Análise dos Resultados e Complexidades

Nome: Gabriel Henrique Silva Duque **R.A:** 0082574

Analisando o gráfico e os tempos obtidos, dá pra perceber que os algoritmos Bubble Sort, Insertion Sort e Selection Sort ficam bem mais lentos à medida que o tamanho dos vetores aumenta. Isso confirma o que vimos em aula sobre o comportamento quadrático deles ($O(n^2)$).

Já o Quick Sort e o Merge Sort se saíram muito melhor, com tempos bem menores e um crescimento mais suave mesmo quando os vetores aumentam bastante. Isso está de acordo com a complexidade O(n log n), que é típica desses dois algoritmos. Quando os vetores são pequenos, a diferença entre eles não é tão grande, provavelmente por causa das constantes envolvidas e da sobrecarga das chamadas recursivas.

Mas, à medida que o tamanho dos vetores cresce, fica evidente que os algoritmos com complexidade O(n log n) são muito mais eficientes do que os quadráticos.

Ranking dos Algoritmos com Base no Tempo Total de Execução

1. Quick Sort - Apresentou o menor tempo total

2. Merge Sort - Teve desempenho muito próximo ao Quick Sort, mas não tão bom quanto

3. Insertion Sort - Apesar de ser um algoritmo quadrático, sua simplicidade faz com que se destaque frente aos outros algoritmos quadráticos para tamanhos menores

4. Selection Sort - Apresentou desempenho inferior ao Insertion Sort, devido ao maior número de trocas realizadas

5. Bubble Sort - Foi o algoritmo com pior desempenho, em linha com o esperado dada a sua ineficiência para vetores maiores.

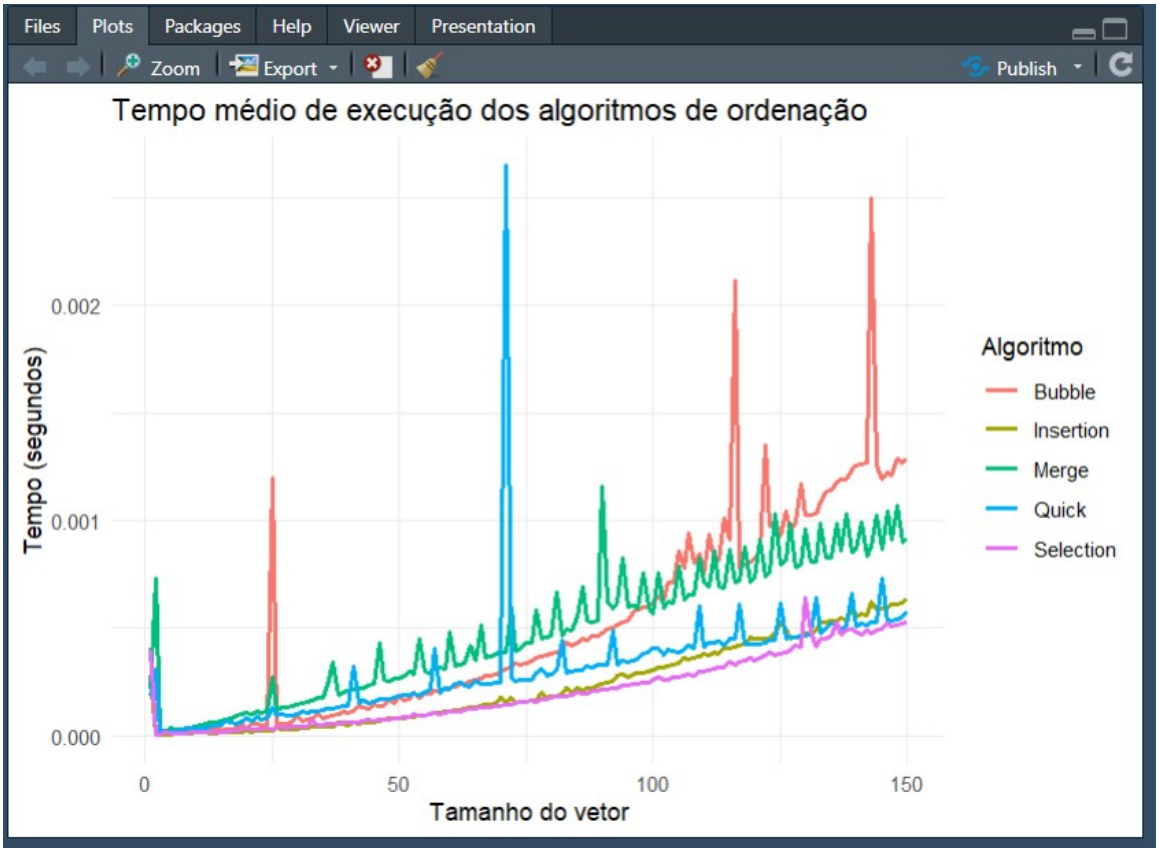
Conclusão

Apesar de o Quick Sort ter mostrado o melhor desempenho médio nos testes, é bom lembrar que ele depende bastante da escolha do pivô e de como o particionamento dos dados é feito. Quando esse particionamento não é dos melhores - como em vetores quase ordenados ou com muitos elementos repetidos - o desempenho pode cair bastante, chegando até ao pior caso, com complexidade $O(n^2)$.

Por outro lado, o Merge Sort tem uma performance mais estável e previsível. Ele mantém a complexidade O(n log n) mesmo que os dados já estejam ordenados ou totalmente bagunçados. Isso faz com que ele seja uma opção mais segura quando a consistência no tempo de execução é importante, mesmo exigindo mais memória por precisar de espaço extra para funcionar.

Então, mesmo que o Quick Sort geralmente seja mais rápido, principalmente quando se usa boas estratégias pra escolher o pivô, o Merge Sort acaba sendo mais confiável em situações que exigem estabilidade e segurança na ordenação, especialmente em grandes volumes de dados.

Gráfico gerado no Rstudio



Código R do Experimento

```
#Algoritmos de Ordenação

rm(list = ls())

#Merge
Merge <- function(A, inicio, meio, fim) {
  esquerda <- A[inicio:meio]
  direita <- A[(meio + 1):fim]
  i <- 1
  j <- 1
  k <- inicio

  while (i <= length(esquerda) && j <= length(direita)) {
    if (esquerda[i] <= direita[j]) {
      A[k] <- esquerda[i]
      i <- i + 1
    } else {
      A[k] <- direita[j]
      j <- j + 1
    }
    k <- k + 1
  }

  while (i <= length(esquerda)) {
    A[k] <- esquerda[i]
    i <- i + 1
    k <- k + 1
  }

  while (j <= length(direita)) {
    A[k] <- direita[j]
    j <- j + 1
    k <- k + 1
  }

  return(A)
}
```

```
#MergeSortRecursivo
MergeSortRecursivo <- function (A, inicio, fim) {
  if (inicio < fim) {
    meio = floor((inicio + fim) / 2)
    A = MergeSortRecursivo(A, inicio, meio)
    A = MergeSortRecursivo(A, meio + 1, fim)
    A = Merge(A, inicio, meio, fim)
  }
  return (A)
}

#QuickSortRecursivo
QuickSortRecursivo <- function(A, inicio, fim) {
  if (inicio < fim) {
    resultado <- Particionar(A, inicio, fim)
    A <- resultado$vetor
    p <- resultado$p
    A <- QuickSortRecursivo(A, inicio, p - 1)
    A <- QuickSortRecursivo(A, p + 1, fim)
  }
  return(A)
}

#Particionamento
Particionar <- function(A, inicio, fim) {
  pivot <- A[fim]
  i <- inicio - 1

  for (j in inicio:(fim - 1)) {
    if (A[j] <= pivot) {
      i <- i + 1
      temp <- A[i]
      A[i] <- A[j]
      A[j] <- temp
    }
  }
  temp <- A[i + 1]
  A[i + 1] <- A[fim]
  A[fim] <- temp
```

```
  return(list(vetor = A, p = i + 1))
}

#BubbleSortIterativo
BubbleSort <- function(A, n) {
  n <- length(A)
  if (n < 2)
    return(A)  # se o vetor tem 0 ou 1 elemento, já está ordenado

  for (i in 1:(n - 1)) {
    swap <- FALSE
    for (j in 1:(n - i)) {
      if (A[j] > A[j + 1]) {
        temp <- A[j]
        A[j] <- A[j + 1]
        A[j + 1] <- temp
        swap <- TRUE
      }
    }
    if (!swap)
      break
  }

  return(A)
}



#InsertionSortIterativo
InsertionSort <- function(A, n) {
  n <- length(A)
  if (n < 2)
    return(A)

  for (i in 2:n) {
    chave <- A[i]
    j <- i - 1
    while (j >= 1 && A[j] > chave) {
      A[j + 1] <- A[j]
      j <- j - 1
    }
```

```
    A[j + 1] <- chave
  }
  return(A)
}

#SelectionSortIterativo
SelectionSort <- function(A, n) {
  n <- length(A)
  if (n < 2)
    return(A)

  for (i in 1:(n - 1)) {
    min <- i
    for (j in (i + 1):n) {
      if (A[j] < A[min]) {
        min <- j
      }
    }
    if (min != i) {
      temp <- A[i]
      A[i] <- A[min]
      A[min] <- temp
    }
  }
  return(A)
}

# Função pra gerar vetor aleatório
gerar_vetor <- function(n) {
  sample(1:(n * 10), n, replace = TRUE)
}

# Vetores pra armazenar os tempos
tempos_bubble <- numeric(150)
tempos_insertion <- numeric(150)
tempos_selection <- numeric(150)
tempos_quick <- numeric(150)
tempos_merge <- numeric(150)
```

```
# Número de execuções
nexec <- 30

# Função aux pra medir tempo médio
medir_tempo_medio <- function(func, tamanho) {
  tempos <- sapply(1:nexec, function(i) {
    vetor <- gerar_vetor(tamanho)
    inicio <- Sys.time()
    func(vetor, tamanho)
    fim <- Sys.time()
    as.numeric(difftime(fim, inicio, units = "secs"))
  })
  mean(tempos)
}

# Função para medir tempo de algoritmos
medir_tempo_medio_qm <- function(func, tamanho) {
  tempos <- sapply(1:nexec, function(i) {
    vetor <- gerar_vetor(tamanho)
    inicio <- Sys.time()
    func(vetor, 1, tamanho)
    fim <- Sys.time()
    as.numeric(difftime(fim, inicio, units = "secs"))
  })
  mean(tempos)
}


# Execução 1 a 150
tempos_bubble    <- sapply(1:150, function(n)
  medir_tempo_medio(BubbleSort, n))
tempos_insertion <- sapply(1:150, function(n)
  medir_tempo_medio(InsertionSort, n))
tempos_selection <- sapply(1:150, function(n)
  medir_tempo_medio(SelectionSort, n))
tempos_quick     <- sapply(1:150, function(n)
  medir_tempo_medio_qm(QuickSortRecursivo, n))
tempos_merge     <- sapply(1:150, function(n)
  medir_tempo_medio_qm(MergeSortRecursivo, n))
```

```
#Gráfico dos algoritmos
library(ggplot2)

dados <- data.frame(
  Tamanho = rep(1:150, 5),
  Tempo = c(
    tempos_bubble,
    tempos_insertion,
    tempos_selection,
    tempos_quick,
    tempos_merge
  ),
  Algoritmo = factor(rep(
    c("Bubble", "Insertion", "Selection", "Quick", "Merge"), each
= 150
  ))
)
ggplot(dados, aes(x = Tamanho, y = Tempo, color = Algoritmo)) +
  geom_line(size = 1) +
  labs(title = "Tempo médio de execução dos algoritmos de
ordenação", x = "Tamanho do vetor", y = "Tempo (segundos)") +
  theme_minimal()
```