Mandatory exercise 1 **INF102**

Deadline: 29th of September (midnight).

This assignment is an individual task, however you are allowed to collaborate and discuss solutions *as long as you do not share code* (see our policy on Collaboration and Cheating). Similarly, for tasks that are not answered with code, you may freely discuss your answers with your peers; but clearly, blindly transcribing answers from others is not allowed.

The assignment is graded on a scale from 0 to 100, and accounts for 10% of your final grade.

Organizational Instructions

First download the zipfile *Oblig1INF102(H19).zip* from files/Oblig 1 on mittuib. The zipfile contains the classes you are required to implement as well as the *Utilities* helper class which contains methods for reading input, generating random numbers, and for timing your program (necessary for Task 5).

The answers to code questions should go into the respective classes for each question (*UFwithDeunion.java* for Task 3, *Sort1.java* for Task 4b, *Sort2.java* for Task 4d, and *QuickSort.java* for Task 5). Note that classes should not be placed in a package.

The answers to non-code questions should be submitted in a pdf named {yourid}.pdf, where yourid is your UiB SEBRA account id. For example, if your id is abc123 the pdf should be named abc123.pdf. You can answer questions in either Norwegian or English.

Every file (the pdf + sourcefiles) should be placed in a folder named {yourid}, which should then be placed in a zipfile named {yourid}.zip. This file should then be handed in on mittuib.

If your submission does not follow these instructions (ie. we need to search for your code), you will be deducted up to 10 points.

Task 1 (20 points)

In the file *bigOQuiz.java*, you will find a number of functions *A(), B(), … , V()*. For each function create a table similar to the one below, where you enter the exact, tilde (~), and order of growth of each function.

Function	Exact	~	O()
A	2n + 1	~2n	O(n)
...			
V			

Task 2 (20 points)

a) An algorithm takes 0.5 ms for input size N=100. How long will it take for input size N=500 if the running time is the following (assume low order terms are negligible):

1) ~N
2) ~N log N
3) ~N^2
4) ~N^3

b) What is the largest instance size that can be solved in 1 minute for each of the running times in a) ?

Task 3 (20 points)

The *deunion* operation allows us to undo the last union operation (which has not yet been undone). For example if N=5 and we run union(0,1), union(1,2) and union(2,3) we will have the sets {0,1,2,3} and {4}. If we then run deunion() twice we will have the sets {0,1}, {2}, [3} and {4}.

a) Implement the class *UFwithDeunion* which supports the following public methods.

setSize(N)	set the total number of elements
find(int a)	return an identifier for the set containing a
union(int a, int b)	merge the set containing a with the set containing b
deunion()	undo the last union operation
elementsInSet(int a)	return the number of elements in the set containing a

Each method should run in time ~log N (except *setSize(n)* which can run in ~N).

b) Why does path compression make deunion hard?

Task 4 (20 points)

One possible way to sort an array containing N elements is to start by comparing elements in positions zero and one and exchanging them if they appear in incorrect order. This is then repeated with each consecutive pair of elements, first the elements in positions one and two, then two and three, and so on all the way up to N-2 and N-1. After repeating this N-1 times (on shorter and shorter parts of the array) the array will be completely sorted.

a) How many comparisons are necessary to sort an array in this way

b) Implement this sorting method for generic input arrays (i.e *sort(Comparable[] array)*). The name of the class should be *Sort1* and it must contain a method *sort(Comparable[] a)*.

Instead of letting every pass go from left to right, we can instead alternate the direction we go through the array after every pass (i.e left to right, right to left, left to right etc).

c) How many comparisons does this method use?

d) Implement this sorting method for generic arrays. The name of the class should be *Sort2* and it must contain a method *sort(Comparable[] a).*

Task 5 (20 points)

In the file *QuickSort.java* you will find a recursive implementation of quicksort. We can revise this implementation as follows. Let a, b and c be positive integers defined in the program and assume that the array contains N elements. Then if

$N < a$	use insertion sort instead of quicksort.
$a \leq N < b$	choose any element as pivot.
$b \leq N < c$	use the median-of-three pivot-selection scheme described in the textbook.
$c \leq N$	the pivot is the median of 9 elements.

Implement the following helper routines for quicksort:

- **a)** shuffle() perform an initial random shuffle of the input.
- **b)** insertionSort() perform insertion sort on the specified array.
- **c)** partitionBy3() use the median element of 3 elements for partitioning.
- **d)** partitionBy9() use the median element of 9 elements for partitioning.

See the file *QuickSort.java* for parameter lists for each routine.

e) Run experiments to determine the best values for a, b, and c when sorting an array of 1 000 000 (1 million) integers. Specifically, you can first determine the best value for a when using a random pivot for all N ≥ a. Then while keeping this value of a fixed, determine the best value for b while using partitionBy3() for all N ≥ b, and finally include c. In order to generate the input array use the function *nextRandomInt()* from the *Utilities* helper class.To get accurate and stable timings you should in each experiment let the program run at least 10 times and take the average time as your running time.

Document your choices for a, b and c by including running times for some of the values you tried.

Appendix: Style Guide

The following rules applies to all source code that is handed in. These guidelines are loosely based on Google Java Style Guide, though with some minor differences.

Most important

- Make your code easy to read.
  - Comment tricky parts of the code.
  - Use descriptive and logical variable names and function names.
  - Break the code into appropriate functions for readability and code reuse; no function should ideally be more than 30 lines of code (with the exception of extremely monotone code, such as sanity tests).
  - Write javadoc comments for functions that are not self-explanatory
- All files should use UTF-8 character encoding.

Also important

- The only whitespace characters allowed are spaces and newlines (tabs are not allowed).
- Each indentation level increases by 4 spaces.
- No line may exceed 120 characters - lines exceeding this limit must be line-wrapped.
- Some exceptions, e.g. very long URL's that don't fit on one line.
- File name should be the same as the name of the class, written in UpperCamelCase.
- Function names, parameter names and variable names should be written in lowerCamelCase.
- Constants should be written in ALL_CAPS.
- No line breaks before open braces ({).
- Blank lines should be used sparingly, but can be used to separate logic blocks and to increase readability.