Algorithms and Data Structure 2

Yiping Deng

February 19, 2018

Problem 1

(a)

The concrete implementation is here: FILE 1: MergeSortVariant.scala

```scala
import scala.annotation.tailrec

object MergeSortVariant {
  /*
  Generic merge function:
  Input: two already sorted list
  Output: merged and sorted list
   */
  def merge[T](xs: List[T], ys: List[T])( implicit ord: Ordering[T]): List[T] = {
    /*
    A tail recursive version
     */
    @tailrec
    def helper(`xs`: List[T], `ys`: List[T], aggr: List[T]): List[T] = (`xs`, `ys`) match {
      case (Nil, l2) =>
        aggr.reverse ::: l2
      case (l1, Nil) =>
        aggr.reverse ::: l1
      case (xsHead :: xsTail, ysHead :: ysTail) =>
        if(ord.lt(xsHead, ysHead))
          helper(xsTail, ysHead :: ysTail, xsHead :: aggr)
        else
          helper(xsHead :: xsTail, ysTail, ysHead :: aggr)
    }
    helper(xs, ys, Nil)
  }

  def mergeSortVariant[T](xs: List[T], insertion_depth: Int)( implicit ord: Ordering[T]): List[T] =
    val len = xs.length
    if(len <= insertion_depth)
      insertionSort(xs)
    else {
      val (split1, split2) = xs.splitAt(len / 2)
      merge(mergeSortVariant(split1, insertion_depth), mergeSortVariant(split2, insertion_depth))
    }
  }

  /*
  Insert a element into a sorted list
   */
  def insert[T](x: T, xs: List[T])(implicit ord: Ordering[T]): List[T] = {
    /*
```

```
    Make it tail recursive so that it does not overflow the stack
     */
    @tailrec
    def helper(x: T, xs: List[T], aggr: List[T]): List[T] = xs match {
      case Nil =>
        (x :: aggr).reverse
      case y :: ys if ord.lt(x, y) =>
        aggr.reverse ::: (x :: y :: ys)
      case y :: ys =>
        helper(x, ys, y :: aggr)
    }
    helper(x, xs, Nil)
  }
  /*
  Implementation of insertion sort
   */
  def insertionSort[T](xs: List[T])(implicit ord: Ordering[T]): List[T] = {
    /*
    Make it tail recursive so that compiler to optimize
     */
    @tailrec
    def helper(xs: List[T], aggr: List[T]): List[T] = xs match {
      case Nil => aggr
      case xsHead :: xsTail => helper(xsTail, insert(xsHead, aggr))
    }
    helper(xs, Nil)
  }
  def generateRandom(n: Int): List[Int] = {
    val r = new scala.util.Random
    1 to n map { _ => r.nextInt() } toList
  }
  def generateWorst(n: Int) = generateRandom(n).sorted
  def generateBest(n: Int) = generateWorst(n).reverse

  def generateData(n:Int, timer: Int => Long): String = {
    val innerdata = (1 to n) map {_ * 10000} map { i => (i, (timer(i) + timer(i) + timer(i) ) / 3)} 
    s"[$innerdata]"
  }
  def kToTime(k: Int, data: List[Int]):Long = {
    Timer.justTime(mergeSortVariant(data, k))
  }

  def main(args: Array[String]):Unit = {
    println("# Generate Plot data")

    val kVal = 20
    println("# Best case")
    println("bestPlot = " + generateData(20, {i:Int => kToTime(kVal, generateBest(i))}))

    println("# Worst case")
    println("worstPlot = " + generateData(20, {i:Int => kToTime(kVal, generateWorst(i))}))

    println("# Average case")
    println("avgPlot = " + generateData(20, {i:Int => kToTime(kVal, generateRandom(i))}))
  }
}
```

FILE 2: Timer.scala

```
object Timer {

  /*
  Generic timer for scala code block
   */
  def printsNano[R](codeBlock: => R): R = {
    //measure the time
    val t0 = System.nanoTime()
    //execute the code block
    val result = codeBlock //call by name
    val t1 = System.nanoTime()
    println(s"Elapsed time: ${t1 - t0} ms")
    result
  }

  def printsMillo[R](codeBlock: => R): R = {
    //measure the time
    val t0 = System.currentTimeMillis()
    //execute the code block
    val result = codeBlock //call by name
    val t1 = System.currentTimeMillis()
    println(s"Elapsed time: ${t1 - t0} ms")
    result
  }

  /*
  timer that returns
   */
  def mkNanoTuple[R](codeBlock: => R): (Long, R) = {
    //measure time
    val t0 = System.nanoTime()

    //call by name
    val result = codeBlock

    val t1 = System.nanoTime()
    (t1 - t0, result)
  }

  def mkMilloTuple[R](codeBlock: => R): (Long, R) = {
    //measure time
    val t0 = System.currentTimeMillis()

    //call by name
    val result = codeBlock

    val t1 = System.currentTimeMillis()
    (t1 - t0, result)
  }

  def justTime[R](codeBlock: => R) : Long = {
    val t0 = System.currentTimeMillis()

    //call by name
    val result = codeBlock

    val t1 = System.currentTimeMillis()
    t1 - t0
```

```
  }

  def avergedJustTime[R](codeBlock: => R): Long = {
    val num = 5
    (1 to num).map( _ => justTime(codeBlock)).sum / num
  }
}
```

(b)

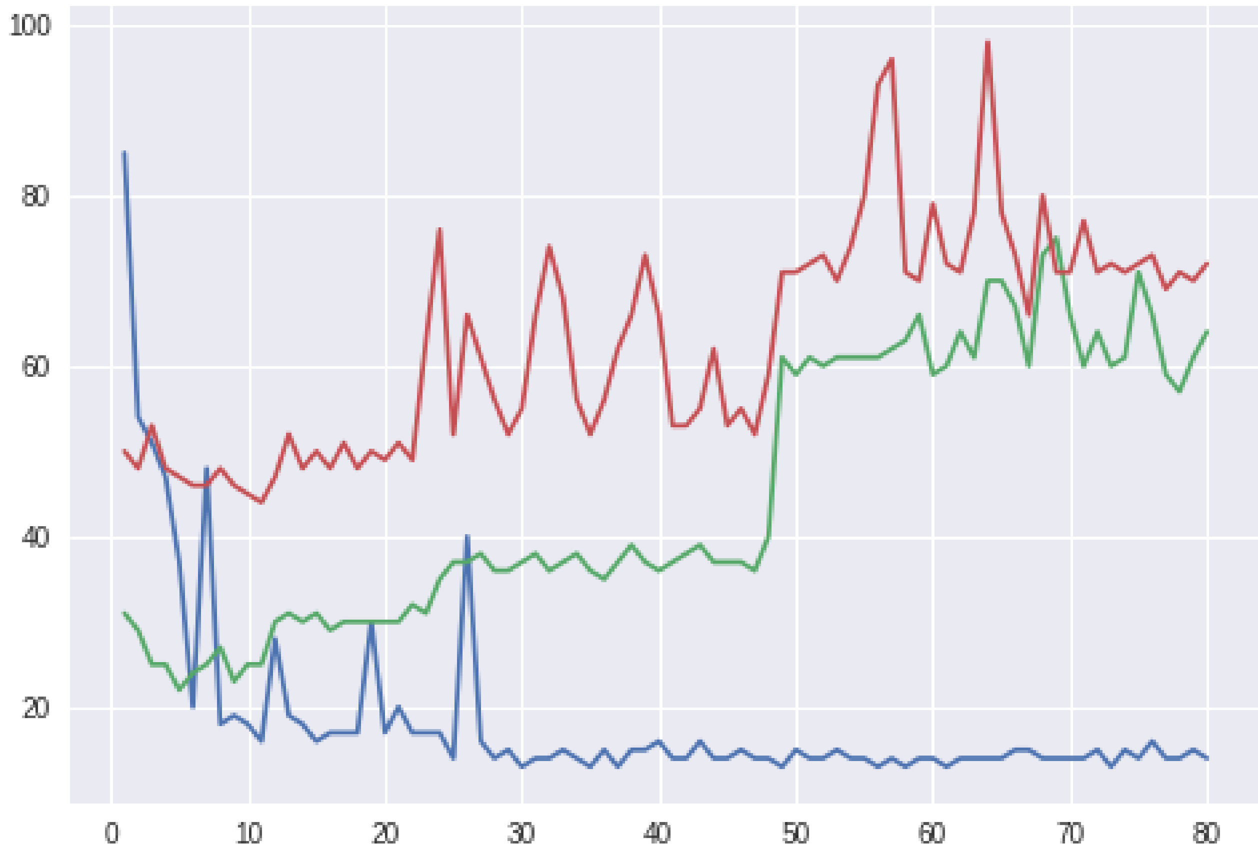
The Scala main function will generate the data necessary for plotting. Plotting is performed by Python.

```
import numpy as np
import matplotlib.pyplot as plt

def breakTuples(tuples):
  xs = []
  ys = []
  for x, y in tuples:
    xs.append(x)
    ys.append(y)
  return xs, ys

def plotTogether(tuples):
  xs, ys = breakTuples(tuples)
  plt.plot(xs, ys)
# Generate Plot data
# Best case
bestPlot = [(1,85), (2,54), (3,51), (4,47), (5,37), (6,20), (7,48), (8,18), (9,19), (10,18), (11,16)
# Worst case
worstPlot = [(1,31), (2,29), (3,25), (4,25), (5,22), (6,24), (7,25), (8,27), (9,23), (10,25), (11,25
# Average case
avgPlot = [(1,50), (2,48), (3,53), (4,48), (5,47), (6,46), (7,46), (8,48), (9,46), (10,45), (11,44),
plotTogether(bestPlot)
plotTogether(worstPlot)
plotTogether(avgPlot)
plt.show()
```

red line denotes worst case, **green** line denotes average case, **blue** line denotes best case. x-axis denotes k, y-axis denotes running time.



Also the graph along with input size n with k = 20

The graph alogn with input size n with k = 50

(c)

As you can see in picture 1, the best case running time is decreasing when k is increasing. Worst case running time is increasing as k increased. Average case reach its minimum when k is around 5

(d)

We can use the graph, in the first picture, find the minimum of running time with respect to k. Make sure the input size is big enough and it is random to represent averge case.

Problem 2

(a)

$a = 36$, $b = 6$, $f(n) = 2n \implies f(n) \in O(n^{log_b a}) \implies T(n) = \Theta(n^2)$

(b)

$a = 5$, $b = 3$, $f(n) = 17n^{1.2} \implies f(n) \in O(n^{log_b a} \implies T(n) = \Theta(n^{log_3 5})$

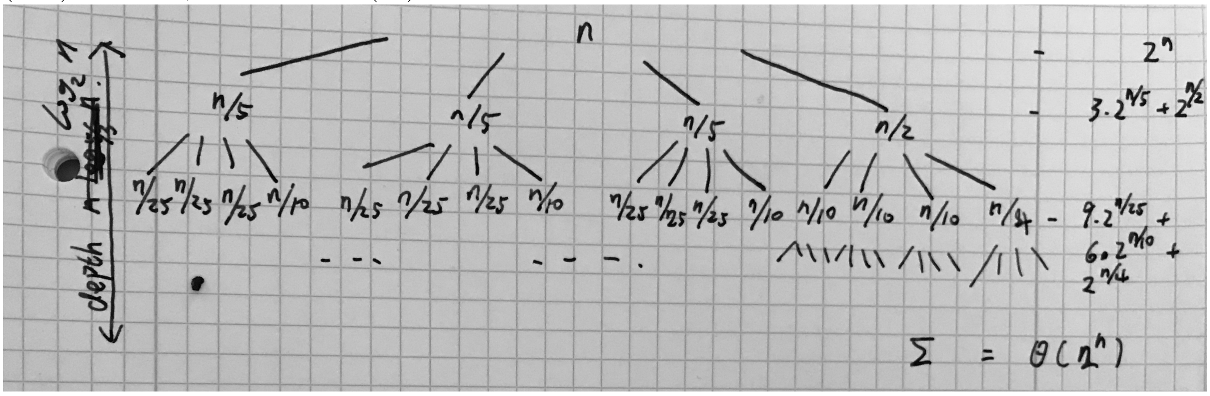
(c)

$a = 12$, $b = 2$, $f(n) = n^2 lg(n) \implies f(n) \in O(n^{log_b a} \implies T(n) = \Theta(n^{log_2 12})$

(d)

Using the tree method and draw the tree, we have

In this result, the first term, 2^n, dominates the rest of term in the summation of the tree$(2^{1/2})^n$, $(2^{1/5})^n$.... Thus, the result is $\Theta(n^2)$



(e)

Again, from tree method, the height of the tree is $log_{5/3}n$ at most, $log_{5/3}n$ at least. At each level of trees, there are n operations. Thus, we can safely conclude that it is $\Theta(n \log n)$