Sairam Soundararajan

Prof. Didulo

CMSC451

12-9-22

Project 2: Reflection of Project 1

When working on Project 1, my main goal was to implement two versions of the Quick Sort algorithm: one being an iterative solution and the other being recursive solution. The main function of quick sort is to divide a large data array into smaller sets based on the selected element p as the pivot of the array. The pivot can either be the first, last, median, or random element of the array. In most cases, the last element of the unsorted array is typically chosen as the pivot. Given an array and its pivot, the aim for partition is to place the pivot at the correct position in a sorted array by putting all smaller values (smaller than p) before the pivot while placing all greater values (greater than p) after the pivot. After choosing the pivot, all elements smaller than the pivot get shifted to the left of the pivot, while all elements greater than the pivot will be shifted to the right of the pivot. All values on either partition of the pivot are now subarrays. The same rule that was applied on the original array will recursively be applied on both subarrays, where each subarray will have its own pivot and continue to have subarrays. This process will continue until the last iteration, where there is only one element in each subarray. Afterwards, we will have a sorted array in ascending numerical order. The following pseudocode for the QuickSort function and partition are provided below:

**Pseudo Code for QuickSort algorithm:**

*/\* low  –> Starting counter,  high  –> Ending counter \*/*

*quickSort(array[], low, high) {*

*if (low < high) {*

*/\* pivot is partitioning counter, array[pivot] is now at right place \*/*

*pivot = partition(array, low, high);*

*quickSort(array, low, pivot – 1);  // Before pivot*

*quickSort(array, pivot + 1, high); // After pivot*

*}*

*}*

**Pseudo code for Partition:**

*// takes last element as pivot*

*// places the pivot element at its correct position in sorted array*

*// places all smaller elements (smaller than pivot) to left of pivot and all greater elements to right of pivot \*/*

*partition (array[], low, high)  
{  
    // pivot (Element to be placed at right position)  
    pivot = array[high];*

*counter = (low – 1)  // Index of smaller element and indicates the   
    // right position of pivot found so far*

*for (count = low; count <= high- 1; count++){*

*// If current element is smaller than the pivot  
        if (array[count] < pivot){  
            counter++;    // increment index of smaller element  
            swap array[counter] and array[count]  
        }  
    }  
    swap array[counter + 1] and array[high])  
    return (counter + 1)  
}*

The following code provided below is what I wrote for the iterative and recursive function of QuickSort:

Function to Recursively Sort the List

public int[] recursiveSort(int[] array, int low, int high) throws UnsortedException {

counter++;

timeStart = System.nanoTime();

int I = partition(array, low, high);

if(low < I – 1) {

recursiveSort(array, low, I – 1);

}

if(high > I + 1) {

recursiveSort(array, I + 1, high);

}

timeEnd = System.nanoTime();

return array;

}

Function to Iteratively Sort the List:

public int[] iterativeSort(int[] array, int low, int high) throws UnsortedException {

counter++;

timeStart = System.nanoTime();

int stack[] = new int[high - low + 1];

int top = -1;

stack[++top] = low;

stack[++top] = high;

while (top >= 0) {

high = stack[top--];

low = stack[top--];

int pivot = partition(array, low, high);

if(low < pivot - 1) {

stack[++top] = low;

stack[++top] = pivot - 1;

}

if(pivot + 1 < high) {

stack[++top] = pivot + 1;

stack[++top] = high;

}

}

timeEnd = System.nanoTime();

return array;

}

Through examining my iterative and recursive methods, I have come to the following conclusions. The time complexity for the iterative quicksort is O(n(log(n))) because it contains a while loop. For the recursive quicksort, the time complexity is O(n2) for the reason that it contains two recursive calls. Referring to the initial time complexities of quicksort, the Big-O notation for both best-case and average-case is O(n(log(n))), while it is O(n2) for the worst-case.

Before finalizing the benchmarking class, I started warming up the JVM by creating “dummy” classes to test before proceeding forward with the program. At the start of the program, I called a function which warms up the JVM. This function creates a thousand instances of a “dummy” class.

For the critical operations, both iterative and recursive algorithms stay fairly the same value. When running the recursive function, we count how many times the function calls itself. For the iterative function, we count how many iterations there are of the while loop.

The following graph show the critical operations for both iterative and recursive quicksort. Notice how each quicksort remains relatively the same.

The following graph shows the execution times of both algorithms. Notice that while the execution time for recursive remains consistent, it varies drastically for the iterative.

When comparing the execution times and critical operations of both algorithms, we see that the recursive quicksort stays consistent throughout the data set sizes for both execution time and critical operation count. The iterative quicksort algorithm, however, considerably differentiates in both execution time and critical operation count. While iterative stays consistent in critical operation count, its execution times fluctuates substantially.

For the iterative sort’s critical operations, the coefficient of variance remains stays consistently at 57%. However, it fluctuates between 43% and 60% for the execution time. For the recursive sort’s critical operations, the coefficient of variance also remains at 57%, while ranging from 52% to 69% for the execution time. Even with 10 data sets from 100 to 1000, there was not a lot of variance in the coefficient. Aside from the iterative sort’s execution time, the coefficient of variance has otherwise stayed relatively constant.

The results of the sorting algorithms do reflect the Big-Oh notations I made earlier about iterative quicksort and recursive quicksort. The iterative sort runs faster than the recursive sort.

Overall, quicksort is an efficient algorithm that can be used for smaller number of data sets, whether it is 10 or 15 elements. However, for a large number of elements, the process can be quite lengthy and less efficient. The most important aspect of using the quicksort algorithm is choosing an element to be the pivot of the array. That way, the sorting algorithm can perform the partition to compare the value of the pivot to all the other elements of the array, thus leading to a sorted array. The feedback I received regarding this project is that the program did not calculate the coefficient of variance of the critical operation counts and cannot be same across all computations.