Algorithms and Data Structure 1

Yiping Deng

February 12, 2018

Problem 1

General proof for limit implies asymptotic notation.
case 1:

$$\lim_{n\to\infty}\frac{f(x)}{g(x)} = c \text{ s.t. } c > 0 \text{ and } c < \infty \implies$$
$$\text{for sufficiently large n,} |\frac{f(x)}{g(x)} - c| < \epsilon \implies$$
$$c - \epsilon < \frac{f(x)}{g(x)} < c + \epsilon \implies$$
$$c_1 < \frac{f(x)}{g(x)} < c_2 \implies$$
$$c_1 g(x) < f(x) < c_2 g(x) \implies$$
$$f \in \Theta(g)$$

case 2:

$$\lim_{n\to\infty}\frac{f(x)}{g(x)} = 0 \implies$$
$$\text{for sufficiently large n,} |\frac{f(x)}{g(x)}| < \epsilon \implies$$
$$f(x) < \epsilon g(x) \text{ since f, g are positive by default, } \epsilon > 0, \text{ it holds for every } \epsilon \implies$$
$$f \in o(g) \implies g \in \omega(f)$$

case 3:

$$\lim_{n\to\infty}\frac{f(x)}{g(x)} = \infty \implies$$
$$\lim_{n\to\infty}\frac{g(x)}{f(x)} = 0 \implies$$
$$\text{for sufficiently large n,} |\frac{g(x)}{f(x)}| < \epsilon \implies$$
$$g(x) < \epsilon f(x) \text{ since f, g are positive by default, } \epsilon > 0, \text{ it holds for every } \epsilon \implies$$
$$g \in o(f) \implies f \in \omega(g)$$

a)

1. $f \in \Theta(g)$ not true.
$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0$$
2. $f \in O(g)$ true.
$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0 < \infty$$

3. $f \in o(g)$ true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$
4. $f \in \Omega(g)$ not true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$
5. $f \in \omega(g)$ not true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$
6. $g \in \Theta(f)$ not true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$
7. $g \in O(f)$ not true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$
8. $g \in o(f)$ not true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$
9. $g \in \Omega(f)$ true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 < \infty$$
10. $g \in \omega(f)$ true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 < \infty$$

b)

$g(n)$ can rewrite as:

$$g(n) = n^{\frac{1}{2}}$$

1. $f \in \Theta(g)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{7n^{0.7}}{n^{0.5}} = \infty$$
2. $f \in O(g)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{7n^{0.7}}{n^{0.5}} = \infty$$
3. $f \in o(g)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{7n^{0.7}}{n^{0.5}} = \infty$$
4. $f \in \Omega(g)$ true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n^{0.5}}{7n^{0.7}} = 0$$
5. $f \in \omega(g)$ true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n^{0.5}}{7n^{0.7}} = 0$$
6. $g \in \Theta(f)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^{0.5}}{7n^{0.7}} = 0$$

7. $g \in O(f)$ true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n^{0.5}}{7n^{0.7}} = 0$$
8. $g \in o(f)$ true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n^{0.5}}{7n^{0.7}} = 0$$
9. $g \in \Omega(f)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{7n^{0.7}}{n^{0.5}} = \infty$$
10. $g \in \omega(f)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{7n^{0.7}}{n^{0.5}} = \infty$$

c)

1. $f \in \Theta(g)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2/log(n)}{nlog(n)} = \lim_{n\to\infty} \frac{n/log(n)}{log(n)} = \lim_{n\to\infty} \frac{n}{log^2(n)} = \infty$$
2. $f \in O(g)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2/log(n)}{nlog(n)} = \lim_{n\to\infty} \frac{n/log(n)}{log(n)} = \lim_{n\to\infty} \frac{n}{log^2(n)} = \infty$$
3. $f \in o(g)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2/log(n)}{nlog(n)} = \lim_{n\to\infty} \frac{n/log(n)}{log(n)} = \lim_{n\to\infty} \frac{n}{log^2(n)} = \infty$$
4. $f \in \Omega(g)$ true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{1}{f(n)/g(n)} = 0$$
5. $f \in \omega(g)$ true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{1}{f(n)/g(n)} = 0$$
6. $g \in \Theta(f)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2/log(n)}{nlog(n)} = \lim_{n\to\infty} \frac{n/log(n)}{log(n)} = \lim_{n\to\infty} \frac{n}{log^2(n)} = \infty$$
7. $g \in O(f)$ true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{1}{f(n)/g(n)} = 0$$
8. $g \in o(f)$ true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{1}{f(n)/g(n)} = 0$$
9. $g \in \Omega(f)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2/log(n)}{nlog(n)} = \lim_{n\to\infty} \frac{n/log(n)}{log(n)} = \lim_{n\to\infty} \frac{n}{log^2(n)} = \infty$$
10. $g \in \omega(f)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2/log(n)}{nlog(n)} = \lim_{n\to\infty} \frac{n/log(n)}{log(n)} = \lim_{n\to\infty} \frac{n}{log^2(n)} = \infty$$

d)

1. $f \in \Theta(g)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{log(3n)}{9log(n)}(log(3n))^2 = \infty$$
2. $f \in O(g)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{log(3n)}{9log(n)}(log(3n))^2 = \infty$$
3. $f \in o(g)$ not true.
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{log(3n)}{9log(n)}(log(3n))^2 = \infty$$
4. $f \in \Omega(g)$ true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{1}{f(n)/g(n)} = 0$$
5. $f \in \omega(g)$ true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{1}{f(n)/g(n)} = 0$$
6. $g \in \Theta(f)$ not true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{1}{f(n)/g(n)} = 0$$
7. $g \in O(f)$ not true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{1}{f(n)/g(n)} = 0$$
8. $g \in o(f)$ not true.
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{1}{f(n)/g(n)} = 0$$
9. $g \in \Omega(f)$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{log(3n)}{9log(n)}(log(3n))^2 = \infty$$
10. $g \in \omega(f)$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{log(3n)}{9log(n)}(log(3n))^2 = \infty$$

Problem 2

(a)

Implementation of selection sort in Scala
File 1: SelectionSort.scala

```
import javax.print.DocFlavor.BYTE_ARRAY

import scala.annotation.tailrec

object SelectionSort {


  /*
  Calculate the minimum
   */
  def indexOfMin[T](start: Int, arr: Array[T])(implicit odr: scala.math.Ordering[T]): Int = {
    var min_idx = start
```

```
  var i = start + 1
  while(i < arr.length){
    if(odr.lt(arr(i),arr(min_idx)))
      min_idx = i
    i = i + 1
  }
  min_idx
}

@inline
def swap[T](i: Int, j:Int, arr: Array[T]): Unit = {
  val tmp = arr(i)
  arr(i) = arr(j)
  arr(j) = tmp
}

/*
Implement selection sort algorithm
 */
def selectionSort[T](arr: Array[T])(implicit odr: scala.math.Ordering[T]): Array[T] = {
  var unsortedIdx = 0;
  while(unsortedIdx < arr.length){
    swap(unsortedIdx, indexOfMin(unsortedIdx, arr), arr)
    unsortedIdx = unsortedIdx + 1
  }
  arr
}

/*
Generate random list of integer
 */
def generateRandomCase(n: Int): Array[Int] = {
  var r = new scala.util.Random
  (1 to n) map {_ => r.nextInt()} toArray
}

/*
Simply sort the list
 */
def generateBestCase(n: Int): Array[Int] = generateRandomCase(n).sorted
/*
Reverse the sorted list
 */
def generateWorstCase(n: Int): Array[Int] = generateRandomCase(n).sortWith(_ >= _) //sort in rever


/*
Test case
 */
def main(args: Array[String]): Unit = {
  val bestCases = 1 to 5 map {_ => generateBestCase(10000)} toList
  val worstCases = 1 to 5 map {_ => generateWorstCase(10000)} toList

  // HotSpot VM warm up
  // Since Java Virtual Machine implement JIT and HotSpot algorithm by default
  // warm up a function will lead to its JIT
  //(1 to 30).foreach(_ => selectionSort(generateWorstCase(1000)))
```

```
    // start calculation
    /*
    println("Worst case:")
    println(worstCases.head mkString " ")
    worstCases foreach {Timer printsMillo selectionSort[Int](_)}
    println("------")

    println("Best case: ")
    println(bestCases.head mkString " ")
    bestCases foreach {Timer printsMillo selectionSort[Int](_)}
    */
    // prepare dataset
    val datasetBest = (1 to 20).map(_ * 1000).map(i => (i, generateBestCase(i))).toList //force eval
    //run multiple times
    val tryTime = 5
    val resultBest = (1 to tryTime).map { i => datasetBest map {
      case (num: Int, arr: Array[Int]) => (num, Timer.mkMilloTuple(selectionSort(arr))._1)
    }}.reduce(_ zip _ map {
      t => (t._1._1, t._1._2 + t._2._2)
    }).map(t => (t._1, t._2 / tryTime))

    val datasetWorst = (1 to 20).map(_ * 1000).map(i => (i, generateWorstCase(i))).toList //force ev
    val resultWorst = (1 to tryTime).map { i => datasetBest map {
      case (num: Int, arr: Array[Int]) => (num, Timer.mkMilloTuple(selectionSort(arr))._1)
    }}.reduce(_ zip _ map {
      t => (t._1._1, t._1._2 + t._2._2)
    }).map(t => (t._1, t._2 / tryTime))

    val datasetAvg = (1 to 20).map(_ * 1000).map(i => (i, generateRandomCase(i))).toList //force eva
    val resultAvg = (1 to tryTime).map { i => datasetAvg map {
      case (num: Int, arr: Array[Int]) => (num, Timer.mkMilloTuple(selectionSort(arr))._1)
    }}.reduce(_ zip _ map {
      t => (t._1._1, t._1._2 + t._2._2)
    }).map(t => (t._1, t._2 / tryTime))
    //print the resulting tuples
    println("# Statistics for (d)")
    println("# Best case")
    println(resultBest)
    println("# Worst case")
    println(resultWorst)
    println("# Average case")
    println(resultAvg)
  }
}
```

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)
  }

}
```

File 3(Unit test): SelectionSortTest.scala

```
import org.scalatest.FlatSpec

class SelectionSortTest extends FlatSpec{
  // Unit Test for selection sort algorithm
  " A index of Minimum function" should "return the index of minimum" in {
    val r = new scala.util.Random
    val rand_arr: Array[Int] = 1 to 100000 map { _ => r.nextInt() } toArray
    val idx = Timer printsMillo SelectionSort.indexOfMin(0, rand_arr)
    assert(rand_arr(idx) == rand_arr.min)
  }
  " A selection sort algorithm" should "sort random list" in{
    val r = new scala.util.Random
    val rand_arr: Array[Int] = 1 to 100000 map { _ => r.nextInt() } toArray
    val selectionSorted = SelectionSort.selectionSort(rand_arr)(scala.math.Ordering[Int])
    assert(selectionSorted.deep == rand_arr.sorted.deep)
  }
```

```
}
```

The Scala code successfully passed the test case. To compile, you can simply execute under code folder

```
# sbt
> compile
> test
```

(b)

Loop invariant:
Initialization: Consider the case when $unsortedIdx == 1$. Under such case, the subarray contains only one element, the minimum of the original array. It is sorted, and its order(or index) will remain unchanged.
Maintanence: Assuming before entering the loop, the subarray A indexed from 0 to $unsortedIdx - 1$ is sorted, and all the element in the subarray is smaller than **any** elements in the remaining subarray $S \backslash A$. Thus, upon execution of the loop, we have $A = A \cup \{min(S \backslash A)\}$. By the assumption, the new element in the subarray A is the biggest one. Hence, the ordering of A remains unchanged.
Termination: Again, loop invariant is satisfied. When terminated, applying $A = A \cup \{min(S \backslash A)\}$, we have a sorted array.

(c)

The code above, upon running, will generate the best and the worst case. And it will feed into the function and note the running time.
Best case example: 2145757357 2146074127 2146212738 2146443596 2147032252
Best case is generated by sorting.
Worst case example: 100 99 98 97 96 95
Worst case is generated by sorting and reverse the array.
Note that best case and worst case is asympototically equivalent. Since the number of comparision is the same. The only difference is on swap operation, which is minimal(number of swap operation is linear)

(d)

The data generated by executing the program is

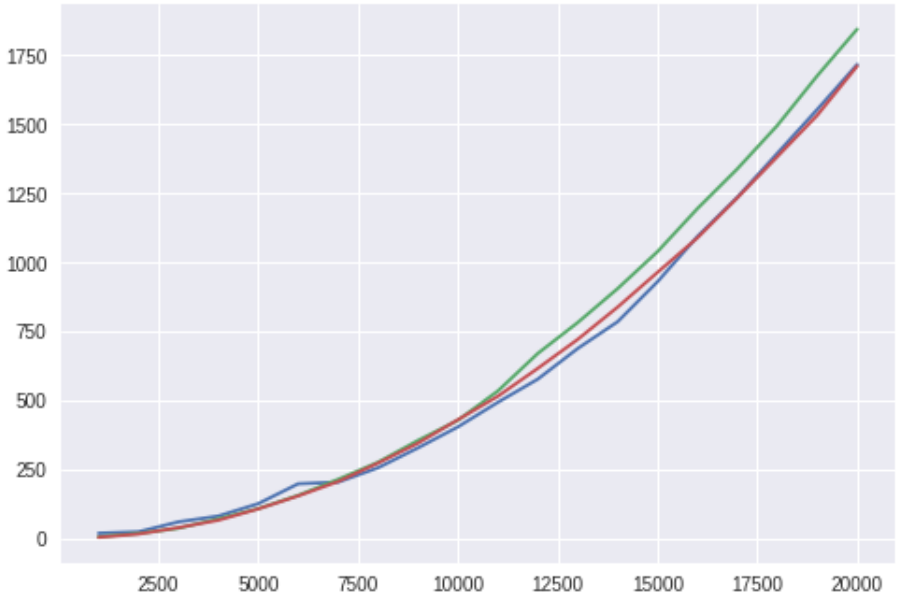
```
import numpy as np
import matplotlib.pyplot as plt

# Best case
tupleBest = [(1000,19), (2000,24), (3000,60), (4000,81), (5000,126), (6000,198), (7000,204), (8000,2
# Average case
tupleAvg = [(1000,4), (2000,16), (3000,39), (4000,66), (5000,107), (6000,154), (7000,206), (8000,273
# Worst case
tupleWorst = [(1000,6), (2000,18), (3000,37), (4000,71), (5000,108), (6000,156), (7000,214), (8000,2
def breaktuples(tup):
  xs = list()
  ys = list()
  for (x_, y_) in tup:
    xs.append(x_)
    ys.append(y_)
  return xs, ys

(xsBest, ysBest) = breaktuples(tupleBest)
(xsWorst, ysWorst) = breaktuples(tupleWorst)
(xsAvg, ysAvg) = breaktuples(tupleAvg)
plt.plot(xsBest, ysBest)
plt.plot(xsWorst, ysWorst)
plt.plot(xsAvg, ysAvg)
```

```
plt.show()
```

First argument is the size of the dataset, second argument is the running time in millosecond. Let's draw the graph and visualize.
Green is the Worst case. Blue is the Best case. Red is the Average case.



(e)

Interpretation: The difference between three cases is minimal. This algorithm can be modeled by the number of comparision made in the process of sorting. In a array of size n, in the i^{th} loop, the number of comparision in indexOfMin function is $n-i+1$, and we perform it n times. The total comparision is:

$$Comparison = \sum_{i=1}^{n}(n-i+1) = \frac{n(n+1)}{2} \in O(n^2)$$

which clearly explains the graph. It is quadratic. The reason between best case and the worst case performance is the assignment in the min function and the swap operation.