# Algorithms and Data Structure 4

Yiping Deng

March 5, 2018

## Problem 1

### a)

The algorithm for bubble sort is shown in 1

**Algorithm 1** Bubble Sort

```
procedure BubbleSort($A$)                          ▷ Input a array $A$
    $swapped \leftarrow true$                      ▷ Make sure it will execute at least once
    while swapped do                               ▷ Keep executing until it is sorted
        $swapped \leftarrow false$
        for $i$ from 1 to A.size - 1 do
            if $A[i] \geq A[i+1]$ then
                $swapped \leftarrow true$
                $swap(A[i], A[i+1])$               ▷ Swap two misplaced elements in the array
            end if
        end for
    end while
end procedure
```

### b)

**Best Case:** If the array is already sorted, the if condition is never satisfied, and the loop is executed once, which will give us $O(n)$ comparisions, and it is also its running time.
**Worst Case:** If the array is inversely sorted, the last element in the original array $A$ will be swapped to the front(its correct position) in $n-1$ loops, with one extra loop of termination. Still, the number of comparision will be $O((n-1) \cdot n) = O(n^2)$
**Average Case:** On uniform distributed array $A$, bubble sort's number of loops are bounded below by $n/2$, and bounded above by $n$, so it is $O(n)$ loop, with each loop takes $O(n)$. It implies a running time of $O(n) \cdot O(n) = O(n^2)$

### c)

- Insertion Sort: Stable sort. If two element is in sorted order, it will not be swapped.
- Merge Sort: Stable sort. If the merge operation is properly implemented, it will preserve the relative order of the object. In the split operation, order is preserved completely.
- Heap Sort: Unstable sort. The sorted array is derived from continuously popping elements from the heap. However, the heapify operation will break the relative ordering.
- Bubble Sort: Stable sort. You will not swap any element in a sorted order, and it keeps the relative order in the list.

### d)

- Insertion Sort: Adaptive. The best case is linear while the average case is quadratic.
- Merge Sort: Not adaptive. The number of merge operation does not change.
- Heap Sort: Not adaptive. Number of heapify call does not change.
- Bubble Sort: Adaptive. The best case is linear while the average case is quadratic.

## Problem 2

### a)

Code is here for HeapSort.scala

```scala
object HeapSort {
  def indexLeft(i: Int) = (2 * i) + 1 // zero indexed system
  def indexRight(i: Int) = indexLeft(i) + 1
  def indexParent(i: Int) = (i - 1) / 2

  def swap[T](a: Array[T], i: Int, j: Int): Unit = {
    //generic swap function for array
    val tmp = a(i)
    a(i) = a(j)
    a(j) = tmp
  }

  def maxHeapify[T](a: Array[T], i: Int, size: Int)(implicit ord: Ordering[T]): Unit = {
    //heapify at index i
    val idxLeft = indexLeft(i)
    val idxRight = indexRight(i)

    //find the largest in the small tree branch
    var largest = -1
    largest = if(idxLeft < size && ord.gt(a(idxLeft), a(i))) idxLeft else i
    largest = if(idxRight < size && ord.gt(a(idxRight), a(largest))) idxRight else largest
    if(largest != i){
      swap(a, i, largest)
      maxHeapify(a, largest, size)
      // let A[i] float down so that A[i] is always larger than its branches
    }
  }

  def maxHeapBuild[T](a: Array[T], size: Int)(implicit ord: Ordering[T]): Unit = {
    val half = size / 2
    for( i <- 0 to half){
      maxHeapify(a, half - i, size) //in a backward order
    }
  }

  def heapSort[T](a:Array[T])(implicit ord: Ordering[T]):Unit = {
    maxHeapBuild(a, a.length) //build a maxheap
    for(i <- (0 until a.length).reverse){
      //in a reverse order sort it
      swap(a, 0, i) //put the last one at the back
      maxHeapify(a, 0, i) //the size is changing
    }
  }
}
```

## b)

The variant code is here

```
object HeapSortVariant {
  import HeapSort._

  def moveBottom[T](a: Array[T], size: Int): Unit = {
    if(size <= 1)
      return
    val top = a(0)
    //move the array
    System.arraycopy(a, 1, a, 0, size - 1)
    a(size - 1) = top
  }
  def recoverBottom[T](a: Array[T], size: Int): Unit  = {
    if(size <= 1)
      return
    //recover the array
    val top = a(size - 1)
    System.arraycopy(a, 0, a, 1, size - 1)
    a(0) = top
  }

  def checkHeap[T](a: Array[T], parent: Int, size: Int)(implicit ord: Ordering[T]): Boolean = {
    val left = indexLeft(parent)
    val right = indexRight(parent)

    var largest = -1
    largest = if(left < size && ord.gt(a(left), a(parent))) left else parent
    largest = if(right < size && ord.gt(a(right), a(largest))) right else largest

    if(left < size && right < size)
      largest == parent && checkHeap(a, left, size) && checkHeap(a, right, size)
    else if (left < size)
      largest == parent && checkHeap(a, left, size)
    else if (right < size)
      largest == parent && checkHeap(a, right, size)
    else
      true
  }

  def tryBottom[T](a: Array[T],  size: Int)(implicit ord: Ordering[T]):Boolean = {
    //try to move to the bottom
    if(size <= 1)
      return true

    moveBottom(a, size)


    /*
    val parent = indexParent(size - 1)
    val left = indexLeft(parent)
    val right = indexRight(parent)

    var largest = -1
    largest = if(left < size && ord.gt(a(left), a(parent))) left else parent
    largest = if(right < size && ord.gt(a(right), a(largest))) right else largest
*/
    if(size == 2 && checkHeap(a, 0, 2))
      true
    //need to make sure the top part is working and the ending leaf is working
    if(checkHeap(a, indexParent(size - 1), size) && checkHeap(a, 0, 3))
      true
    else {
      recoverBottom(a, size)
      false
    }
  }
  def heapSortVariant[T](a: Array[T])(implicit ord: Ordering[T]):Unit = {
    maxHeapBuild(a, a.length)
    for(i <- a.indices.reverse){
      //in a reverse order sort it
      swap(a, 0, i) //put the last one at the back
      if(!tryBottom(a, i)) maxHeapify(a, 0, i) //only heapify when it is not right
    }
  }

  def genArray(n: Int):Array[Int] = {
    val r = new scala.util.Random
    1 to n map { _ => r.nextInt(100)} toArray
  }
  def dataGenerator[T](sort_algo:(Array[Int]) => Unit):String = {
    val input = (1 to 40) map { _ * 1000}
    //generate data set
    val result = input map {
      n =>
        val arr = genArray(n)
        (n, Timer.avergedJustTime(sort_algo(arr)))
    }

    "[" + result.mkString(",") + "]"
  }
  def main(args: Array[String]):Unit = {
    // the program that produces results
    println("normal = " + dataGenerator(heapSort[Int]))
    println("variant = " + dataGenerator(heapSortVariant[Int]))
  }
}
```

Also the test case for both algorithm, proving its correctness.

```
import org.scalatest.FlatSpec

class HeapSortTest extends FlatSpec{
  "A Heap Sort algorithm " should "sort everything" in {
    val r = new scala.util.Random
    val rand_arr = 1 to 10000 map { _ => r.nextInt()} toArray
    val rand_arr_cpy = rand_arr.clone()
    HeapSort.heapSort(rand_arr)
    println(rand_arr)
    assert(rand_arr.deep == rand_arr_cpy.sorted.deep)
  }
  "A Heap Sort Variant" should "sort everything" in {
    val r = new scala.util.Random
    val rand_arr = 1 to 10 map { _ => r.nextInt(100)} toArray
    val rand_arr_cpy = rand_arr.clone()
    HeapSortVariant.heapSortVariant(rand_arr)
    println(rand_arr)
    assert(rand_arr.deep == rand_arr_cpy.sorted.deep)
  }
  "Move to Bottom Method" should "move one element" in {
    val arr = Array(1, 2, 3, 4)
    val expected = Array(2, 3, 4, 1)
    val recovered = arr.clone()
    HeapSortVariant.moveBottom(arr, 4)
    assert(arr.deep == expected.deep)
    HeapSortVariant.recoverBottom(arr, 4)
    assert(arr.deep == recovered.deep)
  }
}
```

## c)

The original implementation is actually faster. Consider the floating process, the original process take $O(lg(n))$ times to put in the right position. While in our case, the floating means more swap operation on the entire array, and in most of the case, the root and its two direct child is not in the right position. Thus, it turns out it will be slower. Because the Prof. explicit mentioned a different way of floating, I do not use heapify function to float it down and fix the head. If you use the code above to generate the data, and draw the graph using python code, you can see it clearly. The green one is much slower.

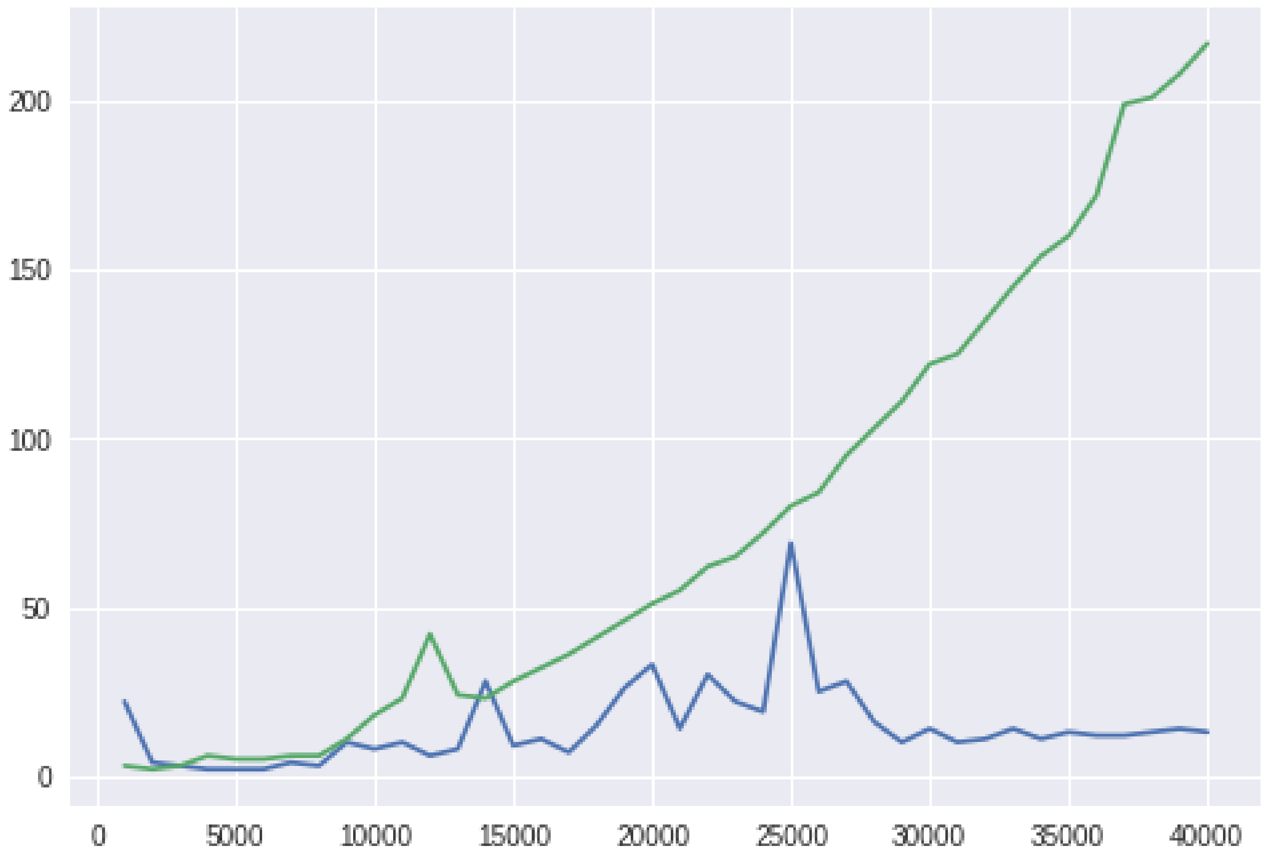
```
import numpy as np
import matplotlib.pyplot as plt

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)

normal = [(1000,22),(2000,4),(3000,3),(4000,2),(5000,2),(6000,2),(7000,4),(8000,3),(9000,10),(10000,
variant = [(1000,3),(2000,2),(3000,3),(4000,6),(5000,5),(6000,5),(7000,6),(8000,6),(9000,11),(10000,
plotTogether(normal)
plotTogether(variant)
plt.show()
```

Let's present the graph here.


200
150
100
50
0
0
5000
10000
15000
20000
25000
30000
35000
40000