Relatório trabalho prático 1

César A. Galvão 19/0011572 Gabriela Carneiro 18/0120816

26 de June de 2022

Contents

Resumo

Nesta atividade foram implementadas em R três métodos de ordenação – seleção, inserção e Quick Sort – e seus desempenhos foram medidos por meio de tempo de execução, número de comparações entre elementos do vetor e número de movimentações dos elementos do vetor. Os três métodos foram testados com tamanhos variados de amostras, variando o tamanho de 500 a 50.000, assim como vetores previamente organizados de quatro formas: ordenados, ordenados de maneira inversa, parcialmente aleatorizados (cerca de 10% dos elementos têm posições aleatórias) e completamente aleatorizados. As duas últimas formas foram executadas 100 vezes e as medidas de interesse são representadas por suas médias. Enquanto o método de seleção fez maior número de comparações, observou-se que o método de inserção fez maior número e movimentações. Conclui-se que o método de ordenação por Quick sort é o mais eficiente dos três que foram analisados.[1]

[1] Todos os documentos desse relatório podem ser verificados no repositório https://github.com/cesar-galvao/Estatistica-computacional

1 Introdução

1.1 Métodos de ordenação

O ordenamento de dados tem papel crucial no campo das ciências da computação. Os vários tipos de algoritmos de ordenação diferem entre si em termos de eficiência, uso de memória, complexidade, dentre outros fatores. Por esse motivo, esses algoritmos tem sito extensamente analisados por meio de pesquisas. Nesse contexto, cada algoritmo desenvolvido para ordenamento de dados tem seus benefícios e limitações, sendo que o desempenho deles pode ser otimizado de diversas maneiras (Zutshi & Goswami, 2021)[2].

Na ciência da computação, a eficiência entre algoritmos é comparada geralmente em termos do tempo de execução. Muitos dos algoritmos de ordenação têm complexidade linear ou quadrática. Destes muitos tem complexidade quadrática $O(n^2)$ como pior caso e complexidade linear como melhor caso, sendo que o desempenho dos algoritmos $O(n^2)$ tende a cair conforme a quantidade de dados aumenta. Ainda, alguns algoritmos tem a complexidade $O(n \log n)$ tanto no pior caso quanto em seu melhor caso (Zutshi & Goswami, 2021).

1.1.1 Ordenação por seleção

O método de ordenação por seleção percorre a sequencia de dados e "seleciona" cada elemento que não está em sua posição correta. Em seguida, ele o troca de lugar com o elemento que está na primeira posição. O algoritmo executa esse mesmo procedimento para o restante da lista de dados.

Estas operações quando executadas têm complexidade quadrática, sendo que a busca pelo menor elemento da operação custa $n-1$ passos na primeira iteração. Em seguida a operação custa $n-2$ na segunda operação e assim por diante. Dessa forma, o custo total é dado pela soma da progressão aritmética $\sum_{i=1}^{n-1} a_i$ de razão $r = 1$, com $a_1 = 1$ e $a_{n-1} = (n-1)$. Dessa forma, o tempo de execução do algoritmo é:

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

1.1.2 Ordenação por inserção

Esse tipo de método de ordenação "insere" cada elemento desordenado dos dados em sua posição correta. Ou seja, esse tipo de ordenação tem como rotina base a inserção ordenada, que basicamente compara o elemento que está sendo ordenado com os elementos anteriores a ele e só efetua uma movimentação quando tal elemento é maior que elemento imediatamente e sua esquerda ou quando ele ocupa a primeira posição da sequência. As operaçãoes, assim como o método do ordenação por seleção, são de ordem quadrática, em seu pior caso. Em seu melhor caso, quando os elementos já estão ordenados, o custo total é de $O(n)$.

Assim como no caso de ordenação mencionado anteriomente, o custo total é dado pela soma da progressão aritmética $\sum_{i=1}^{n-1} a_i$ de razão $r = 1$, com $a_1 = 1$ e $a_{n-1} = (n-1)$. Dessa forma, o tempo de execução do algoritmo é:

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

[2]Zutshi, A, Goswami, D. (2021) *Systematic review and exploration of new avenues for sorting algorithm*

1.1.3 Quick Sort

O Quick Sort geralmente é implementado recursivamente, como foi realizado neste relatório. Um pivô aleatório é escolhido e todos os elementos são reorganizados em torno do pivô em ordem crescente mantendo-o como referência. A escolha do pivô determina a complexidade do Quick Sort sendo que em algumas versões, nos melhores casos, o algoritmo tem complexidade de $O(n \log n)$. No entanto, se o pivô for selecionado em uma das extremidades da sequencia, ele passa a ter complexidade de $O(n^2)$.

Escolher um pivô aleatorimente é um boa estratégia para diminuir significativamente a probabilidade de ocorrência do pior caso, já que, para o pior caso acontecer, o pivô escolhido deveria ser sempre o pior tendo uma probabilidade de ocorrência

$$p = \frac{1}{n!}. \tag{3}$$

1.1.4 Comparação entre métodos de ordenação

Comparando os diferentes tipos de estratégia de ordenação, é esperado que o método de ordenação por seleção efetue menos trocas do que o método de ordenação por inserção, pois há uma troca apenas por iteração. Desse modo, o algoritmo efetua n trocas. Já o método de inserção efetua ao menos uma troca por iteração. Contudo, este efetua menos comparações do que o algoritmo de seleção, pois nem sempre o elemento a ser inserido de forma ordenada deve ir até o final do vetor. Isso só ocorre no pior caso, isto é, quando os elementos estão ordenado em ordem decrescente. O algoritmo de seleção precisa comparar todos os elementos restantes a cada iteração para determinar qual é o menor deles.

Ambos os métodos citados acima estão na mesma classe de complexidade ($O(n^2)$). No entanto, o método de inserção apresenta melhor desempenho do que o método de seleção na prática.

Entre os três métodos, o Quick Sort é teoricamente o mais eficiente, tendo uma complexidade de $O(n \log n)$. Em seu pior cenário, o método atinge a mesma complexidade dos outros dois tipos avaliados.

2 Método

2.1 Testes de desempenho

Os testes de desempenho realizados para cada um dos algoritmos de ordenação tomaram em conta duas dimensões: tamanho de amostra e ordenação prévia do vetor a ser ordenado.

Para poder comparar os métodos, três medidas foram feitas para todos os tamanhos de amostra: contagem da quantidade de movimentações, quantidade de comparações entre elementos e tempo de execução. Vetores ordenados nas duas direções tiveram testes realizados apenas uma vez, enquanto aqueles que possuíam algum elemento aleatório foram realizados cem vezes e os valores considerados para comparação são as médias dessas cem realizações de teste.

2.2 Amostragem

As amostras foram criadas utilizando a função `gera.amostra(size, ordem, seed)` disponível no código anexo. Os argumentos correspondem a tamanho, ordenamento e seed para a geracao da aleatoriedade desejada. A função retorna uma lista de dois elementos: o vetor e o método de ordenação desejado.

O tamanho da amostra permitido na função é qualquer número inteiro. Não foram realizados controles para tamanho 0 e números negativos, porém qualquer número estritamente positivo tem retorno válido. Para os testes, foram utilizados cinco tamanhos de amostra: 500, 1.000, 5.000, 10.000 e 50.000.

O argumento *ordem* da função recebe uma das quatro ordenações testadas, quais sejam: completamente ordenado, inversamente ordenado, completamente aleatorizado e parcialmente aleatorizado. Este caso compreende 10% de aleatorizacao dos elementos do vetor.

2.3 Implementação do método de seleção

O método seleção foi implementado na função `selecao()`, que recebe a lista da função de amostragem como argumento. Inicialmente, o elemento $a_i, i = 1, 2, ..., n-1$, é tomado como mínimo ($min = i$) e é comparado aos elementos $a_j, j = 2, 3, ..., n$ seguintes. Se $a_j < a_{min}$, esses elementos são invertidos e o mínimo é redefinido, $min = j$. essas operações são realizadas em um laço `for` de dois níveis, conforme o código anexo.

2.4 Implementação do método de inserção

O método seleção foi implementado na função `insercao()`, que recebe a lista da função de amostragem como argumento. Considerando $i = 2, 3, ..., n$ como o iterador primário e $j = 1, 2, ..., n-1$, tem-se x_i, x_j e x_{temp}. Na primeira iteração, defini-se $x_{\text{temp}} = x_i$. Enquanto x_j e $j > 0$, redefine-se $x_{j+1} = x_j$ e $j = j - 1$ até que uma das condições não sejam mais satisfeitas. Neste momento, reatribui-se $x_{j+1} = x_{\text{temp}}$ e itera-se sobre o próximo valor de i.

2.5 Implementação do Quick Sort

Este algoritmo foi implementado em dois níveis: a funcao `quick_base()`, objeto principal de análise, aplica o algoritmo enquanto `quick_completa()` permite o retorno das medidas de interesse. A ordenação inicia com a seleção de um pivô, aqui selecionado como a posição mediana do comprimento do vetor, e redistribui os elementos do vetor. Enquanto os elementos menores que o pivô são passados para a esquerda dele, os maiores são passados para a direita. Em seguida, a função recursivamente opera sobre os braços do vetor até que todo ele esteja ordenado.

3 Resultados

3.1 Tempo de execução

Na medida em que o número de elementos nas amostras cresce, o tempo de execução das ordenações também cresce, conforme esperado. Quando as amostras são geradas com ordenação de elementos em ordem aleatória, é possível observar que para ordená-la os métodos de ordenação por inserção e Quick Sort levam menos tempo que o método de seleção, fato que mostra que o metodo de seleção é mais oneroso que os outro dois. O mesmo pode ser observado quando as amostra está invertida, ordenada ou parcialmente ordenada.

Porém, de forma inesperada para as amostras de tamando 50.000, o método de seleção levou mais tempo para ordenar as amostras que já estavam ordenadas do que as amostras que estavam completamente invertidas. Isso não é esperado, pois no caso das amostras invertidas tanto o método de seleção quanto o método se inserção tem seus piores cenários, tendo desempenho quadrático.

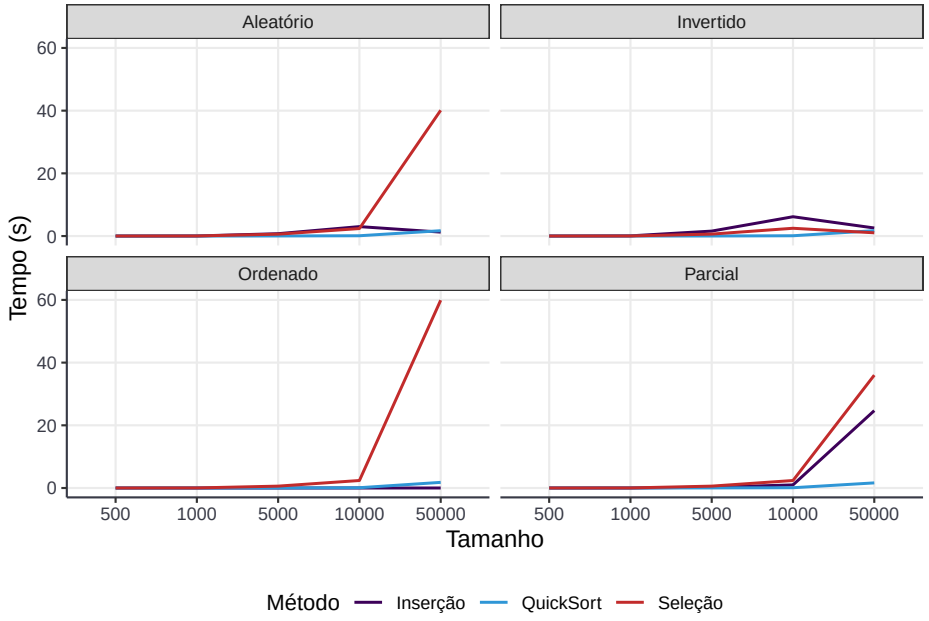

Figure 1: Tempo de execução dos métodos desagregado por ordenamento original da amostra

3.2 Número de comparações

Conforme pode ser observado na figura 2, o número de comparações também tende a crescer conforme o número de elementos das amostras aumenta. Ainda, para todos os tipos de amostra o método de seleção foi o que fez o maior número de comparações, principalmente para as amostras maiores de 10.000 e 50.000 elementos. Esse resultado já é esperado, conforme já citado no item 1.1.4 da introdução. Cabe apontar que, no caso do ordenamento Invertido, a linha correspondente ao método de Inserção está sobreposta àquela do método de Seleção.

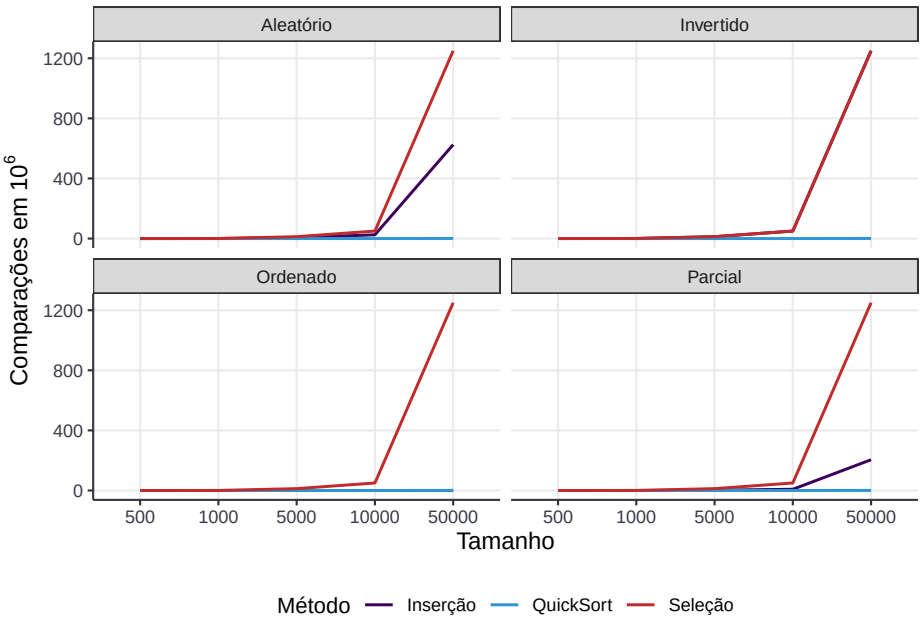

Figure 2: Quantidade de comparações dos métodos desagregado por ordenamento original da amostra

3.3 Número de movimentações

De maneira análoga, o número de movimentações também deve crescer a medida que o número de elementos das amostras cresce. Na figura 3 é possível observar que o método de ordenação por inserção sempre faz o maior número de movimentações, conforme já citado anteriormente no item 1.1.4 da introdução, principalmente quando as amostras estão invertidas, quando esse método tem complexidade quadrática. Nas amostras ordenadas, não houve movimentações o que já era esperado. Esse resuldado corrobora que as implementações da contagem das movimentações foram implementadas de forma correta no desenvolvimentos das funções dos três métodos de ordenação.

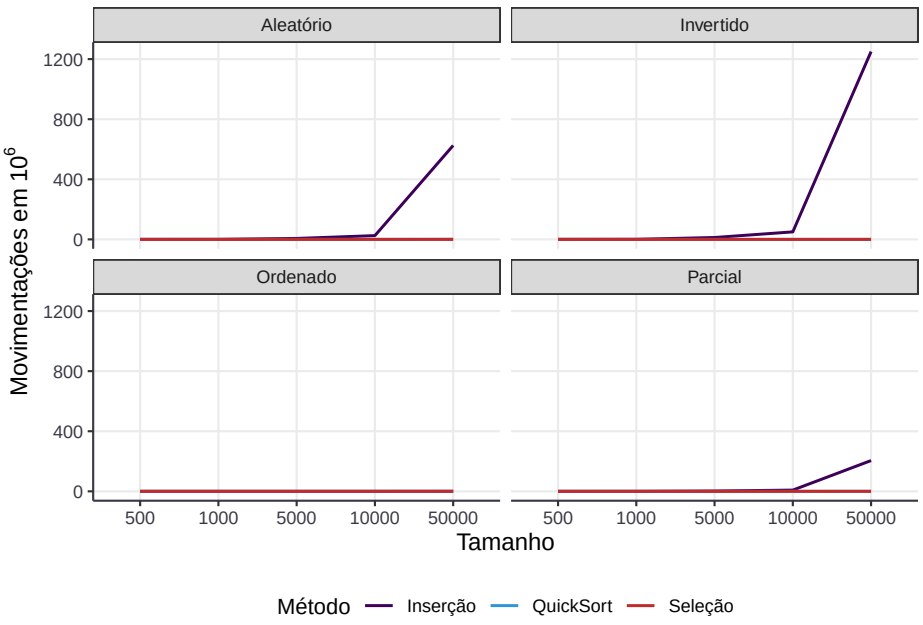

Figure 3: Quantidade de movimentações dos métodos desagregado por ordenamento original da amostra

4 Conclusao

Por meio das análises feitas é possível notar que o número de elementos na amostra afeta o tempo de execução da ordenação, assim como a complexidade da amostra, i.e. se ela está invertida, ordenada ou em alguma ordem aleatória. O tempo de execução das amostras invertidas foi inesperadamente menor que o tempo de execução das amostras já ordenadas, mas não foi possível identificar o motivo deste resultado. Ademais, essas diferenças no tempo de execução só foram percebidas para amostras maiores.

Para o número de comparacões e de movimentações, os algoritmos tiveram os resultados esperados, com o método de seleção fazendo o maior número de comparações que o método de inserção e este fazendo mais movimentações que aquele. Por fim, o Quick Sort foi o mais eficiente dos métodos de ornenação.

Anexo A - tabela com dados de execução dos algoritmos

Método	Tamanho	Ordenamento	Tempo	Comparações	Movimentações
Inserção	500	Aleatório	0.0073818 secs	63175.45	63674.45
Inserção	500	Invertido	0.0159225 secs	125249.00	125748.00
Inserção	500	Ordenado	0.0000772 secs	499.00	998.00
Inserção	500	Parcial	0.0024790 secs	21346.50	21845.50
Inserção	1000	Aleatório	0.0300949 secs	251098.85	252097.85
Inserção	1000	Invertido	0.0620458 secs	500499.00	501498.00
Inserção	1000	Ordenado	0.0001290 secs	999.00	1998.00
Inserção	1000	Parcial	0.0100131 secs	83807.18	84806.18
Inserção	5000	Aleatório	0.7530582 secs	6247376.11	6252375.11
Inserção	5000	Invertido	1.5809329 secs	12502499.00	12507498.00
Inserção	5000	Ordenado	0.0006828 secs	4999.00	9998.00
Inserção	5000	Parcial	0.2473113 secs	2052134.50	2057133.50
Inserção	10000	Aleatório	3.0103041 secs	24992830.95	25002829.95
Inserção	10000	Invertido	6.1660368 secs	50004999.00	50014998.00
Inserção	10000	Ordenado	0.0013168 secs	9999.00	19998.00
Inserção	10000	Parcial	0.9891398 secs	8209190.67	8219189.67
Inserção	50000	Aleatório	1.2550940 secs	625139777.91	625189776.91
Inserção	50000	Invertido	2.5709911 secs	1250024999.00	1250074998.00
Inserção	50000	Ordenado	0.0066781 secs	49999.00	99998.00
Inserção	50000	Parcial	24.6829854 secs	204828410.49	204878409.49
QuickSort	500	Aleatório	0.0016056 secs	4693.96	3134.49
QuickSort	500	Invertido	0.0008912 secs	3510.00	750.00
QuickSort	500	Ordenado	0.0009573 secs	3753.00	0.00
QuickSort	500	Parcial	0.0013527 secs	4245.99	1624.32
QuickSort	1000	Aleatório	0.0042355 secs	10498.21	6969.87
QuickSort	1000	Invertido	0.0055308 secs	8006.00	1500.00
QuickSort	1000	Ordenado	0.0020545 secs	8498.00	0.00
QuickSort	1000	Parcial	0.0031550 secs	9649.42	3935.13
QuickSort	5000	Aleatório	0.0350165 secs	65796.11	42854.85
QuickSort	5000	Invertido	0.0199394 secs	52286.00	7500.00
QuickSort	5000	Ordenado	0.0200574 secs	54774.00	0.00
QuickSort	5000	Parcial	0.0306256 secs	62592.02	26409.63
QuickSort	10000	Aleatório	0.0971727 secs	143824.18	92592.51
QuickSort	10000	Invertido	0.0954213 secs	114548.00	15000.00
QuickSort	10000	Ordenado	0.0973439 secs	119535.00	0.00
QuickSort	10000	Parcial	0.0882310 secs	138064.31	58672.14
QuickSort	50000	Aleatório	1.7015193 secs	848051.17	543859.65
QuickSort	50000	Invertido	1.7174673 secs	692262.00	75000.00
QuickSort	50000	Ordenado	1.7960176 secs	717248.00	0.00
QuickSort	50000	Parcial	1.6348659 secs	845808.05	364271.70
Seleção	500	Aleatório	0.0060219 secs	124750.00	1485.98
Seleção	500	Invertido	0.0064356 secs	124750.00	999.00
Seleção	500	Ordenado	0.0070200 secs	124750.00	499.00
Seleção	500	Parcial	0.0058984 secs	124750.00	1321.76
Seleção	1000	Aleatório	0.0241992 secs	499500.00	2984.90
Seleção	1000	Invertido	0.0268190 secs	499500.00	1999.00
Seleção	1000	Ordenado	0.0241041 secs	499500.00	999.00
Seleção	1000	Parcial	0.0240066 secs	499500.00	2827.52
Seleção	5000	Aleatório	0.5993001 secs	12497500.00	14980.26

(continued)

Método	Tamanho	Ordenamento	Tempo	Comparações	Movimentações
Seleção	5000	Invertido	0.6259341 secs	12497500.00	9999.00
Seleção	5000	Ordenado	0.5981762 secs	12497500.00	4999.00
Seleção	5000	Parcial	0.5999767 secs	12497500.00	14840.56
Seleção	10000	Aleatório	2.3946125 secs	49995000.00	29979.50
Seleção	10000	Invertido	2.4841545 secs	49995000.00	19999.00
Seleção	10000	Ordenado	2.3931191 secs	49995000.00	9999.00
Seleção	10000	Parcial	2.3942290 secs	49995000.00	29790.08
Seleção	50000	Aleatório	40.1216324 secs	1249975000.00	149976.26
Seleção	50000	Invertido	1.0326209 secs	1249975000.00	99999.00
Seleção	50000	Ordenado	59.9039979 secs	1249975000.00	49999.00
Seleção	50000	Parcial	36.0013029 secs	1249975000.00	149808.78

Anexo B - código comentado

O código a seguir está estruturado da seguinte forma:

1. Definição da função geradora de amostras e testes de funcionamento;

2. Métodos de ordenamento;
   2.1. Definição da função de seleção;
   2.2. Definição da função de inserção;
   2.3. Definição da função de Quick Sort;
   2.3.1. Definição da função Quick Sort base;
   2.3.2. Definição da função envoltória para realizar a medição de tempo e completar os demais outputs necessários;
3. Testes de comparação
   3.1. Testes de comparação para amostras em ordem direta ou inversa;
   3.2. Testes de comparação para amostras parcialmente ou totalmente aleatórias;
   3.3. Cálculo das médias das medidas do item 3.2.

```
# funcao geradora de amostras ----
gera.amostra <- function(size, ordem, seed){
  #funcao para facilitar teste de argumento ORDEM na funcao
  `%notin%` <- Negate(`%in%`)

  #opcoes de argumento
  opcoes <- c("ordenado", "invertido", "aleatorio", "parcial")
  #testa ordem da funcao
  if (ordem %notin% opcoes) {
    stop("Selecione uma das opcoes possíveis: \nordenado, invertido, aleatorio ou parcial.",
         call. = FALSE)}

  if(missing(seed)){message("Argumento não usado: random seed.")}
  else {set.seed(seed)}


  #testa tamanho da amostra
  if (size - as.integer(size) != 0){
    message("Tamanho de amostra será convertido para o \nmenor número inteiro mais próximo.")
  }
  #converte para realizar os calculos
  size <- as.integer(size)

  #gera amostra
  if (ordem == "ordenado"){
    vec <- 1L:size
} else if (ordem == "invertido"){
    vec <- size:1L
} else if (ordem == "aleatorio"){
    vec <- sample(c(1L:size), size)
} else if (ordem == "parcial") {
  vec <- 1L:size
  amostra <- sample(vec, size = as.integer(size*0.1))
  locais <- sample(as.integer(size*0.1))
  antigos_velhos <- matrix(c(amostra, locais), ncol = 2)
```

```
  vec[locais] <- antigos_velhos[,1]
  vec[amostra] <- antigos_velhos[,2]

  }
  return(list(amostra = vec, ordem = ordem))
}
## exemplos de amostras ----
gera.amostra(15L, "parcial") #uso correto sem seed
gera.amostra(15.2, "aleatorio", seed = 1234) #uso com decimal e com seed
gera.amostra(15L, "aleatorio", seed = 1234) #demonstra que o vetor é igual se o seed é igual
gera.amostra(15L, "aleatorio") #demonstra que o vetor é diferente sem seed fornecido
gera.amostra(10L, "cassildis") #demonstra interrupcao da execucao da funcao

# Metodos de ordenacao ----

## Selecao ----
selecao <- function(amostra, .show_all = FALSE){

  #amostra <- gera.amostra(size, ordem) - removido

  x <- amostra$amostra
  size <- length(amostra$amostra)

  # if (length(x) == 1){ - removido. Considerar apenas as amostrar geradas, numéricas
  #   x <- as.character(x)
  #   x <- unlist(strsplit(tolower(x), split = "*"))
  # }

  comparacoes <- 0
  movimentacoes <- 0 #copias, reatribuicoes
  t1 <- Sys.time()

  #tamanho do vetor
  n <- length(x)
  #loop de ordenacao
  for(i in 1:(n-1)){ #para i de 1 ao penultimo
    min = i#considere o primeiro como o menor
    for(j in (i+1):n){ #para j do segundo ao ultimo
      if(x[j] < x[min]) {#compara x[j] ao x[i] (considerado minimo)
        min <- j #se x[j] for menor, o indice minimo deve ser j
      }
      comparacoes <- comparacoes+1 #aumenta o contador de comparacoes
    }
    temp <- x[min] #objeto temporario com o menor valor - COPIA
    movimentacoes <- movimentacoes + 1

    #objetos originais para comparacao
    xmin_antigo <- x[min]
    xi_antigo <- x[i]

    #objetos novos apos reposicionamento - REPOSICIONAMENTOS (talvez)
    x[min] <- x[i] #antigo indice do minimo se torna o maior valor
    x[i] <- temp #antigo indice maior recebe valor minimo,
```

```
                ## armazenado na variavel temporaria

    #testa se houve reposicionamento
    if(xmin_antigo != x[min]){movimentacoes <- movimentacoes + 1}
    if(xi_antigo != x[i]){movimentacoes <- movimentacoes + 1}
  }
  t <- Sys.time()-t1

  if(.show_all == FALSE){
    return(data.frame(
      metodo = "selecao",
      tamanho = size,
      ordenamento = amostra$ordem,
      # objeto_original = amostra,
      # objeto_ordenado = x,
      tempo = t,
      comparacoes = comparacoes,
      movimentacoes = movimentacoes)
    )
  } else {
    return(list(
      metodo = "selecao",
      tamanho = size,
      ordenamento = amostra$ordem,
      objeto_original = amostra$amostra,
      objeto_ordenado = x,
      tempo = t,
      comparacoes = comparacoes,
      movimentacoes = movimentacoes)
    )
  }
}

### exemplo ----
amostra <- gera.amostra(20, "parcial", seed = 1234)
selecao(amostra)
selecao(amostra, .show_all = T)


## Insercao ----

insercao <- function(amostra, .show_all = FALSE){
  x <- amostra$amostra
  size <- length(amostra$amostra)
  comparacoes <- 0
  movimentacoes <- 0 #copias, reatribuicoes
  t1 <- Sys.time()

  #tamanho do vetor
  n <- length(x)
  #loop de ordenacao
  for(i in 2:n){ # a partir de i = 2
    temp = x[i] # selecione o i-esimo
```

```
    movimentacoes <- movimentacoes+1
    j = i-1

    while(x[j] > temp && j > 0){ #
      comparacoes <- comparacoes+1
      x[j+1] <- x[j]
      movimentacoes <- movimentacoes+1
      j <- j-1
    }
    comparacoes <- comparacoes+1
    #1 comparacao se x[j] > temp == FALSE
    x[j+1] <- temp
    movimentacoes <- movimentacoes+1
  }


  t <- Sys.time()-t1

  if(.show_all == FALSE){
    return(data.frame(
      metodo = "insercao",
      tamanho = size,
      ordenamento = amostra$ordem,
      # objeto_original = amostra,
      # objeto_ordenado = x,
      tempo = t,
      comparacoes = comparacoes,
      movimentacoes = movimentacoes)
    )
  } else {
    return(list(
      metodo = "insercao",
      tamanho = size,
      ordenamento = amostra$ordem,
      objeto_original = amostra$amostra,
      objeto_ordenado = x,
      tempo = t,
      comparacoes = comparacoes,
      movimentacoes = movimentacoes)
    )
  }
}

### exemplo ----
amostra <- gera.amostra(20, "invertido", seed = 1234)
insercao(amostra)
insercao(amostra, .show_all = T)

## QuickSort ----


quicksort_base <- function(vetor, esq = 1, dir = length(vetor),
                      movimentacoes = 0, comparacoes = 0){
```

```r
pivot <- vetor[as.integer((esq+dir)/2)] # pega ponto médio como pivo
i <- esq
j <- dir

while(i <= j){
  #compara i com j no while(i <= j)
  comparacoes <- comparacoes+1

  # enquanto o pivo for maior que o numero à esquerda
  # avanca com i até achar um numero maior que o pivot
  # compara x[i] com pivot
  while(vetor[i] < pivot){
    i <- i+1
    comparacoes <- comparacoes+1
  }

  # enquanto o pivo for menor que o numero à direita
  # retrocede com j até achar um numero menor que o pivot
  # compara x[j] com pivot
  while(vetor[j] > pivot){
    j <- j-1
    comparacoes <- comparacoes+1
  }

  #se o índice i for menor que o índice j
  #realiza movimentacoes
  #prossegue com os contadores
  if(i <= j){
    temp <- vetor[i]
    vetor[i] <- vetor[j]
    vetor[j] <- temp
    if(i<j){movimentacoes <- movimentacoes+3}
    i <- i + 1
    j <- j - 1
  }
  # #se os indices forem iguais
  # #cruza os apontadores
  # if(i == j){
  #   i <- i + 1
  #   #j <- j - 1
  # }
}

# prints pra auditoria de um digito que estava escapando do ordenamento
#print(vetor)
#print(glue::glue("i = {i}, j = {j}, pivot = {pivot}, es = {esq}, dir = {dir}"))

#quicksort da esquerda
if(esq < j){
  resultado <- quicksort_base(vetor, esq = esq, dir = j,
                              movimentacoes = movimentacoes, comparacoes = comparacoes)
  vetor <- resultado$vetor
  movimentacoes <- resultado$movimentacoes
```

```
    comparacoes <- resultado$comparacoes
  }
  #quicksort da direita
  if(i < dir){
    resultado <- quicksort_base(vetor, esq = i, dir = dir,
                            movimentacoes = movimentacoes, comparacoes = comparacoes)
    vetor <- resultado$vetor
    movimentacoes <- resultado$movimentacoes
    comparacoes <- resultado$comparacoes
  }
  return(list(
    vetor = vetor,
    movimentacoes = movimentacoes,
    comparacoes = comparacoes
  ))
}

quick_completa <- function(amostra, .show_all = FALSE){

  vetor <- amostra$amostra

  t1 <- Sys.time()
  resultados <- quicksort_base(vetor)
  t <- Sys.time()-t1

  if(.show_all == FALSE){
    return(data.frame(
      metodo = "quicksort",
      tamanho = length(amostra$amostra),
      ordenamento = amostra$ordem,
      # objeto_original = amostra$amostra,
      # objeto_ordenado = resultados$vetor,
      tempo = t,
      comparacoes = resultados$comparacoes,
      movimentacoes = resultados$movimentacoes)
    )
  } else {
    return(list(
      metodo = "quicksort",
      tamanho = length(amostra$amostra),
      ordenamento = amostra$ordem,
      objeto_original = amostra$amostra,
      objeto_ordenado = resultados$vetor,
      tempo = t,
      comparacoes = resultados$comparacoes,
      movimentacoes = resultados$movimentacoes)
    )
  }
}

### exemplo
amostra <- gera.amostra(20, "parcial", seed = 1234)
quick_completa(amostra)
```

```r
quick_completa(amostra, .show_all = T)

# Limpeza ----
rm(list=setdiff(ls(), c("gera.amostra", "selecao", "insercao",
                        "quicksort_base", "quick_completa")))
gc()

# Testes de comparação ----

# tamanhos de vetores: 0.5K, 1K, 5K, 10K, 50K
tamanhos <- c(500L, 1000L, 5000L, 10000L, 50000L)
# ordenações possiveis: ordenados, 10% desordenados, aleatorio, e ordem invertida
ordens_simples <- c("ordenado", "invertido")
ordens_medias <- c("parcial", "aleatorio")


## computa tabela de ordenacoes simples se arquivo nao existe ----
if(!dir.exists("./dados gerados tp1/")){
  dir.create("./dados gerados tp1/")}

if(!file.exists("trabalho pratico 1/dados gerados tp1/comps_simples.Rdata")){

  comps_simples <- data.frame()

  ### loop de amostras nao aleatorizadas ----

  for(i in tamanhos[1:5]){ #poderia ser otimizado para nao gerar amostras toda vez
      for(j in ordens_simples){
       amostra <- gera.amostra(i,j, seed = 12345)
       comps_simples <- rbind(comps_simples, selecao(amostra),
                              insercao(amostra), quick_completa(amostra))
    }
  }


  ### salva o arquivo, se nao existe ----
  saveRDS(comps_simples, "trabalho pratico 1/dados gerados tp1/comps_simples.Rdata")

} else {comps_simples <- readRDS("trabalho pratico 1/dados gerados tp1/comps_simples.Rdata")}

## computa tabela com multiplas iteracoes nos casos aleatorizados se nao existe ----
if(!file.exists("trabalho pratico 1/dados gerados tp1/comps_iter.Rdata")){

  comps_iter <- data.frame()

  ### loop amostras aleatorizadas
  for(k in 1:100){
    #print(paste("k = ", k)) - para auditoria
    for(i in tamanhos[1:5]){ #poderia ser otimizado para nao gerar amostras toda vez
      #print(paste("i = ", i)) - para auditoria
      for(j in ordens_medias){
        amostra <- gera.amostra(i,j, seed = k)
        comps_iter <- rbind(comps_iter, selecao(amostra),
```

```
                            insercao(amostra), quick_completa(amostra))
        #COMO NO ANTERIOR, BOTAR TODOS NO RBIND
      }
    }
  }



  ### salva o arquivo, se nao existe ----
  saveRDS(comps_iter, "trabalho pratico 1/dados gerados tp1/comps_iter.Rdata")

} else {comps_iter <- readRDS("trabalho pratico 1/dados gerados tp1/comps_iter.Rdata")}

## calculo das médias dos casos aleatorizados ----
library(dplyr)
medias <-comps_iter %>%
  group_by(metodo, tamanho, ordenamento)%>%
  summarise(across(tempo:movimentacoes, mean))

saveRDS(medias, "trabalho pratico 1/dados gerados tp1/medias_iter.Rdata")

# limpeza do ambiente, deixar apenas funcoes e objetos finais

rm(list=setdiff(ls(), c("gera.amostra", "selecao", "insercao",
                        "quicksort_base", "quick_completa",
                        "comps_iter", "comps_simples", "medias")))
gc()
```