09/02/2017

Report Tutorial Course 1

Algorithmics and Advanced Programming

WU Zheng

Report Tutorial Course 1

Algorithmics and Advanced Programming

The input data

The running time depends mainly on the size of the input data to be processed. I used the following utility class to generate random data.

```java
import java.util.Date;
import java.util.Random;

public class RandomData {
    public static int[] generate1d(int nbVals, int min, int max) {
        int[] res = new int[nbVals];
        Random generator = new Random(new Date().getTime());
        for (int i = 0; i != nbVals; ++i) {
            res[i] = (int)((Math.abs(generator.nextLong()) % ((long)max - min)) +
min);
        }
        return res;
    }
}
```

The function $generate1d()$ returns an array of $nbVals$ numbers ranging from min to max. The numbers are generated randomly.

Minimum of an array

As shown below, this is the algorithm I designed to solve the problem of finding the minimum element of an array of numerical values. In the worst case, `if (anInputData < min)` and `i++` and `i < inputData.length` will be executed n times, `int min = inputData[0]` and `min = anInputData;` and `return min;` will be executed one time. Then the number of elementary operations performed by the algorithm is $3n + 3$. Using the big O notation, its time complexity is $O(n)$.

```java
private static int findMinimumValue(int[] inputData) {
    int min = inputData[0];
    for (int anInputData : inputData) {
        if (anInputData < min) min = anInputData;
    }
    return min;
}
```

After using the function $generate1d()$ to generate arrays of different sizes ranging from 500,000 to 10,000,000 by 100,000. I ran the algorithm designed for the generated arrays and calculate the running time, and made a plot of the running time versus the number of elements. Results shown in Figure 1.1.

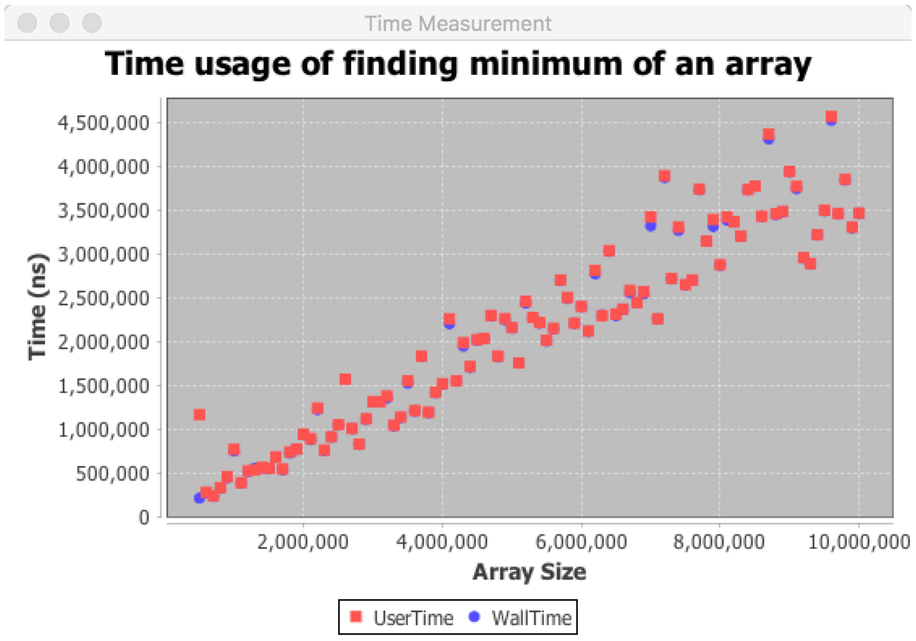

Figure 1.1

The plot shows a shape of a straight line. When the array size increases, the time usage increases proportionally. That is a linear equation, so the experimental results agree with the theorethical complexity of my algorithm, the time complexity is $O(n)$.

Sorting Algorithms

The basic operations of sorting algorithms are comparison and exchange. In order to exchange values in a (fixed-size) array, one has to swap elements. This can be achieved by the following code:

```
private static void swap(int[] data, int i, int j) {
    int tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
}
```

The `int tmp = data[i];` and `data[i] = data[j];` and `data[j] = tmp;` will be executed one time only. The algorithmic complexity of this function is $O(1)$.

Selection sort

Selection sort is a sorting algorithm based on the fact that when an array is sorted (in ascending order), given an element i, this is the minimum of the elements of indices greater or equal to the index of i. Running times are compounded (multiplied) in nested loops, as we can see in the following implementation selection sort functions.

```
private static int findMinimumIndex(int[] inputData, int begin, int end) {
    int res = begin;
    for (int i = begin + 1; i < end; i++) {
        if (inputData[i] < inputData[res]) {
            res = i;
        }
    }
    return res;
}
private static void selectionSort(int[] inputData) {
    if (inputData.length < 2) return;
    for (int i = 0; i < inputData.length; i++) {
        swap(inputData, i, findMinimumIndex(inputData, i, inputData.length));
    }
}
```

Assume that we have n elements in the array. In the worst case, we made $(n-1)$ comparisons to find the first smallest, $(n-2)$ comparisons to find the second smallest, finally we made 1 comparison to find the last smallest one. The total comparisons is $(n(n-1))/2$, the theorethical complexity of this algorithm is $O(n^2)$.

I made a plot of the running time versus the number of elements. To get readable result, I adjusted the array size range to 5,000-100,000.

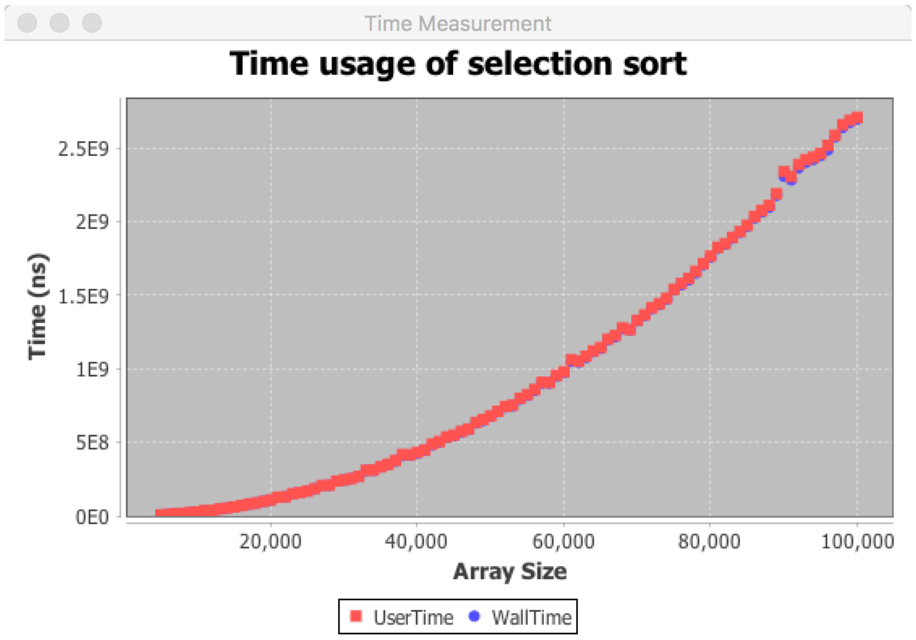

Figure 2.1 - Time usage of selection sort

The curve appears to agree with the graph of a quadratic equation. The experimental results agree with the theorethical complexity of my algorithm, the time complexity is $O(n^2)$.

Bubble sort

Bubble sort takes (all) adjacent pairs of elements and orders them. This process is repeated until all adjacent pairs are ordered. The implementation functions are shown below.

```
private static void bubbleSort(int[] inputData) {
    if (inputData.length < 2) return;
    boolean hadToSwap;
    do {
        hadToSwap = false;
        for (int i = 0; i != inputData.length - 1; ++i) {
            if (inputData[i] > inputData[i + 1]) {
                swap(inputData, i, i + 1);
                hadToSwap = true;
            }
        }
    } while (hadToSwap);
}
```

Assume that we have n elements in the array. In the worst case, we made $(n-1)$ swaps in the first loop, $(n-2)$ swaps in the second loop, finally we made 1 swap between the last two numbers. The total count of swap is $(n(n-1))/2$, and the theorethical complexity of this algorithm is $O(n^2)$.

In previous part, I adjusted the array size range to 5,000-100,000, but I found it may take too long to finish computing. So I changed it to 5,00-10,000. Here is the result.

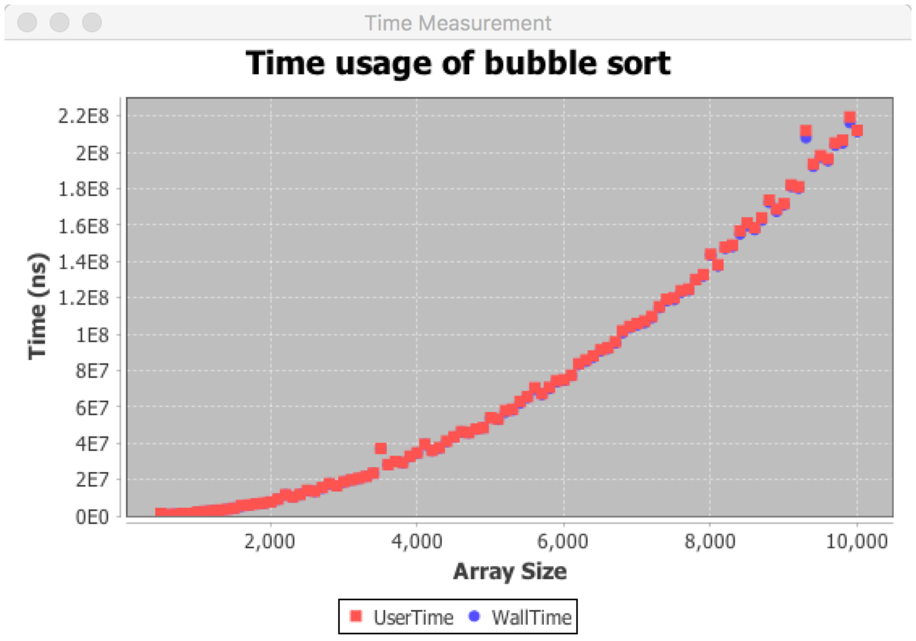

Figure 2.2 – Time usage of bubble sort

The curve appears to agree with the graph of a quadratic equation. The experimental results agree with the theorethical complexity of my algorithm, the time complexity is $O(n^2)$, the same as the selection sort.

But I found selection sort almost always outperforms bubble sort and gnome sort. To answer this question we need to focus on a detailed analysis[1] of those two algorithms.

Merge Sort

Merge sort is based on the *Divide and conquer* paradigm. Given the following implementation to merge two sorted consecutive intervals.

```
private static void mergeSorted(int inputData[], int begin, int middle, int end) {
    int[] tmp = new int[middle - begin];
    System.arraycopy(inputData, begin, tmp, 0, tmp.length);
    int i = 0, j = middle, dest = begin;
    while ((i < tmp.length) && (j < end)) {
        inputData[dest++] = (tmp[i] < inputData[j]) ? tmp[i++] : inputData[j++];
    }
    while (i < tmp.length) {
        inputData[dest++] = tmp[i++];
    }
}
```

Assume that we have n elements in both two intervals, `while ((i < tmp.length) && (j < end))` and `while (i < tmp.length)` traverse the whole auxiliary array `tmp`. So in the worst case, the number of compartions and assignment in this function is n. Its memory usage is n.

Given the following implementation of a merge sort.

```
private static void mergeSort(int[] inputData) {
    mergeSort(inputData, 0, inputData.length);
}
private static void mergeSort(int[] inputData, int begin, int end) {
    if ((end - begin) < 2) {
        return;
    }
    int middle = (end + begin) / 2;
    mergeSort(inputData, begin, middle);
    mergeSort(inputData, middle, end);
    mergeSorted(inputData, begin, middle, end);
}
```

Define that $T(N)$ is the number of comparisons to mergesort an input of size N. According to our implementation, we have:

[1] *http://cs.stackexchange.com/questions/13106/why-is-selection-sort-faster-than-bubble-sort*

$$\begin{cases} T(1) = 0 \\ T(N) = T\left(\frac{N}{2}\right) + T\left(\frac{N}{2}\right) + N \ (N > 1) \end{cases}$$

Assume that N is a power of 2, and we claim that, if $T(N)$ satisfies this recurrence, then $T(N) = NlgN$.
Proof (by induction on N):

- Base case: $N = 1, T(1) = 1lg1 = 0$
- Inductive hypothesis: $T(N) = NlgN$
- Goal: Show that $T(2N) = 2Nlg(2N)$.
    - $T(2N) = 2T(N) + 2N$ (given)
    - $= 2NlgN + 2N$ (inductive hypothesis)
    - $= 2N(lg(2N) - 1) + 2N$ (algebra)
    - $= 2Nlg(2N)$ (QED)

The algorithmic complexity of this algorithm is $O(nlgn)$.

After adjusting the array size range to 50,000-1,000,000, the result shows the same trend as a linearithmic function.

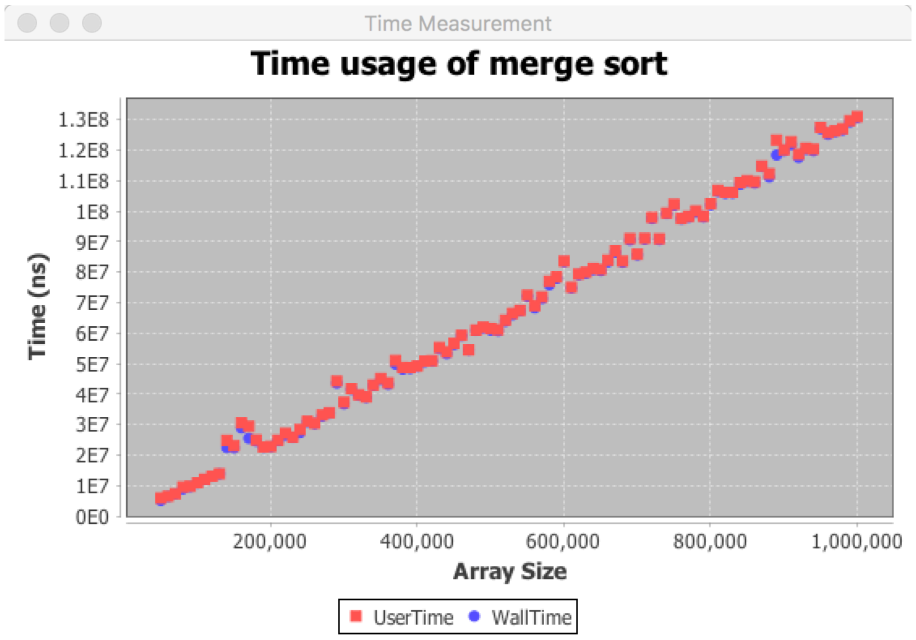

Figure 2.3 – Time usage of merge sort

The experimental results agree with the theorethical complexity of my algorithm, the time complexity is $O(nlgn)$.

Quick sort

Another way to "divide and conquer" for a sorting algorithm is to pre-process the data before dividing so as to make the merging trivial. Quicksort does so by partitioning the data according to a pivot element.

This enables in-place sorting without any external memory. Given the following implementation of a quick sort.

```
private static int partition(int[] inputData, int begin, int end, int pivotIdx) {
    swap(inputData, pivotIdx, --end);
    pivotIdx = end;
    int pivot = inputData[pivotIdx];
    // invariant is that everything before begin is known to be < pivot
    // and everything after end is known to be >= pivot
    while (begin != end) {
        if (inputData[begin] >= pivot) {
            swap(inputData, begin, --end);
        } else {
            ++begin;
        }
    }
    swap(inputData, pivotIdx, begin);
    return begin;
}
private static void quickSort(int[] inputData) {
    quickSort(inputData, 0, inputData.length);
}
private static void quickSort(int[] inputData, int begin, int end) {
    if ((end - begin) < 2) {
        return;
    }
    int m = partition(inputData, begin, end, (end + begin) / 2);
    quickSort(inputData, begin, m);
    quickSort(inputData, m + 1, end); // +1 for convergence
}
```

The Master Theorem[2] applies to recurrences of the following form:

$$T(n) = aT(N/b) + f(n)$$

where $a \geq 1, b > 1$ are constants and $f(n)$ is an asymptotically positive function.

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)

[2] *https://en.wikipedia.org/wiki/Master_theorem*

- $f(n)$ is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.

There are 3 cases:

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} log^{k+1} n)$.
   Most of the time, $k = 0$.
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$.
   Regularity condition: $af(n/b) <= cf(n)$ for some constant $c < 1$ and all sufficiently large n.

And each partition of the quick sort divides a problem into two problems, that is:

$$T(n) = 2T(n/2) + O(n)$$

Where $O(n)$ is the time complexity of $partition()$. Therefore $a = 2, b = 2, f(n) = O(n) \Rightarrow n^{\log_b a} = n$ (case 2), $T(n) = \Theta(n \log n)$. But in the worst case, the partition does not perform well:

$$T(n) = T(n-1) + T(1) + O(n)$$

Which time complexity is $O(n^2)$. And its memory complexity is $O(n)$.

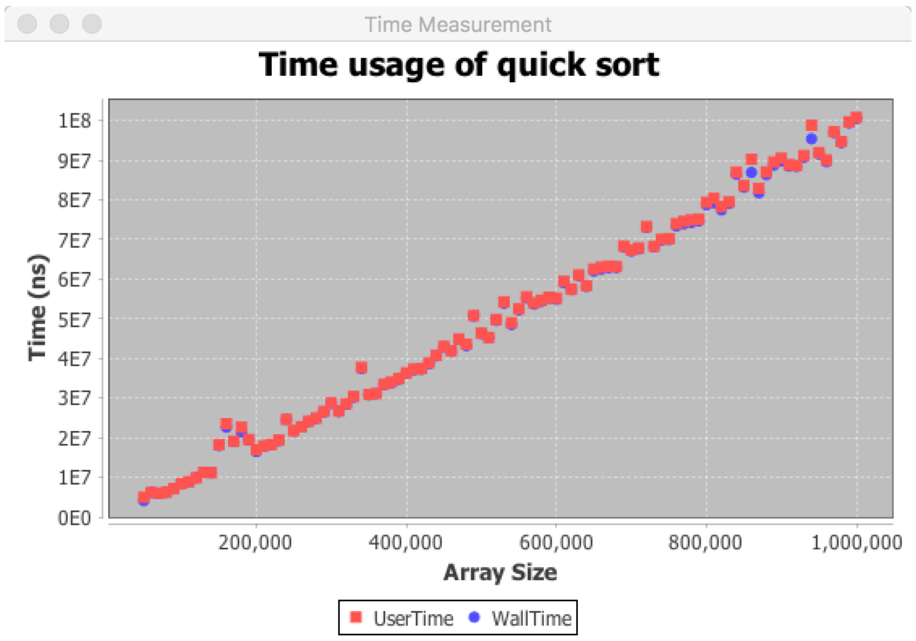

Figure 2.4 – Time usage of quick sort

After adjusting the array size range to 50,000-1,000,000, the result shows the same trend as a linearithmic function. The experimental results agree with the theorethical complexity of the algorithm $O(nlgn)$. The quick sort is faster than mergesort in practice because of lower cost of other high-frequency operations.

Comparison of sorting algorithms

The sizes of the arrays range from 500 to 10,000. I made the plot below by combining these four sorting algorithms. Bubble sort has the worst performance with the average and the worst complexity of $O(n^2)$. Selection sort has the same complexity but in practice it is faster than bubble sort. Their memory complexity is $O(1)$. While merge sort and quick sort need more memory, they are much more faster with the average complexity of $O(nlgn)$.

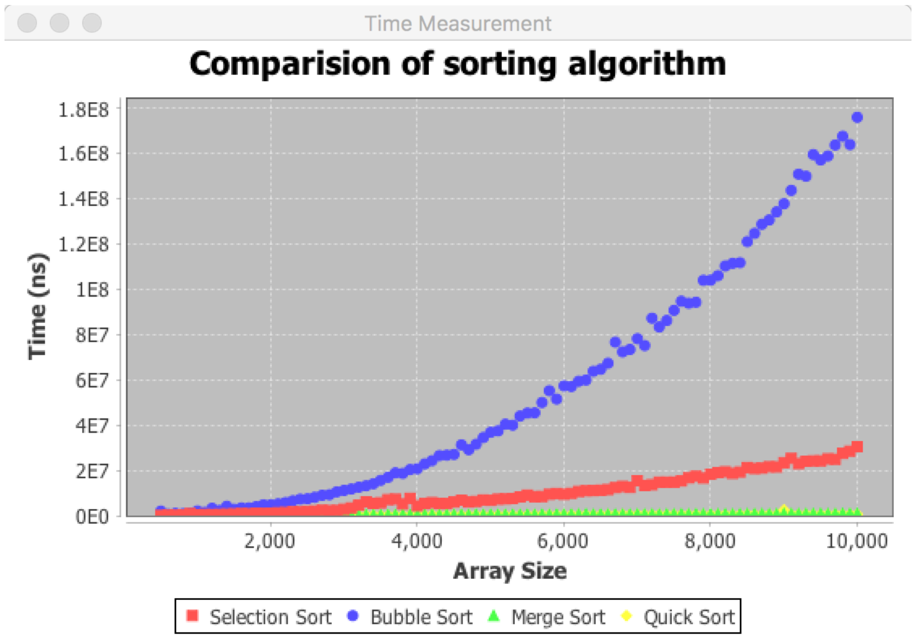

Figure 2.5 – Comparison of sorting algorithms

Binary search

Consider the problem of searching a value in a sortes array. An e cient algorithm to solve this problem is called binary search. The principle is to compare the searching value with the median. Whether the median is lower or greater than the searched value, this one comparison allows to discard half (respectively the upper or lower half) of the array, by transitivity of the ordering relationship. The following algorithm is an implementation of binary search.

```
// index of first element >= v
private static int binarySearch(int[] data, int v) {
    return lowerBound(data, v, 0, data.length);
}
private static int lowerBound(int[] data, int v, int begin, int end) {
    if (begin == end) {
        return begin;
    }
    int m = (begin + end) / 2;
    return data[m] < v ? lowerBound(data, v, m + 1, end) : lowerBound(data, v, begin,
m);
}
```

Its memory complexity is O(n). Define that $T(N)$ is the number of comparisons to perform a search in an input of size N. According to our implementation, we have:

$$\begin{cases} T(1) = 1 \\ T(N) = T\left(\frac{N}{2}\right) + O(1)(N > 1) \end{cases}$$

In this case, $a = 1, b = 2, n^{log_b a} = 1$, Its complexity is $O(lgn)$.

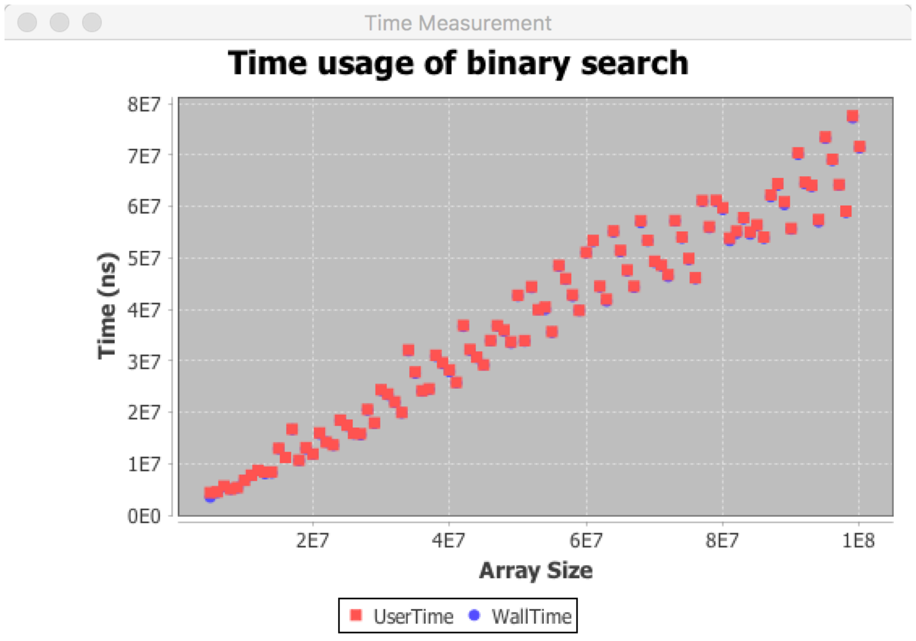

Figure 3.1 – Time usage of binary search

Appendix I. Experiment Code

```java
package com.darwindev;

import java.awt.Shape;
import java.lang.management.ManagementFactory;
import java.lang.management.ThreadMXBean;

import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.data.xy.XYDataset;
import org.jfree.data.xy.XYSeries;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RefineryUtilities;
import org.jfree.util.ShapeUtilities;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.xy.XYSeriesCollection;

/**
 * Created by Zheng on 08/02/2017.
 *
 */
public class TimeMeasurementTargets extends ApplicationFrame {

    private static int findMinimumValue(int[] inputData) {
        int min = inputData[0];
        for (int anInputData : inputData) {
            if (anInputData < min) min = anInputData;
        }
        return min;
    }

    private static int findMinimumIndex(int[] inputData, int begin, int end) {
        int res = begin;
        for (int i = begin + 1; i < end; i++) {
            if (inputData[i] < inputData[res]) {
                res = i;
            }
        }
        return res;
    }

    private static void swap(int[] data, int i, int j) {
        int tmp = data[i];
        data[i] = data[j];
        data[j] = tmp;
    }

    private static void selectionSort(int[] inputData) {
        if (inputData.length < 2) return;
        for (int i = 0; i < inputData.length; i++) {
            swap(inputData, i, findMinimumIndex(inputData, i, inputData.length));
        }
    }
```

```java
private static void bubbleSort(int[] inputData) {
    if (inputData.length < 2) return;
    boolean hadToSwap;
    do {
        hadToSwap = false;
        for (int i = 0; i != inputData.length - 1; ++i) {
            if (inputData[i] > inputData[i + 1]) {
                swap(inputData, i, i + 1);
                hadToSwap = true;
            }
        }
    } while (hadToSwap);
}

private static void mergeSorted(int inputData[], int begin, int middle, int end) {
    int[] tmp = new int[middle - begin];
    System.arraycopy(inputData, begin, tmp, 0, tmp.length);
    int i = 0, j = middle, dest = begin;
    while ((i < tmp.length) && (j < end)) {
        inputData[dest++] = (tmp[i] < inputData[j]) ? tmp[i++] : inputData[j++];
    }
    while (i < tmp.length) {
        inputData[dest++] = tmp[i++];
    }
}

private static void mergeSort(int[] inputData) {
    mergeSort(inputData, 0, inputData.length);
}

private static void mergeSort(int[] inputData, int begin, int end) {
    if ((end - begin) < 2) {
        return;
    }
    int middle = (end + begin) / 2;
    mergeSort(inputData, begin, middle);
    mergeSort(inputData, middle, end);
    mergeSorted(inputData, begin, middle, end);
}

private static int partition(int[] inputData, int begin, int end, int pivotIdx) {
    swap(inputData, pivotIdx, --end);
    pivotIdx = end;
    int pivot = inputData[pivotIdx];
    // invariant is that everything before begin is known to be < pivot
    // and everything after end is known to be >= pivot
    while (begin != end) {
        if (inputData[begin] >= pivot) {
            swap(inputData, begin, --end);
        } else {
            ++begin;
        }
    }
    swap(inputData, pivotIdx, begin);
    return begin;
}
```

```
    private static void quickSort(int[] inputData) {
        quickSort(inputData, 0, inputData.length);
    }

    private static void quickSort(int[] inputData, int begin, int end) {
        if ((end - begin) < 2) {
            return;
        }
        int m = partition(inputData, begin, end, (end + begin) / 2);
        quickSort(inputData, begin, m);
        quickSort(inputData, m + 1, end); // +1 for convergence
    }

    // index of first element >= v
    private static int lowerBound(int[] data, int v) {
        return lowerBound(data, v, 0, data.length);
    }

    private static int lowerBound(int[] data, int v, int begin, int end) {
        if (begin == end) {
            return begin;
        }
        int m = (begin + end) / 2;
        return data[m] < v ? lowerBound(data, v, m + 1, end) : lowerBound(data, v,
begin, m);
    }

    private static int binarySearch(int[] inputData, int val) {
        for (int i = 0; i != inputData.length; ++i) {
            inputData[i] = 2 * i;
        }
        return lowerBound(inputData, val);
    }

    private XYSeries[] getTimeSeries()
    {
      /* You need to customize this function, this
       * exception is just a reminder to do so
       */
//        if (true)
//        {
//            throw new UnsupportedOperationException("getTimeSeries()");
//        }

        ThreadMXBean bean = ManagementFactory.getThreadMXBean( );

      /*
       * Set this array to the accurate length for your series
       */
//        XYSeries[] res = new XYSeries[]{
//          new XYSeries( "UserTime" ),
//          new XYSeries( "WallTime" )
//    };
        XYSeries[] res = new XYSeries[]{
                new XYSeries( "Selection Sort" ),
                new XYSeries( "Bubble Sort" ),
                new XYSeries( "Merge Sort" ),
                new XYSeries( "Quick Sort" ),
        };

//        for(int i = 500000; i <= 10000000 ; i += 100000)
//        for(int i = 5000; i <= 100000 ; i += 1000)
//        for(int i = 500; i <= 10000 ; i += 100)
//        for(int i = 50000; i <= 1000000 ; i += 10000)
//        for(int i = 50000; i <= 1000000 ; i += 10000)
        for (int n = 0; n < res.length; n++) {
            for (int i = 500; i <= 10000; i += 100) {
         /*
          * Here, do the setUp(), the part of your test
          * that need to be done before each calculus
          * but does count in the time mesured
          */
                int[] random_test_data = RandomData.generate1d(i, 0, 10000000);
                int find_num = RandomData.generate1d(1, 0, random_test_data.length -
1)[0];

         /*
          * This is two way to calculate the time spent
          * cpu time should be more accurate, but its precision
          * depends on your system (it can be nanoseconds)
          */
//             long beginWallClockTime = System.nanoTime();
                long beginCpuTime = bean.getCurrentThreadCpuTime();

         /*
          * Here, call the code you want to test
          */
//       findMinimumValue(random_test_data);
//             selectionSort(random_test_data);
//             bubbleSort(random_test_data);
//             mergeSort(random_test_data);
//             quickSort(random_test_data);
//             binarySearch(random_test_data, find_num);
                switch (n) {
                    case 0:
                        selectionSort(random_test_data);
                        break;
                    case 1:
                        bubbleSort(random_test_data);
                        break;
                    case 2:
                        mergeSort(random_test_data);
                        break;
                    case 3:
                        quickSort(random_test_data);
                        break;
                    default:
                        break;
                }

//             long wallClockDuration = System.nanoTime() - beginWallClockTime;
                long cpuTimeDuration = bean.getCurrentThreadCpuTime() - beginCpuTime;
          /*
           * Add these duration to your result the way you want
           */
//              res[0].add(i,wallClockDuration);
                res[n].add(i, cpuTimeDuration);

          /*
           * Forcing GC to run, so it reduce the chance of it
           * running during your time measurement phase
           */
                System.gc();
                System.out.println(i);
            }
        }
        return res;
    }

    /* This function goal is just to make sure everything
    * in Java and the virtual machine is loaded correctly
    * before actually measuring time.
    *
    * Otherwise, first time measurement can be more important
    * than the real algorithm execution time.
    *
    * Just run your algorithm for about a second of time
    */
    private void warmUp()
    {
//          for (int i = 0; i < 10000; ++i)
//          for (int i = 0; i < 100; ++i)
//          for (int i = 0; i < 100; ++i)
//          for (int i = 0; i < 10000; ++i)
        for (int i = 0; i < 10; ++i)
        {
            int[] random_warmup_data = RandomData.generate1d(5000, 0, 10000000);
            int find_num = RandomData.generate1d(1, 0, random_warmup_data.length -
1)[0];

//             findMinimumValue(random_warmup_data);
//             selectionSort(random_warmup_data);
//              bubbleSort(random_warmup_data);
//              mergeSort(random_warmup_data);
//              quickSort(random_warmup_data);
//              binarySearch(random_warmup_data, find_num);
            selectionSort(random_warmup_data);
            bubbleSort(random_warmup_data);
            mergeSort(random_warmup_data);
            quickSort(random_warmup_data);
        }
    }

    private XYDataset createDataset( )
    {
        warmUp();
        final XYSeriesCollection dataset = new XYSeriesCollection( );

      /*
```

```
        * You might want to add a loop here
        */
         XYSeries[] series = getTimeSeries();


         for(XYSeries serie : series )
         {
             dataset.addSeries(serie);
         }


         return dataset;
     }

     private static void draw_graph() {
         ThreadMXBean bean = ManagementFactory.getThreadMXBean( );
         TimeMeasurementTargets chart = new TimeMeasurementTargets(
//                  "Time Measurement", "Time usage of finding minimum of an array"
//                  "Time Measurement", "Time usage of selection sort"
//                  "Time Measurement", "Time usage of bubble sort"
//                  "Time Measurement", "Time usage of merge sort"
//                  "Time Measurement", "Time usage of quick sort"
//                  "Time Measurement", "Time usage of binary search"
                 "Time Measurement", "Comparision of sorting algorithm"
         );
         chart.pack( );
         RefineryUtilities.centerFrameOnScreen( chart );
         chart.setVisible( true );
     }

     /*
     * The constructor is actually doing most of the work here,
     * but you don't need to change it
     */
     private TimeMeasurementTargets(String applicationTitle, String chartTitle)
     {
         super(applicationTitle);
         JFreeChart xylineChart = ChartFactory.createScatterPlot(
                 chartTitle ,
                 "Array Size" ,
                 "Time (ns)" ,
                 createDataset() ,
                 PlotOrientation.VERTICAL ,
                 true , false , false);

         Shape cross = ShapeUtilities.createDiagonalCross(3, 1);

         ChartPanel chartPanel = new ChartPanel( xylineChart );
         chartPanel.setPreferredSize( new java.awt.Dimension( 560 , 367 ) );
         final XYPlot plot = xylineChart.getXYPlot( );

       /*
        * OPTIONNAL :
        *
        * If you want to customize the series layout, but the program does it
        * quite well by itself
        */
        /*XYItemRenderer renderer = plot.getRenderer( );
        renderer.setSeriesPaint( 2 , Color.RED );
        renderer.setSeriesPaint( 1 , Color.GREEN );
        renderer.setSeriesPaint( 0 , Color.YELLOW );
        renderer.setSeriesShape(0, cross);

        plot.setDomainCrosshairVisible(true);
        plot.setRangeCrosshairVisible(true);*/

         setContentPane( chartPanel );
    }

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