California State University, Northridge

Data Structures and Program Design

COMP 182

Lab 3: Time Complexity for Mergesort

Written by
Kyle Nguyen

Professor
Mohammed Abdelrahim

October 10, 2017

1 Introduction

The mergesort algorithm was provided by `http://rosettacode.org/wiki/Sorting_algorithms/Merge_sort#Java`, where the algorithm would take a list of any size and sort the list in ascending order. How merge sort works is it's a divide and conquer algorithm where it splits the unsorted sequence of numbers into pairs by dividing each subsequence larger than 2 by half. It makes the algorithm very efficient when sorting and once the set of numbers are reduced to their smallest size they are sorted and joined together two at a time to the final solution to the original problem.

2 Time Analysis for Mergesort

The objective is to determine the time complexity of mergesort. Merge sort takes n elements in the entire list.

```java
public static <E extends Comparable<? super E>> List<E>
    mergeSort(List<E> m) {
  if (m.size() <= 1)
    return m;

  int middle = m.size() / 2;
```

The mergesort class has a method which takes a list m with n elements in it. If the list has only 1 element, it returns since the list is already sorted, if it is greater than 1 it will divide the number of elements m in half. It creates a left and a right where the left would start from 0 to the *middle* and the right would start from the *middle* to the end or size of the list $m.size()$. These statements are simple statements will execute in constant time for some constant

$$c1$$

```java
  List<E> left = m.subList(0, middle);
  List<E> right = m.subList(middle, m.size());
```

These assignments will iterate n times through the list, where n is the length of the list. Running at least one of these statements would be $c2$

where it is some constant, then the cost of running them would be

$$c2 * n$$

```
    right = mergeSort(right);
    left = mergeSort(left);
    List<E> result = merge(left, right);
```

The list is then assigned to a recursive function that keeps finding dividing the left and right indices until they have both reached the midpoints of their set. Looking at the recursive call for *mergeSort*, we are saying that for some time $T(n)$, the mergeSort for right would cost

$$T(\frac{n}{2})$$

and left would also be

$$T(\frac{n}{2})$$

. The finally, $Line3$, will take one element from the left or right at a time and write them into another list. So all the loops were running at a total length of the $left$ list and $right$ list giving us a time of n times where some constant $c3$ will be

$$c3 * n$$

2.1 Solving Recurrence Relation

By putting everything together we will be able to calculate the time complexity for the algorithm mergesort.

$$T\left(n\right) = 2T\left(\frac{n}{2}\right) + \left(c2 + c3\right)n + c1$$

We can say $(c2 + c3)$ is another constant $c4$ which results in:

$$T\left(n\right) = 2T\left(\frac{n}{2}\right) + \left(c4\right)n + c1$$

We can drop the lowest term since the constant $c1$ is negligible in comparison to the other terms resulting in:

$$T\left(n\right) = 2T\left(\frac{n}{2}\right) + \left(c4\right)n$$

We can simplify it further by making it:

$$T\left(n\right)=2[2T\left(\frac{n}{4}\right)+\left(c4\right)\left(\frac{n}{2}\right)]+\left(c4\right)n$$

$$T\left(n\right)=4T\left(\frac{n}{4}\right)+2\left(c4\right)n$$

It can go further making it:

$$T\left(n\right)=4[2T\left(\frac{n}{8}\right)+\left(c4\right)(\frac{n}{4})]+2\left(c4\right)n$$

$$T\left(n\right)=8T\left(\frac{n}{8}\right)+3\left(c4\right)n$$

And:

$$T\left(n\right)=16T\left(\frac{n}{16}\right)+4\left(c4\right)n$$

Then we create some generic term k we can make it:

$$T\left(n\right)=2^kT\left(\frac{n}{2^k}\right)+k\left(c4\right)n$$

By reducing it in terms of constant time:

$$\frac{n}{2^k}=1$$

$$2^k=n$$

$$k=log_2n$$

We can rewrite it as:

$$T\left(n\right)=2^{(log_2n)}T\left(\frac{n}{2^{(log_2n)}}\right)+\left(log_2n\right)\left(c4\right)n$$

$$T\left(n\right)=\left(c5\,n\right)T\left(1\right)+c4\,n\,log_2n$$

Let c6 be some constant $(c5*T(1))$

$$T\left(n\right)=c6\,n+c4\,n\,log_2n$$

$$T\left(n\right)=c4\,n\,log_2n$$

3 Results and Conclusions

By solving the recurrence relation for the mergesort algorithm does yield a time complexity of $O(n) = n\,log(n)$ and the plotted points does indeed show an average sublinear growth rate for all cases which is indeed the Time Complexity for mergesort.

3.1 Benchmark of Mergesort Graph

In order to plot the points of how fast the algorithm mergesort sorts the input of increasing size, I've tested inputs of size 1 - 10,000 at intervals of 1,000. For each input size, 10 benchmarks were performed and an average was created. The time in which it was calculated in was in nanoseconds (ns) and then converted into milliseconds on the graph for clarity reasons. When plotted, it yields a graph resembling the sublinear graph which was indeed the conclusion earlier shown. The benchmarked result was then bounded by $c_1\, n\, log_2 n$ and $c_2\, n\, log_2 n$ showing that mergesort's Time Complexity is indeed: $O(n\, log\, n)$, $\Theta(n\, log\, n)$, and $\Omega(n\, log\, n)$.

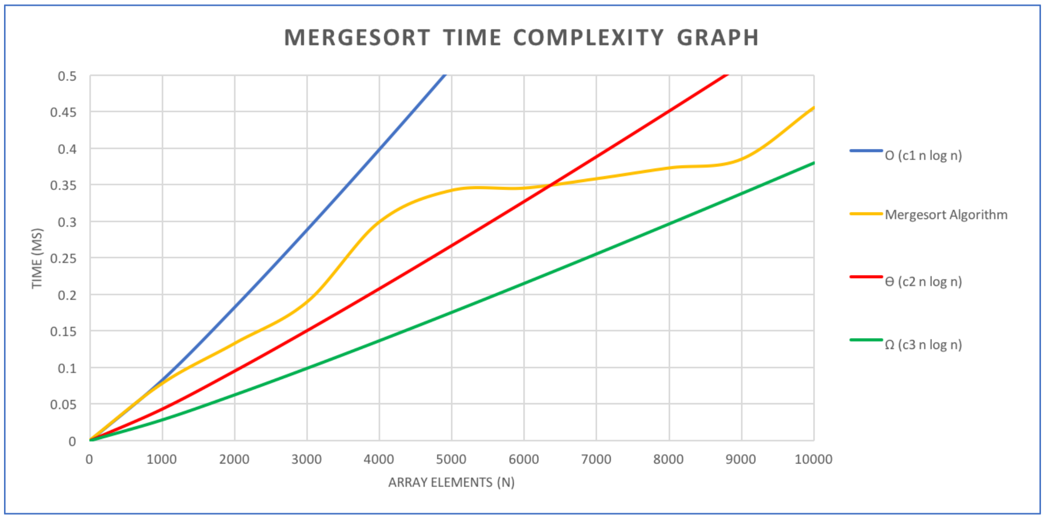

Figure 1: Graph of Mergesort Algorithm with varying inputs from (1-10,000).

3.2 Code to Create Jagged Array

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;
import java.util.stream.Collectors;

public class main {
   public static final int INITIALIZE_RESOURCES = 100000; //
       Variable to initialize resources, else poor results
   public static final int INITIALIZE_RESOURCES2 = 0; // Variable to
       initialize resources, else poor results

   public static void main(String[] args) {
      averageBenchmark();
   }

   /***************************
    * AVERAGE BENCHMARK
    ***************************/
   public static void averageBenchmark() {
      // Array to test inputs from (1 to 10,000)
      int[] numElements = new int[] { INITIALIZE_RESOURCES,
          INITIALIZE_RESOURCES2, INITIALIZE_RESOURCES2, 1, 1000,
          2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000 };

      // Benchmark results for a total of 10 times
      for (int i = 1; i <= 10; i += 1) {
         System.out.println("Test " + i);
         benchmark(numElements.length, numElements); // (to what
             element in numElements, numElement's values)
         System.out.println("");
      }
   }

   /***************************
    * BENCHMARK ALGORITHM
    ***************************/
   public static void benchmark(int x, int[] y) {
        int[][] myTests = getArray(x, y);

        for (int[] jagArray : myTests) {
            algTime(jagArray);
        }
    }

    /***************************
     * CREATE JAGGED ARRAY
     ***************************/
    public static int[][] getArray(int x, int[] y) {
        // Create Jagged array with numElement rows
        int[][] result = new int[x][];

        // Create rows with numElement's values
        for (int i = 0; i < x; i += 1) {
            int[] inputinput = new int[y[i]];
            result[i] = populate(inputinput);
        }

        return result;
    }

    /***************************
     * POPULATE ARRAY
     ***************************/
    public static int[] populate(int[] input) {
        Random rand = new Random();

        // Iterate through and populate with numbers from (-5000,
             10,000)
        for (int i = 0; i < input.length; i += 1) {
            input[i] = rand.nextInt(10000) - 5000;
        }

        return input;
    }

    /***************************
     * ALGORITHM TIME
     ***************************/
  public static void algTime(int[] input) {
     // Array to Integer List
     List<Integer> myList = arrayToList(input);

     // Algorithm Timer
     long start = System.nanoTime();
     Merge.mergeSort(myList);
     long finish = System.nanoTime();

     long algTime = finish - start;

     // If it's initalizing resources don't print anything (garbage)
     if (input.length == INITIALIZE_RESOURCES || input.length ==
         INITIALIZE_RESOURCES2) {

     }

     // Not garbage, print stuff
     else {
        System.out.println("It takes " + algTime + " nano seconds
            to sort an array of size " + input.length);
     }
  }

  /***************************
   * ARRAY TO INTEGER LIST
   ***************************/
  private static List<Integer> arrayToList(int[] input) {
     // Create List of Integers
     List<Integer> list = new ArrayList<>();

     // Loop through and add to the list
     for (int i : input) {
        list.add(i);
     }

     return list;
  }
}
```

3.2.1 Code Review

The code above generates 10 jagged arrays with varying sizes in the array and randomly populates them. It is then sorted using mergesort and timed for how long it took to sort the array of different sizes. A graph of the number of elements vs. the time it took to sort them was plotted. The first three values in the array $numElements$ were variables to ignore the time it took to initialize the resources since the first couple of tests were outliers since Java takes away time to initialize all the resources making the first couple of tests invalid. This was a workaround I created to create consistent results. The arrays will then be populated with random numbers and then converted into a Integer list and ran through the aglorithm mergesort provided. A speed test of how long it took to sort the items in the list was taken over the course of 10 tests. These numbers were later recorded and plotted into the graph of time in (ms) versus the $number of elements$.