4.
Logging is an important part of debugging, which is hard to achieve on mobile devices, where application development and execution take place on different systems.
Android includes a framework that provides centralized logging for the entire environment, including applications and the Android system.
The logging framework includes the logd daemon, the liblog library, the framework API classes and the logcat tool.
The logd daemon is used from Android 5.0, and acts as a centralized user-mode logger.
It uses 4 sockets: one for control, one for reading, one for writing and a netlink socket for the integration with SELinux. These sockets are the only way to communicate with logd.

5.
The sockets are not accessed directly, but through the liblog library. So the applications use the Log class to send log messages, which in turn calls the liblog library through JNI, that opens the /dev/socket/logdw socket and writes the log message.
For reading logs, logcat will connect to the /dev/socket/logdr through the liblog API and instruct the LogReader instance of logd to provide the logs by sending parameters to it.

6.
A log message has the following format:

1) Priority - that indicates the severity of a log message. We have 6 priorities - verbose, debug, info, warning, error and assert.
2) Tag - which identifies the software component that generated the message. It can be any string. Logcat can use this tag to filter messages. The tag should not be too long.
3) The actual message. At the end of each messages a newline character is automatically added.

7.
In the Java code you can log messages by using the Log class:

```
Log.v(String tag, String msg) -> verbose
▯Log.d(String tag, String msg) -> debug
▯Log.i(String tag, String msg) -> information
▯Log.w(String tag, String msg) -> warning
▯Log.e(String tag, String msg) -> error
▯Log.wtf(String tag, String msg) -> assert
▯Log.println(int priority, String tag, String msg) -> any priority
```

For example, if you call

```
Log.i("MyActivity", "Get item number " + position);
```

Logcat will display the following line:

```
I/MyActivity( 1557): Get item number 1
```

8.
The logging API for the native code is provided through android/log.h. Therefore, if we need to use the logging functions, we need to include this header file in the native code.

In addition, we need to modify the Android.mk for dinamically linking the native code with the logging library. We will configure the build system variable LOCAL_LDLIBS to include -llog. We need to do this before include $(BUILD_SHARED_LIBRARY) in order to obtain the necessary effect.

9.
We have a set of logging functions for the native code:
1) __android_log_write - that will generate a simple text message, without special formatting. It will receive as parameters: the priority, tag and text message.
2) __android_log_print - that will generate a message by using a formatted string, exactly like printf (the formatting syntax is similar). It will receive as parameters: the priority, tag, string format and a variable number of parameters, as specified in the format.

10.
3) __android_log_vprint - is similar with the previous one, but will receive additional parameters in a va_list, instead of a succession of parameters. It is useful when we want to display all parameters received by the current function. In this example, we have a va_list variable that includes all parameters of the function log_verbose.
4) __android_log_assert - is used for logging an assertion failure. It will not receive the priority as parameter, because it will automatically be set to “assert”. If we have a debugger attached, a SIGTRAP signal will be sent to the process, in order to allow the inspection through the debugger. The parameters are: the condition, tag and message.

11.
Logcat is a command that displays log messages. You can either run it in the command line through adb, or from Android Studio graphical interface.
You can set the log level, search through logs and apply various filters.
A log is displayed in this format: date time PID-TID/package priority/tag: message
For example: 12-10 13:02:50.071 1901-4229/com.google.android.gms V/AuthZen: Handling delegate intent.

12.
Usually, we want to have some messages in the debug phase and others in the release phase. But the logging API does not provide the possibility to exclude certain messages based on priority. Here, I present a method based on the preprocessor that will display only certain log messages (from a certain priority upwards).
We create a header file. First, we define a macro MY_LOG_PRINT that displays a message with all possible details: file, line, etc.
Then, we define conditions for each level (here we have an example for “warning”). If the actual logging level is less or equal to warning, then we display the message with the previous macro. Otherwise, we execute the macro MY_LOG_LOOP that does nothing.
We will have such conditions for each priority.
These are wrappers over the logging functions, that allow us to select only some priorities at compile time.

13.

After defining that header file, we will include it in the native code and will use the macros for logging messages of a certain priority.
In addition, we need to update Android.mk in order to specify the tag, priority level and to apply the configuration. For specifying the tag, we have MY_LOG_TAG:=\”my_native_code\”
For specifying the priority level depending on the type of build (release or debug), we can use this code. The build system variable APP_OPTIM specifies the type of build. If it is release, only the messages with the priority error and upward. Otherwise, all log messages will be displayed.
Finally, we need to add the following compiling flags in order to enable this configuration.

14.
The descriptors STDOUT and STDERR are not visible by default on Android (the output is sent to /dev/null).
For redirecting the messages sent to STDOUT and STDERR to the logging system, we need to execute the following commands. After restarting the application, those messages will be displayed in logcat with the tag stdout and stderr (with priority Information).
This configuration is temporary and will be enabled only until the device is rebooted. For making this configuration permanent, we need to modify /data/local.prop on the device.

16.
For investigating the state of an application, we can use a debugger.
Android NDK provides support for debugging native applications through GNU Debugger (GDB).
The ndk-gdb script can be used for managing error conditions and displaying error messages.
For using GDB, we need to:
1) Compile the native code using ndk-build or Android Studio. The NDK build system generates certain files that make debugging possible.
2) In the manifest file, we need to set the attribute android:debuggable to be true in the application tag.
3) We need to use at least Android 2.2. The previous versions do not support debugging.

17.
The ndk-gdb script will configure the debugging session. It will go through the following steps:
1) ndk-gdb launches the application through the Activity Manager, through ADB. The Activity Manager will send a request to Zygote to create a new process. Zygote forks itself to create the new process. The application is now running.
2) After obtaining the PID of the new process, ndk-gdb will start the GDB server and will attach it to the application
3) The script will configure port forwarding through ADB to make the GDB server accessible to the host system.
4) Then, it will copy the binaries for Zygote and shared libraries on the host system.
5) Finally, it will start the GDB client on the host system. In this moment, the debugging session is active and can receive commands. The commands are sent through the debug port.

18.
On this slide, we have an example on how to use ndk-gdb from the command line.
1) In the project directory, we remove all generated files and directories: bin, obj, libs.
2) We compile the native code using ndk-build
3) We need to use the command “android update project -p” in order to obtain the file build.xml
4) We need to compile and package the whole project in debugging mode using “ant debug”
5) We install the application using “ant installd”
6) We start the application and attach the debugger with “ndk-gdb --start”
7) After the debugger has been attached, we have the GDB prompt. Then we can run commands.

19.
These are some useful GDB commands:
1) break - for creating a breakpoint at a certain function (function name / file name + line number)
2) clear - to delete all breakpoints
3) enable/disable/delete - operations with breakpoints
4) next - runs the next line of code
5) continue - continue execution
6) backtrace - display the call stack
7) backtrace full - display the call stack and the local variables
8) print - display a variable, expression, memory address, register
9) display - continues to print the value after each step
10) info thread - lists running threads
11) thread - selects a certain thread

20.
Android Studio includes a debugger that allows you to debug applications running on the emulator or on real devices. It can be accessed by pressing the Debug button in the toolbar. It allows you to select the device running the application that you want to debug. You can use it to set breakpoint in both Java and native code. It allows you to inspect variables and expressions at runtime. In addition it can capture screenshots or record videos of the running application.
Android Studio runs LLDB debugger in the Debug window for debugging native code.

21.
This is a screenshot of the Android Studio Debugger. It shows the current thread and the object tree for a certain variable.

22.
For adding a breakpoint you need to locate the line of code and press Ctrl+F8. Then you should click on “Attach debugger to Android process” in order to update the running application.

When code execution reaches a certain breakpoint, you can examine the object tree for a certain variable, evaluate an expression, move to the next line of code, move to the first line inside a method call, move to the next line outside the current method, and continue the normal execution of the application.

23.
Dalvik Debug Monitoring Server (DDMS) is SDK tool for Android application debugging.
It is integrated in Android Studio and can be accessed by launching Android Debug Monitor and then clicking on DDMS.
It can be easily used for both emulator and real devices. It can also be used in the command line from tools/ddms in the SDK.
It includes multiple functionalities such as: port-forwarding, screen capture, thread info, heap info, process state, radio state, incoming call, SMS spoofing, location spoofing, etc.

24.
When started, DDMS connects to ADB. When a device is connected, a virtual machine monitoring service is created between ADB and DDMS. The service will notify DDMS when a virtual machine is started or terminated.
When a VM is started, DDMS obtains the PID and opens a connection to the VM debugger through the ADB daemon (adbd) that runs on the device. Then DDMS can communicate with the VM by using a custom wire protocol.

25.
This is a screenshot of DDMS running in the Android Device Monitor (in Android Studio).

26.
DDMS allows us to see how much heap memory is used by a certain process on the Android device:
1) Select a process from the Devices tab
2) Click on the Update Heap to enable the collection of information about the heap
3) Click on Cause GC from the Heap tab to run the Garbage Collector. This way, we will have only the currently used objects (their type, amount of memory allocated for each type)
4) If we click on a certain type of object, we will see the number of allocated objects of that type.

27.
Here you have a screenshot of DDMS showing the Devices and Heap tabs. In the Devices tab you can select a process and in Heap tab you can see all types of objects and the occupied heap memory.

28.
DDMS can track memory allocations to see which classes and threads allocate objects.
Real-time tracking allows the detection of memory allocation problems.
1) Select the process from the Devices tab
2) Click on “Start Tracking” from the Allocation Tracker tab
3) Click on “Get Allocations” to see which objects have been allocated in the meanwhile

4) Click on “Stop Tracking” in the end
We can obtain information about the exact method, file and line where the objects have been allocated.
DDMS can also be used to obtain information about the threads running in a certain process:
1) Select the process from the Devices tab
2) Click on “Update Threads”
3) In the Threads tab you can see information about the running threads.

29.
This is a screenshot of DDMS, including the Devices and Allocation Tracker tabs. In the Devices tab you select a process and in the Allocation Tracker tab you can see the objects that have been allocated with information about the method, file and line.

30.
DDMS also provides information about the network traffic generated by applications. You can distinguish between different types of traffic by assigning a tag to a socket before using it (using the TrafficStats class - tagSocket() and untagSocket()). In the screenshot we can see the network traffic for different tags.

32.
In addition to debuggers, we can use several tools for troubleshooting in order to identify the cause of a certain problem.
When a native application crashes, a tombstone containing a large amount of information about the crashed process is generated and saved in the directory /data/tombstones/. The information is also displayed in logcat.

33.
A tombstone includes: the build fingerprint, PID, TIDs, and process name, signal and fault address, CPU registers, call stack, stack content of each call.
In a call stack, the lines starting with # indicate a call. The line that starts with #00 is the place where it crashed and the next lines are the previous call functions. After #00, we can see the address where the error has occurred (the program counter).

34.
This is an example of tombstone. You can see the build fingerprint, PID, TID, process name, signal and fault address, registers, call stack, and contents of the stack.

35.
We can use ndk-stack in order to translate the stack trace by adding filenames and line numbers.
We need to use the command: “adb logcat | $NDK/ndk-stack -sym $PROJECT_PATH/obj/local/armeabi”.
This way, we can see the exact file and line where there the program crashed.
This is an example of stack trace that has been translated in a more readable form using ndk-stack.

36.
Android provides an mode in which all calls through JNI are verified, that is called CheckJNI. To enable this mode on a certain device, we need to use the following commands, which are different for rooted and regular devices. In logcat, we will observe a line that tells you that CheckJNI has been turned on.
When CheckJNI detects a possible error, it generates a log message that starts with “JNI WARNING”.

37.
The libc debug mode can be enabled for troubleshooting problems with the memory.
To enable this mode we need to use these commands. The value of the parameter can be 1, 5 or 10.
1 - detects memory leaks
5 - fills the allocated memory in order to detect overruns
10 - fills the allocated memory and adds a sentinel to detect overruns.
Then a problem is detected by the libc debug mode, it will generate an error message and terminate program execution. In addition, it will display a call stack. Here you have an example where a memory allocation error is detected.

38.
Valgrind can be used for advanced memory inspection. It is an open-source tool for memory debugging, memory leaks detection and profiling.
Valgrind provides support for Android. For this, we need to build Valgrind from sources, and the binaries will be in the Inst directory. Then we need to put the binaries on the target device using adb push Inst /data/local/. We have to configure execution permissions for the binaries on the device. We also need a helper script that includes the following. We put this script on the device in /data/local/Inst/bin and give it execution permissions.

39.
For running an application under Valgrind, we need to inject the script in the startup sequence, with adb shell setprop wrap.com.example.testapp "logwrapper /data/local/Inst/bin/valgrind_wrapper.sh". This command will set the property warp.packagename to logwrapper and the path to the script.
This way, we can visualize with logcat all error messages that are generated by Valgrind when running the application.

40.
Strace is a tool for intercepting system calls made by a certain application and the signals that are receive by it.
For each system call, it will display the name, arguments and returned value.
It is very useful in analyzing closed source programs when investigating their behavior.
The Android emulator includes strace, so it can be used directly. There are versions of strace that can run on devices, but you need to copy the binary on the device first.
First, we need to obtain the PID by using the ps command, then we run strace on that PID and it will display all system calls made by that process.

42.
Profilers are used for obtaining information about the performance of a component.
GNU Profiler (gprof) is a Unix-based profiling tool. It uses instrumentation and sampling to obtain the time spent in each function.
Instrumentation is usually made at compile-time by gcc, when the option -pg is used. During execution, the collected sampling data will be stored in the file gmon.out which will be used by gprof to generate profiling reports.
Although Android NDK includes the gprof tool, the NDK toolchain does not include the implementation of the function __gnu_mcount_nc which is necessary for measuring the time spent in a function.
So, for using gprof on native code we can use an open source project called Android NDK Profiler.

43.
To install Android NDK Profiler, we need to download the zip archive, unzip it, move the contents to NDK_HOME/sources and rename the directory to android-ndk-profiler.
To enable and use the profiler:
1) Update Android.mk to statically bind the previously installed library
2) In the native code, include the header file prof.h if the profiler is enabled (this is a variable defined in Android.mk)
3) To start collecting data for profiling we must call the function monstartup and give it the library name as argument (our library, the one that is analysed) -> to verify all memory addresses of the library functions.
4) To stop data collection we use the function moncleanup.

44.
After the execution, the collected data is stored by default in /sdcard/gmon.out. For this, the application needs write permissions for the sdcard.
The next step is to copy the file gmon.out from the sdcard to the host system.
Then we will run gprof (the binary from the Android NDK toolchain) and give it as argument our library (the analysed one) and the file gmon.out.
Gprof will analyse the data and generate a report with the following sections: flat profile and call graph. For each function, it computes the time spent in that function and how many times it was called and the call order of the functions.

45.
Systrace is a tool for analyzing the performance of an application. It collects system and application execution data and generates interactive reports. It helps you identify the performance issues and improve performance.
It shows the system and applications on a common timeline. It collects a large amount of data including: execution time, CPU load, CPU frequency, disk input and output activities, threads, etc.

46.
For running Systrace, you need to:

1) Have a device with Android 4.1 (API 16) or higher
2) Enable Developer Options and USB Debugging
3) Root access on the device
You can use it from the command line or from the graphical user interface. This is an example on how to record the trace from the CLI. The command generates a html file, which can be opened in a browser.

47.
You can analyse the trace generated by Systrace in order to identify the performance problems in your application: you can inspect frames and look for long running-frames (that take more than 16.6 millisecond - yellow and red). The trace also gives you alerts that may indicate the performance problems.
Here is a screenshot of systrace. You can see the yellow and red frames that are frequent for this application.

48.
The tracing signals used by systrace may not capture everything the application is doing. Sometimes you may need to add your own tracing signals to the code. From Android 4.3, you can add instrumentation to your code by using the Trace class.
You will start instrumentation by calling Trace.beginSection() and end it by calling Trace.endSection().
You can have nested trace calls, endSection() will end the most recent beginSection(). However, it is mandatory to end the trace on the same thread in which it was started.

49.
Traceview is another tool that allows you to visualize execution logs that are created when using the Debug class to log traces in your code. It is used for both debugging and profiling.
Traceview displays the logged traces in two panels:

- The timeline panel - which displays the start and end of each thread and method
- The profile panel - which provides information about the time spent in each method

50.
There are two ways to generate the trace logs:

- Use the startMethodTracing() and stopMethodTracing() of the Debug class. This method is precise because you specify exactly where the tracing starts and stops.
- Use the method profiling feature of DDMS. This is less precise because you have no control over where the logging starts and stops. This is useful when you don't have access to the source code of the application or if you don't need precise log timing.

This is an example on how to use the Debug class to start and stop tracing.

51.
This is a screenshot of the timeline panel. Each row represents a thread and the horizontal axis is time. Each method is represented with a different color. The thin lines below the first row represent the extent of all calls to that method.

52.
This is a screenshot of the profile panel, which includes the time spent in each method (both inclusive and exclusive times). Exclusive time is the time actually spent in the method. The inclusive time is the time spent in the method plus the time spent in the called methods.
The last column displays the number of calls to this method plus the number of recursive calls.