Alexander Laudino 2021-10-10

Dr. Farrett CSC-162-IN1

Lab Assignment 5 – Sorting Algorithms Reference Documents

Description of Sort Algorithms

1. **Insertion Sort –** The insertion-sort algorithm works by repeatedly inserting a new element into a sorted sublist until the entire list is sorted. To insert *list[i]* into *list[0..i-1]*, save *list[i]* into a temporary variable, *temp*. Move *list[i-1]* to *list[i]* if *list[i-1] > temp*, move *list[i-2]* to *list[i-1]* if *list[i-2] > temp*, and so on, until *list[i-k] <= temp* or *k > i*. Assign temp to *list[i-k+1]*. The method is implemented with a nested *for* loop. The outer loop (with loop control variable *i*) is iterated in order to obtain a sorted sublist, which ranges from *list[0]* to *list[i]*. The inner loop (controlled by variable *k*) inserts *list[i]* into the sublist from *list[0]* to *list[i-1]*.
2. **Bubble Sort –** The bubble sort algorithm works by sorting an array in several passes. On each pass, successive neighboring pairs are compared. If a pair is in decreasing order, its values are swapped; otherwise, the values remain unchanged. After the first pass, the last element becomes the largest in the array, with the second pass, the second-to-last element becomes the second largest element in the array. This continues until the entire array is sorted.
3. **Merge Sort –** The merge sort algorithm is a recursive algorithm that takes an array and divides it into two halves and then recursively applies the merge sort to each successive half. This process continues until each subarray contains only one element. The algorithm then merges the subarrays into larger sorted subarrays by comparing the values and adding the smaller of the values to a new temporary sorted list. This continues until only one array remains.
4. **Quick Sort –** The quick sort algorithm selects an element, called a pivot, in the array. It divides the array into two parts so that all elements in the first part are less than or equal to the pivot, and all elements in the second half are greater than the pivot. The selection of the pivot affects the performance of the algorithm, with the algorithm choosing one that divides the two parts evenly. The algorithm is then recursively applied to the first and second halves of the array. The algorithm is comprised of two overloaded methods, the first is used to sort the array, and the second is a helper method that sorts a subarray with a specific range, and a third method that partitions the array using the pivot.

Requirements

Requirements for SortingAlgorithms

Requirement	Priority
Modify insertion, bubble, merge, and quick sort algorithms to include count of each element swap	High
Four identical arrays of at least 20 integers	High
Method to generate array(s)	Medium
Method to copy arrays	Medium
Method to get array(s)	Medium
Four variables to hold number of swaps for each algorithm	High
Methods to get swap variables	Medium
Method to get user input	Low
Constructors to initialize and set arrays	Low
Method to display results	High

Requirements for DemoSortingAlgorithms

Requirement	Priority
Call each method using same array	High
Display the number of swaps made by each algorithm	High

Pseudo-code

This program performs the Insertion Sort, Bubble Sort, Merge Sort, and Quick Sort algorithms on an int array of at least 20 integers, such that each method keeps a count of the number of swaps it makes. This demo application tests each algorithm using four identical arrays and displays the number of exchanges made by each algorithm.

DemoSortingAlgorithms

Begin

1. Create new arraylist of SortAlgorithms objects as algos
2. Create new SortAlgorithms object using user input and add to algos
3. Display results for user input list
4. Initialize and set array of 20 integers as list
5. Create new SortAlgorithms object using list and add to algos
6. Display results for list
7. Create new SortAlgorithms object using empty constructor and add to algos
8. Display results for 20 integer list
9. Create new SortAlgorithms object using 30 as parameter for constructor and add to algos
10. Display results for 30 integer list
11. Create new SortAlgorithms object using 40 as parameter for constructor and add to algos
12. Display results for 40 integer list

End

The variables, constructors, and methods the program depends on.

SortingAlgorithms

// ----- Variables -----

Initialize scanner object for user input as INPUT

Initialize random object for generating integers as rand

Initialize arraylist of integers to hold original list as oList

Initialize arraylist of integers to hold insertion sort list as iList

Initialize arraylist of integers to hold bubble sort list as bList

Initialize array of integers to hold merge sort list as mList

Initialize array of integers to hold quick sort list as qList

Initialize insertion sort swap count as iSwaps to zero

Initialize bubble sort swap count as bSwaps to zero

Initialize merge sort swap count as mSwaps to zero

Initialize quick sort swap count as qSwaps to zero

// ----- Constructors -----

Begin SortAlgorithms()

1. generateLists()

End

Begin SortAlgorithms(n)

1. generateLists(n)

End

Begin SortAlgorithms(nList)

1. generateLists(nList)

End

Begin SortAlgorithms(uList)

1. generateLists(uList)

End

// ----- Methods -----

Begin generateLists()

1. For i = 0, i < 20, i++
2. If a randomly generated integer % 2 = 0
3. Add random integer in range 0 to 1000 to original arraylist
4. Else
5. Add random integer in range (0 to 1000 * - 1) to original arraylist
6. Copy original arraylist to insertion sort arraylist
7. Copy original arraylist to bubble sort arraylist
8. Set size of merge sort array to size of original arraylist
9. Copy original arraylist to merge sort array
10. Set size of quick sort array to size of original arraylist
11. Copy original arraylist to quick sort array

End

Begin generateLists(n)

1. For i = 0, i < n, i++
2. If a randomly generated integer % 2 = 0
3. Add random integer in range 0 to 1000 to original arraylist
4. Else
5. Add random integer in range (0 to 1000 * - 1) to original arraylist
6. Copy original arraylist to insertion sort arraylist
7. Copy original arraylist to bubble sort arraylist
8. Set size of merge sort array to size of original arraylist
9. Copy original arraylist to merge sort array
10. Set size of quick sort array to size of original arraylist
11. Copy original arraylist to quick sort array

End

Begin generateLists(nList)

1. Copy nList array to original arraylist
2. Copy nList array to insertion sort arraylist
3. Copy nList array to bubble sort arraylist
4. Set size of merge sort array to length of nList array
5. Copy nList array to merge sort array
6. Set size of quick sort array to length of nList array
7. Copy nList array to quick sort array

End

Begin generateLists(uList)

1. Copy uList arraylist to original arraylist
2. Copy uList arraylist to insertion sort arraylist
3. Copy uList arraylist to bubble sort arraylist
4. Set size of merge sort array to size of uList arraylist
5. Copy uList arraylist to merge sort array
6. Set size of quick sort array to size of uList arraylist
7. Copy uList arraylist to quick sort array

End

Begin copyArray(fromList, toList)

1. For integer i in fromList arraylist
2. Add i to toList arraylist

End

Begin copyArray(fromList, toList)

1. For i = 0, i < length of toList, i++
2. Get i from fromList arraylist and set it to toList[i]

End

Begin copyArray(fromList, toList)

1. For i = 0, i < length of fromList, i++
2. Add fromList[i] to toList

End

Begin copyArray(fromList, toList)

1. For i = 0, i < length of fromList, i++
2. toList[i] = fromList[i]

End

Begin setUserList()

1. n = inputNumber()
2. Create new arraylist object as uList

3. Print “Please enter integers in the range of -1000 to 1000.”
4. For index = 1, index < n + 1, index++
5. i = inputNumber(index)
6. Add i to uList arraylist
7. Return uList

End

Begin inputNumber()

1. num = 0
2. continueInput = true
3. do
4. try
5. Print “Enter number of elements in list (Must be at least 20): “
6. num = INPUT.nextInt()
7. If num < 20
8. Print “Try again. (Incorrect input: Must be at least 20.)”
9. Continue
10. Else
11. continueInput = false, to break out of loop
12. Catch InputMisMatchException
13. Print “Try again. (Incorrect input: an integer is required.)”
14. Clear INPUT cache
15. while continueInput is true
16. return num

End

Begin inputNumber(index)

1. num = 0
2. continueInput = true
3. do
4. try
5. Print “Enter number“ + index + “: “
6. num = INPUT.nextInt()
7. If num > 1000 or num < -1000
8. Print “Try again. (Incorrect input: -1000 through 1000 required.)”
9. Continue
10. Else
11. continueInput = false, to break out of loop
12. Catch InputMisMatchException
13. Print “Try again. (Incorrect input: an integer is required.)”
14. Clear INPUT cache
15. while continueInput is true
16. return num

End

Begin getOList()

1. Return oList

End

Begin displayList(numList)

1. For integer n in numList arraylist
2. Print n + " "

End

Begin displayList(numList)

1. For int n in numList array
2. Print n + " "

End

Begin insertionSort()

1. For int i = 0, i < iList.size(), i++
2. currentElement = iList.get(i)
3. int k
4. For (k = i – 1), k >= 0 and iList.get(k) > currentElement, k—
5. Set iList (k+1) to k
6. Increment insertion sort swap counter
7. Set iList (k + 1) to currentElement
8. Increment insertion sort swap counter

End

Begin getISwaps()

1. Return iSwaps

End

Begin getIList()

1. Return iList

End

Begin bubbleSort()

1. needNextPass = true
2. for int k = 1, k < bList.size() and needNextPass, k++
3. needNextPass = false
4. for int i = 0, i < bList.size() – k, i++
5. If bList.get(i) > bList.get(i + 1)
6. int temp = bList.get(i)
7. Set bList i to bList.get(i + 1)
8. Increment bubble sort swap counter
9. Set bList (i+1) to temp
10. Increment bubble sort swap counter
11. needNextPass = true

End

Begin getBSwaps()

1. Return bSwaps

End

Begin getBList()

1. Return bList

End

Begin mergeSort(list)

1. If list.length > 1
2. Merge sort the first half
3. Merge sort the second half
4. Merge first half with second half into list

End

Begin merge(list1, list2, temp)

1. Set current1 to zero
2. Set current2 to zero
3. Set current3 to zero
4. While current1 < list1.length and current2 < list2.length
5. If list1[current1] < list2[current2]
6. Temp[current3++] = list1[current1++]
7. Increment merge sort swap counter
8. Else
9. temp[current3++] = list2[current2++]
10. Increment merge sort swap counter
11. While current1 < list1.length
12. temp[current3++] = list1[current1++]
13. Increment merge sort swap counter
14. While current2 < list2.length
15. temp[current3++] = list2[current2++]
16. Increment merge sort swap counter

End

Begin getMSwaps()

1. Return mSwaps

End

Begin getMList()

1. Return mList

End

Begin quickSort(list)

1. quickSort(list, 0, list.length – 1)

End

Begin quickSort(list, first, last)

1. if last > first
2. int pivotIndex = partition(list, first, last)
3. quickSort(list, first, pivotIndex – 1)
4. quickSort(list, pivotIndex + 1, last)

End

Begin partition(list, first, last)

1. Choose the first element as the pivot
2. Set index for forward search as low = first + 1
3. Set index for backward search as high = last
4. While high > low
5. Search forward from left, while low <= high and list[low] <= pivot
6. low++
7. Search backward from right, while low <= high and list[high] > pivot
8. high--
9. Swap two elements in the list, If high > low
10. int temp = list[high]
11. list[high] = list[low]
12. Increment quick sort swaps counter
13. list[low] = temp
14. Increment quick sort swaps counter
15. While high > first and list[high] >= pivot
16. high—
17. If pivot > list[high]
18. list[first] = list[high]
19. Increment quick sort swaps counter
20. list[high] = pivot
21. Increment quick sort swaps counter
22. Return high
23. Else
24. Return first

End

Begin getQSwaps()

1. Return qSwaps

End

Begin getQList()

1. Return qList

End

Begin displayResults()

1. Print “Unsorted list of “ + oList.size() + “ elements: “
2. displayList(getOList())
3. Print “----- INSERTION SORT -----”

4. Print “Insertion sort list – Pre-sort: “
5. displayList(getIList())
6. insertionSort()
7. Print “Insertion sort list – Post-sort: “
8. displayList(getIList())
9. Print “Insertion sort swaps: “ + getISwaps()
10. Print “----- BUBBLE SORT -----”
11. Print “Bubble sort list – Pre-sort: “
12. displayList(getBList())
13. bubbleSort()
14. Print “Bubble sort list – Post-sort: “
15. displayList(getBList())
16. Print “Bubble sort swaps: “ + getBSwaps()
17. Print “----- Merge SORT -----”
18. Print “Merge sort list – Pre-sort: “
19. displayList(getMList())
20. mergeSort()
21. Print “Merge sort list – Post-sort: “
22. displayList(getMList())
23. Print “Merge sort swaps: “ + getMSwaps()
24. Print “----- QUICK SORT -----”
25. Print “Quick sort list – Pre-sort: “
26. displayList(getQList())
27. quickSort()
28. Print “Quick sort list – Post-sort: “
29. displayList(getQList())
30. Print “Quick sort swaps: “ + getQSwaps()

End

UML


DemoSortingAlgorithms
algos : ArrayList<SortAlgorithms>
SortingAlgorithms
-INPUT : Scanner class object
-rand : Random class object
-oList : ArrayList<Integer>
-iList : ArrayList<Integer>
-bList : ArrayList<Integer>
-mList : int[]
-qList : int[]
-iSwaps : int
-bSwaps : int
-mSwaps : int
-qSwaps : int
+SortAlgorithms()
+SortAlgorithms(n : int)
+SortAlgorithms(nList : int[])
+SortAlgorithms(uList : ArrayList<Integer>)
-generateList() : void
-generateList(n : int) : void
-generateList(nList : int[]) : void
-generateList(uList : ArrayList<Integer>) : void
-copyArray(fromList : ArrayList<Integer>, toList : ArrayList<Integer>) : void
-copyArray(fromList : ArrayList<Integer>, toList : int[]) : void
-copyArray(fromList : int[], toList : ArrayList<Integer>) : void
-copyArray(fromList : int[], toList: int[]) : void
+setUserList() : ArrayList<Integer>
-inputNumber() : int
-inputNumber(index : int) : int
-getOList() : ArrayList<Integer>
-displayList(numList : ArrayList<Integer>) : void
-displayList(numList : int[]) : void
-insertionSort() : void
-getISwaps() : int
-getIList() : ArrayList<Integer>
-bubbleSort() : void
-getBSwaps() : int
-getBList() : ArrayList<Integer>
-mergeSort(list : int[]) : void
-merge(list1 : int[], list2 : int[], temp : int[]) : void
-getMSwaps() : int
-getMList() : int[]
-quickSort(list : int[]) : void
-quickSort(list : int[], first : int, last : int) : void
-partition(list : int[], first : int, last : int) : int
-getQSwaps() : int
-getQList() : int[]
+displayResults() : void