Assignment 1: 3SUM and 4SUM

Riko Jacob and Matti Karppa

Hand-in date: 2022-09-20 09:59

1 Introduction

Your task is to recreate the experiments and the report as described in `report-example.pdf`. In the experiments, you will be comparing three different implementations to solve the 3SUM and 4SUM problems, as introduced in the first lecture. The assignment is to be solved alone. Create a report in LaTeXand return it as a single Zip file containing the report in PDF format and all code you have produced on LearnIT by the deadline. You may use your favorite programming language and environment for the implementation and the experimental framework, but the implementation examples will be provided Java and the experimental framework examples in Python. We recommend that you follow the examples and write the implementations in Java and create the framework in Python, use Visual Studio Code as your IDE, and use a UNIX command line for running the experiments, such as WSL on Windows, or the native command-line environment on Mac, and this is the only environment we will give support for, but you are free to choose any other environment. We recommend that you use Overleaf to write the report. The remainder of this instruction document contains detailed step-by-step instructions for recreating the experiments, and constructing the report.

2 Set up your environment (optional)

Follow the instructions on the LearnIT page under the title *Recommended software* to set up your environment. Note that since the course staff has no access to clean experimental systems other than those that they use for their daily work, the instructions have only been tested partially; if you encounter any problems, please ask for help at the Teams channel, or at the lab session.

While you are free to use any tools you wish, the recommended set of tools is WSL (for Windows), git, Python 3 + associated libraries (NumPy, Matplotlib) through Anaconda, Java 11, and Overleaf. The course staff will only help with these tools, using other tools is at your own risk.

3 Fork & clone the repository

Head over to `https://github.itu.dk/algorithms/ThreeFourSum2022` and *fork* the repository, creating a repository of your own under `https://github.itu.dk`. You will use this repository as basis for your experiments, and you are expected to store your code and your report there.

Clone the repository to your local computer using `git`. Typically, you would issue a command such as the following:

```
$ git clone git@github.itu.dk:algorithms/ThreeFourSum2022.git
                        ThreeFourSum2022
```

Of course, replace the URL with the URL of your personal fork. You can find out the URL by pressing the green button labeled "code".

4 Start working on your report in Overleaf

Under the repository, you will find a skeleton template for the report. Head over to `http://overleaf.itu.dk/`, log in (or sign up if you haven't done so already), and start a new project. If you choose a blank project, Overleaf may create a `main.tex` for you. You can delete that file; instead, upload the `report-skeleton.tex` you just downloaded with `git`.

Hit the green button that says *recompile*. On the right hand side, the compiled version of the report template. Adjust the author and title information of your new report, recompile your new report, and make sure that the changes become visible.

5 Create a new Gradle project for 3SUM

This section is optional, but recommended if you want to use Gradle for your project. Since we are starting to work on the 3SUM first, let us create a new Gradle project for the problem. Start by creating a directory called `threesum`, then initialize the directory as a Gradle project. We assume that we are operating under our repository.

```
$ mkdir threesum
  $ cd threesum
  $ gradle init

Select type of project to generate:
  1: basic
  2: application
```

```
  3: library
  4: Gradle plugin
Enter selection (default: basic) [1..4] 2

Select implementation language:
  1: C++
  2: Groovy
  3: Java
  4: Kotlin
  5: Scala
  6: Swift
Enter selection (default: Java) [1..6] 3

Split functionality across multiple subprojects?:
  1: no - only one application project
  2: yes - application and library projects
Enter selection (default: no - only one application project) [1..2] 1

Select build script DSL:
  1: Groovy
  2: Kotlin
Enter selection (default: Groovy) [1..2] 1

Generate build using new APIs and behavior (some features may
change in the next minor release)? (default: no) [yes, no] no
Select test framework:
  1: JUnit 4
  2: TestNG
  3: Spock
  4: JUnit Jupiter
Enter selection (default: JUnit Jupiter) [1..4] 1

Project name (default: threesum): threesum
Source package (default: threesum): threesum

> Task :init
Get more help with your project: https://docs.gradle.org/7.5/
samples/sample_building_java_applications.html

BUILD SUCCESSFUL in 27s
2 actionable tasks: 2 executed
```

This should create a directory structure such as the following:

```
./app/build.gradle
./app/src/test/resources
./app/src/test/java/threesum/AppTest.java
./app/src/main/resources
./app/src/main/java/threesum/App.java
./gradle/wrapper/gradle-wrapper.jar
./gradle/wrapper/gradle-wrapper.properties
./gradlew
./.gitignore
./.gitattributes
./.gradle
./.gradle/file-system.probe
./gradlew.bat
./settings.gradle
```

Obs. If you did not get an interactive menu as described above, it might be due to an old version of Gradle. Alternatively, you can issue the following command:

```
$ gradle init --type java-application --test-framework junit
```

This should yield the same file and directory structure.

Whichever way you did it, you should now have a skeleton project that you can build and run. Your implementation code will go to the file `./app/src/main/java/threesum/App.java`, which you can of course rename, and your test code goes to `./app/src/test/java/threesum/AppTest.java`. Now might be a good time to commit your changes in the repository since you want to keep all of these files, but not necessarily the files that will be built when you first build your code. You can do this by issuing the following command:

```
$ git add .
$ git commit -m "initial version"
```

The first command adds all files in the present directory in the commit, the second file actually stores the changes in the *local* repository. You also want to copy the changes over to the *remote* repository by issuing

```
$ git push
```

Let us now try to build our project. Running the following command will compile the code and run unit tests.

```
$ ./gradlew build
```

If the build was successful, try running the actual executable app by running

```
$ ./gradlew run
```

This should print `Hello World!` on the screen.

The filename `App.java` is a bit off, so let us rename it to `ThreeSum.java`. To rename a tracked file in `git`, we can issue the command[1]

```
$ git mv app/src/main/java/threesum/App.java
  app/src/main/java/threesum/Threesum.java
```

Likewise, let us rename `AppTest.java` to `ThreeSumTest.java`:

```
$ git mv app/src/test/java/threesum/AppTest.java
  app/src/test/java/threesum/ThreeSumTest.java
```

However, the build will now fail because Java expects the class name to match the filename. So open up the .java files and rename the classes `App` and `AppTest` to `ThreeSum` and `ThreeSumTest`, respectively, to match the filenames. Additionally, remember to change the class name `App` to `ThreeSum` on the first lines of the `main` method of `Threesum` and the `appHasAGreeting` method of `ThreeSumTest`. The build should succeed now

However, you still need to make one more change to fix running the application, as you will need to update the information about the main class by editing the file `app/build.gradle` and replacing the line

```
application {
    // Define the main class for the application.
    mainClass = 'threesum.App'
}
```

with

```
application {
    // Define the main class for the application.
    mainClass = 'threesum.ThreeSum'
}
```

With these changes, also `./gradlew run` should work again.

However, when running experiments we *really* don't want to use `./gradlew run`, but we want to create a self-contained JAR package. This can be done easily by adding a new target in `app/build.gradle`:

[1]This is a single line that is longer than the paragraph, not two commands.

```
jar {
    manifest {
        attributes 'Main-Class': 'threesum.ThreeSum'
    }
}
```

This instructs Gradle to build a JAR package that contains all dependencies, and whose main class is the same `ThreeSum` class as before. You can then build the JAR package by issuing

```
$ ./gradlew jar
```

This should create the JAR file under `./app/build/libs/app.jar`. You can then directly launch it by running

```
$ java -jar app/build/libs/app.jar
```

Check that your JAR package works.

6 Solve the 3SUM problem in three different ways

Let us recall the 3SUM problem. The problem statement[2] is as follows: Given a list of n numbers, are there any three numbers a, b, c in the list such that $a + b + c = 0$?

There are three basic solutions. The first one is to try all triplets (a, b, c) in the list, for example, by three nested for loops, and compute their sum. This solution is obviously $\Theta(n^3)$ in the number of elements. This solution is given in pseudocode in Algorithm 1.

A less obvious solution works by reducing the problem to the related 2SUM problem: iterate over all elements a of the list, and check if there exists a pair of elements b, c in the remainder that sum up to $b + c = -a$. The trick is that if we first sort the list, once the target value is known, we can perform a linear scan of the remainder: start by setting b next value from a, and c as the last value; since $a \leq b \leq c$, if $a + b + c < 0$, we know that we have to increase b to get closer to 0, and if $a + b + c > 0$, we need to decrease c, and if $c \leq b$, we know that there are no two values that sum up to $-a$. The complexity of this solution is: $\Theta(n \log n + n^2) = \Theta(n^2)$: when

[2]Note that there are several variants of the problem. Here we allow the solution to contain the same number multiple times, provided the list originally contained the number multiple times.

Algorithm 1 The cubic algorithm for 3SUM.

```
1: Input: a list of n integers x
2: Output: a triple a, b, c of integers in the list x such that a + b + c = 0 if
   such a triple exists, or none otherwise
3: function ThreeSumCubic(x)
4:     for i ← 1, 2, ..., n do
5:         a ← x[i]
6:         for j ← i + 1, i + 2, ..., n do
7:             b ← x[j]
8:             for k ← j + 1, j + 2, ..., n do
9:                 c ← x[k]
10:                if a + b + c = 0 then
11:                    return (a, b, c)
12:                end if
13:            end for
14:        end for
15:    end for
16:    return none
17: end function
```

n is large, the sorting does not factor into play, as the time is dominated by the quadratic search. This solution is given in pseudocode in Algorithm 2.

Another less obvious basic solution is to store all elements of the list in a hash map along with their indices in the original list. Then, iterate over each pair of elements (a, b) and query the hash map to see if $-a - b$ is in the list. Assuming the elements of the hash table can be accessed in $O(1)$ time, this solution is also $\Theta(n^2)$. This solution is given in pseudocode 3.

Observe that there is an extra check on line 13 of Algorithm 3 that the index j is strictly less than the index k returned from the hash map. Why is that? Hint: consider what happens if you have a list of two elements a, b satisfying $a + a + b = 0$. Why do we not need to check that $i < j$?

An algorithm is an abstract representation of the solution to a problem. In order to evaluate an algorithm in practice, it needs to be *implemented* in a programming language. We will give the implementations of Algorithms 1, 2, and 3 Java in Listings 1, 2, and 3, respectively.

Your task. Copy the code of the implementations of the algorithms into your project. If you want to use some other programming language than Java, you can use the code use as basis for creating an implementation of your own. Check that the code works by calling the routines with some suitable arguments.

Algorithm 2 The quadratic algorithm for 3SUM.

Input: a list of n integers x
Output: a triple a, b, c of integers in the list x such that $a + b + c = 0$ if such a triple exists, or *none* otherwise
function ThreeSumQuadratic(x)
    Sort(x)
    for $i \leftarrow 1, 2, \ldots, n$ **do**
        $a \leftarrow x[i]$
        $\ell \leftarrow i + 1$
        $r \leftarrow n$
        while $\ell < r$ **do**
            $b \leftarrow x[\ell]$
            $c \leftarrow x[r]$
            if $a + b + c = 0$ **then**
                return (a, b, c)
            else if $a + b + c < 0$ **then**
                $\ell \leftarrow \ell + 1$
            else
                $r \leftarrow r - 1$
            end if
        end while
    end for
    return *none*
end function

Listing 1: Implementation of the cubic algorithm in Java.

```
public static int[] threeSumCubic(int[] x) {
    int n = x.length;
    for (int i = 0; i < n; ++i) {
        int a = x[i];
        for (int j = i+1; j < n; ++j) {
            int b = x[j];
            for (int k = j+1; k < n; ++k) {
                int c = x[k];
                if (a + b + c == 0) {
                    return new int[] { a, b, c };
                }
            }
        }
    }
    return null;
}
```

Algorithm 3 The hash map algorithm for 3SUM.

Input: a list of n integers x
Output: a triple a, b, c of integers in the list x such that $a + b + c = 0$ if such a triple exists, or *none* otherwise
function ThreeSumHashMap(x)
    Initialize hash map H
    for $i \leftarrow 1, 2, \ldots, n$ **do**
        $H[x[i]] \leftarrow i$
    end for
    for $i \leftarrow 1, 2, \ldots, n$ **do**
        $a \leftarrow x[i]$
        for $j \leftarrow i + 1, i + 2, \ldots, n$ **do**
            $b \leftarrow x[j]$
            $k \leftarrow H[-a - b]$ ▷ Assume the hash map returns 0 if the element is not present
            if $k \neq 0$ **and** $j < k$ **then**
                return $(a, b, x[k])$
            end if
        end for
    end for
    return *none*
end function

Listing 2: Implementation of the quadratic algorithm in Java.

```
1  import java.util.Arrays;
2
3  public static int[] threeSumQuadratic(int[] x) {
4      int n = x.length;
5      int[] y = x.clone();
6      Arrays.sort(y);
7      for (int i = 0; i < n; ++i) {
8          int a = y[i];
9          int left = i+1;
10         int right = n-1;
11         while (left < right) {
12             int b = y[left];
13             int c = y[right];
14             if (a+b+c == 0) {
15                 return new int[] { a, b, c};
16             }
17             else if (a+b+c < 0) {
18                 ++left;
19             }
20             else {
21                 --right;
22             }
23         }
24     }
25
26     return null;
27 }
```

Listing 3: Implementation of the hash map algorithm in Java.

```
1  import java.util.HashMap;
2
3  public static int[] threeSumHashMap(int[] x) {
4      int n = x.length;
5      HashMap<Integer,Integer> H =
6          new HashMap<Integer,Integer>();
7      for (int i = 0; i < n; ++i) {
8          H.put(x[i], i);
9      }
10     for (int i = 0; i < n; ++i) {
11         int a = x[i];
12         for (int j = i+1; j < n; ++j) {
13             int b = x[j];
14             int c = -a - b;
15             Integer k = H.get(c);
16             if (k != null && j < k) {
17                 return new int[] { a, b, c };
18             }
19         }
20     }
21     return null;
22 }
```

7 Add tests to check the correctness of your implementation

When you implement algorithms, it is important to write tests to verify the correctness of your implementation. It might be a good idea to apply the paradigm of *Test-Driven Development (TDD)*: write some unit tests first, make sure that they fail, and then implement the missing functionality. It is important to check that the tests fail to make sure that the tests are actually evaluated.

If you created the Gradle project following instructions in Section 5, you can easily create unit tests by adding new methods in the `ThreeSumTests.java` file. The tests must be annotated with `@Test`. If you are unfamiliar with JUnit, have a look at the JUnit website[3] for some examples about

[3] `https://github.com/junit-team/junit4/wiki/Assertions`

assertions. Assertion makes a statement that should be true, and the test fails if the assertion does not hold. You are going to need at least the assertions `assertNull` to verify that a test case that should not return an solution indeed does return `null`, `assertNotNull` to verify that the method actually returns a solution, and `assertArrayEquals` to verify that the method returns the exact solution specified. The tests can be run by executing

```
$ ./gradlew test
```

Listing 4 shows how to check trivial cases with the cubic algorithm.

Listing 4: Simple assertions in JUnit.

```
1  package threesum;
2
3  import org.junit.Test;
4  import static org.junit.Assert.assertNull;
5  import static org.junit.Assert.assertNotNull;
6  import static org.junit.Assert.assertArrayEquals;;
7
8  public class ThreeSumTest {
9      @Test
10     public void testCubic() {
11         assertNull(ThreeSum.threeSumCubic(
12             new int [] { 1, 2, 3 }
13         ));
14         assertNotNull(ThreeSum.threeSumCubic(
15             new int [] { 1, 2, -3 }
16         ));
17         assertArrayEquals(new int[] { 1, 2, -3 },
18            ThreeSum.threeSumCubic(new int [] {
19                1, 2, -3
20            })
21         );
22     }
23 }
```

Your task. Verify the correctness of the routines given to you by writing suitable test cases. You should test all three routines, and with variable cases. Include cases where there are 0, 1, or multiple triplets summing to 0. Include cases where there are fewer than 3 integers in the list. Include a case where there is an empty list. Include cases where there are several equal integers and where all integers are distinct.

Create another version of the hash map implementation where you remove

the comparison (`&& j < k`). Find a case that provides incorrect output (reports an invalid triple), and write a test that verifies that the original implementation provides correct output and that the version without comparison does not.

Write one or two paragraphs in the report on how you tested your code for correctness.

8 Input/Output

Our stand-alone application is not very interesting unless it can run the desired algorithms with the given input. In order to do this, we need to specify how the algorithm is chosen, how input is passed, and how output is returned to the caller. We are going to do this as follows:

- The program will take one *command-line argument*[4] that selects the algorithm,
- The input is passed through *standard input*[5] in the following format: the first line will contain an integer that tells the number n of integers in the list, followed by another line that contains n space-separated integers,
- The output is produced to *standard output*[6] as follows: if a suitable triple (a, b, c) is found, the integers a, b, and c are written space separated into output, otherwise the string `null` is written to mark that no solution could be found.

Standard input can be accessed via `System.in` in Java, and reading such data is easy with the `Scanner` class. Listing 5 shows a Java routine that reads data in the format specified above, and returns an array if n `int`s. Note that this routine is not very error-tolerant, as it contains no kinds of safeguards against malformed input, and as such, is probably inappropriate for production code that is exposed to outside users.

[4] `https://en.wikipedia.org/wiki/Command-line_interface#Arguments`

[5] `https://en.wikipedia.org/wiki/Standard_streams#Standard_input_(stdin)`

[6] `https://en.wikipedia.org/wiki/Standard_streams#Standard_output_(stdout)`

Listing 5: A Java routine that reads the number n, followed by n integers from standard input and returns them as an `int` array.

```
1  import java.util.Scanner;
2
3  public static int[] readData() {
4      Scanner s = new Scanner(System.in);
5      int[] x = null;
6      try {
7          int n = s.nextInt();
8          x = new int[n];
9          for (int i = 0; i < n; ++i) {
10             x[i] = s.nextInt();
11         }
12     }
13     finally {
14         s.close();
15     }
16     return x;
17 }
```

Command-line arguments are simple to access in Java: they are contained in the `args` array that is always passed to the `main` method. We simply need to select the algorithm based on what is the first element of the array. We shall specify, that the first element must be `cubic`, `quadratic`, or `hashmap`, and select the algorithm and store the result as an array.

Finally, standard output is accessed by printing to `System.out`. We just need to check whether the array we got from our algorithm is `null`, and potentially use formatted printing to pretty-print our result. The full `main` method is shown in Listing 6. Again, the code is not very error-resistant, and specifically fails to address the cases where input is malformed, no command-line parameters are passed, or the command-line parameter does not match one of our specified choices.

Listing 6: Main method of our Java program that reads data from standard input, chooses the algorithm based on the command line parameters, and writes the output to standard output.

```
1  public static void main(String[] args) {
2      int[] x = readData();
3      int[] y = null;
4
5      if ("cubic".equals(args[0])) {
6          y = threeSumCubic(x);
7      }
8      else if ("quadratic".equals(args[0])) {
9          y = threeSumQuadratic(x);
10     }
11     else if ("hashmap".equals(args[0])) {
12         y = threeSumHashMap(x);
13     }
14
15     if (y == null) {
16         System.out.println("null");
17     }
18     else {
19         System.out.println(String.format("%d %d %d",
20             y[0], y[1], y[2]));
21     }
22 }
```

A potential session could look like following, assuming the project has been built as a JAR package using Gradle as suggested:

```
$ java -jar app/build/libs/app.jar cubic
3
1 2 3
null
$ java -jar app/build/libs/app.jar cubic
3
1 2 -3
1 2 -3
```

Your task. Implement the main method and any auxiliary functions you require, and check that you can run your self-contained program on the command-line, and that it produces the expected output.

9 Run Java applications from Python

We are going to build our experimental framework in Python. We will start by setting up the possibility of running Java applications. This can be done through the `subprocess` module. The module allows the creation of subprocesses which will independently run whatever code we wish, and we can communicate with the subprocess by feeding it data through the standard input, and receive data from standard output.

Listing 7 shows how to do this in Python. The global variable `TIMEOUT` sets a timeout at 30 seconds, after which an exception is raised if the process has not terminated. The special `PIPE` objects instruct the subprocess to receive and send data to the calling Python process; if these are not provided as arguments to the `Popen` function, the standard streams are connected to the calling process of the Python interpreter. Finally, instead of a main method, the convention in Python is to separate the executable application code from library code by an `if`-block. This block shows how the `run_java` function is expected to be called.

Assuming the code is placed one level above in the directory hiearchy from the Gradle project in a file, such as `experiments.py` or whatever you want to call the file, the expected output of the code is

```
null

1 2 -3
```

Your task. Copy the code from Listing 7 into a file, and experiment with it. Make sure you know how to call your algorithms from Python, and that you get the expected output.

Listing 7: Calling a Java application from Python using the `subprocess` module.

```
#!/usr/bin/env python3

import subprocess

TIMEOUT = 30

# run the given jar package,
# provide the given arg as the command-line
# argument,
# feed the given input string to the stdin of the
# process,
# and return the stdout from the process as string
def run_java(jar: str, arg: str, input: str)->str:
    p = subprocess.Popen(['java','-jar',jar,arg],
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE)
    (output,_) = p.communicate(input.encode('utf-8'),
        timeout=TIMEOUT)
    return output.decode('utf-8')

if __name__ == '__main__':
    print(run_java('threesum/app/build/libs/app.jar',
        'cubic','3\n1 2 3'))
    print(run_java('threesum/app/build/libs/app.jar',
        'cubic','3\n1 2 -3'))
```

10 Generating input data

The parameter the scaling with respect which we are interested in is the *number of elements* in a list n, but the routines we described take actual lists, not just the number of elements. Furthermore, we want the input to exhibit *worst case behavior* because our algorithms have an *early termination* property: the execution is terminated as soon as a positive example is found. We are usually interested in worst-case performance which in the case of 3SUM is the one where we cannot find a suitable triple. Furthermore, we want to be able to ensure that every time we run the experiments, we use the same data, so as to make our experiments reproducible.

Our plan is as follows: we are going to create lists of variable length n consisting of random *positive* integers. This ensures that there will not be a satisfying triple, and this is the worst case. We will make M repetitions for each value n, so there will be M different lists of n elements, for each n we will try.

We will fix some value $i_{\max}$ to be the number of different values of n that we will try, and we will use the following formula to determine the values that we will use:

$$n_i = 30 \cdot 1.41^i \,,$$

for $i = 0, 1, \ldots, i_{\max} - 1$. This means that every subsequent number of elements n_i is approximately a factor of $\sqrt{2}$ away from the previous value; this should provide a sufficient resolution for determining the scalability, while at the same time providing a sufficient range of different values.

For reproducibility, we are going to use a fixed *seed* that will initialize the random number generator to produce the exact same sequence of numbers each time. In addition, we are going to construct the input data beforehand, and we are going to provide the same input data to all algorithms.

Listing 8 shows how to do this using NumPy. NumPy is a very useful library that provides, in particular, advanced linear algebra functions. In this case, however, we use its faculties for constructing random numbers using a high-quality pseudorandom number generator (PRNG). The input will be stored in a *dictionary* called `INPUT_DATA`. The keys of the dictionary correspond to the values of n we are using that are stored in the list `NS`. Each value `INPUT_DATA[n]` is a list of M lists, each of which contains n `int`s. The maximum value is set at $2^{28} - 1$ to prevent negative values from occurring in the sums because of integer overflows.

Listing 8: Input generation in Python.

```python
from typing import List, Dict
import numpy as np # type: ignore

TIMEOUT = 30

# how many different values of n
I_MAX: int = 30
# the different values of n
NS: List[int] = [int(30 * 1.41**i) \
    for i in range(I_MAX)]
# how many repetitions for the same n
M: int = 5
# seed for the pseudorandom number generator
SEED: int = 314159
# the PRNG object
rng = np.random.default_rng(SEED)
# The generated input:
# The dictionary maps n to a list of lists
# each list contains M lists of n ints
INPUT_DATA: Dict[int,List[List[int]]] = {
    n : [rng.integers(1, 2**28, n) \
         for _ in range(M)] \
    for n in NS
}
```

Your task. Copy the above code to create input. Play with the different parameters to see how they interact with one another.

Caveats. Standard libraries may contain substandard pseudorandom number generators, and some algorithms are sensitive to the correlations that such poor random number generators produce! This is particularly true if one uses the `rand` function from the C standard library, as the typical implementations are very poor *Linear Congruential Generators* with short periods and large amounts of correlation between the numbers produced. Therefore, it may sometimes be necessary to be even aware of the properties of the random numbers generators that are being used to avoid systematic erros from creeping into the experiments.

11 Add framework for measuring runtimes

For making measurements, we will create a function that takes in as argument the choice of algorithm which will be passed to the Java application, and the input data for the particular run. As result, the function will return the number of seconds it took for the Java application to execute.

In Python, the current time can be recorded by using the `time` module, and specifically the function `time()`. To get the current time, one would call `time.time()`. The precise meaning of this value is implementation dependent, but on most systems it is a floating point number that records the number of seconds since 1970-01-01 00:00:00 (UTC), commonly known as Unix time.

Listing 9 shows a potential implementation of `measure`, followed by a potential invocation of the function. As the `run_java` function assumes the input is given as a string, we start by converting the input into the correct format. Then, we record the start time, run the Java application, and record the end time. The assertion on line 14 is to ensure that the expected result was provided by the application: if this assertion were to fail, we would know something is wrong.

Your task. Implement the measurement function by copying the code from Listing 9. Try varying the arguments to make sure you understand how it works. In particular, try varying the `TIMEOUT` parameter and try to see what happens when the timeout is exceeded.

A word of warning. We measure the wall clock time here, not CPU time, and the measurements are not particularly accurate. As they can only represent macroscopic time scales with reasonable accuracy, they are not suited for microbenchmarks. There are a multitude of reasons for this, starting from the fact that we are running our application under an operating system that implements pre-emptive multitasking: other load will be invariably present on the system and we cannot easily control how the operating system chooses to allocate resources for our program.

You should not rely on your measurements if the runtime of your function is too low; in practice, you should only rely on measurements when they are in the order of hundreds of milliseconds, or preferably seconds. And even then, you should always perform several iterations to average out random fluctuations. Note that the runtime also includes the time it takes to set up the virtual machine and the application, which will dominate the runtime for small inputs.

Also, it might be wise to perform sanity checks. If you expect your code to run for seconds, but the function returns nanoseconds or microseconds, something is off, and vice versa.

Listing 9: Implementation of the measurement function.

```
from typing import List
import time

def measure(algorithm: str, jar: str,
    input: List[int])->float:
    input_string: str = f'{len(input)}\n' + \
        ' '.join(map(str,input))
    start: float = time.time()
    result_string: str = run_java(jar, algorithm,
        input_string)
    end: float = time.time()
    assert result_string.strip() == 'null'
    return end - start

if __name__ == '__main__':
    print(measure('cubic',
        'threesum/app/build/libs/app.jar',
        INPUT_DATA[30][0]))
```

12 Performing benchmarks

Once you have the machinery set up for performing individual measurements, you should start planning for a set of experiments. In practice, since we are interested in the growth of the runtime of the algorithm, we want to (i) try different values of the parameter n, and (ii) perform several repetitions at each value of n to average out random fluctuations in the measurements.

The structure of the routine that we need should be something like the following: Iterate over each parameter value n, and for each n, iterate M times call the measurement function with the particular data associated with the (n, i) pair corresponding to the input size and the repetition i; if all M repetitions were successful, store the runtimes, but if timeout was exceeded, terminate and drop the results for the last parameter value. As a result, return a list of (n, t) pairs where we record the parameter value and the runtime.

An example function following this paradigm is seen in Listing 10. The list `results` will be populated with individual results. The results for a particular value of n are stored in the temporary list `result_n`, and all M values are stored in `results` only if timeout is not exceeded. If you played

with the timeout in the previous task, you should have noticed that exceeding the timeout will cause a `TimeoutExpired` exception to be raised. What the code does is catch this exception and terminate the benchmark. As such, it is possible that some values of n are not measured if smaller values caused timeout to be exceeded.

Listing 10: Implementation of the benchmarking of a single algorithm with various input sizes.

```
def benchmark(algorithm: str, jar: str)-> \
    List[Tuple[int,float]]:
    results: List[Tuple[int,float]] = list()

    for n in NS:
        try:
            result_n: List[Tuple[int,float]] = list()
            for i in range(M):
                input: List[int] = INPUT_DATA[n][i]
                diff: float = measure(algorithm,jar,
                    input)
                result_n.append((n,diff))
            results += result_n
        except subprocess.TimeoutExpired:
            break
    return results
```

Your task. Implement the benchmark function by copying the code of Listing 10. Check that you know how to call it.

Caveats. While we are happy with this structure for this particular assignment, there are some things that we need to be wary of. For example, we generate all our input beforehand and pass it to the subroutine at once. This works when the input is sufficiently small that the combined input across all iterations can fit the RAM of a computer at once, but in some cases this might be prohibitive from the point of RAM use. In such cases, the code should be modified such that the input is not provided as a simple input parameter, but it is generated or retrieved upon need before performing the individual measurement.

13 Putting it all together

We now have all the ingredients we need to actually perform the experiments. The plan is as follows: (i) iterate over a list of *instances* that determine which

algorithms to benchmark, (ii) run the `benchmark` function to obtain a list of (n, t) pairs, that is, problem size-runtime pairs, and (iii) store the values in a comma-separated values (CSV) file for further processing. CSV files are regular text files where observations are presented on rows, and individual values corresponding to an observation are separated by commas. The data can then be read with Pandas, NumPy, Excel, Matlab, or a multitude of other tools at ease.

Listing **??** shows how to do this. We start by opening the file `results.csv` for writing. We then use a `DictWriter` from the `csv` module to write a CSV file that includes a header for the fields. Then, we iterate over all of the instances we want to experiment with, obtain results using the `benchmark` function, and finally write the results row-by-row in the output file.

Listing 11: The full experiment code.

```
import csv

INSTANCES: List[Tuple[str,str]] = [
    ('cubic', 'threesum/app/build/libs/app.jar'),
    ('quadratic', 'threesum/app/build/libs/app.jar'),
    ('hashmap', 'threesum/app/build/libs/app.jar')
]

if __name__ == '__main__':
    with open('results.csv','w') as f:
        writer = csv.DictWriter(f,
            fieldnames = ['algorithm','n','time'])
        writer.writeheader()
        for algorithm, jar in INSTANCES:
            results: List[Tuple[int,float]] = \
                benchmark(algorithm,jar)
            for (n,t) in results:
                writer.writerow({
                    'algorithm' : algorithm,
                    'n' : n,
                    'time' : t
                })
```

Your task. Implement the full set of experiments using the subroutines developed in previous sections.

14 Computing statistics

The experimental framework developed in previous sections is self-contained and produces a single file, `results.csv`, as output. We can thus move to postprocessing the raw results into more useful data. We will start constructing a new Python file from scratch, so the following code will go to a separate file; that way, we do not need to run the experiments again every time we want to make changes to the postprocessing of the data.

We will start by computing some very basic statistics, such as the mean and standard deviation of the runtimes. Listing 12 shows how to read the CSV file, convert the results into a dictionary that uses the algorithm as a key to map to another dictionary that uses the parameter n as a key to map the parameter value to a list of measurements, and then uses Numpy functions to compute the mean and standard deviation. The result will be a dictionary that maps the algorithm name to a Numpy array of three columns. The first column will contain the problem size n, the second column will contain the averate runtime for that particular problem size, and the third column will contain the standard deviation.

Your task. Copy the code of Listing 12 in a separate file, such as `postprocess.py`, and make sure you understand how the data is stored and how to access the data.

Listing 12: Computing simple statistics.

```
#!/usr/bin/env python3

import csv
from typing import Dict, List
import numpy as np # type: ignore

def read_results(filename: str)-> \
    Dict[str,Dict[int,List[float]]]:
    results: Dict[str,Dict[int,List[float]]] = dict()
    with open(filename,'r') as f:
        reader = csv.DictReader(f)
        for row in reader:
            algorithm: str = row['algorithm']
            n: int = int(row['n'])
            t: float = float(row['time'])
            if algorithm not in results:
                results[algorithm] = dict()
            if n not in results[algorithm]:
                results[algorithm][n] = list()
            results[algorithm][n].append(t)
    return results

def compute_mean_std(raw: Dict[int,List[float]])-> \
    np.ndarray:
    result = np.zeros((len(raw),3))
    for i, n in enumerate(sorted(raw)):
        result[i,0] = n
        result[i,1] = np.mean(raw[n])
        result[i,2] = np.std(raw[n], ddof=1)
    return result

if __name__ == '__main__':
    raw_results: Dict[str,Dict[int,List[float]]] = \
        read_results('results.csv')
    refined_results: Dict[str,np.ndarray] = dict()
    for algorithm in raw_results:
        refined_results[algorithm] = \
            compute_mean_std(raw_results[algorithm])
```

15 Generating tables

In order to report our results to the public, we need to be able to present it in a human-readable form. For very particular results and fine-grained, we can use tables. We will practice this by presenting the results of all of our algorithms in tabular form.

As the report is written in LaTeX, it is natural to use the tabular facilities of LaTeXto include a table of results in the report. The syntax for creating such tables is a little unwieldy, though, so it makes sense to create a script that automagically generates the tables from the results. This also makes it easy to update the values if the experiments are rerun; doing this manually would be a lot of work.

Listing 13 how to create a LaTeXtabular from the results in Python. Following the example in the preceeding section, the function could be called by issuing the following statement:

```
write_latex_tabular(refined_results['cubic'],
        'threesum_cubic_tabular.tex')
```

One possible point that may need adjustment is the number of decimals to include in the string representation of the values. Here, the values are set to be represented at 6 decimals, and whether this is appropriate or not, depends on the range of values in the results.

Listing 13: Writing a LaTeXtabular from the results.

```
1  def write_latex_tabular(res: np.ndarray,
2      filename: str):
3      with open(filename, 'w') as f:
4          f.write(r'\begin{tabular}{rrr}' + '\n')
5          f.write(r'$n$ & Average (s) & ' + \
6              'Standard deviation (s)')
7          f.write(r'\\\hline' + '\n')
8          for i in range(res.shape[0]):
9              fields = [str(int(res[i,0])),
10                 f'{res[i,1]:.6f}',
11                 f'{res[i,2]:.6f}']
12             f.write(' & '.join(fields) + r'\\'+'\n')
13         f.write(r'\end{tabular}' + '\n')
```

Your task. Include a LaTeXtable in your report. In both cases, the resulting table can be included in the LaTeXdocument by

```
\input{threesum_cubic_tabular.tex}
```

The `\input` command simply pastes whatever content the file has to the place where the command is placed. This should be done within the `table` environment to enable labels and captions for the table. The source code of the `report-skeleton.tex` file shows how to do this.

16 Making plots

While tables provide an exact listing of the data, it may be difficult to see larger trends in a table of numbers. Visualizing the data by plotting can be a useful tool.

Listing 14 shows how to plot the data using `matplotlib` which is the de facto standard plotting library for Python. The code in `matplotlib` is organized in terms of *figures* that contain one or more *axes*. The function `subplots()`, without any arguments, creates a new figure with a single axis. The axis object does the heavy lifting of creating the actual plots. The `errorbar` function can be used to create a lineplot of the average values with error bars corresponding to the standard deviations of the observations. Observe the order of the arguments: the x values, the y values (averages), and the y-axis error values (standard deviations). We also include markers for the actual data points. It is important that the figure is saved in PDF format because this preserves the vector structure of the plot, and can be included in PDF formatted documents without loss of quality.

Note that the resulting plot has a logarithmic scale on the x and y axes. This means that, rather than increasing by a constant amount, the values increase in powers along the axes. This is justified in the case of this assignment because the difference in the growth of the runtime of the different algorithms is so large that otherwise the plot of the faster, quadratic algorithm would amount to little more than a straight line. You are welcome to try out what happens if the scale is linear! Furthermore, by the rules for logarithms, we know that $\log n^k = k \log n$, so this means that if the x axis is logarithmic as well, the power k should match the slope as n grows.

Your task. Create a plot that shows the runtimes of the algorithms as a function of n on a logarithmic scale. Include the resulting PDF in your report.

Listing 14: Plotting with Matplotlib.

```
import matplotlib.pyplot as plt # type: ignore

def plot_algorithms(res: Dict[str,np.ndarray],
    filename: str):
    (fig, ax) = plt.subplots()
    algorithms = ['cubic', 'quadratic', 'hashmap']
    for algorithm in algorithms:
        ns = res[algorithm][:,0]
        means = res[algorithm][:,1]
        stds = res[algorithm][:,2]
        ax.errorbar(ns, means, stds, marker='o',
            capsize = 3.0)
    ax.set_xlabel('Number of elements $n$')
    ax.set_ylabel('Time (s)')
    ax.set_xscale('log')
    ax.set_yscale('log')
    ax.legend(['Cubic algorithm',
        'Quadratic algorithm', 'Hashmap algorithm'])
    fig.savefig(filename)
```

17 Extending your framework to 4SUM

You should now have all the tools for conducting an experiment. It is now time to extend your framework to a related, but slightly more challenging problem: the 4SUM. The problem is almost the same: given a list of n integers, do there exist four integers in the list a, b, c, and d, such that $a + b + c + d = 0$?

Algorithms 4, 5, and 6 provide pseudocode for three different algorithms that mimic those of 3SUM, but adapted for the 4SUM problem. Note that when implementing the code, particularly that of Algorithm 6, some care must be taken, as the hash map maps integers to pairs of integers; you will probably need to store arrays in the Java `HashMap`.

Your task. Implement Algorithms 4, 5, and 6. In your report, discuss the running times and tradeoffs between the algorithms: how do they scale theoretically in terms of runtime and space complexity? Test your implementation for correctness. Perform a similar benchmark as you did with 3SUM to compare the runtimes of these algorithms. Produce a plot that shows the runtimes of the algorithms as a function of n in a logarithmic scale.

Algorithm 4 The quartic algorithm for 4SUM.

```
1: Input: a list of n integers x
2: Output: a quadruple a, b, c, d of integers in the list x such that a + b +
   c + d = 0 if such a triple exists, or none otherwise
3: function FourSumQuartic(x)
4:     for i ← 1, 2, ..., n do
5:         a ← x[i]
6:         for j ← i + 1, i + 2, ..., n do
7:             b ← x[j]
8:             for k ← j + 1, j + 2, ..., n do
9:                 c ← x[k]
10:                for ℓ ← k + 1, k + 2, ..., n do
11:                    d ← x[ℓ]
12:                    if a + b + c + d = 0 then
13:                        return (a, b, c, d)
14:                    end if
15:                end for
16:            end for
17:        end for
18:    end for
19:    return none
20: end function
```

18 Finish up the report

We have now described all the necessary tools and tasks that you need to write your report. Use the file `report-skeleton.tex` as a starting point.

Return your assignment on LearnIT by the handin time as a Zip file containing a single PDF report and all of the code you produced.

Algorithm 5 The cubic algorithm for 4SUM.

```
1: Input: a list of n integers x
2: Output: a quadruple a, b, c, d of integers in the list x such that a + b +
   c + d = 0 if such a triple exists, or none otherwise
3: function FourSumQuartic(x)
4:     Sort(x)
5:     for i ← 1, 2, ..., n do
6:         a ← x[i]
7:         for j ← i + 1, i + 2, ..., n do
8:             b ← x[j]
9:             ℓ ← j + 1
10:            r ← n
11:            while ℓ < r do
12:                c ← x[ℓ]
13:                d ← x[r]
14:                if a + b + c + d = 0 then
15:                    return (a, b, c, d)
16:                else if a + b + c + d < 0 then
17:                    ℓ ← ℓ + 1
18:                else
19:                    r ← r − 1
20:                end if
21:            end while
22:        end for
23:    end for
24:    return none
25: end function
```

Algorithm 6 The hash map algorithm for 4SUM.

Input: a list of n integers x
Output: a quadruple a, b, c, d of integers in the list x such that $a + b + c + d = 0$ if such a triple exists, or *none* otherwise
function FourSumHashMap(x)
    Initialize hash map H
    for $i \leftarrow 1, 2, \ldots, n$ **do**
        for $j \leftarrow i + 1, i + 2, \ldots, n$ **do**
            $H[x[i] + x[j]] \leftarrow (i, j)$
        end for
    end for
    for $i \leftarrow 1, 2, \ldots, n$ **do**
        $a \leftarrow x[i]$
        for $j \leftarrow i + 1, i + 2, \ldots, n$ **do**
            $b \leftarrow x[j]$
            $(k, \ell) \leftarrow H[-a - b]$ ▷ Assume the hash map returns a pair of integers, and $(0, 0)$ if the element is not present
            if $(k, \ell) \neq (0, 0)$ **and** $j < k$ **then**
                $c \leftarrow x[k]$
                $d \leftarrow x[\ell]$
                return (a, b, c, d)
            end if
        end for
    end for
    return *none*
end function