University of Manitoba  
COMP 2140, Winter 2019  
Assignment 3

Due Date: March 18, at 8:00pm

Don’t be satisfied with stories, how things have gone with others. Unfold your own myth.

Rumi

Please pay attentions to the followings when preparing your assignment:

• There is both written and programming components in this assignment.

• You need to submit your solutions for the written parts electronically via Crowdmark. You need to submit your programming questions via UM-learn as you did in Assignment 1. Place all Java files in a single folder that will be available for Assignment 3.

• There are 5 problems with a total mark of 66. All questions are mandatory.

• As always, I encourage you to ask your questions on Piazza. Those students who help others on Piazza will receive bonus marks (by ‘helping’, I mean removing confusions). It is likely that I drop hints on Piazza in response to the questions that are asked publicly. In the pursuit of fairness, I hesitate to drop a hint in response to private questions or emails. So, be active on Piazza!

• This is an individual assignment. You are welcome to discuss questions with your friends (or enemies). But you should write the answers individually, and you should fully understand what you are writing. Please read http://www.cs.umanitoba.ca/~kamalis/winter19/infoCOMP2140.pdf for guidelines on academic integrity.

• Good luck!
Problem 1  Multiple Parenthesis [5+15 marks]

A sequence of parenthesis is said to be ‘balanced’ if each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested. For example, (())((() is balanced and (())()) is not. A multiple parenthesis is the same concept except that parenthesis have types and each open symbol and the pairs of parentheses are properly nested. For example, (()(())) is balanced and (()))() is not balanced.

Problem 1 Multiple Parenthesis [5+15 marks]

A sequence of parenthesis is said to be ‘balanced’ if each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested. For example, (())((() is balanced and (())()) is not. A multiple parenthesis is the same concept except that parenthesis have types and each open symbol and the pairs of parentheses are properly nested. For example, (()(())) is balanced and (()))() is not balanced.

Problem 1 Multiple Parenthesis [5+15 marks]

A sequence of parenthesis is said to be ‘balanced’ if each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested. For example, (())((() is balanced and (())()) is not. A multiple parenthesis is the same concept except that parenthesis have types and each open symbol and the pairs of parentheses are properly nested. For example, (()(())) is balanced and (()))() is not balanced.

(a) Explain an efficient strategy for checking whether a given multiple parenthesis sequence is balanced or not. Describe your solution using English words. For an input sequence of length \( n \), your algorithm should run in time \( O(n) \). Note that you might need to use some of the data structures we learned in the class. Answer: Let \( S \) be an initially empty stack. Process arrays \( p \) and \( b \) from index 0 to \( n - 1 \). At index \( i \), if the parenthesis is ‘open’ (if \( b[i] == \text{false} \)), push \( p[i] \) to \( S \). If the parenthesis is close (if \( b[i] == \text{true} \)), pop a value \( x \) from the stack; if \( p[i] == x \), then the type of this closed parenthesis and its matching one is the same. Otherwise, when \( p[i] \neq x \), the sequence is not balanced and the algorithm returns ‘false’, e.g., for sequence \( (()) \). While attempting the pop operation, if the stack is empty, the number of open paranthesis is less than the close ones and the algorithm returns ‘false’ (e.g., \( (((()) \)) \)). When all paranthesis are processed, we should check the stack; if it is empty, return ‘true’; otherwise, the number of open paranthesis is more than closed ones and the algorithm returns ‘false’, e.g., for sequence \( (()) \).

Marking Scheme: Any efficient, linear time approach to this problem requires a stack (which could be implemented using, e.g., a linked list). If you used some other approach, you will get at most 2 out of 5 (because your algorithm would run much slower). For a stack-based approach, you should indicate the three possible cases when your algorithm returns ‘false’ (when the stack is empty when popping, when the parentheses are not matching, and when the array is not empty at the end); missing any of these cases results in deduction of 1 mark. You should mention when the algorithm returns ‘true’ (at the end if it did not return ‘false’).

(b) Implement your algorithm from part (a). For that, create a class named Parenthesis which have a method \textbf{public static boolean checkBalanced(int[]} p, boolean[] b). The output of the method is \textbf{true} if the sequence is balanced and \textbf{false} otherwise. In case you use a data structure learned in the class (e.g., a queue or a stack), you have to implement them in your code (that is, you cannot use Java’s libraries).

p.s. A test file will be provided shortly.

Problem 2  Circular Shifts [15 marks]

Assume you are given a linked list formed by \( n \) nodes such that each node stores an integer. Write an efficient Java code that prints all circular shifts of the odd numbers in the linked list in array of \( n \) strings. For example, if the list is \( 1 \rightarrow 2 \rightarrow 15 \rightarrow 14 \rightarrow 23 \), the output will be an array \textbf{str} of strings such that \textbf{str}[0] = “1, 15, 23”, \textbf{str}[1] = “15, 23, 1”, and \textbf{str}[2] = “23, 1, 15”. Note that the numbers are separated by ‘,’ and there is no space between them.

You need to create a class named Shifts which has the following method: \textbf{public static String[]} giveShifts(LinkedList<Integer> list). Here \textbf{list} is a linked list that maintains integers. Refer to slides 13 and 14 (of the module on stacks and linked lists) to see how \textbf{Node} and \textbf{LinkedList} should be implemented. The output of the method is an array of strings as described. In case you use a data
structure learned in the class (e.g., a queue or a stack), you have to implement them in your code (that is, you cannot use Java’s libraries).

p.s. A test file will be provided shortly.

**Problem 3 Detecting the Right Sorting Algorithm [6 \times 3 = 18 marks]**

For each of the following questions, a situation is described in which you have to use a sorting algorithm. Among the proposed sorting algorithms, indicate the most suitable algorithm. Note that you can select only one algorithm. For each situation, you have to consider factors such as time complexity, space complexity (e.g., being in-place), and applicability of the sorting algorithms. In all cases, assume \( n \) is a very large number. You need to justify your answer in one or two sentences.

(a) A set of \( n \) integers in the range \([-100, 100]\) should be sorted.

<table>
<thead>
<tr>
<th>Insertion Sort</th>
<th>Quick Sort</th>
<th>Marge Sort</th>
<th>Counting Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Answer:</strong> Counting Sort as it runs in linear time. In general, when integers in small range are being sorted, counting sort is preferred.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) An array of \( n \) real numbers should be sorted. The array was already sorted but a bug in a function developed by a lazy programmer caused only a few of the numbers get out of order. So, the array is almost sorted.

<table>
<thead>
<tr>
<th>Insertion Sort</th>
<th>Quick Sort</th>
<th>Marge Sort</th>
<th>Counting Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Answer:</strong> We use Insertion sort. As we saw in the class, the insertion sort runs in ( O(n) ) when the input is almost sorted. It is in-place and does not use additional memory (as opposed to Quick-sort). Note that counting sort is not useful for sorting real numbers.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) An array of \( n \) real numbers should be sorted. The array is so big that it occupies more than half of the main memory of your laptop. You need to sort the array using your laptop; so you need to be careful about memory usage of your algorithms.

<table>
<thead>
<tr>
<th>Quick Sort</th>
<th>Marge Sort</th>
<th>Counting Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Answer:</strong> We use Quick Sort as Merge Sort is not in-place and needs ( O(n) ) additional memory (which does not fit in the memory of your laptop). Counting sort is no good for real numbers.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(d) We want to sort an array of \( n \) integers with values in the range \([0, n \log n]\).

<table>
<thead>
<tr>
<th>Marge Sort</th>
<th>Counting Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Answer:</strong> We use Merge-Sort. Both algorithms have time complexity ( O(n \log n) ). The extra space used by Merge-Sort is ( O(n) ) while the extra space for Counting Sort is ( O(n \log n) ).</td>
<td></td>
</tr>
</tbody>
</table>

(e) You want to sort records of \( n \) students in University of Manitoba. You are given an array with each index having a pointer to a class Student. In the sorted array, international students appear before Canadian ones. For two international students, the one with the higher GPA appears first. For two Canadian students, the one with lower GPA appears first.

<table>
<thead>
<tr>
<th>Insertion Sort</th>
<th>Counting Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Answer:</strong> We use the Insertion Sort. Counting Sort cannot be used since you do not have integers. You can, however, map each student to a number and use a counting sort. This requires a tailored mapping that guarantees all international students are mapped to smaller numbers (e.g., their most significant digit is 0 while those of Canadian students is 1) and the digits with lower significant indicate a mapping from the GPA. You will get 3, 2, 1, or 0 marks depending on your explanations if you chose counting sort.</td>
<td></td>
</tr>
</tbody>
</table>

(f) You want to sort a stream of integers which arrive one by one in an online manner. As the numbers arrive, you need to sort them without knowing when the whole stream ends.
Answer: Insertion Sort as it maintains a sorted subset of items without having the whole input. The other two algorithms need to have the whole input to ‘divide’ it before ‘conquering’.

Marking Scheme: for each part except for (e), 1 mark for the correct answer and 2 marks for the correct justification.

Problem 4 Quick Sort [8 marks]

(a) A stable sorting algorithm is one in which the relative order of all identical elements (or keys) is the same in the output as it was in the input. Indicate whether quick sort (as we saw in the class) is stable or not. You need to justify your answer via an example or a proof.

Answer: Quick-sort with the partition method that we saw in the class is Not stable. Consider an input \( a = \{5, 2, 6, 4, 6', 1\} \). In the first swap, the partition algorithm swaps 1 and 6 and the array becomes \( a = \{2, 1, 4, 5, 6', 6\} \) after the partition. When recursing on the right, the order of 6 and 6’ does not change, i.e., the final sorted array will be \( a = \{1, 2, 4, 5, 6', 6\} \) in which the initial ordering of 6 and 6’ is changed.

Marking Scheme: This part has 3 marks: 1 mark for mentioning that the algorithm is not stable, 2 for a justification; this justification should include an example. A description without example does not suffice (and gets at most 2 marks).

(b) Consider an implementation of quick-sort in which the pivot is always selected in a way that at least \( n/4 \) items are smaller than it and at least \( 2n/3 \) items are larger than it. Provide a recursion for the best-case time complexity of the algorithm and solve the recursion to express the best-case time complexity using big-Oh notation.

Answer: Note that in this case, the pivot is in the range \((n/4, n/3)\). The best-case running time happens when the two recursive calls have sizes as close as possible to each other. In this case, the best case happens when the pivot is at \( n/3 \). In this case, the two recursive call will have sizes roughly \( n/3 \) and \( 2n/3 \). The recursive function for the time complexity is \( T(n) \leq T(n/3) + T(2n/3) + cn \) and \( T(1) = d \) for some constants \( c \) and \( d \).

To solve the recursion, we can write \( T(n) = T(n/3) + T(2n/3) + cn = T(n/9) + T(2n/9) + T(2n/9) + T(4n/9) + 2cn \). After recursing \( \log_{3/2} n \) times, all recursive components will become constant and the accumulated time will be \( \log_{3/2} n \times cn = O(n \log n) \).

Marking Scheme: 2 marks for the correct recursion; 2 mark for mentioning that recursion takes \( O(n \log n) \) and 1 mark for the right solution to the recursion.

Problem 5 Lower Bounds [5 marks]

Recall that in a sorted array of \( n \) comparable items, we can use binary search to search for a given item in \( O(\log n) \). Prove that binary search is the optimal searching algorithm in a sorted array. You need to use a decision tree approach to show that no algorithm can search in a sorted array in time less than \( O(\log n) \).

Answer: Given a sorted array, the searched item can be at any index item in the array. So, there are \( n \) possible inputs. Consider a decision tree with \( n \) leaves. As mentioned in the class, the height of this tree will be \( O(\log n) \). So, the number of comparisons that is required to search in the array is at least \( \log(n) \).
Marking Scheme: 2 mark for mentioning the number of possible inputs, 1 mark for mentioning decision tree, 2 mark for argument on the height of the tree.