Relatório do Trabalho de Ordenação e Estatísticas de Ordem

Técnicas de Programação Avançada — Ifes — Campus Serra

Aluno: INSIRA O NOME DO ALUNO

Prof. Jefferson O. Andrade

29 de setembro de 2018

Sumário

Lista de Códigos Fonte

Lista de Figuras

1. Introdução

Este documento tem o propósito duplo de servir de modelo para confecção do relatório que deve ser entregue como parte do Trabalho de Ordenação e Estatísticas de Ordem da disciplina de Técnicas de Programação Avançada, e também de descrever dois algoritmos de ordenação não discutidos em sala de aula.

1.1. Ambiente de Desenvolvimento

Para o desenvolvimento do trabalho foi utilizada a linguagem Scala, e para automação de algumas das atividades de preparação de dados foram utilizados scripts do interpretador de comandos (shell) Bash.

Para edição do código fonte foi utilizado o editor de textos GNU Emacs, com a distribuição Spacemacs. Para suporte ao desenvolvimento de código em Scala foi utilizado o *plug-in* ENSIME, do Emacs, que acrescenta características típicas de um ambiente integrado de desenvolvimento.

2. Geração de Dados para Testes

Originalmente, pretendia-se utilizar um sistema online de geração de dos aleatórios para testes. Entretanto, todos os sistemas online gratuitos encontrados tinham severas limitações de número de registros que poderiam ser criados. Tipicamente limitando este número em 500 registros ou menos. Como seria necessária a geração de arquivos com mais de 1 milhão de registros para os testes dos algoritmos de ordenação, a opção de utilizar um sistemas online ficou inviabilizada.

Deste modo, foi desenvolvido um programa específico para geração dos dados para teste dos algoritmos de ordenação. Este programa foi desenvolvido na linguagem Scala, e é um programa relativamente simples.

2.1. Programa para Geração de Dados

O programa desenvolvido foi chamado de Mokata, que é um amálgama entre as palavras em ingês *mock* e *data*. A versão atual do Mokata (versão 0.1.0) foi desenvolvida especificamente para gerar os dados para este trabalho, o que levou a um programa pouco flexível e com diversas decisões tomadas arbitrariamente e expressa de modo fixo em código, i.e., *hard-coded*. Espera-se que o Mokata evolua para, eventualmente, poder ser usado para geração de dados aleatórios de testes para uma grande variedade de programas, podendo ser configurado para gerar uma quantidade variável de campos, e uma grande variedade de tipos diferentes de campos de dados, de modo consistente. Por exemplo, mantendo consistência entre nome e gênero, ou entre nome e e-mail.

O programa Mokata recebe dois argumentos obrigatórios, o número de registros que devem ser gerado e o nome do arquivo de saída onde os registros devem ser gravados, e um argumento opcional, a semente aleatória que será usada para a geração dos dados. Se a semente não for especificada, o valor padrão 12345 será utilizado.

```
1  def genMockData(config: Config) {
2    val writer = CSVWriter.open(config.outputFile)
3    val header = List("email", "gender", "uid", "birthdate", "height", "weight")
4    writer.writeRow(header)
5
6    var uidSet: TreeSet[Long] = TreeSet.empty
7    for (i <- 1 to config.recordCount) {
8      val row = genMockRow(uidSet)
9      writer.writeRow(row)
10   }
11   writer.close()
12 }
```

Código Fonte 1: Função com o loop principal que gera o arquivo de dados do programa Mokata.

O Código Fonte 1 mostra a função contendo o loop principal do programa Mokata que gera as linhas de dados individualmente e as grava no arquivo de saída. A função `genMockData` recebe os argumentos de configuração do programa através do parâmetro `config`.

O Código Fonte 2 mostra a função que gera as linhas de dados individuais. Note que existe uma *dependência funcional estatística* entre altura e peso. A função `genMockWeight` recebe a altura como parâmetro, juntamente com um indicador de desvio padrão para o peso. O peso é calculado aleatoriamente, porém com base em uma distribuição probabilística gaussiana com centro no *Índice de Massa Corporal* (BMI) 25 e desvio padrão de acordo com o indicado como parâmetro. Outras dependências deverão ser criadas em futuras versões do programa, à medida que novos tipos de campos forem acrescentados à ele.

2.2. Bibliotecas Externas e Build

Para a análise e processamento dos argumentos de linha de comando foi utilizado a biblioteca scopt. Esta biblioteca foi escolhida fundamentalmente por sua simplicidade de

```scala
def genMockRow(uidSet: TreeSet[Long]): List[String] = {
  val email = genMockEmail
  val gender = genMockGender
  val uid = genMockUserId(uidSet)
  val bdate = genMockBirthdate.toString
  val height = genMockHeight(175, 15)
  val weight = genMockWeight(height)
  val row = List(email, gender, uid, bdate, height.toString,
  ↪  round(weight).toString)
}
```

Código Fonte 2: Função que gera cada linha individual de dados.

uso. Embora haja bibliotecas mais sofisticadas, estas exigiam maior tempo de estudo para o aprendizado, e os recursos adicionais oferecidos não seriam necessários para o que tínhamos em mente.

De modo semelhante, a escolha da biblioteca utilizada para a gravação dos arquivos CSV foi feita baseada na simplicidade de uso. A biblioteca escolhida foi a scala-csv. Esta biblioteca define um *reader* e um *writer* para arquivos CSV que retornam listas de *strings* e gravam listas de *strings*, respectivamente. Embora a biblioteca não ofereça muito, é o suficiente para as necessidades do Mokata e o tempo de aprendizado é extremamente curto – a página da biblioteca no GitHut apresenta um conjunto de exemplos que permite saber o necessário para utilizá-la em menos de 5 minutos.

```scala
lazy val root = (project in file("."))
  .settings(
    name := "Mokata",
    version := "0.1.0",
    scalaVersion := "2.12.6",
    libraryDependencies += "com.github.scopt" %% "scopt" % "3.7.0",
    libraryDependencies += "com.github.tototoshi" %% "scala-csv" % "1.3.5"
  )
```

Código Fonte 3: Arquivo de configuração do projeto Mokata no sbt.

Para o processo de *build* foi utilizada a ferramenta sbt. O Código Fonte 3 mostra o arquivo de configuração da ferramenta sbt para o projeto *Mokata*, permitindo ver a configuração para inclusão das bibliotecas scopt e scala-csv.

O sbt também é necessário para integração do *plug-in* ENSIME ao editor Emacs, visto que o arquivo de configuração do ENSIME é gerado pelo comando *ensimeConfig* acrescentado ao sbt através de um *plug-in* do sbt desenvolvido pela equipe do ENSIME.

O processo padrão de "*packaging*" do sbt gera um arquivo JAR que pode ser executado no *Java Virtual Machine* (JVM), porém com dependência. Para que esse arquivo JAR seja executado pela JVM é necessário indicar no *classpath* o arquivo JAR com a biblioteca padrão da linguagem Scala, bem como os arquivos JAR com as bibliotecas de terceiros (neste caso a scopt e a scala-csv). Esse processo é inconveniente. Para reduzir essa inconveniência, foi usado o *plug-in* sbt-assembly. Este *plug-in* cria arquivos JAR que já contém todas as dependências do projeto, eliminando a necessidade do usuário ter que

baixar essas dependências e especificá-las como parâmetros, ou na variável de ambiente CLASSPATH.

2.3. Geração dos Arquivos de Dados

Para automatizar o processo de geração dos arquivos de dados foi gerado um script do interpretador de comandos Bash. O código fonte deste script é mostrado como apêndice na Subseção A.1.

O script da Subseção A.1, gravado no arquivo `mkdata.sh`, define três listas: `PRIMES`, `SIZES`, e `NAMES`. A lista `PRIMES` contém números primos de 5 dígitos que serão usados como semente de geração de números randômicos pelo programa Mokata. A lista foi obtida no site Prime Curious. A lista `SIZES` contém o tamanho (número de registro) dos arquivos de serão gerados, e a lista `NAMES` contém uma lista de identificadores que serão utilizados para compor o nome dos arquivos que serão gerados. É necessário que a lista `PRIMES` e a lista `NAMES` tenham pelo menos a mesma quantidade de elementos da lista `SIZES`. Caso contrário, ocorrerá um erro na execução do script.

3. Implementação do Trabalho

Como dito anteriormente, o trabalho foi implementado na linguagem Scala, utilizando o sistema de *build* sbt, e Emacs/ENSIME como ambiente de desenvolvimento. O projeto consiste de apenas três arquivos:

- `MySort.scala` – Contém o código para o programa principal.
- `Sorting.scala` – Contém a implementação dos algoritmos de ordenação.
- `Utils.scala` – Contém a implementação da função que conta o número de linhas de um arquivo.

3.1. Programa Principal

O programa principal é responsável por processar os argumentos de linha de comando, ler os dados do arquivo CSV de entrada, executar o algoritmo de ordenação, coletar o tempo de execução do algoritmo de ordenação, e gravar os dados ordenados no arquivo CSV de saída.

O Código Fonte 4 mostra a função principal do programa MySort. Optamos por seguir a arquitetura proposta no Algoritmo 2 do enunciado do trabalho, ou seja, ter um único programa principal que recebe como parâmetro qual algoritmo de ordenação deve ser aplicado. É possível ver nas linhas de 7 a 12 do Código Fonte 4 que o campo `algorithmId` da configuração do programa é usado para selecionar o algoritmo.

A função `loadInputFile`, merece um comentário à parte. Uma forma relativamente simples de fazer a leitura do arquivo de entrada seria construir uma lista simplesmente encadeada à medida que fôssemos lendo os registros e, ao final, retornar esta lista. Entretanto, precisamos de um vetor para passar aos algoritmos de ordenação, então as listas teriam que ser convertidas em vetores. Eta foi nossa primeira abordagem. Porém esta abordagem estava causando um erro de exaustão de memória com o arquivo de 5 milhões de registros. Em virtude disso, foi necessário mudar de estratégia, e passamos a utilizar

```scala
def run(config: Config): Unit = {
  validateConfig(config);

  val persons = loadInputFile(config.inputFile)

  val sortInitTime = System.nanoTime()
  config.algorithmId match {
    case "shell" => Sorting.shellsort(persons)
    case "comb" => Sorting.combsort(persons)
    case _ => throw new RuntimeException(
      s"Error! Invalid algorithm id: ${config.algorithmId}")
  }
  val sortFinishTime = System.nanoTime()

  writeOutputFile(config.outputFile, persons)
  reportTime(config.algorithmId, persons.size, sortFinishTime - sortInitTime)
}
```

Código Fonte 4: Função principal do programa MySort.

```scala
def loadInputFile(inputFile: File): Array[Person] = {
  val numLines = Utils.countLines(inputFile)
  val persons = new Array[Person](numLines - 1)
  val reader = CSVReader.open(inputFile)
  reader.readNext()
  var i = 0
  reader foreach {fields => persons(i) = Person.fromList(fields); i += 1}
  reader.close()
  persons
}
```

Código Fonte 5: Função que faz a leitura dos registros do arquivo de entrada.

uma função que conta o número de linhas do arquivo, então alocamos o vetor já com seu tamanho total e em seguida passamos a ler os registros e armazená-los já diretamente no vetor. O Código Fonte 5 apresenta a função que faz a leitura dos arquivos de entrada.

3.2. Estruturas de Dados

Para modelar os dados que o programa deve manipular foi definida uma estrutura de dados, ou uma *case class* na terminologia de Scala, chamada `Person`. Esta estrutura de dados é composta por seis campos, correspondentes as colunas dos arquivos CSV de entrada.

O Código Fonte 6 apresenta a definição do tipo `Person`. Note que também foi definida para valores este tipo uma operação (método) chamada `toList` que converte o objeto uma lista de strings correspondente. Esta operação será utilizada para gravação dos dados no arquivo de saída.

O Código Fonte 7 apresenta a definição do *objeto companheiro* (*companion object*) da

```scala
case class Person(uid: String, email: String, gender: Char,
                  birthdate: LocalDate, height: Int, weight: Int)
{
  def toList: List[String] = {
    List(email,
         gender.toString,
         uid,
         birthdate.toString,
         height.toString,
         weight.toString)
  }
}
```

Código Fonte 6: Definição da estrutura de dados `Person`.

```scala
object Person {
  def fromList(fields: Seq[String]): Person = {
    val email = fields(0)
    val gender = fields(1).charAt(0)
    val uid = fields(2)
    val birthdate = LocalDate.parse(fields(3))
    val height = fields(4).toInt
    val weight = fields(5).toInt
    Person(uid, email, gender, birthdate, height, weight)
  }
}
```

Código Fonte 7: Objeto companheiro da classe `Person`, que define a operação `fromList`.

classe `Person`. Esse objeto define uma função chamada `fromList` que realiza a operação inversa da função `toList`, i.e., recebe uma lista de strings e constrói um objeto do tipo `Person` à partir dos valores da lista.

```scala
implicit val ordPerson: Ordering[Person] = new Ordering[Person]() {
  def compare(x: Person, y: Person): Int = x.uid.compare(y.uid)
}
```

Código Fonte 8: Definição do valor implícito do tipo `Ordering[Person]` que será usado como forma padrão de comparação entre dois valores do tipo `Person`.

O Código Fonte 8 apresenta a definição de um valor implícito do tipo `Ordering[Person]`. Esta interface define uma função de comparação que será usada como padrão toda vez que uma operação qualquer definir um parâmetro deste tipo como implícito. Os algoritmos de ordenação implementados neste trabalho utilizaram esta estratégia, i.e., definiram parâmetros implícitos do tipo `Ordering[T]`, de modo que ao chamar as funções de ordenação não é necessário passar explicitamente o objeto que faz a comparação.

3.3. Algoritmos de Ordenação

O Trabalho de Ordenação e Estatísticas de Ordem pede a implementação de cinco algoritmos de ordenação diferentes. A saber: ordenação por seleção (*selection sort*), ordenação por inserção (*insertion sort*), *merge sort*, *quicksort*, e *heapsort*.

Neste exemplo de relatório, entretanto, foram implementados dois outros algoritmos que não estão na lista pedida. Para não estragar a diversão dos alunos durante a implementação.

Os algoritmos implementados são o *Shell sort* e o *Comb sort*, ambos podem ser considerados generalizações de algoritmos já vistos, e apresentam boas performances.

3.3.1. Algoritmo Shell Sort

O *Shell sort* foi desenvolvido por Donald Shell. Este algoritmo utiliza a ideia de enxergar a sequência a ser ordenada como sendo composta por várias subsequencias intercaladas com um certo intervalo (*gap*) entre elementos de cada sequência. Por exemplo, suponhamos um vetor de 10 posições (0 a 9), e um intervalo de 5 isso geraria 5 subsequencias de 2 elementos: 0 e 5; 1 e 6; 2 e 7; 3 e 8; 4 e 9. Cada subsequência é ordenada por *insertion sort*. Em seguida diminui-se o *gap* e repete-se o processo. Qualquer sequência de *gaps* que termine com um *gap* de tamanho 1 garante a ordenação (porque com o *gap* de tamanho 1 o *shell sort* se torna idêntico ao *insertion sort*).

A grande melhoria do *shell sort* em relação ao *insert sort* se dá nas primeiras iterações, com valores de *gap* maiores, que faz com que valores pequenos que estejam no final do vetor sejam rapidamente movidos para as posições iniciais.

A complexidade do *shell sort* depende muito dos tamanhos dos *gaps* escolhidos e pode variar, no pior caso, desde $\Theta(n^2)$ até $\Theta\left(n^{4/3}\right)$. A página da Wikipédia sobre Shellsort possui um apanhado sobre a complexidade do *shell sort* para várias sequências de *gaps* diferentes que já foram propostas na literatura.

Nossa implementação utiliza a sequência de *gaps* proposta por Robert Sedgewick, onde cada termo de ordem k da sequência é definido pela expressão $g(k) = 4^k + 3 \cdot 2^{k-1} + 1$, e o termo de ordem zero é 1 (i.e., 1, 8, 23, 77, 281, ...). Para esta sequência de *gaps* a complexidade no pior caso é dada por $\Theta\left(n^{4/3}\right)$.

O Código Fonte 9 mostra nossa implementação do *shell sort*. Na linha 1 temos a função de calcula os termos de ordem k da sequência. na linha 3 é definida uma *steam*, i.e., uma sequência sem limite definido, de *gaps* onde o primeiro termo é 1 e os demais são formados pelos termos de ordem k (com k de 1 em diante) da sequência de Sedgewick. Na linha 6, pegamos todos os elementos da *stream* que sejam menores que o tamanho do vetor em ordem reversa para formar a sequência de *gaps*.

Em nossa implementação, a função *shellsort* recebe dois parâmetros. O primeiro é o vetor, `a`, que será ordenado. O segundo é uma implementação da interface `Ordering[T]` que permite fazer a comparação entre dois valores de um tipo `T`. Note que o parâmetro `ord` é definido como `implicit`, o que significa que se houver um valor implícito declarado no escopo da chamada da função `shellsort`, este parâmetro não precisa ser passado. Nosso programa principal define um objeto implícito para comparação de objetos do tipo `Person`.

```
def gterm(k: Int): Int = pow(4, k).toInt + 3 * pow(2, k-1).toInt + 1

val gapStream: Stream[Int] = 1 #:: (Stream.from(1) map {k => gterm(k)})

def shellsort[T](a: Array[T])(implicit ord: Ordering[T]): Array[T] = {
  val gaps = (gapStream takeWhile {g => g < a.size}).toSeq.reverse
  for (gap <- gaps) {
    for (i <- gap until a.size) {
      val temp = a(i)
      var j = i
      while (j >= gap && ord.compare(a(j-gap), temp) > 0) {
        a(j) = a(j - gap)
        j -= gap
      }
      a(j) = temp
    }
  }
  a
}
```

Código Fonte 9: Implementação do algoritmo Shell sort.

3.3.2. Algoritmo Comb Sort

O *Comb sort* foi desenvolvido por Włodzimierz Dobosiewicz. Assim como o *shell sort*, o *comb sort* também pode ser visto como uma generalização de outro algoritmo de ordenação que utiliza *gaps* para definir subsequencias intercaladas, e ordenar primeiramente estas subsequencias. No caso do *comb sort*, ele é uma generalização do Algoritmo da Bolha, notório pelo desempenho pobre. Surpreendentemente, o desempenho do algoritmo *comb sort* é consideravelmente mais rápido.

Diferentemente do que acontece com o *shell sort*, o algoritmo *comb sort* não é muito sensível às sequências de intervalos e esta sequência tipicamente é definida como sendo $[n/k, n/k^2, n/k^3, \ldots, 1]$, onde k é chamado de *fator de encolhimento*. Tipicamente, o algoritmo *comb sort* apresenta seu melhor desempenho com fator de encolhimento $k = 1.3$.

4. Análise de Desempenho dos Algoritmos

Como é possível ver no Código Fonte 4, a linha 6 coleta o tempo do sistema imediatamente antes de acionar o algoritmo de ordenação selecionado, e imediatamente após a execução do algoritmo de ordenação o tempo do sistema é coletado novamente, na linha 13. O tempo transcorrido é relatado ao final da execução do programa. Entretanto, há outras questões a considerar. Pode haver variação no tempo de execução do programa em função de uma série de fatores externos ao nosso código. Por isso, a forma mais segura de analisar a performance é executar cada programa várias vezes para um mesmo arquivo de entrada e trabalhar com a média dos tempos coletados. As seções seguintes descrevem como foi feita a coleta dos dados de tempo de execução e como esses dados foram analisados.

Todos os testes foram executado em um *notebook* Acer Aspire V15, com processador Intel Core i7, e 32Gb de RAM.

```
def combsort[T](a: Array[T])(implicit ord: Ordering[T]): Array[T] = {
  var gap = a.size
  val shrink = 1.3
  var sorted = false

  while (!sorted) {
    gap = floor(gap/shrink).toInt
    if (gap > 1) {
      sorted = false
    }
    else {
      gap = 1
      sorted = true
    }

    for (i <- gap until a.size) {
      if (ord.compare(a(i-gap), a(i)) > 0) {
        swap(a, i-gap, i)
        sorted = false
      }
    }
  }
  a
}
```

Código Fonte 10: Implementação do algoritmo Comb sort.

4.1. Coleta e Pré-processamento de Dados

Como dito anteriormente, cada algoritmo de ordenação – no nosso caso o *shell sort* e o *comb sort* – foi executado uma quantidade de vezes sobre cada um dos arquivos de dados e os tempos de execução reportados foram coletados e armazenados em um arquivo. Mais especificamente, cada algoritmo de ordenação foi executado 25 vezes para cada um dos 25 arquivo de dados. O script do interpretador de comandos Bash, apresentado na Subseção A.2, realiza esta função.

Uma vez coletados os tempos de execução é necessário processar o arquivo com os tempos de execução para extrair as médias por algoritmo e por número de registros. Também decidimos que seria interessante extrair os tempos mínimos e máximos de execução para cada algoritmos e arquivo de dados. Para essa finalidade, escrevemos um *script* em Scala, utilizando o shell Ammonite que é um interpretador de comandos Scala com algumas facilidades de importação de bibliotecas e código de outros *scripts*.

O código fonte do *script* criado para a preparação dos dados de tempo de execução é mostrado na Subseção A.3. O *script* espera o identificador do algoritmo para o qual devem ser calculadas as estatísticas, o nome do arquivo de dados de entrada (contendo os dados de tempo de execução), e o nome do arquivo de saída, onde serão gravadas as estatísticas.

A Tabela 1 e a Tabela 2 mostram as estatísticas extraídas dos arquivos com os tempos de execução para os algoritmos *Shell sort* e *Comb sort* respectivamente. Todos os tempos são dados em milissegundos.

4.2. Análise dos Resultados

n	**Média**	**Mínimo**	**Máximo**
10	25.089	23.945	29.668
25	25.161	24.400	26.548
50	25.527	24.528	26.202
75	26.033	24.981	26.897
100	27.127	25.863	29.318
250	28.302	27.070	30.997
500	29.148	27.534	30.694
750	29.884	28.139	39.250
1000	34.032	30.927	42.856
2500	36.259	32.669	47.124
5000	38.612	35.672	45.959
7500	42.054	39.064	48.990
10 000	46.161	42.210	57.681
25 000	99.861	88.734	113.409
50 000	120.227	106.230	162.819
75 000	154.800	149.695	161.732
100 000	152.896	145.787	162.129
250 000	402.177	388.248	417.022
500 000	870.255	790.333	902.306
750 000	1392.446	1259.931	1479.716
1 000 000	1876.332	1743.529	2208.050
2 500 000	5368.353	5075.158	5727.995
5 000 000	12 293.675	11 808.986	12 819.200
7 500 000	20 171.195	19 706.090	20 946.589
10 000 000	28 607.621	28 071.576	29 787.436

Tabela 1: Estatísticas dos tempos de execução do algoritmo *Shell sort*.

n	**Média**	**Mínimo**	**Máximo**
10	19.646	18.861	23.237
25	20.339	19.439	23.961
50	20.811	20.228	22.437
75	21.643	20.124	22.683
100	21.957	21.016	24.184
250	22.712	21.729	24.302
500	23.644	22.364	24.826
750	25.040	23.697	26.779
1000	29.205	25.642	37.010
2500	34.831	30.562	44.168
5000	38.451	33.331	48.389
7500	43.733	39.086	54.439
10 000	47.882	43.521	58.947
25 000	135.800	105.689	168.108
50 000	201.437	192.229	220.072
75 000	257.638	250.156	269.658
100 000	268.129	262.803	274.089
250 000	791.280	759.587	813.961
500 000	1815.228	1658.002	1883.421
750 000	2809.356	2565.111	2930.332
1 000 000	3619.447	3370.810	3920.030
2 500 000	10 817.512	10 101.095	12 034.163
5 000 000	23 126.487	22 457.472	24 445.174
7 500 000	38 064.868	36 808.737	39 785.756
10 000 000	53 861.360	52 773.852	55 404.615

Tabela 2: Estatísticas dos tempos de execução do algoritmo *Comb sort*.

A. Scripts de Shell

A.1. Script de shell para geração dos arquivos de dados

```
#!/bin/bash

MOKATA_JAR=~/Dropbox/workspace/Mokata/target/scala-2.12/Mokata-assembly-0.1.0.jar
MOKATA="java -jar $MOKATA_JAR"
DATA_DIR=/home/jefferson/data

PRIMES=(75353 75361 75787 75869 75913 75931 76157 76367 76543 76667 77141 77171\
              77269 77377 77463 77569 77611 77773 77777 78233 78317 78557 78697\
              78721 78787 78887 79397 79561 79999 80387 80557 80609 80809 80909\
              81018 81181 81341 81457 81517 81619 81647 81649 81770 81839 81918\
              83257 83431 84211 84551 84631 85201 85837 86269 86351 86453 86969\
              87811 88357 88469 88789 88956 89051 89069 89189 89821 89898 90053\
              90089 90441 90709 90863 91019 92377 92549 92857 93239 93581 93701\
              94049 94111 94397 94583 94649 94849 94889 94949 95063 95273 95471\
              95479 95731 95959 96469 96576 96769 96847 96857 97343 97379 97579\
              97829 98041 98057 98257 98317 98389 98407 98411 98519 98639 98641\
              98689 98789 98867 98899 98999 99119 99499 99923 99929 99969 )

SIZES=(10 25 50 75 \
          100 250 500 750 \
          1000 2500 5000 7500 \
          10000 25000 50000 75000\
          100000 250000 500000 750000 \
          1000000 2500000 5000000 7500000 \
          10000000)

NAMES=(10e0 25e0 50e0 75e0 \
            10e1 25e1 50e1 75e1 \
            10e2 25e2 50e2 75e2 \
            10e3 25e3 50e3 75e3 \
            10e4 25e4 50e4 75e4 \
            10e5 25e5 50e5 75e5 \
            10e6)

echo Criating data directory ...
mkdir -p $DATA_DIR

echo Generating data files ...
N=`expr ${#SIZES[*]} - 1`
echo "Number of files => ${#SIZES[*]}"
echo "Limit of iteration => $N"
for i in $(seq 0 $N)
do
    SEED=${PRIMES[$i]}
    COUNT=${SIZES[$i]}
    ID=${NAMES[$i]}
    FILE="data_${ID}.csv"
    echo "Generating $COUNT records ..."
    CMD="$MOKATA -s $SEED -n $COUNT -o $DATA_DIR/$FILE"
    eval $CMD
done
echo "Done."
```

A.2. Script de shell para execução dos algoritmos de ordenação

```
#!/bin/bash

# Diretório onde estão armazenados os arquivos de dados de entrada e onde serão
# armazenados os arquivos de saída do programa de ordenação.
DATA_DIR=/home/jefferson/data

# Diretório base para o trabalo de programação de LFA
TRAB2_DIR=/home/jefferson/Dropbox/2018-2/tpa/trab2

# Diretório onde serão gerados os arquivos de coleta de dados sobre tempo de
# execução dos programas de ordenação.
REPORT_DIR=$TRAB2_DIR/report-data

# Arquivo JAR correspondendo ao código executável em byte-code do programa de
# ordenação.
MYSORT_JAR=$TRAB2_DIR/mysort/target/scala-2.12/MySort-assembly-0.1.0.jar

# Identificadores dos arquivos de dados de entrada e de saída. Todos os arquivos
# de entrada tem nomes que sequem o padrão `data_ID.csv`, e todos os arquivos de
# saída terão nomes que seguem o padrão `sorted_ID.csv`; onde ID é um elemento
# da lista abaixo.
FILE_IDS=(10e0 25e0 50e0 75e0\
          10e1 25e1 50e1 75e1\
          10e2 25e2 50e2 75e2\
          10e3 25e3 50e3 75e3\
          10e4 25e4 50e4 75e4\
          10e5 25e5 50e5 75e5\
          10e6)

# Data e hora de execução deste script de shell. Será usado para criar os nomes
# dos arquivos de dados de tempo, de log de trabalho e de alerta de término de
# execução.
TIMESTAMP=`date +%Y-%m-%d_%H-%M-%S`

# Nome do arquivo de dados de tempo de execução dos programas de ordenação.
REPORT_FILE="data_${TIMESTAMP}.dat"

# Número de vezes que cada programa de ordenação será executado para cada
# arquivo de entrada de dados.
NUM_RUNS=25

# Data e hora (com precisão de milisegundos) de início da execução dos
# programas. Será gravado no arquivo de alerta de término de execução.
INIT_TIME=`date --rfc-3339=ns`

# Nome do arquivo de log de trabalho. Nesse arquivo serão gravados todos os
# comando executados por este script de shell para evocar os programas de
# ordenação com seus respectivos parâmetros de arquivos de entrada e saída.
WORK_FILE="$REPORT_DIR/working_${TIMESTAMP}.txt"

# Cria o arquivo de log de trabalho.
touch $WORK_FILE

# Para cada ID na lista,...
for id in ${FILE_IDS[*]}
```

```
do
    echo -n "==> $id "

    # Para cada uma das N vezes,...
    for n in `seq $NUM_RUNS`
    do
        # Para cada algoritmo de ordenação,...
        for algid in "shell" "comb"
        do
            INFILE="$DATA_DIR/data_${id}.csv"
            OUTFILE="$DATA_DIR/sorted_${id}.csv"
            CMD="java -jar $MYSORT_JAR -a $algid -i $INFILE -o $OUTFILE >>
            ↪  $REPORT_DIR/$REPORT_FILE"
            echo $CMD >> $WORK_FILE
            echo -n .
            eval $CMD
        done
    done
    echo ""
done

# Data e hora (com precisão de milisegundos) de término da execução dos
# programas. Será gravado no arquivo de alerta de término de execução.
FINISH_TIME=`date --rfc-3339=ns`

# Grava o arquivo de alerta de término de execução.
COMPLETED="completed_${TIMESTAMP}.txt"
echo $INIT_TIME >> $REPORT_DIR/$COMPLETED
echo $FINISH_TIME >> $REPORT_DIR/$COMPLETED
echo "Done."
```

A.3. Script de shell para pré-processamento do tempos de execução

```
#!/usr/bin/env amm

import java.io.File
import java.io.PrintWriter
import scala.io.Source
import scala.util.matching.Regex

/** Expressão regular usada para \quebrar" a linha lida do arquivo de registro
  * de tempos de execução. Dada linha deverá ter a seguinte estrutura:
  *
  * <identificador-do-algoritmo> <número-de-registros> <tempo-em-nanosegundos>
  *
  * A expressão regular irá reconhecer estes elementos em 3 grupos diferentes.
  */
val pat: Regex = "\\s*([0-9a-zA-Z]+)\\s+([0-9]+)\\s+([0-9]+)\\s*".r


/** Converte uma linha lida do arquivo de registro de tempos de execução e
  * converte essa linha para um trio-ordenado, com os 3 campos da linha já
  * processados. Se a linha lida não tiver a estrurura correta returna um None.
  */
def parseLine(line: String): Option[(String, Int, Long)] = {
  pat.findFirstMatchIn(line) match {
```

```
24      case Some(m) =>
25        val algId = m.group(1)
26        val count = m.group(2).toInt
27        val time = m.group(3).toLong
28        val t = (algId, count, time)
29        Some(t)
30      case None => None
31    }
32  }
33
34
35  @main
36  def prepare(algorithmId: String, inputFile: File, outputFile: File) {
37    val source = Source.fromFile(inputFile)
38    val tuples = source.getLines() flatMap {line => parseLine(line)}
39    val filtered = tuples filter {case (algId, count, time) => algId == algorithmId}
40    val pairs = filtered map {case (algId, count, time) => (count, time)}
41    val grp = pairs.toSeq groupBy {case (count, time) => count} mapValues {l => l map
      ↪ {case (count, time) => time}}
42    val scale = 1000000.0
43    val stats = grp mapValues {l => ((l.sum / l.size)/scale, l.min/scale, l.max/scale)}
44    val writer = new PrintWriter(outputFile)
45    writer.write("n, mean, min, max\n")
46    stats.keys.toSeq.sorted foreach {
47      count =>
48      val stat = stats(count)
49      writer.write(s"$count, ${stat._1}, ${stat._2}, ${stat._3}\n")
50    }
51    writer.close()
52  }
```