Project 3: Practical analysis of algorithms

1 Why do we do it?

We have so far figured out the best and worst cases for several sorting algorithms. But, the really important stuff is the average case behavior of these algorithms, which is tough to get. As an example, we will go through a theoretical analysis of the average case for the Insertion sort in the next chapter on probabilistic analysis.

A practical way to analyze algorithms is to add counters in the algorithms to get the number of the quantities that we are interested, e.g., the number of data comparisons, and then compare these results to the known functions, such as $n$, $n \log n$, $n^2$, etc, to find out the exact and/or upper bounds for these quantities. We can then choose the least expansive ones based on the orders of these magnitudes.

We will see how to follow this approach of algorithm analysis by studying the average number of comparisons and movements of several sorting algorithms through this project, which we have already worked on in the last Project.

2 What to do?

1. Collect the codes for the sorting programs that you developed for the previous project, i.e., BubbleSort, InsertionSort, MergeSort, and SelectionSort.

2. Make necessary changes so that those algorithms will sort out a list of integers.

3. Add in counters in appropriate places within these algorithms to keep respective records for, e.g., data comparisons and data movement. For example, the following shows how to add counters into the insertion sort algorithm to get the correct number of comparisons with $C_{\text{Insertion}}$ and movements with $M_{\text{Insertions}}$.

   ```
   M_{\text{Insertion}} <- 0
   C_{\text{Insertion}} <- 0
   INSERTION-SORT(A)
       for j <- 2 to length[A]
           do
   ```

---

1 Several students could not work out some of the code. In this case, feel free to use those posted on the Project page under Project 2. Regarding MergeSort, considering the situation, the one without using bed rocks, i.e., MergeSort.java, might make things easier.

2 You just create arrays of integers. Check out Footnote 1 in Project 1 assignment for the array type in Java and examples.
key <- A[j]
//Increment the counter for move
MInsertion++
i <- j-1
//Increment the counter for comparisons for the very first one
CInsertion++
while i>0 and A[i]>key
  do
    A[i+1]<-A[i]
    //Increment the counter for move
    MInsertion++
    i<-i-1
    CInsertion++ //For the others, including the last one
A[i+1]<-key
//Increment the counter for move
MInsertion++

The following shows the process of using these counters

1 Generate a list of integers, list
2 CInsertion=0
3 MInsertion=0
4 INSERTION-SORT(list)
5 Print CInsertion
6 Print MInsertion

**Question 1:** We usually increment the counters after the fact, e.g., in Lines 5, 14, and 19. **Why do we do it before the fact in Lines 9, and 16?**

**Question 2:** The best way to understand an algorithm is to trace through... . What will be values of both counters when you apply it to (1, 2, 3), (1, 3, 2) and (3, 2, 1)?

For example, when applied to the input list (1, 2, 3), the final values of CInsertion and MInsertion will be 2 and 4, respectively. Are they correct? Why? (Cf. Pages 28 and 29 of Chapter 2 notes) How about the others?

Such an analysis should help you to figure out an answer to **Question 1** so that you can properly insert such counters for the other sorting algorithms.

4. Add such a pair of counters into the other sorting algorithms to keep a record of the correct number of movements and comparisons.

5. Use the random number generator that Java provides via, e.g., java.util.Random, to come up with test data of decent size, e.g., n = 10, 20, 50, 100, 200, 500, 1000, and
collect the average value of the data comparison and data movement numbers in a spreadsheet.

To iron out probable noise, run the respective sorting algorithm 5,000 times\(^3\) for each of the above values of \(n\).

6. Compare the data that you collected in the previous step with the theoretical results as shown in Table 1. They might not be identical, but should be pretty close\(^4\).

To further clarify your comparative results, you want to make use of the chart mechanism, as provided by, e.g., Microsoft Excel, to demonstrate that their growth rates are in the same orders as the theoretical yard sticks.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comparison</th>
<th>Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>(\frac{n^2}{2} + O(n))</td>
<td>See Assignment 7</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>(\frac{n^2}{2} + O(n))</td>
<td>(\frac{n}{4} + O(n))</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>(\frac{n^2}{2} + O(n))</td>
<td>(3n + O(1))</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>(n \log(n) + O(n))</td>
<td>(n \log(n) + O(n))</td>
</tr>
</tbody>
</table>

Table 1: Theoretical formulas

7. The expression of the average number of data movement for the bubble sort is quite messy. \(\text{😊}\) This is another place when practical analysis is valuable.

Find out the average number of movements made in the bubble sort for \(n = 10, 20, 50, 100, 200, 500, \) and \(1000\), compare the growth rate of the results with that of \(n, n \log n, \) and \(n^2\), and make an educated guess as what is the closest match of the average number of movements associated with the bubble sort algorithm out of the aforementioned bounds.

3 What should be handed in?

Send me, via Canvas, the source code, as well as a lab report, as a zip file, containing the following:

1. Your answer to **Question 1** as backed up by your answer to **Question 2**.

2. An explanation regarding your placement of the counters to collect the data for both comparison and movement.

---

\(^3\)Notice that \(10! = 3,628,800\). So, we cannot afford to generate all the possible permutations.

\(^4\)If they are significantly different, the placement of your counters must be incorrect.
3. A spreadsheet that contains both the data as collected through your program, and
calculated with the formulas as shown in Table 1, and their comparisons, as represented
in meaningful charts. For example, a table as shown in Table 2 can be used to compare
the results for one of the four sorting algorithms.

<table>
<thead>
<tr>
<th>n</th>
<th>Practical Comp</th>
<th>Theoretical Comp</th>
<th>Practical Move</th>
<th>Theoretical Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Data Comparison

4. What is your justification, e.g., what led you to believe that, it takes, on average, \( \frac{n^2}{4} \) comparisons, to sort a list with \( n \) randomly picked elements using InsertionSort? You can either do it theoretically, using the notions that we have learned in the order of growth chapter that the ratio of the theoretical and practical data is a constant in the long run; or use a curve to show that those for the theoretical data and that for the practical one do match.

5. Your educated guess of the exact bound for the average number of the data movement as associated with the bubble sort algorithm, and its justification.

### 4 A general grading guideline

\( \geq 1 \): A syntactically correct and executable program

\( \leq 2 \): Counters are properly placed as backed up by your answers to the questions (Steps 1 and 2)

\( \leq 3 \): Both the collected, and calculated, data in spreadsheet(s) (Step 3).

\( \leq 4 \): Justification of your comparative conclusion, using, e.g., charts of growth rate comparison (Step 4)

\( \leq 5 \): The exact bound for the average number of the data movement as associated with the bubble sort algorithm, and why (Step 5)