CPSC 335

Project 2
Creation, Optimization and Performance Analysis of Selection Sort and Merge Sort Algorithms

Professor Wortman
Jesus Contreras

Selection Sort Code

```
//This program will test the execution time
//of several input values for the selection
//sort algorithm
import java.util.*;

public class selectionSort{
      public static void main (String[] args){

            Scanner scan = new Scanner(System.in);
            System.out.println("Please enter value of n :");
            int n = scan.nextInt();
            //create an array for selection sort to process
            int array[]  = new int[n];
            //currentTimeMillis returns current time in milliseconds
            double t1 = System.currentTimeMillis();
            //(pass the array, of size n) through selection_srt
            //algorithm
            selection_srt(array, n);
            double t2 = System.currentTimeMillis();
            //take the difference of current time and current time
            //before execution of selection_srt algo. The difference
            //of both times yields the execution time of the algo.
            double  total_time = ((t2-t1));
            System.out.println("Total execution time (in milli-
            seconds):" + total_time);

      }

      public static void selection_srt(int array[], int n){
            for(int x=0; x<n; x++){
                  int index_of_min = x;
                  for(int y=x; y<n; y++){
                        if(array[index_of_min]<array[y]){
                              index_of_min = y;
                        }
                  }
                  int temp = array[x];
                  array[x] = array[index_of_min];
                  array[index_of_min] = temp;

            }
      }
}
```

MergeSort Code

```
//This program will test the execution time
//of several input values for the merge
//sort algorithm


public class MergeSort {

	private int[] numbers;
	private int number;

	public void sort(int[] values) {
		this.numbers = values;
		number = values.length;

		mergesort(0, number - 1);
	}

	private void mergesort(int small, int big) {
		// Check if small is smaller than big (if not the case then
		//array is already sorted)
		if (small < big) {
			// finds the index of the "middle" element
			int middle = (small + big) / 2;
			// will sort left side of the array
			mergesort(small, middle);
			// will sort right side of the array
			mergesort(middle + 1, big);
			// combines the two sorted arrays
			merge(small, middle, big);
		}
	}

	private void merge(int small, int middle, int big) {


		int[] anotherArray = new int[number];

		// Copies both parts into anotherArray
		for (int i = small; i <= big; i++) {
			anotherArray[i] = numbers[i];
		}

		int i = small;
		int j = middle + 1;
		int k = small;
```

```
            /* copies smallest values from either the left or the right
            side back
             to the original array*/
            while (i <= middle && j <= big) {
                  if (anotherArray[i] <= anotherArray[j]) {
                        numbers[k] = anotherArray[i];
                        i++;
                  } else {
                        numbers[k] = anotherArray[j];
                        j++;
                  }
                  k++;
            }
            // copies the rest of the left side of the array into
            //target array
            while (i <= middle) {
                  numbers[k] = anotherArray[i];
                  k++;
                  i++;
            }
            anotherArray = null;

      }
}

import java.util.Scanner;

public class MergesortTest {
      public static void main (String[] args){

            final int[] numbers;


            Scanner scan = new Scanner(System.in);
            System.out.println("Please enter value of n :");
            int n = scan.nextInt();
            numbers  = new int[n];

            MergeSort sorter = new MergeSort();
            long t1 = System.currentTimeMillis();
            sorter.sort(numbers);
            long t2 = System.currentTimeMillis();
            long total_time = (t2 - t1);
            System.out.println("Total execution time (in milli-
            seconds):" + total_time);


      }
}
```

Data For Scatter Plots + Questions

Selection Sort:

n	time(mseconds)
25000	1422
50000	5838
75000	12551
100000	22709
125000	36926
150000	53941
175000	74354
200000	96837
225000	124722
250000	155842

MergeSort:

n	time(mseconds)
25000	1045
50000	4195
75000	9296
100000	15556
125000	25514
150000	37621
175000	51044
200000	65905
225000	82217
250000	101498

(Optimized) MergeSort:

n	time(mseconds)
25000	9
50000	16
75000	21
100000	26
125000	31
150000	37
175000	44
200000	51
225000	54
250000	60

Questions:

1. Theoretically it would be reasonable to assume MergeSort has a constant performance advantage over SelectionSort, however in practice for small values of n there was no real significant increase of performance. As the values of n increase the runtime execution of mergesort is faster than that of SelectionSort, thus its efficiency is more apparent.

2. By average there was about a 25% increase in performance by comparing the runtime(s) of MergeSort to that of Selection Sort, and a 50% increase of performance at our highest values of N (size of processed array). Implementing an optimal algorithm is certainly worth the extra effort. Note: The expected performances are O(n^2) and O(nlogn) for Selection Sort and MergeSort, respectively.

3. (3 and 4 are answered under 4)

4. Absolutely, by implementing a non-recursive MergeSort the performance was radically improved. This huge improvement in performance was a pleasant surprise as the runtime was nearly linear or just about O(n). In this case optimizing the algorithm was more work than coming up with the MergeSort itself. It was worth the effort since fine tuning MergeSort actually produced an extremely more efficient performance than just switching from Selection Sort to a baseline MergeSort.

MergeSort Code (Optimized)

```
public class MergeSort {

	//private int[] numbers;
	//private int number;

	//non-recursive mergeSort
	public static void mergesort(int[] array){
		if(array.length < 2){
			//If the array is a single element then logically
			there is nothing to sort
			//and therefore the array is already sorted.
			return;
		}
		//Size of subarrays, not stagnant.
		int sizer = 1;
		//begLeft, will be the beginning index for the left
		subarray
		//begRight, will be the beginning index for the right
		subarray
		int begLeft;
		int begRight;

		while(sizer < array.length){
			begLeft = 0;
			begRight = sizer;
			while(begRight + sizer <= array.length){
				merge(array, begLeft, begLeft + sizer,
				begRight, begRight + sizer);

				begLeft = begRight + sizer;
				begRight = begLeft + sizer;
			}

			if(begRight < array.length){
				merge(array, begLeft, begLeft + sizer,
				begRight, array.length);
			}
			sizer *= 2;
		}
	}
```

```
//Merge to sorted blocks
            public static void merge(int[] array, int begLeft, int
            endLeft, int begRight, int endRight){
            //establish arrays for merging
            int [] right = new int[endRight - begRight + 1];
            int[] left = new int[endLeft - begLeft + 1];

            //Copy elements to arrays for merging
            for(int i = 0, k = begRight; i < (right.length -1); ++i,
            ++k)

            right[i] = array[k];
            for(int i = 0, k = begLeft; i < (left.length -1); ++i, ++k)
                  left[i] = array[k];
            right[right.length-1] = Integer.MAX_VALUE;
            left[left.length-1] = Integer.MAX_VALUE;

            //Merge the two sorted arrays to the original one
            for(int k = begLeft, m = 0, n = 0; k < endRight; ++k) {
                  if(left[m] <= right[n]) {
                        array[k] = left[m];
                        m++;
                  }
                  else {
                        array[k] = right[n];
                        n++;
                  }

            }
      }
}
```