CSCI 2235
Programming Project 01 - Sort Comparison

Assigned: September 02, 2019
Due: September 27, 2019 @ 23:00h

1 Purpose

- Gain an understanding of the experimental process used to compare algorithms.
- Learn about the power of the $O(n \log n)$ sorting algorithms.

2 Problem Statement

Your boss wants you to build a sorting method that will sort an array of random values (no order whatsoever) as quickly as possible. These arrays will be between 10,000 and 100,000 elements, and usually Integers from 0 to 100,000 (but your method needs to use generic data types just in case). You were just going to use quick sort until the subarrays get to about 10 elements, then use insertion sort for those 10 element subarrays. He also suggested comparing your approaches against a know simplistic approach such as InsertionSort. To complete this work you will conduct an experiment to compare the results of the following sorts on large arrays:

- MergeSort
- QuickSort
- InsertionSort
- SelectionSort
- TimSort – A combination of MergeSort and InsertionSort where InsertionSort is used when the size of the subarrays is 10 or less.
- HybridSort – A combination of QuickSort and InsertionSort where InsertionSort is used when the size of the subarrays is 10 or less.

To do the comparisons I would suggest using a method similar to that used in Programming Assignment 01. Also similar to Programming Assignment 01 this assignment will use a Strategy Pattern based approach where each of the sorts implement the ArraySort interface provided to you.

3 Copying Generic Arrays

In Java we cannot directly create generic arrays. To get around this problem as follows:

1. add the following import: `import java.util.Arrays;` to the top of your class file.
2. Assuming you have a generic parameter `E`, you can create a new `E` array from an exiting one (i.e., copy) as follows:

```
/**
 * Constructs an exact copy of the input array
 * @param otherArray The array to copy
 * @param E The type parameter of the data in the array
 * @return A copy of the provided array
 */
public <E> E[] copyArray(E[] otherArray) {
    E[] array = Arrays.copyOf(otherArray, otherArray.length);
}
```

4 Assignment

1. Start with the provided code.
2. Review the interface ArraySort in the package edu.isu.cs2235.algorithms
3. Create the package edu.isu.cs2235.algorithms.impl
4. Implement each of the following classes, which implement the interface ArraySort (**Note: these classes should be implemented in the package** edu.isu.cs2235.algorithms.impl):
    - InsertSort
    - SelectionSort
    - MergeSort
    - QuickSort
    - TimSort
    - HybridSort
5. Upon complete implementation run the tests associated with each class using the command `$ gradle test` in the root directory of the project. You can also execute the tests from within your IDE.
6. Data collection

```
For each algorithm
    For i = 10,000 to 100,000 by 10,000 do
        for i = 0 to 49 by 1 do
            Create a random array of size i filled with random values between 1
               and 100,000
            Execute the selected algorithm marking the start and end times
            Take the difference between start and end and store it.
        Find the average time for the algorithm at that level and store that
           value for reporting
Report results to the Standard Output
Report results to a CSV (comma separated values) file named 'report.csv'
```

7. Using this report.csv file generate a chart (using MS Excel or its equivalent). Then import this into a text editor and prepare a short writup regarding your results (similar to Programming Assignment 01).
8. Produce a PDF file called "writeup.pdf" and save it to the root directory of your project. Final writeup should be in MLA format with your name and proper header. Failure to follow this format and include your name will result in a 0 in a zero credit for the writeup section. See the Purdue Owl MLA Guideline for information on proper MLA formatting.
9. ZIP COMPRESS YOUR PROJECT FOLDER TO A FILE NAMED AS FOLLOWS: [first]_[last]_pp01.zip (where first is your first name and last is your last name and the square brackets are not included but rather indicate required).
10. Submit the file to moodle by the date and time noted at the top of this document. **Please note that if your file name does not conform to the file name specifications noted, it will not be graded and you will receive a zero!**

NOTE: To run your code you will need to create a class in package edu.isu.cs2235 which contains the method **public static void** main(String[] args) {...} . I would suggest that this be the experimentation program and have the name Driver. If you use a name for the class other than Driver you will need to change the last line of build.gradle in the project root directory to reflect the different name.

4.1 Merge Sort

The following algorithms present two methods for implementing merge sort. The first (Algorithm 1A) is an in-place MergeSort, and the second (Algorithm 1B) is the more typical multiple copy version of MergeSort. Either approach is fine.

Algorithm 1A: In Place MergeSort

```
function sort(A)
  copy = copy(A)
  mergeSort(copy, A, 0, length(A))
end function

function mergeSort(A, result, start, end)
  if end = start then
    return
  end if
  if end - start < 2 then
    if result[start] > result[end] then
      swap(result, start, end)
    end if
    return
  end if
  mid = (end - start) / 2 + start
  mergeSort(result, A, start, mid)
  mergeSort(result, A, mid + 1, end)
  i = start
  j = mid + 1
  for index = start to end do
    if j > end or (i <= mid and A[i] < A[j]) then
      result[index] = A[i]
      i = i + 1
    else
```

```
      result[index] = A[j]
      j = j + 1
    end if
  end for
end function

function swap(A, first, second)
  temp = A[first]
  A[first] = A[second]
  A[second] = temp
end function

function copy(A, start, end)
  T = [] // new empty array of size end - start + 1
  for i = start to end do
    T[i] = A[i]
  end for
  return T
end function
```

Algorithm 1B: MergeSort (typical)

```
function sort(A)
  mergeSort(A)
end function

function mergeSort(A)
   if (length(A) <= 1)
      return A
   else
      lower = mergeSort(copy(A, 0, length(A) / 2)
      upper = mergeSort(copy(A, length(A) / 2, length(A) - 1)

      return merge(A, lower, upper)
  end if
end function

function merge(A, lower, upper)
  i = 0
  j = 0

  while i + j < A.length
    if (i > length(lower) - 1)
       A[i + j] = upper[j]
       j = j + 1
    else if (j > length(upper) - 1)
       A[i + j] = lower[i]
       i = i + 1
    else if (lower[i] <= upper[j])
       A[i + j] = lower[i]
       i = i + 1
    else
       A[i + j] = upper[j]
```

```
      j = j + 1
    end if
  end while
end function
```

4.2 Algorithm 2: QuickSort

```
function sort(A)
  quicksort(A, 0, n - 1)
end function

function quicksort(A, left, right)
  if left < right then
    pi = partition(A, left, right)
    quicksort(A, left, pi - 1)
    quicksort(A, pi + 1, right)
  end if
end function

function partition(A, left, right)
  pivot = A[left]

  i = left + 1
  j = right
  while i <= j do
    if A[i] > pivot and A[j] <= pivot then
      swap(A, i, j)
      i = i + 1
      j = j - 1
    else
      if A[i] <= pivot
        i = i + 1
      end if
      if A[j] > pivot
        j = j - 1
      end if
    end if
  end while

  swap(A, left, j)
  return j
end function

function swap(A, first, second)
  temp = A[first]
  A[first] = A[second]
  A[second] = temp
end function
```

4.3 Algorithm 3: InsertionSort

```
function sort(A)
  for i = 1 to n - 1 do
    insert(A, i, A[i])
  end for
end function

function insert(A, pos, value)
  i = pos - 1
  while i >= 0 and A[i] > value do
    A[i + 1] = A[i]
    i = i - 1
  end while
  A[i + 1] = value
end function
```

4.4 Algorithm 4: SelectionSort

```
function sort(A)
  for firstUnsorted = 0 until firstUnsorted = length(array) - 1 do
    min = firstUnsorted + 1
    for i = min until length(array) - 1 do
      if array[i] < array[min] then
        min = i
      end if
    end for
    swap(firstUnsorted, i)
    firstUnsorted += 1
  end for
end function
```

4.5 Algorithm 5: TimSort

As noted above this is a combination of both MergeSort and InsertionSort. I leave the implementation details of this up to you. Just note that it must invoke the use of InsertionSort when the size of the subarrays is 10 or less.

4.6 Algorithm 6: HybridSort

As noted above this is a combination of both QuickSort and InsertionSort. I leave the implementation details of this up to you. Just note that it must invoke the use of InsertionSort when the size of the subarrays is 10 or less.

5 Submission

1. Construct a PDF for the algorithm comparison and place into the root directory of your project. Name the file “writeup .pdf”
2. Zip compress your entire project directory into a file name [firstname]_[lastname].zip (note: the square brackets are not part of the file name, but simply denote required information).
3. Submit the zip file to moodle in the dropbox for PP01

6 Grading (50 Points)

- (2.5 points) Insertion Sort implemented correctly.
- (2.5 points) Selection Sort implemented correctly.
- (5 points) Merge Sort implemented correctly.
- (5 points) Quick Sort implemented correctly.
- (10 points) Tim Sort implemented correctly.
- (10 points) Hybrid Sort implemented correctly.
- (15 points) Writeup