**Assignment 6: Analysis of QuickSort Pivots**

Proposed experimental procedure:

Firstly, using Assignment 5’s code I will record some run times of the quicksort code without any modification made to how it chooses its pivot.

Secondly implement a few lines of code to compare the values of the left, right and middle array points.

Using this new implementation of quicksort, run the code again and take some run times.

I hope to find that on the new implementation there is a substantial decrease in time for the code to run as it will split the array to be sorted more evenly.

A potential problem I might run into is that on trying to compare the strings I am unable to code it in an efficient manner.

Code for Running Experiments:

Object pivotValue = **null**;

// Set the pivot to be the best element

**if**(c.compare(s[leftBound],s[rightBound])<0 &&

c.compare(s[leftBound],s[rightBound/2])<0 && //quick compare between the left, right and middle values

c.compare(s[rightBound],s[rightBound/2])<0) //if array sorted, picks the middle value

pivotValue = s[rightBound];

**else** **if**(c.compare(s[leftBound],s[rightBound])<0 &&

c.compare(s[leftBound],s[rightBound/2])<0 &&

c.compare(s[rightBound/2],s[rightBound])<0)

pivotValue = s[rightBound/2];

**else**

pivotValue = s[leftBound];

//pivotValue = s[rightBound];

**private** **static** String[] randomArray(**int** s) {

String[] arr = **new** String[s];

**for**(**int** i = 0; i<s; i++) {

String rand = "" + (**int**)(Math.*random*()\*500);

arr[i] = rand;

}

**return** arr;

}

**public** **static** **void** main(String args[]) {

**double**[] Quick = **new** **double**[9];

String print = JOptionPane.*showInputDialog*("Size of Array to sort:");

**int** s = Integer.*parseInt*(print);

String[] arr = *randomArray*(s);

String[] arr1 = *randomArray*(s); // arrays of same size to test

String[] arr2 = *randomArray*(s);

System.***out***.println("");

System.***out***.println("Compares || Moves || Time");

// we can also use insertionSort directly

String[] copy1a = arr.clone();

String[] copy1b = arr1.clone(); // clone arrays for use

String[] copy1c = arr2.clone();

*moveCount* = 0;

*compareCount* = 0;

String[] copy3a = arr.clone();

String[] copy3b = arr1.clone(); // clone arrays for use

String[] copy3c = arr2.clone();

**for**(**int** i = 0; i<3; i++) {

*time* = System.*nanoTime*();

*compareCount* = 0;

*moveCount* = 0;

**if**(i == 0) {

*quickSort*(copy3a, **new** StringComparator());

}

**if**(i == 1) {

*quickSort*(copy3a, **new** StringComparator()); // go through each sort

}

**if**(i == 2) {

*quickSort*(copy3a, **new** StringComparator());

}

*time* = System.*nanoTime*() - *time*;

**switch** (i) {

**case** 0 :Quick[0] = *compareCount*;

Quick[1] = *moveCount*;

Quick[2] = *time*;

**case** 1 :Quick[3] = *compareCount*; // record details

Quick[4] = *moveCount*;

Quick[5] = *time*;

**case** 2 :Quick[6] = *compareCount*;

Quick[7] = *moveCount*;

Quick[8] = *time*;

}

}

System.***out***.println("" + *array2String*(Quick)); // print out numbers

}

Details of Experiments and Results:

I ran the code for each implementation 3 times and documented the results for the time.

My code is designed, on one run, to sort the array and then try and sort the sorted array again another 2 times. For the purpose of this experiment, I felt it was sufficient to just include the 2 attempts at resorting for each run.

For each run I used a different array size.

|  |  |  |
| --- | --- | --- |
| Old Implementation/s | 1st re-sort | 2nd re-sort |
| 1st run(1000) | 0.00699 | 0.0105 |
| 2nd run(5000) | 0.3912 | 0.1465 |
| 3rd run(10000) | 0.821 | 0.642 |

|  |  |  |
| --- | --- | --- |
| New Implementation/s | 1st re-sort | 2nd re-sort |
| 1st run(1000) | 0.00357 | 0.00209 |
| 2nd run(5000) | 0.1336 | 0.0881 |
| 3rd run(10000) | 0.421 | 0.407 |

One quick note before I go onto the conclusion, it seems that I broke the code and haven’t been able to find a work around for it. I noticed that and high numbers, e.g. 25000 would incur a stack overflow so that’s why I cheated and kept the values low for experimenting, although I still think you can pull conclusions from the data.

Final Conclusions and Comparison with Theory:

First off then the new implementation is faster and that correlates with the theory. Basically, splitting the array into even halves all the way down keeps the optimal complexity for quicksort on already sorted arrays.

My implementation is more than likely causing problems from what I can tell looking at the data. The first re-sort is always a step above the second although I have no idea why. There is obviously something slowing it down the first time but nothing comes to mind.

We can then conclude that this is a better implementation of quicksort for a specific circumstance, already sorted array, and can conclude with experimental data that using quicksort with optimal pivots will skew us closer to the best complexity.