Comparing Selection and Quicksort Algorithms Times

Lab # 5

By

Kamila Jusino

CS 303 Algorithm and Data Structure

October 11, 2024

1. Problem Specification

The goal for this assignment was to implement selection sort and quicksort algorithms. In total we had to create three different working algorithms because for quicksort a simple version was made, as well as “RandomizedQuickSort” which used a random pivot for partioning. All three algorithms were compared with the provided text file ("input_16.txt","input_32.txt", etc... and "input_Random.txt","input_ReversedSorted.txt","input_sorted.txt".) The code outputs the sorted and unsorted version of the arrays before and after each implementation of the algorithm for input_16.txt. Beyond that the runtime is shown for each text file the algorithms are ran through. All the outputs were evaluated and compared with the results in time complexity analysis inside the lab report.

2. Program Design

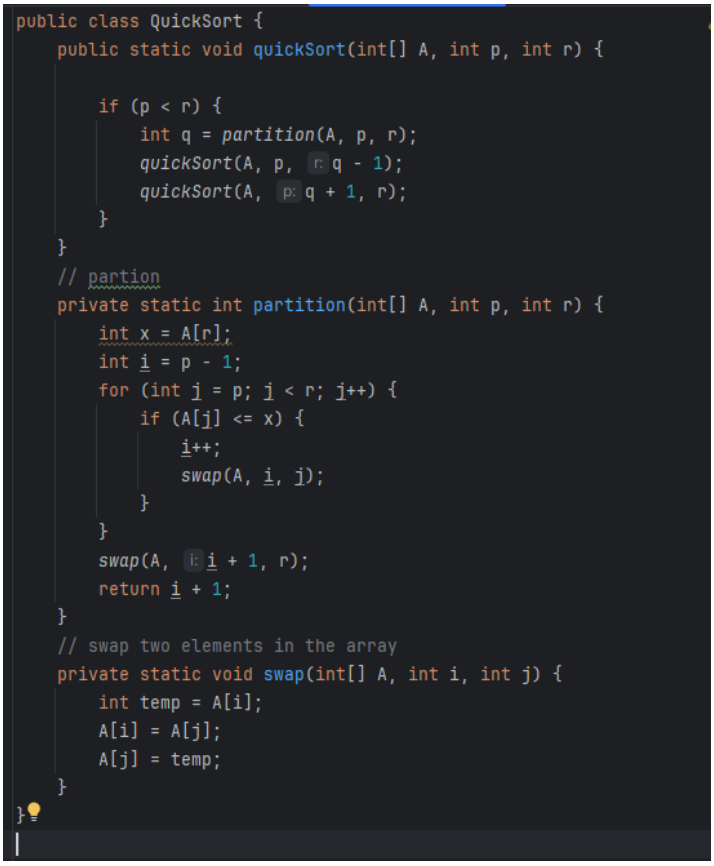
- The first step for this assignment was to create SelectionSort. This code follows the pseudocode provided. In selection sort there are two subarrays. One is sorted and one is unsorted. In this snippet it can be shown that an iteration of the array occurs where it is repeatedly looking for the smallest element to swap with the first element of that portion. This iteration continues until the array is sorted. for (int I = 0; I < N; I++) picks a place in the array. for (int J = I + 1; J < N; J++) searches through the unsorted subarray for the smallest element and swaps the elements with the place picked by the first for loop.

```
public class SelectionSort {
    public static void selectionSort(int[] A) {
        int N = A.length;

        for (int I = 0; I < N; I++) {
            int smallSub = I;
            for (int J = I + 1; J < N; J++) {
                if (A[J] < A[smallSub]) {
                    smallSub = J;
                }
            }
            // Swap
            int temp = A[I];
            A[I] = A[smallSub];
            A[smallSub] = temp;
        }
    }
}
```

- The Second Step was to create the quicksort method. QuickSort uses divide and conquer, making it a recursive algorithm. It is split into 3 steps. Choosing a pivot value,

partitioning, and sorting both parts. This method sorts of subarray and partions the array with the chosen pivot. While partioning changes the elements based on what needs to go to its left and right. Finally, the swap method is a helper method that rearranges elements during partioning.

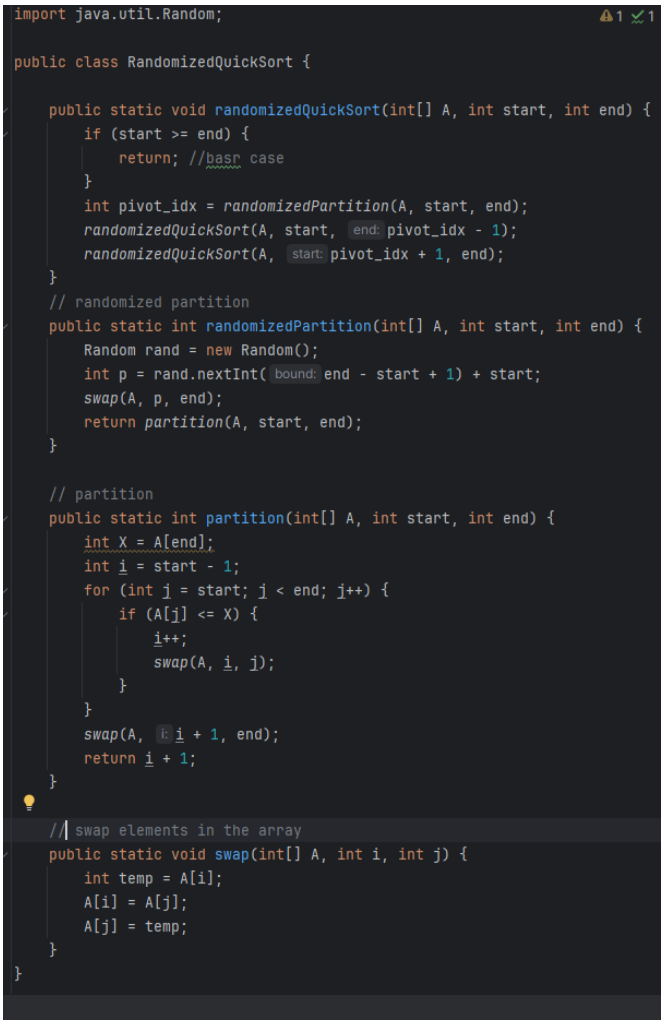


```
public class QuickSort {
    public static void quickSort(int[] A, int p, int r) {

        if (p < r) {
            int q = partition(A, p, r);
            quickSort(A, p, q - 1);
            quickSort(A, q + 1, r);
        }
    }
    // partion
    private static int partition(int[] A, int p, int r) {
        int x = A[r];
        int i = p - 1;
        for (int j = p; j < r; j++) {
            if (A[j] <= x) {
                i++;
                swap(A, i, j);
            }
        }
        swap(A, i + 1, r);
        return i + 1;
    }
    // swap two elements in the array
    private static void swap(int[] A, int i, int j) {
        int temp = A[i];
        A[i] = A[j];
        A[j] = temp;
    }
}
```

- The Third step was to create RandomizedQuickSort class. This is where the quick sort algorithm is bettered by using a randomized pivot for partioning. It functions of regular quick sort remain like it recursiveness. This code starts with a base case check. It then uses the method to randomly select a pivot. Later it changes with the last element before beginning the partioning method. Once that method is called, elements are changed to

maintain what needs to be on the left and what needs to be on the right. The swap method is again used to change elements.

- 

```java
import java.util.Random;

public class RandomizedQuickSort {

    public static void randomizedQuickSort(int[] A, int start, int end) {
        if (start >= end) {
            return; //basr case
        }
        int pivot_idx = randomizedPartition(A, start, end);
        randomizedQuickSort(A, start, end: pivot_idx - 1);
        randomizedQuickSort(A, start: pivot_idx + 1, end);
    }
    // randomized partition
    public static int randomizedPartition(int[] A, int start, int end) {
        Random rand = new Random();
        int p = rand.nextInt(bound: end - start + 1) + start;
        swap(A, p, end);
        return partition(A, start, end);
    }

    // partition
    public static int partition(int[] A, int start, int end) {
        int X = A[end];
        int i = start - 1;
        for (int j = start; j < end; j++) {
            if (A[j] <= X) {
                i++;
                swap(A, i, j);
            }
        }
        swap(A, i: i + 1, end);
        return i + 1;
    }

    // swap elements in the array
    public static void swap(int[] A, int i, int j) {
        int temp = A[i];
        A[i] = A[j];
        A[j] = temp;
    }
}
```

The next step was to create my main function:

```java
public static void main(String[] args) throws IOException {
    String[] fileNames = {
            "input_16.txt",
            "input_32.txt",
            "input_64.txt",
            "input_128.txt",
            "input_256.txt",
            "input_512.txt",
            "input_1024.txt",
            "input_2048.txt",
            "input_4096.txt",
            "input_8192.txt",
            "input_Random.txt",
            "input_ReversedSorted.txt",
            "input_Sorted.txt"
    };

    for (String fileName : fileNames) {
        System.out.println("\n============================");
        System.out.println("File: " + fileName);
        System.out.println("==============================");
        int[] originalArray = readInputFile(fileName);

        long startTime, endTime;
```

- This snippet of the code shows an array of file names containing the names of the text files provided to us.
    - A for loop iterates over each file and prints into the console for more concise data viewing. (Show array size by title name)
- ReadInputFile(fileName) reads the file and returns an array of int called original array

```
// Basic Quick Sort                                                    ⚠1 ^
int[] quickSortArray = copyArray(originalArray);
if (fileName.equals("input_16.txt")) {
    System.out.println("\n--- Unsorted Array before QuickSort ---");
    System.out.println(Arrays.toString(quickSortArray));
}
startTime = System.nanoTime();
QuickSort.quickSort(quickSortArray, p: 0, r: quickSortArray.length - 1);
endTime = System.nanoTime();
System.out.println(" QuickSort Time: " + (endTime - startTime) + " ns");
if (fileName.equals("input_16.txt")) {
    System.out.println("Array after QuickSort:");
    System.out.println(Arrays.toString(quickSortArray));
}

// Randomized Quick Sort
int[] randomizedQuickSortArray = copyArray(originalArray);
if (fileName.equals("input_16.txt")) {
    System.out.println("\n--- Unsorted Array before RandomizedQuickSort ---");
    System.out.println(Arrays.toString(randomizedQuickSortArray));
}
startTime = System.nanoTime();
RandomizedQuickSort.randomizedQuickSort(randomizedQuickSortArray, start: 0, end: randomizedQuick
endTime = System.nanoTime();
System.out.println("RandomizedQuickSort Time: " + (endTime - startTime) + " ns");
if (fileName.equals("input_16.txt")) {
    System.out.println("Array after RandomizedQuickSort:");
    System.out.println(Arrays.toString(randomizedQuickSortArray));
}
```

```
    // Selection Sort
    int[] selectionSortArray = copyArray(originalArray);
    if (fileName.equals("input_16.txt")) {
        System.out.println("\n--- Unsorted Array before SelectionSort ---");
        System.out.println(Arrays.toString(selectionSortArray));
    }
    startTime = System.nanoTime();
    SelectionSort.selectionSort(selectionSortArray);
    endTime = System.nanoTime();
    System.out.println("Selection Sort Time: " + (endTime - startTime) + " ns");
    if (fileName.equals("input_16.txt")) {
        System.out.println("Array after SelectionSort:");
        System.out.println(Arrays.toString(selectionSortArray));
    }

    System.out.println();
}
```

-
- This screenshot shows the call of each sort algorithm
    - QuickSort:
        - A new array is created
        - Time starts being recorded
        - Method is called to sort
        - After sorting end time is recorded. Then start and end time are substracted

        - Time is printed in nanoseconds
    - RandomizedQuickSort:
        - A copy of the orginal array is made
        - The start time is recorded
        - Method sort is called to sort
        - Times are subtracted
        - Time is printed in nanosecond
    - Selection:
        - Another copy is made of the orginal array
        - Start time is recorded
        - Times are subtracted
        - Method is called to sort
        - Time is printed in nanoseconds
- Print statements made to ensure the console showed the sorted array after the algorithm was called

```java
// Reads the input file and returns an array of integers
public static int[] readInputFile(String fileName) throws IOException {
    InputStream is = Main.class.getResourceAsStream( name: "/" + fileName); // Ensure files ar
    List<Integer> keysList = new ArrayList<>();

    try (BufferedReader br = new BufferedReader(new InputStreamReader(is))) {
        String line;
        while ((line = br.readLine()) != null) {
            String[] numbers = line.split( regex: "\\s+");
            for (String number : numbers) {
                if (!number.trim().isEmpty()) {
                    keysList.add(Integer.parseInt(number.trim()));
                }
            }
        }
    }

    // Convert List<Integer> to int[]
    int[] keysArray = new int[keysList.size()];
    for (int i = 0; i < keysList.size(); i++) {
        keysArray[i] = keysList.get(i);
    }
    return keysArray;
}

public static int[] copyArray(int[] original) {
    int[] copy = new int[original.length];
    System.arraycopy(original, srcPos: 0, copy, destPos: 0, original.length);
    return copy;
}
```

- Method readInputFile reads file containing integers separated by white space and returns the integers found as an array of integers
- Method copyArray(int[] orginal) create a copy of the given integer array

3. Test Cases

I used the file containing 16 elements for my testing phase. Unlike the previous assignments this was a good starting point for figuring out and coding out the algorithms. Issue the pseudocode this assignment was because I felt that the pseudocode required unnecessary parts and could have been optimized but regardless the assignment was to follow the pseudocode and that was accomplished by my code successfully. Another error that I ran across was having to redo how files were read in because unlike previous assignments they were separated by white space rather than commas. Asides from that once the algorithms were working correctly and the results were analyzed to make sure the times looked right the other files were passed through. When an error occurred, it was important to debug and decrease the data size again.

4. Analysis and Conclusions

As can be seen in the following table and screenshot. Overall, RandomizedQuickSort performed the best once it got the larger datasets. Selection sort however worked best for smaller datasets because it is simpler in nature and provides a smaller overhead. Even though they have the same time complexity because every dataset and occurrence are different at times the regular Quicksort did contain the better time. However, in general the algorithms did perform as mentioned above.

File	QuickSort (ns)	RandomizedQuickSort (ns)	SelectionSort (ns)
input_16.txt	1000000	2155600	1029700
input_32.txt	16300	44900	15500
Input_64.txt	27400	74200	35500
Input_128.txt	78900	143300	121900
Input_256.txt	136300	356100	508200
Input_512.txt	197200	522800	1391100
Input_1024.txt	142000	259200	1525700
Input_2048.txt	316500	578400	6447100
Input_4096.txt	513100	1080400	15478400
Input_8192.txt	816100	1561400	43136500

Input_Random.tx	82300	173800	736400
Input_ReversedSorted.txt	672500	158700	1278400
Input_Sorted.txt	1306000	153400	909700

In conclusion, for the material studied in class we know that quicksort has a best and average time complexity of O(n*logn) and a worst of O(n^2). Although it holds a slow worse case, it is efficiency in average case is a good option for sorting because unlike other sorting methods it sorts in place. Another factor to consider is that it does well with large data sets because of how the data grows. Quick sort uses divide and conquer meaning it is a recursive algorithm. The Randomizes quicksort holds the same time complexities but yet it is more effective because it chooses a random pivot. In doing so, there is a more balanced partion on average and actually reduces the chances of getting to a wort case time. This means that RandomizedQuick sort does indeed outperform quicksort. To reiterate, randomizing still holds a best and average time complexity of O(n*logn) and a worst of O(n^2). As can be seen in my results RandomizedQuick performs better on a more “diverse dataset” because of its ability to randomize the pivot leading to better partioning. In contrast, Selection sort is simple. Its simplicity is the reason it does better on smaller datasets like other linear sorting algorithms. It holds the time complexity of O(n^2) for all cases. In comparison to insertion as seen in other assignments it would actually do worse

because of this complexity. Insertion holds the best time complexity of O(n) and worse of O(n^2).

```
--- Unsorted Array before QuickSort ---
[8, 12, 5, 7, 9, 14, 2, 15, 2, 8, 9, 9, 8, 3, 8, 7]
 QuickSort Time: 1000000 ns
Array after QuickSort:
[2, 2, 3, 5, 7, 7, 8, 8, 8, 8, 9, 9, 9, 12, 14, 15]

--- Unsorted Array before RandomizedQuickSort ---
[8, 12, 5, 7, 9, 14, 2, 15, 2, 8, 9, 9, 8, 3, 8, 7]
RandomizedQuickSort Time: 2155600 ns
Array after RandomizedQuickSort:
[2, 2, 3, 5, 7, 7, 8, 8, 8, 8, 9, 9, 9, 12, 14, 15]

--- Unsorted Array before SelectionSort ---
[8, 12, 5, 7, 9, 14, 2, 15, 2, 8, 9, 9, 8, 3, 8, 7]
Selection Sort Time: 1029700 ns
Array after SelectionSort:
[2, 2, 3, 5, 7, 7, 8, 8, 8, 8, 9, 9, 9, 12, 14, 15]


==========================
File: input_32.txt
==============================
 QuickSort Time: 16300 ns
RandomizedQuickSort Time: 44900 ns
Selection Sort Time: 15500 ns


==========================
File: input_64.txt
==============================
 QuickSort Time: 27400 ns
RandomizedQuickSort Time: 74200 ns
Selection Sort Time: 35500 ns
```

```
File: input_128.txt
 ==============================
  QuickSort Time: 78900 ns
 RandomizedQuickSort Time: 143300 ns
 Selection Sort Time: 121900 ns


 ===========================
 File: input_256.txt
 ==============================
  QuickSort Time: 136300 ns
 RandomizedQuickSort Time: 356100 ns
 Selection Sort Time: 508200 ns


 ===========================
 File: input_512.txt
 ==============================
  QuickSort Time: 197200 ns
 RandomizedQuickSort Time: 522800 ns
 Selection Sort Time: 1391100 ns


 ===========================
 File: input_1024.txt
 ==============================
  QuickSort Time: 142000 ns
 RandomizedQuickSort Time: 259200 ns
 Selection Sort Time: 1525700 ns
```

```
File: input_2048.txt
==============================
 QuickSort Time: 316500 ns
RandomizedQuickSort Time: 578400 ns
Selection Sort Time: 6447100 ns


==========================
File: input_4096.txt
==============================
 QuickSort Time: 513100 ns
RandomizedQuickSort Time: 1080400 ns
Selection Sort Time: 15478400 ns


==========================
File: input_8192.txt
==============================
 QuickSort Time: 816100 ns
RandomizedQuickSort Time: 1561400 ns
Selection Sort Time: 43136500 ns


==========================
File: input_Random.txt
==============================
 QuickSort Time: 82300 ns
RandomizedQuickSort Time: 173800 ns
Selection Sort Time: 736400 ns
```

```
==========================
File: input_Random.txt
==============================
 QuickSort Time: 82300 ns
RandomizedQuickSort Time: 173800 ns
Selection Sort Time: 736400 ns


==========================
File: input_ReversedSorted.txt
==============================
 QuickSort Time: 672500 ns
RandomizedQuickSort Time: 158700 ns
Selection Sort Time: 1278400 ns


==========================
File: input_Sorted.txt
==============================
 QuickSort Time: 1306000 ns
RandomizedQuickSort Time: 153400 ns
Selection Sort Time: 909700 ns


Process finished with exit code 0
```

5. References

Array Help Credit:

GeeksforGeeks

https://www.geeksforgeeks.org/arrays-in-java/

Professor Unan Slide Show

https://stackoverflow.com/questions/5785745/make-copy-of-an-array

https://www.youtube.com/watch?v=_86FMWNfGOc

https://chatgpt.com/share/66ee3ea9-5200-8011-97c5-fcdb0380596e

Running Time Help Credit:

StackOverFlow

https://stackoverflow.com/questions/5204051/how-to-calculate-the-running-time-of-my-program

BroCode

https://www.youtube.com/watch?v=dOYieTlItMM

Professor Chen's slide show

File Input Help Credit:

https://chatgpt.com/share/66ee3d6a-a7b8-8011-90b6-abe7e6c76210

GeeksforGeeks

https://www.geeksforgeeks.org/different-ways-reading-text-file-java/

StackOverFlow

https://stackoverflow.com/questions/69356108/accessing-a-txt-file-in-the-src-folder

Proffesor Unan Slide Show

Selection Sort Help Credit:

https://www.youtube.com/watch?v=g-PGLbMth_g&pp=ygUec2VsZWN0aW9uIHNvcnQgYW5pbWF0aW9uIHZpZGVv

https://www.youtube.com/watch?v=dsqsnngsoD8

https://www.geeksforgeeks.org/selection-sort-algorithm-2/

Professor Chen's slide show

Quick sort Help Credit:

https://www.youtube.com/watch?v=Hoixgm4-P4M&t=33s

https://www.youtube.com/watch?v=h8eyY7dIiN4&t=76s

https://www.geeksforgeeks.org/quick-sort-algorithm/

https://www.w3schools.com/dsa/dsa_algo_quicksort.php

Professor Chen's slide show