COMP 7035
LINEAR DATA STRUCTURES & ALGORITHMS

ASSIGNMENT 2 – Divide & conquer and greedy algorithms

SUBMISSION & DUE DATE

This assignment should be submitted to Canvas before 11:59pm on **Friday 28/11/2025**. The usual late submission penalties apply in accordance with MTU's marks and standards policy.

Please submit a single ZIP file with your student number and name in the filename. Your submission should contain **exactly 6 files**:

- *DivideAndConquer.java*
- *Greedy.java*
- *MergeSort.java*
- *QuickSort.java*
- *HeapOfBinaryTries.java*
- *Huffman.java*

Please do **NOT** include any other files in your submission.

To ensure that your answers are your own during the oral assessment of your code the submitted Java files must not contain any comments. **If your code contains any comments your work will not be graded!**

EVALUATION PROCEDURE

You can achieve a total of 25 points for the submission as indicated in the tasks. For each subtask you are given full marks for correct answers in your submission, 70% for minor mistakes, and 35%, 15%, or 0% for major mistakes or omissions depending on severity.

To mitigate against any form of plagiarism, including but not limited to the use of generative AI tools, **you MUST also present and explain your solution and confidently answer questions** about your submission to be graded. This oral assessment will take place during the scheduled labs after the submission deadline (or the week after, should we run out of time during these sessions).
If you do not show up your submission will not be graded unless there are approved IECs!

Your ability to fully explain your own submission will be crucial. You will only receive full marks if you are able to confidently present and explain all your work. Your assignment grade will be capped at 70% or 50% in case of minor issues with your explanations, and at 35%, 15%, or 0% in case of major problems with your explanations during the oral assessment.

The oral assessment will cover everything you uploaded onto Canvas, so be very careful what you include into your submission and make sure that you **only submit what you fully understand** to avoid your final marks being capped; in particular, do NOT include anything, including snippets taken from labs or lectures, unless you are able to explain it in detail or risk the whole rest of the assignment being marked down.

TASK 1 (Divide & conquer and greedy algorithms, 8 points)

In this task you will implement some simple divide & conquer and greedy algorithms. Download the Java source files *Assignment_LDSA_2.zip* from Canvas and edit the files *DivideAndConquer.java* and *Greedy.java* to implement the missing code where indicated.

A. Complete the following functions in the `DivideAndConquer` class using the divide and conquer algorithm schema (any solution, even if correct, that does not follow this schema will be marked as zero points!).

a. The Fibonacci numbers are a sequence of numbers defined recursively as

$$F_n = \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ F_{n-1} + F_{n-2} & otherwise \end{cases}$$

Implement the function

```
public static int fibonacci(int n)
```

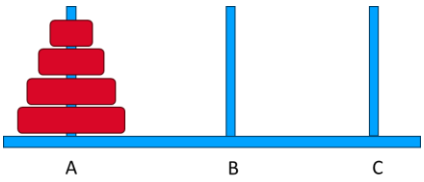
that calculates and returns the n^{th} Fibonacci number [1 point].

b. A binary search can be used to efficiently find an element in a sorted array. Implement the function

```
public static int search(int[] A, int v)
```

that takes as input a sorted array A and a value v and returns the index within the array at which the value is stored or -1 otherwise [1 point]. You can assume that the array is sorted already.

c. The Towers or Hanoi is a puzzle in which we want to find the best strategy to move the discs from rod A to rod C (see below) using an intermediate rod B so that at no point a larger disc is put on top of a smaller disc.



Implement the function

```
public static void hanoi(int n, char A, char B, char C)
```

that takes as input the number of discs on rod A and three characters ‘A’, ‘B’, and ‘C’ to indicate the names of the rods and outputs the optimal strategy on the console (e.g. A -> B, A -> C, B -> C, etc.) [2 points].

B. Complete the following functions in the `Greedy` class using the greedy algorithm schema (any solution, even if correct, that does not follow this schema will be marked as zero points!).

   a. The activity selection problem tries to find the maximum number of non-overlapping activities given their start and finish times. These activities are represented by the *Activity* class provided as part of the Java source files for this assignment. Implement the function

```
public static LinkedList<Activity>
    activitySelection(LinkedList<Activity> activities)
```

   that takes as input a list of activities and should produce as output a maximum list of non-overlapping activities selected from the input [2 points]. Use the function

```
public Boolean overlap(Activity activity)
```

   in the *Activity* class to determine if two activities overlap. You can assume that the input activities are sorted according to finish time already.

   b. The change making problem tries to find the minimum number of coins/notes for a given amount of money. A greedy strategy, which works for the denominations of the Euro currency (1c, 2c, 5c, 10c, 20c, 50c, €1, €2, €5, €10, €20, €50), is to always use the coin/note with the largest possible value and continue until the full amount is paid (e.g. €12.34 = €10 + €2 + 20c + 10c + 2c + 2c). Implement the function

```
public static LinkedList<Integer>
        makeChange(int amount, int[] denominations)
```

   that takes as input the amount to be paid and a list of available denominations and outputs a minimal list of coins/notes that adds up to the desired amount [2 points]. You can assume that the input list of denominations is ordered from the largest to the smallest.

TASK 2 (Sorting algorithms, 8 points)

In this task you will implement two efficient sorting algorithms based on the divide and conquer strategy. Download the Java source files *Assignment_LDSA_2.zip* from Canvas and edit the files *MergeSort.java* and *QuickSort.java* to implement the missing code where indicated.

A. Complete the `MergeSort` class to implement the merge sort algorithm as follows

   a. First implement the function

```
private static int[] merge(int[] A1, int[] A2)
```

   that takes two sorted integer arrays as input and produces as output a new sorted integer array containing all elements from the two inputs merged [2 points].

   b. Then implement the function

```
public static int[] mergesort(int[] A)
```

that takes an unordered integer array as input and returns a new sorted integer array as output containing all elements from the input [2 points]. Apply a divide & conquer strategy based on merging sorted arrays using the function implemented in the previous subtask.

B. Complete the `QuickSort` class to implement the quick sort algorithm as follows
   a. First implement the function

```
private static int partition(int[] A, int p, int r)
```

   that takes as input an integer array A and two indexes $p < r$ and then reorders the elements inside the subarray $A[p..r]$ so that for some $q \in \{p, \dots, r\}$ all elements in in the subarray $A[p..q]$ are all smaller than $A[q]$ and all elements in the subarray $A[q \dots r]$ are all larger than $A[q]$ [2.5 points]. The function should perform this reordering in place by pairwise swapping of elements and return the index q as output.

   b. Then implement the function

```
private static void quicksort(int[] A, int p, int r)
```

   that takes as input an integer array A and two indexes p and r and reorders the elements in the subarray $A[p..r]$ in place so that they are sorted when the function returns [1.5 points]. Apply a divide & conquer strategy based on the partition function implemented in the previous subtask.

TASK 3 (Huffman codes, 9 points)

In this task you will implement a heap based greedy algorithm to calculate optimal binary codes based on the character frequency in a string. Download the Java source files *Assignment_LDSA_2.zip* from Canvas and edit the files *HeapOfBinaryTries.java* and *Huffman.java* to implement the missing code where indicated.

A. At the core of an efficient implementation of Huffman codes is a heap data structure that stores the binary tries used to represent the prefix codes being built by the algorithm. The latter is already implemented in the `BinaryTrie` class provided as part of the Java source files for this assignment. To implement the required heap data structure, complete the following functions in the `HeapOfBinaryTries` class

   a. Implement the function

```
private void heapify(int i)
```

   that reorders the elements of the array

```
private BinaryTrie[] A
```

   such that $A[i]$ and all its descendants satisfy the heap property (i.e. that $A[i]$ is smaller than all its descendants) provided that these descendants already satisfied the heap property themselves prior to the call [2.5 points]. Use the function

```
public Boolean compare(BinaryTrie T)
```

provided in the `BinaryTrie` class to determine the order of elements in the heap.

b. Implement the constructor

```
public HeapOfBinaryTries(BinaryTrie[] A)
```

to create a heap from an array of binary tries [1 point]. Use the heapify function implemented in the previous subtask and ensure that the fields

```
private BinaryTrie[] A;
private int heapsize;
```

are initialised correctly.

c. Implement the function

```
public BinaryTrie extractMin()
```

that returns and removes the minimum element from the heap [1.5 points]. Make sure that all variables are updated correctly and that the heap property is maintained for the remaining elements.

d. Implement the function

```
public void insert(BinaryTrie x)
```

to insert a new single element into the heap [2.5 points]. Make sure that all variables are updated correctly and that the heap property is maintained.

B. The `Huffman` class provided as part of the Java source files for this assignment already has functionality for counting the frequency of characters in an input string and for encoding and decoding strings based on a binary trie representing a suitable prefix code. To calculate this required binary trie, implement the following function

```
private static BinaryTrie findOptimalCode(HeapOfBinaryTries H)
```

It takes a heap as input that contains all code tries representing a single character occurring in the input string. The output should be a prefix code trie for all characters, calculated using the greedy Huffman algorithm by consecutively combining the lowest frequency partial codes until only a single code trie is left in the heap [1.5 points]. The implementation should use the heap data structure developed in the previous subtask to efficiently manage and identify which tries to merge and use the constructor

```
public BinaryTrie(BinaryTrie left, BinaryTrie right)
```

of the BinaryTrie class to merge tries when required.