Jayden Younger

Ian Riley

CS-2003-01(24/SP): Fnd Algorithm Comp Appl

24 March 2024

CS-2003-Project 2-Lab report

Intro

In this lab, we experimented with 8 sorting algorithms (Bubble, Insertion, Selection, Quick, Merge, Tree, Heap, and Radix sort) to determine which algorithm had the fastest execution time and the least significant operations.

Setup

To test each algorithm, we set up 5 arrays with n number of elements generated through a seed. The method that was used to gather these 5 arrays was provided by our instructor:

```
public static int[] getBenchData(long seed, int n) {
        Random rng = new Random(seed);
                int[] data = new int[n];
                for (int i = 0; i < n; i++) {
                data[i] = rng.nextInt(Integer.MAX_VALUE);
                }
         return data;
 }
```

In this method, we have to input Seed and N, Seed can be numbers of my choosing but note that we **Do not use** the same seed for different values of n and should use the same seed across algorithms when generating data for the same test, and N is the number of elements that we want to generate. We generated 5 arrays with the following elements:

Arr1: array of a 100 elements
Arr2: array of a 1000 elements
Arr3: array of a 10000 elements
Arr4: array of a 100000 elements
Arr5: array of a 1000000 elements

Alongside our bench data, we had to time our algorithms, the way I did it was by utilizing the System.currentTimeMillis() like this:

```
long start time = System.currentTimeMillis();
int Sorting algorithm = Sorting method();
long stop time = System.currentTimeMillis();
Long elapsed time = stopTime - startTime;
```

Sorting algorithms

Bubble Sort

```
public static int bubbleSort(int[] arr, int length, int j, int length2) {
  int sigOps = 0;
 int n = arr.length;
    if (n <= 1) // Array of length 0 or 1 is already sorted
        return sigOps;

    boolean swapped;
    do {
        swapped = false;
        for (int i = 0; i < n - 1; i++) {
            if (arr[i] > arr[i+1]) {
                int temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
                swapped = true;
                sigOps++;
            }
        }
    } while (swapped);
    return sigOps;
}
```

The Bubble sort algorithm is our simplest algorithm, in this algorithm, we repeatedly swap elements that are out of order. What I have noticed with this algorithm is that it **can not** handle any large arrays.

Insertion Sort

```java
public static int insertionSort(int[] arr, int length, int k, int length2) {
    int sigOps = 0; // Variable to count significant operations
      for (int i = 1; i < length; i++) {
        int key = arr[i];
        int j = i - 1;
        sigOps++; // initializing key
        sigOps++; // initializing j

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
            sigOps += 2; // each comparison and array assignment
        }
        arr[j + 1] = key;
        sigOps++; // final array assignment
      }
    return sigOps;
}
```

In the insertion sort algorithm, we split the array into sorted and unsorted. Then values from the unsorted part are picked and placed into the sorted part. The problem with this algorithm based on the benchmarks is that it is an O(N^2) solution but does hold consistent times based on my benchmarks.

Selection Sort

```
public static int selectionSort(int[] arr, int length, int i, int length2) {
    int sigOps = 0; // Variable to count significant operations
    // Iterate through the array
    for (i = 0; i < length - 1; i++) {
        // Find the index of the minimum element in the unsorted part of the arra
        int minIndex = i;
        for (int j = i + 1; j < length; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
            sigOps++; // Count comparison operation
        }
        // Swap the found minimum element with the first element
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
        sigOps += 3; // Count swap operations
    }
    return sigOps; // Return the total significant operations
}
```

The algorithm works by picking either the smallest or largest element from an array and moving it to the sorted part of the array. This method of sorting an array still does not work well with large data sets.

Quick Sort

```java
public static int quickSort(int[] arr, int length2, int i, int length3) {
 int sigOps = 0;
        if (arr == null || arr.length <= 1)
            return sigOps;

        int length = arr.length;
        Stack<Integer> stack = new Stack<>();
        stack.push(item:0);
        stack.push(length - 1);

        while (!stack.isEmpty()) {
            int end = stack.pop();
            int start = stack.pop();

            if (start < end) {
                int pivotIndex = partition(arr, start, end);
                sigOps++;

                stack.push(start);
                stack.push(pivotIndex - 1);

                stack.push(pivotIndex + 1);
                stack.push(end);
            }
        }
        return sigOps;
    }
    private static int partition(int[] arr, int left, int right) {
      int pivot = arr[right];
      int i = left - 1;

      for (int j = left; j < right; j++) {
          if (arr[j] <= pivot) {
              i++;
              int temp = arr[i];
              arr[i] = arr[j];
              arr[j] = temp;
          }
      }

      int temp = arr[i + 1];
      arr[i + 1] = arr[right];
      arr[right] = temp;

      return i + 1;
  }
```

This was the worst algorithm to mess with and I had to resort to an iterative version of the quick-sort algorithm due to a lot of encounters with it causing a stack overflow. This algorithm yielded the most consistent significant operation with it being the number of operations minus 1.

Merge Sort

```java
public static int mergeSort(int[] arr, int length, int i, int length2) {
    // Check if the array is already sorted
    if (length <= 1) {
        return 0;
    }

    // Calculate the middle
    int middle = length / 2;

    // Create subarrays
    int[] leftArr = new int[middle];
    int[] rightArr = new int[length - middle];

    // Fill subarrays
    for (int j = 0; j < middle; j++) {
        leftArr[j] = arr[j];
    }
    for (int j = middle; j < length; j++) {
        rightArr[j - middle] = arr[j];
    }

    // Recursively sort left and right subarrays and count operations
    int opsLeft = mergeSort(leftArr, middle, i, length2);
    int opsRight = mergeSort(rightArr, length - middle, i, length2);

    // Merge the sorted subarrays and count operations
    int opsMerge = merge(arr, leftArr, rightArr, length, middle);

    // Return the total number of significant operations
    return opsLeft + opsRight + opsMerge;
}
```

```java
private static int merge(int[] arr, int[] leftArr, int[] rightArr, int length, int middle) {
    // Merge the two sorted subarrays
    int ops = 0;
    int i = 0, j = 0, k = 0;
    while (i < middle && j < length - middle) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k++] = leftArr[i++];
        } else {
            arr[k++] = rightArr[j++];
        }
        ops++; // Increment operation count
    }

    // Copy remaining elements
    while (i < middle) {
        arr[k++] = leftArr[i++];
        ops++; // Increment operation count
    }

    // Copy remaining elements
    while (j < length - middle) {
        arr[k++] = rightArr[j++];
        ops++; // Increment operation count
    }

    return ops;
}
```

The algorithm divides the array and sorts the 2 halves and then combines the 2 sorted halves. The algorithm did seem to efficiently sort an array in a reasonable amount of time but did require additional memory to store the merged sub-arrays during the sorting process.

Tree Sort

```
public static int treeSort(int[] arr, int length, int i, int length2) {
    int sigOps = 0; // Initialize sigOps
    // is already sorted
    if (length <= 1) {
        return sigOps; // No significant operations
    }

    // Recursively sort the left subarray
    sigOps += treeSort(arr, length2, i, length2 / 2);

    // Recursively sort the right subarray
    sigOps += treeSort(arr, length - length2, i + length2, (length - length2) / 2);

    // Merge the sorted subarrays
    sigOps += treemerge(arr, i, length2, length - length2, length2 / 2);

    return sigOps;
}

public static int treemerge(int[] arr, int start, int length1, int length2, int gap) {
    int sigOps = 0; // Initialize sigOps
    int[] merged = new int[length1 + length2];
    int i = 0;
    int j = length1;

    // Merge elements from both subarrays into the temporary array
    for (int k = 0; k < length1 + length2; k++) {
        if (i < length1 && (j >= length1 + length2 || arr[start + i] <= arr[start + j])) {
            merged[k] = arr[start + i];
            i++;
        } else {
            merged[k] = arr[start + j];
            j++;
        }
        sigOps++;
    }
    // Copy the merged array back into the original array
    for (int k = 0; k < length1 + length2; k++) {
        arr[start + k] = merged[k];
        sigOps++;
    }
    return sigOps; // Return total significant operations
}
```

In tree sort, we create a binary tree and perform in-order traversal on the tree to get the elements in sorted order. In my test, it did well with handling large arrays with the 100000 element array taking 54 milliseconds.

Heap Sort

```java
public static int heapSort(int[] arr, int length, int i, int length2) {
    // Build max heap
    for (int j = length / 2 - 1; j >= 0; j--) {
        sigOps += heapify(arr, length, j);
    }
    // Heap sort
    for (int j = length - 1; j > 0; j--) {
        // Swap root (maximum value) with last element
        int temp = arr[0];
        arr[0] = arr[j];
        arr[j] = temp;
        sigOps++;

        // Heapify root element
        sigOps += heapify(arr, j, i:0);
    }
    return sigOps;
}

public static int heapify(int[] arr, int length, int i) {
    int sigOps = 0;
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    // Check if left child is larger than root
    if (left < length && arr[left] > arr[largest]) {
        largest = left;
    }
    sigOps++;
    // Check if right child is larger than largest so far
    if (right < length && arr[right] > arr[largest]) {
        largest = right;
    }
    sigOps++;
    // If largest is not root
    if (largest != i) {
        // Swap root with largest
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        sigOps++;
        // Recursively heapify
        sigOps += heapify(arr, length, largest);
    }
    return sigOps;
}
```

We build a heap with the array and Swap, remove, and heapify the sorted part of the heap. The algorithm did well with handling large smaller arrays with it taking 0 milliseconds for an array of 100 and a 1000. This did come with the drawback of a lot of significant operations.

Radix Sort

```
public static int radixSort(int[] arr, int length, int i, int length2) {
  int sigOps = 0;
  // Find the maximum number to know the number of digits
  int max = getMax(arr, length);
  // Perform counting sort for every digit
  for (int exp = 1; max / exp > 0; exp *= 10) {
      sigOps += countingSort(arr, length, exp);
  }
  return sigOps;
}
// A utility function to get the maximum element from an array
private static int getMax(int[] arr, int length) {
  int max = arr[0];
  for (int i = 1; i < length; i++) {
      if (arr[i] > max) {
          max = arr[i];
      }
  }
  return max;
}
private static int countingSort(int[] arr, int length, int exp) {
  int[] output = new int[length];
  int[] count = new int[10];
  int sigOps = 0;
  // Store count of occurrences
  for (int i = 0; i < length; i++) {
      count[(arr[i] / exp) % 10]++;
      sigOps++;
  }
  // Change count so that count now contains actual position of this digit in output
  for (int i = 1; i < 10; i++) {
      count[i] += count[i - 1];
      sigOps++;
  }
  // Build the output array
  for (int i = length - 1; i >= 0; i--) {
      output[count[(arr[i] / exp) % 10] - 1] = arr[i];
      count[(arr[i] / exp) % 10]--;
      sigOps++;
  }
  // Copy the output array to arr[]
  for (int i = 0; i < length; i++) {
      arr[i] = output[i];
      sigOps++;
  }
  return sigOps;
}
```

For the final algorithm, we sort the array linearly by processing them digit by digit. So far Radix sort is a decently fast algorithm that does have a large amount of operations that are going on in the back ground.

Benchmark data

Bubble Sort		
Element amount	Significant operations	Execution time (milliseconds)
100	2595	0
1000	246051	5
10000	24849562	93
100000	1795432724	15218
1000000	760127204	1573534

Insertion Sort		
Element amount	Significant operations	Execution time (milliseconds)
100	297	0
1000	2997	0
10000	29997	0
100000	299997	1
1000000	2999997	1

Selection Sort		
Element amount	Significant operations	Execution time (milliseconds)
100	5247	0
1000	502497	1
10000	50024997	7
100000	705282701	704
1000000	1786293661	68309

Quick Sort		
Element amount	Significant operations	Execution time (milliseconds)
100	99	0
1000	999	2
10000	9999	12
100000	99999	779
1000000	999999	141184

Merge Sort		
Element amount	Significant operations	Execution time (milliseconds)
100	672	0
1000	9976	1
10000	133616	2
100000	1668928	5
1000000	19951424	46

Tree Sort		
Element amount	Significant operations	Execution time (milliseconds)
100	1544	0
1000	21952	1
10000	287232	1
100000	3537856	6
1000000	41902848	54

Heap Sort		
Element amount	Significant operations	Execution time (milliseconds)
100	2020	0
1000	30124	0
10000	405868	1
100000	5052511	8
1000000	60337207	67

Radix Sort		
Element amount	Significant operations	Execution time (milliseconds)
100	3708	1
1000	36108	1
10000	360108	3
100000	3600108	9
1000000	36000108	80

Works Cited

Gallagher, Patrick, and Sandeep Jain. “Radix Sort - Data Structures and Algorithms Tutorials.” *GeeksforGeeks*, 16 February 2024, https://www.geeksforgeeks.org/radix-sort/. Accessed 5 April 2024.

Gupta, Deepika, and Sandeep Jain. “Selection Sort – Data Structure and Algorithm Tutorials.” *GeeksforGeeks*, 5 January 2024, https://geeksforgeeks.org/selection-sort/. Accessed 5 April 2024.

Jain, Sandeep. “Heap Sort - Data Structures and Algorithms Tutorials.” *GeeksforGeeks*, 29 March 2024, https://geeksforgeeks.org/heap-sort/. Accessed 5 April 2024.

Jain, Sandeep. “Insertion Sort - Data Structure and Algorithm Tutorials.” *GeeksforGeeks*, 16 February 2024, https://www.geeksforgeeks.org/insertion-sort/. Accessed 5 April 2024.

Jain, Sandeep. “Merge Sort - Data Structure and Algorithms Tutorials.” *GeeksforGeeks*, 4 March 2024, https://geeksforgeeks.org/merge-sort/. Accessed 5 April 2024.

Jain, Sandeep. “QuickSort - Data Structure and Algorithm Tutorials.” *GeeksforGeeks*, 22 March 2024, https://www.geeksforgeeks.org/quick-sort/. Accessed 5 April 2024.

Jaiswal, Saurabh, and Sandeep Jain. “Tree Sort.” *GeeksforGeeks*, 11 September 2023, https://geeksforgeeks.org/tree-sort/. Accessed 5 April 2024.

Tiwari, Nikita, and Sandeep Jain. “Bubble Sort - Data Structure and Algorithm Tutorials.” *GeeksforGeeks*, 21 November 2023, https://www.geeksforgeeks.org/bubble-sort/. Accessed 5 April 2024.