Honour Code

1. Do not copy the answers from any source.
2. Do not use generative AI tools to generate your code.
3. You may not receive or share any part of your code with anyone other than your partner.
4. Cheating students will face the Committee on Discipline.

Assignment Details

In this assignment you will implement several sorting algorithms you've seen in class. We provide you with "experiments" that run your functions on random arrays and draw graphs of the runtimes. These graphs will allow you to get a feel for your functions' performance, and the situations in which one algorithm is more suitable than another. The graphs generated will be part of your submission.

The provided project skeleton contains several files that implement the plotting/graphing. You must add a new file `Sorter.java`. The file contains the class `Sorter` in which you must implement the below functions.

Listing 1: Sorter's public functions

```java
public static <T extends Comparable<T>> void quickSort(T[] array);

public static <T extends Comparable<T>> void mergeSortNoRecursion(T[] array);

public static void radixSort(Long[] array, int bitsPerDigit);
```

1. `quickSort`
   You should select the pivot by choosing a random element from the array. Think about how you can modify the pseudocode which always uses the right-most element of the sub-array as the pivot.
   Note: this function sorts an array of `T` objects which implement the `Comparable` interface (similar to the Heap homework).
2. `mergeSortNoRecursion`
   This implementation must not use any recursive calls, it should be purely iterative (loops). We've discussed this briefly in class. You can also recall the demo of recursive mergeSort from the recitation. In what order were the sub-arrays merged together? Think how to implement the same behaviour without recursion.
   Note: this function sorts an array of `T` objects which implement the `Comparable` interface (similar to the Heap homework).
3. `radixSort`
   This function should implement the radix sort algorithm you have seen in class. This implementation will be specialised for the case when the base is a power of two as follows. You may assume that the base is a power of two between 2^1 and 2^{20}. You may also assume that all numbers are non-negative (your code does not need to verify these assumptions).

   Listing 2: Radix sort's auxiliary function

   ```java
   static void countingSort(Long[] arr, int bitsPerDigit, int digitIndex);

   static int extractDigit(Long key, int bitsPerDigit, int digitIndex);
   ```

You should implement a private auxiliary method called `extractDigit`, which takes as input a `Long` number `key`, and returns the `digitIndex`'th digit of `key` when it is treated as a number in base $2^{\texttt{bitsPerDigit}}$. For example, if `bitsPerDigit` is 4, then the base to be used is $16 = 2^4$, and the 4 least significant bits of `key` represent the digit at index 0 (i.e., the least significant digit) in the base-16 representation of `key`. The following 4 bits represent the digit at index 1, and so on. The output digit should be returned to the user as an int. You have encountered bitwise operations in other courses, e.g., Systems programming in C. You must implement `extractDigit` using bit manipulation (e.g. shifting, masking, etc.) **Do not** use division, modulo, etc. in your implementation of `extractDigit`.

Next, you should implement the private `countingSort` auxiliary method which sorts an array of `Long`s. In this implementation of counting sort, the key of each element n in the array is not the entire number n, but rather the `digitIndex`'th digit of n when viewed as a number in base $2^{\texttt{bitsPerDigit}}$. This digit can be extracted by calling `extractDigit`.

Finally, you should implement `radixSort` which treats the `Long`s in the input array as numbers in base $2^{\texttt{bitsPerDigit}}$. By finding the maximum value in the input array, deduce the number of digits to be used in the radix sort. Implement radix sort with the number of digits you computed using the auxiliary function `countingSort` to perform the stable sort according to each digit.

When implementing `radixSort` it is strongly recommended to:

(a) Read and fully understand the pseudocode of `radixSort` and `countingSort`
(b) Write (and test) `extractDigit`.
(c) Write (and test) `countingSort`
(d) Finally, write (and test) `radixSort`.

Guidelines and Advice

You must **test each of your sorting functions thoroughly**. It's not enough to run each function on a few inputs, you must think of every possible edge-case and a (very) wide variety of inputs. Vary array length, vary distributions of values, sort arrays which are already sorted, sort very short arrays and exceptionally long arrays (to name a few cases).
If this sounds like overkill, be aware that standard library implementations of sorting functions sometime have bugs discovered years after implementation because small edge-cases weren't considered by the developers. **Sorting functions which fail for certain types of arrays will result in a large loss of points in the grade.**

Write code to perform this testing for you, manual testing will never find all bugs. Even though you should not submit the code that tests your functions, it is vital to have such a testing framework if you wish to get a good grade in this assignment.
Note: we provide a basic test and some useful utilities in a test file. You can run the file via Maven/your IDE just like in HW02. You can use this test as a template for additional tests (the tests can all go in this file). Your code will be tested against many additional tests which we do not provide. Be certain to write your own tests.

Functions should have the performance characteristics described in class. E.g., if your `mergeSort` is not $O(n \cdot \log(n))$ worst-case, something is very wrong and you will lose points.
Once you are satisfied that a function is correct, you can start optimising it. The same pseudocode can be implemented in many ways, some will be faster than others. These optimisations will not affect the asymptotic complexity of the functions, but they can make your functions faster by a constant factor.

Once you're satisfied that your functions are correct you should run the `main` method in `SortAnalyser.java`. This will create a folder `output` and render graphs of your functions' runtimes. The code can take a few minutes to run depending on your computer's performance. Be sure you inspect each of the graphs to see if the relative performance of your sorting functions aligns with your understanding. **The PNG files generated in the folder `output` should be included in the PDF you submit**. The PDF must be named `graphExplanation.pdf`. In the PDF, you must **include a short explanation/discussion of the experiments** in each graph. The total explanation does not need to be over a page (less is fine). Explain (if you can) why in some experiments algorithm x is faster than algorithm y, but in other experiments y is faster than x. The titles and labels of these graphs should be self-explanatory, but if you have any doubts you can read the code in `SortAnalyser.java` to see what each experiment really does.

Recommended (but optional): Create Project in Intellij IDEA

Intellij IDEA can be downloaded freely on (almost) any platform, we strongly recommend using it for this homework but it's perfectly fine to use any other IDE you like. The following instructions should get your project ready in IDEA, but the specific details can vary by platform/version. If you Google "Intellij IDEA import existing Maven project", you can find guides for your specific operating system.

1. Extract `HW5.zip` to a folder called `HW5`.
2. In IDEA, create a new "Project from existing sources", and choose the folder `hw05` (this folder can be found inside `HW5`).
3. Select the option "Import project from external model". Select "Maven" from the list.
4. Click "Create".

The Java file you write should be placed in the folder `src/main/java` (alongside the provided Java files like `SortAnalyser.java`). All your code should be in `Sorter.java`. Since the graphing code we provide has 3$^{\text{rd}}$-party dependencies **you must ensure you have imported the project correctly as a Maven project**. If you do not know how to do this, it was covered in Jon's office hours during HW02. Here's a link to the relevant part of the recording. **Failing to import correctly will prevent you from running the graphing code we provide, making it much harder to know if your code is performing as you expect** and making it impossible to submit the graphs generated by `SortAnalyser.java`.

Misc. Guidelines

- As in HW02, you will need to understand generics and the *Comparable* interface, and some other topics of Java. Generics and Interfaces are used in (almost) every non-trivial Java codebase, so it's worth understanding them well. If you feel comfortable with HW02, you are already prepared.
- You are allowed to:
  - Add new private functions to help you (and to keep the code clean and readable).
- You are not allowed to:
  - Remove or modify the signature of public functions in any class.
  - Rename classes or public members/fields.
  - Import anything unnecessary that might defeat the purpose of implementing sorting functions, the official solution has only one import in `Sorter.java` (*hint:* it's `java.util.Random`).

Submission Instructions

The assignment may be submitted in pairs, this is not mandatory but it is recommended.

Take the time to go over your code and ensure it is understandable, not excessively long, and well formatted. With a little thought, implementations can be kept reasonably short. Deviations from these guidelines will result in a point penalty.

Ensure your code compiles and runs. **Code which does not compile will not be graded.** Submit a single zip file containing only the following files (no additional files, folders, etc.):

- `Sorter.java`
- `graphExplanation.pdf`

The zip should be named according to the following format.
If submitting alone: `ID_FIRSTNAME-LASTNAME.zip`
E.g.: `1234_John-Doe.zip`

If submitting as a pair: `ID1_FIRSTNAME1-LASTNAME1_ID2_FIRSTNAME2-LASTNAME2.zip`
E.g.: `1234_John-Doe_5678_Jane-Doe.zip`

A jar file `submission_check.jar` is included with this assignment. You can tell if your submission (zip file) is properly named and formatted by running the following in a terminal/shell:

```
java -jar submission_check.jar example/path/123456_Jon-Simon.zip
```

Note: If your name consists of more than two parts, you can follow the below examples: `James-T-Kirk`, `Jean-Luc-Picard`, `Albus-Percival-Wulfric-Brian-Dumbledore`, etc.

Deadline

The deadline is the 31st of May. You may get an automatic extension up to the 4th of June.