INTRODUÇÃO

O presente relatório foi produzido para a atividade 01 da disciplina de Estruturas de Dados Básicas 2, pelo grupo composto por Hélio Lima Carvalho, Pedro Galvão do Amaral Neto, Raoni Marti Silvestre Silva e Thiago Freire Cavalcante.

Os pseudocódigos foram produzidos por Hélio Lima Carvalho, a análise teórica foi realizada por Thiago Freire Cavalcante e Raoni Marti Silvestre Silva. As implementações solicitadas pela atividade foram realizadas por todos os componentes do grupo.

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:

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

1.2.2.3 MÉTODO DA ARVORE DE RECURSÃO A árvore de chamadas do bubbleSort começa com um nó raiz que representa o problema original de tamanho n que dará origem a um filho, que por sua vez, dará origem a um filho e assim por diante até atingirmos o caso base, em que o tamanho do array é $n = 1$, como mostrado a seguir:

```
T(n)------------------1n
  |
  |

T(n-1)----------------1(n-1)
  |
  |
T(n-2)----------------1(n-2)
 ...
T(1)
```

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

Calculando h:

$$\begin{aligned} T(n-h) = T(1) &\implies n - h = 1 \\ &\implies h = n - 1 \end{aligned}$$

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

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{h} (n - i) \\
&= \sum_{i=0}^{h} n - \sum_{i=0}^{h} i \\
&= n^2 + \frac{n(n-1)}{2} \\
&= O(n^2)
\end{aligned}
$$

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 Para todos os casos, teremos:

Caso base:

O caso n = 1 quebra pois log(1) = 0, então usaremos n = 2 como caso base.

$n = 2; T(2) = 2(1) + 2 = 4$

$$T(2) \leq 2c \log 2 \implies 4 \leq 2 * c * 1 \implies c \geq 2$$

Passo indutivo:

Hipotese: $T(k) \leq kc \log k, (\forall k)[1 \leq k \leq n]$

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

Logo $T(n) = \Theta(n \log n)$.

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:

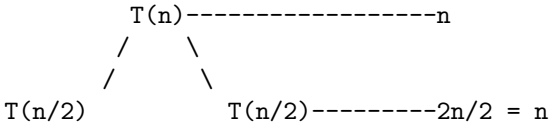
$$
\begin{aligned}
T(n) &= 2^{log_2 n} T(\frac{n}{2^{log_2 n}}) + n.log_2 n \\
&= n.T(\frac{n}{n}) + n.log_2 n \\
&= n + n.log_2 n
\end{aligned}
$$

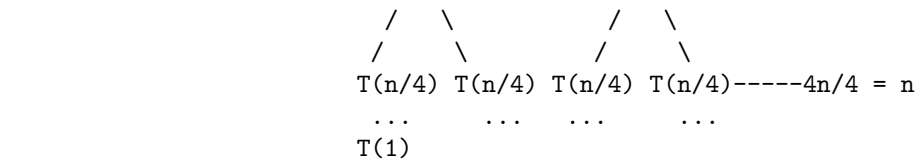
Portanto, a complexidade total é:

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

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:





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:

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

1.3 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:

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

1.3.2.1 MÉTODO DA SUBSTITUIÇÃO Para o pior caso do quicksort, teremos como recorrência $T(n) = T(n-1) + n$.

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).

Para o melhor caso, teremos como recorrência $T(n) = 2T(\frac{n}{2}) + n$.

Caso base:

O caso n = 1 quebra pois log(1) = 0, então usaremos n = 2 como caso base.

$n = 2; T(2) = 2(1) + 2 = 4$

$$T(2) \leq 2c \log 2 \implies 4 \leq 2 * c * 1 \implies c \geq 2$$

Passo indutivo:

Hipotese: $T(k) \leq kc \log k, (\forall k)[1 \leq k \leq n]$

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

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

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 A árvore de chamadas do quickSort no pior caso começa com um nó raiz que representa o problema original de tamanho n que dará origem a um filho, que por sua vez, dará origem a um filho e assim por diante até atingirmos o caso base, em que o tamanho do array é $n = 1$, como mostrado a seguir:

```
T(n)------------------1n
  |
  |

T(n-1)----------------1(n-1)
  |
  |
T(n-2)----------------1(n-2)
 ...
T(1)
```

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

Calculando h:

$$\begin{aligned} T(n-h) = T(1) &\implies n - h = 1 \\ &\implies h = n - 1 \end{aligned}$$

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

$$\begin{aligned} T(n) &= \sum_{i=0}^{h}(n-i) \\ &= \sum_{i=0}^{h} n - \sum_{i=0}^{h} i \\ &= n^2 + \frac{n(n-1)}{2} \\ &= O(n^2) \end{aligned}$$

Já no melhor caso, o quickSort se comporta igual ao mergeSort.

1.3.2.4 MÉTODO DO TEOREMA MESTRE Pelo Teorema Mestre, podemos resolver uma recorrência que possua a forma:

$$T(n) = aT(\frac{n}{b}) + \Theta(n^k)$$

sendo $a, b > 1$ e $k \geq 0$.

Para a recorrência do quickSort no pior caso, não é possível aplicar o teorema.

No melhor caso, a recorrência é igual ao mergeSort, logo temos $a = 2$, $b = 2$ e $k = 1$.

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

Note que

$$2 = 2^1 \implies a = b^k$$

O teorema Mestre diz que, nesse caso, $T(n)$ é $\theta(n^k \log n)$. Portanto,

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

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), ou seja, 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). Dessa forma, a complexidade de idadeRep2 é O(n log n).

2.2.3 Análise de Resultados

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

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.

2.3 FUNÇÕES RECURSIVAS - BUSCABINARIA E BBINREC

2.3.1 BuscaBinaria

A função buscaBinaria implementa o algoritmo de busca binária, em sua versão iterativa, para encontrar a posição de um elemento (chave) em um vetor ordenado. Inicialmente, os índices da esquerda (esq) e direita (dir) são definidos como os extremos do vetor. Em cada iteração do loop while, o índice do meio (m) é calculado. Se o valor no índice m for menor que a chave, o índice esquerdo

é movido para m + 1; caso contrário, o índice direito é ajustado para m. Esse processo continua até que esq e dir se encontrem. Quando o loop termina, a função verifica se o elemento encontrado é a chave buscada; se for, retorna o índice esq, caso contrário, retorna -1.

Essa implementação percorre a lista em tempo O(log n), onde n é o tamanho do vetor, já que a cada iteração o intervalo de busca é reduzido pela metade. Essa implementação é eficiente para vetores grandes.

2.3.2 bBinRec

A função bBinRec implementa a busca binária de forma recursiva para encontrar a posição de uma chave em um vetor. A função começa verificando se o intervalo de busca está esgotado: se esq for maior ou igual a dir e o elemento no índice esq não for igual à chave, a função retorna -1, indicando que a chave não foi encontrada. Em seguida, calcula o índice do meio (m) como o ponto médio entre esq e dir. Se o elemento no índice m for igual à chave, m é retornado como o índice onde a chave foi encontrada.

Caso contrário, a função realiza uma chamada recursiva: se o valor em vetor[m] for maior que a chave, chama-se bBinRec para o subvetor da esquerda (esq até m - 1); se for menor, a chamada é feita para o subvetor da direita (m + 1 até dir). Esse processo de divisão continua até que a chave seja encontrada ou que o intervalo de busca se esgote.

A complexidade da função bBinRec é O(log n), onde n é o número de elementos no vetor, pois a cada chamada recursiva o intervalo de busca é reduzido pela metade.

2.3.3 Análise de Resultados

A tabela a seguir mostra os resultados de tempo de execução das funções buscaBinaria e bBinRec com vetores de 100, 1000 e 10000 elementos.

Tamanho	buscaBinaria	bBinRec
100	5 s	4 s
1000	6 s	4 s
10000	7 s	4 s

É possível perceber nos resultados apresentados acima que a função bBinRec é consistentemente mais rápida que buscaBinaria para os três tamanhos de vetor testados (100, 1000 e 10000 elementos). Enquanto buscaBinaria apresenta um aumento gradual no tempo de execução conforme o tamanho do vetor cresce (de 5 s para 7 s), bBinRec mantém um tempo constante de 4 s para todos os tamanhos. Essa diferença pode ser atribuída à natureza iterativa de buscaBinaria, que, embora eficiente, envolve verificações adicionais a cada iteração. Em

contrapartida, a recursão de bBinRec permite uma eliminação direta de metade do vetor em cada chamada sem repetir verificações, o que resulta em uma leve vantagem de desempenho.

Entretanto, a recursão pode consumir mais memória de pilha, o que, para vetores muito grandes, pode se tornar uma limitação. A partir desses resultados, pode-se concluir que bBinRec é mais eficiente em termos de tempo, especialmente para vetores de tamanho moderado, embora a eficiência de memória de buscaBinaria possa ser preferível em casos específicos onde o uso de pilha é uma preocupação.

2.4 ALGORITMOS DE ORDENAÇÃO - BUBBLESORT, QUICKSORT E MERGESORT

2.4.1 Bubblesort

2.4.1.1 Versão Iterativa A função iterative_bubble_sort implementa o algoritmo de ordenação bolha (ou bubble sort) de forma iterativa em Rust. A função primeiro calcula o tamanho do vetor (len), então utiliza dois loops aninhados para percorrer o vetor e ordenar seus elementos em ordem crescente. No loop externo, que controla o número de passagens, o índice i representa a quantidade de elementos já ordenados ao final do vetor. O loop interno, controlado por j, percorre os elementos não ordenados e compara cada par adjacente (array[j] e array[j + 1]). Caso array[j] seja maior que array[j + 1], os dois elementos são trocados de lugar com array.swap(j, j + 1). Esse processo é repetido até que o vetor esteja completamente ordenado.

A complexidade de tempo do iterative_bubble_sort é $O(n^2)$ no pior e no caso médio, onde n é o número de elementos no vetor. Isso ocorre porque, para cada elemento, o algoritmo potencialmente percorre todo o vetor novamente, resultando em $\times$ comparações. A complexidade no melhor caso é O(n), quando o vetor já está ordenado, pois o algoritmo pode ser otimizado para detectar que não há necessidade de trocas e parar antes.

2.4.1.2 Versão Recursiva A função recursive_bubble_sort implementa o algoritmo de ordenação bolha (ou bubble sort) de maneira recursiva em Rust. A função recursive_bubble_sort inicia a chamada à função auxiliar recursive_bubble, que aplica o algoritmo de ordenação bolha recursivamente. Na recursive_bubble, a função primeiro verifica se o tamanho do vetor (n) é 1, caso em que a ordenação está completa e a recursão termina. Caso contrário, recursive_bubble realiza uma passagem de ordenação chamando a função bubble_pass, que compara e troca elementos adjacentes recursivamente ao longo do vetor. Após essa passagem, recursive_bubble é chamada novamente com n - 1, reduzindo gradativamente o tamanho do vetor a ser ordenado até que todos os elementos estejam na posição correta.

A complexidade de tempo dessa implementação é $O(n^2)$ no pior e no caso médio, onde n é o número de elementos no vetor, semelhante ao bubble sort iterativo.

Isso ocorre porque a função precisa realizar n passagens, cada uma envolvendo até n comparações recursivas em bubble_pass. No melhor caso, onde o vetor já está ordenado, a função ainda possui complexidade $O(n^2)$, pois a implementação atual não detecta se o vetor já está ordenado e sempre executa as passagens completas. A implementação recursiva ocupa mais espaço na pilha devido às chamadas recursivas, o que pode ser uma limitação em casos de vetores muito grandes.

2.4.1.3 Análise de Resultados A tabela a seguir mostra os resultados de tempo de execução das implementações da ordenação de bolha (bubblesort) iterativa e recursiva com vetores de 1000, 10000 e 100000 elementos.

Tamanho	Bubblesort Iterativo	Bubblesort Recursivo
1000	3.20ms	3.67ms
10000	338.42ms	360.27ms
100000	34.23s	35.83s

Ambos os algoritmos exibem tempos de execução semelhantes em cada tamanho de vetor, mas a implementação iterativa é consistentemente um pouco mais rápida. Com 1000 elementos, o bubble sort iterativo executa em 3.20 ms, enquanto o recursivo demora 3.67 ms. À medida que o tamanho do vetor aumenta, essa diferença se torna mais significativa: com 10000 elementos, o iterativo leva 338.42 ms contra 360.27 ms para o recursivo, e com 100000 elementos, o iterativo demora 34.23 s, enquanto o recursivo leva 35.83 s.

Essa diferença ocorre porque o bubble sort recursivo tem o mesmo comportamento de tempo assintótico ($O(n^2)$) que a versão iterativa, mas carrega o custo adicional de múltiplas chamadas recursivas, que consomem mais memória de pilha e exigem mais tempo para o gerenciamento das chamadas de função. A recursão, portanto, se torna um fator de sobrecarga crescente conforme o tamanho do vetor aumenta.

2.4.2 Quicksort

2.4.2.1 Versão Iterativa A função iterative_quick_sort implementa o algoritmo de ordenação rápida (ou quick sort) de forma iterativa em Rust. Em vez de usar chamadas recursivas, esta implementação utiliza uma pilha (stack) para armazenar os índices das sublistas a serem ordenadas. Inicialmente, os índices do início (0) e do fim (tamanho - 1) do vetor são empilhados. Em seguida, a função entra em um loop while, onde o limite direito (right) e o limite esquerdo (left) das sublistas são retirados da pilha. A função particionar é chamada para particionar o vetor, colocando elementos menores que o pivô à esquerda e maiores à direita, retornando o índice do pivô final.

Após a partição, a função verifica se há sublistas à esquerda e à direita do pivô que precisam ser ordenadas. Se houver, seus índices são empilhados para serem

processados em iterações subsequentes. Esse processo continua até que a pilha esteja vazia, indicando que o vetor está totalmente ordenado.

A complexidade de tempo do iterative_quick_sort é O(n log n) no caso médio e $O(n^2)$ no pior caso, onde n é o número de elementos no vetor. A eficiência de O(n log n) no caso médio ocorre porque a função divide repetidamente o vetor ao meio, ordenando cada metade de forma independente. No entanto, no pior caso (por exemplo, se o vetor já estiver quase ordenado ou se sempre for escolhido um pivô ineficiente), a complexidade pode degradar para $O(n^2)$.

2.4.2.2 Versão Recursiva A função recursive_quick_sort implementa o algoritmo de ordenação rápida (ou quick sort) de maneira recursiva em Rust. A função inicializa a ordenação chamando a função sorting, que é responsável por particionar o vetor e chamar-se recursivamente em cada sublista. Na sorting, é verificado se o índice esquerdo (left) é menor que o direito (right). Se for, a função partition é chamada para reorganizar o vetor de forma que todos os elementos menores que o pivô fiquem à esquerda e os maiores à direita, retornando o índice final do pivô (index_pivot). A sorting então chama-se recursivamente para ordenar as sublistas à esquerda e à direita do pivô.

A função partition utiliza o primeiro elemento da sublista como pivô e faz comparações a partir dos extremos (left e right) até que ambos se cruzem. Durante esse processo, ela compara os elementos com o pivô e realiza trocas, ajustando os limites left e right conforme necessário. Quando o loop termina, o pivô é trocado com o elemento em right, completando a partição.

A complexidade de tempo da recursive_quick_sort é O(n log n) no caso médio e $O(n^2)$ no pior caso, onde n é o número de elementos, similar à sua implementação iterativa. O caso médio ocorre quando o vetor é dividido aproximadamente ao meio em cada partição, enquanto o pior caso ocorre quando o pivô escolhido resulta em partições muito desbalanceadas, como em vetores quase ordenados.

2.4.2.3 Análise de Resultados A tabela a seguir mostra os resultados de tempo de execução das implementações da ordenação rápida (quicksort) iterativa e recursiva com vetores de 1000, 10000 e 100000 elementos.

Tamanho	Quicksort Iterativo	Quicksort Recursivo
1000	147.87µs	149.96µs
10000	1.21ms	1.77ms
100000	20.38ms	55.42ms

Em todas as instâncias, o quicksort iterativo apresenta tempos de execução mais rápidos do que a versão recursiva, e essa diferença se torna mais pronunciada à medida que o tamanho do vetor aumenta. Para um vetor de 1000 elementos, a diferença é pequena: o quicksort iterativo leva 147.87 µs, enquanto o recursivo leva 149.96 µs. No entanto, com 10000 elementos, o iterativo demora 1.21

ms, enquanto o recursivo leva 1.77 ms. Entretanto, para 100000 elementos, a diferença aumenta de forma significativa, com o quicksort iterativo executando em 20.38 ms e o recursivo demorando 55.42 ms. A diferença nos tempos de execução com o vetor de 100000 elementos indica algum tipo de ineficiência ligada ao maior custo espacial da implementação recursiva.

Esses resultados indicam que a implementação iterativa é mais eficiente, principalmente para vetores maiores, devido ao menor custo de gerenciamento de chamadas de função e uso de memória. O quicksort recursivo, embora seja uma abordagem tradicional e intuitiva para esse algoritmo, acumula sobrecarga de memória de pilha devido às sucessivas chamadas recursivas, o que leva a um desempenho inferior conforme o vetor cresce.

2.4.3 Mergesort

2.4.3.1 Versão Iterativa A função iterative_merge_sort implementa o algoritmo de ordenação por mesclagem (merge sort) de maneira iterativa em Rust. A função começa chamando a função merge_sorting, que divide e ordena o vetor em pedaços de tamanho crescente, substituindo o vetor original pelo resultado ordenado. Em merge_sorting, o vetor é particionado em sublistas de tamanho i, que dobra em cada iteração do while externo, permitindo que pedaços ordenados sejam mesclados em intervalos crescentes até que todo o vetor seja ordenado. Dentro do loop interno, a função identifica os limites mid e right_arr de cada sublista a ser mesclada e chama a função auxiliar merge para combinar as duas partes.

A função merge recebe duas sublistas (arr_1 e arr_2) e as combina em uma única lista ordenada (result). Ela usa dois índices (i e j) para percorrer ambas as sublistas, adicionando os elementos menores a result em cada etapa até que todos os elementos de uma das listas sejam esgotados. Os elementos restantes de qualquer uma das sublistas são então adicionados a result.

A complexidade de tempo do iterative_merge_sort é O(n log n) tanto no caso médio quanto no pior caso, onde n é o número de elementos no vetor. Isso ocorre porque o algoritmo divide o vetor em duas metades repetidamente e, em seguida, mescla as metades ordenadas, cada etapa exigindo O(n) operações.

2.4.3.2 Versão Recursiva A função recursive_merge_sort implementa o algoritmo de ordenação por mesclagem (merge sort) de forma recursiva em Rust. A função principal chama merge_sorting, que divide recursivamente o vetor em duas metades até que cada sublista contenha apenas um elemento. A cada divisão, merge_sorting calcula o índice central (mid), particiona o vetor em duas sublistas (left_arr e right_arr), e então chama-se recursivamente para ordenar cada uma dessas metades. Em seguida, as duas sublistas ordenadas (left e right) são combinadas pela função auxiliar merge, que mescla ambas em uma única lista ordenada.

A função merge recebe as duas sublistas (arr_1 e arr_2) e percorre-as utilizando dois índices (i e j). Ela compara os elementos das duas sublistas e adiciona o menor elemento ao vetor de resultado (result) em cada passo, até que todos os elementos de uma das listas sejam adicionados. Depois disso, os elementos restantes da outra sublista são adicionados a result.

A complexidade de tempo do recursive_merge_sort é O(n log n) tanto no caso médio quanto no pior caso, onde n é o número de elementos no vetor. Isso ocorre porque a função divide o vetor repetidamente em duas metades, realizando comparações e mesclagens, cada etapa exigindo O(n) operações.

2.4.3.3 Análise de Resultados A tabela a seguir mostra os resultados de tempo de execução das implementações da ordenação de mesclagem (mergesort) iterativa e recursiva com vetores de 1000, 10000 e 100000 elementos.

Tamanho	Mergesort Iterativo	Mergesort Recursivo
1000	463.17µs	550.43µs
10000	4.14ms	5.16ms
100000	40.57ms	49.43ms

Em todas as situações, a versão iterativa é mais rápida do que a recursiva, e essa diferença se acentua conforme o tamanho do vetor aumenta. Para um vetor de 1000 elementos, o mergesort iterativo leva 463.17 µs, enquanto o recursivo demora 550.43 µs. Com um vetor de 10000 elementos, o iterativo executa em 4.14 ms e o recursivo em 5.16 ms. Já para 100000 elementos, o iterativo leva 40.57 ms, enquanto o recursivo demora 49.43 ms.

Esses resultados indicam que o mergesort iterativo é consistentemente mais eficiente do que o recursivo, principalmente para vetores maiores. Embora ambos possuam complexidade de tempo O(n log n), a implementação recursiva exige mais recursos de memória de pilha devido às chamadas recursivas, o que resulta em maior sobrecarga de execução. Em contraste, a versão iterativa evita essas chamadas e, consequentemente, é mais rápida e eficiente em termos de uso de memória.

Conclui-se que, embora a versão recursiva do mergesort seja conceitualmente mais próxima do design do algoritmo, a versão iterativa oferece desempenho superior, especialmente para grandes conjuntos de dados. Para listas menores, a diferença é menos significativa, mas a versão iterativa ainda se mostra vantajosa em termos de eficiência.

2.4.4 Comparação entre Algoritmos

Entre os algoritmos analisados, quicksort e mergesort são muito mais eficientes que o bubble sort, especialmente com vetores maiores. Ambos possuem complexidade O(n log n), o que os torna adequados para grandes conjuntos de dados.

O bubble sort, por outro lado, com complexidade $O(n^2)$, é viável apenas para listas muito pequenas.

Em todos os algoritmos, as versões iterativas foram consistentemente mais rápidas que as recursivas, principalmente em vetores grandes. As versões recursivas consomem mais memória de pilha e enfrentam sobrecarga de chamada de função, resultando em tempos de execução ligeiramente maiores. Essa diferença é menos visível em vetores menores, mas se intensifica em conjuntos maiores.

Para cenários onde a eficiência e o uso de memória são essenciais, quicksort e mergesort iterativos são as melhores escolhas. O bubble sort, devido à sua ineficiência, deve ser evitado em casos práticos de ordenação de grandes conjuntos de dados.

Dessa forma, os resultados mostram que o quicksort e o mergesort (principalmente as versões iterativas) são altamente eficazes para ordenar grandes vetores, enquanto o bubble sort é impraticável para a maioria dos casos.