Optimize your apps to make them
stable, efficient, and fast

Pro
Android Apps
Performance Optimization

Hervé Guihot

Apress®

Pro Android Apps Performance Optimization

Hervé Guihot

Apress®

Pro Android Apps Performance Optimization

ISBN-13 (pbk): 978-1-4302-3999-4

ISBN-13 (electronic): 978-1-4302-4000-6

President and Publisher: Paul Manning
Lead Editor: James Markham
Technical Reviewer: Charles Cruz, Shane Kirk, Eric Neff

Coordinating Editor: Corbin Collins
Copy Editor: Jill Steinberg
Compositor: MacPS, LLC
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

Any source code or other supplementary materials referenced by the author in this text is available to readers at `www.apress.com`. For detailed information about how to locate your book’s source code, go to `http://www.apress.com/source-code/`.

Contents at a Glance

Contents ... iv
About the Author ... viii
About the Technical Reviewers ... ix
Acknowledgments ... x
Introduction ... xi
Chapter 1: Optimizing Java Code ... 1
Chapter 2: Getting Started With the NDK ... 33
Chapter 3: Advanced NDK ... 73
Chapter 4: Using Memory Efficiently ... 109
Chapter 5: Multithreading and Synchronization ... 133
Chapter 6: Benchmarking And Profiling ... 163
Chapter 7: Maximizing Battery Life ... 177
Chapter 8: Graphics ... 207
Chapter 9: RenderScript ... 231
Index ... 265

Contents

Contents at a Glance ... iii
About the Author ... viii
About the Technical Reviewers ... ix
Acknowledgments ... x
Introduction ... xi

Chapter 1: Optimizing Java Code ... 1
How Android Executes Your Code ... 2
Optimizing Fibonacci ... 4
From Recursive To Iterative ... 5
BigInteger ... 6
Caching Results ... 11
android.util.LruCache<K, V> ... 12
API Levels ... 13
Fragmentation ... 16
Data Structures ... 17
Responsiveness ... 20
Lazy initializations ... 22
StrictMode ... 23
SQLite ... 25
SQLite Statements ... 25
Transactions ... 28
Queries ... 30
Summary ... 31
Chapter 2: Getting Started With the NDK ... 33
What Is In the NDK? ... 34
Mixing Java and C/C++ Code ... 37
Declaring the Native Method ... 37
Implementing the JNI Glue Layer ... 38
Creating the Makefiles ... 40
Implementing the Native Function ... 41
Compiling the Native Library ... 43
Loading the Native Library ... 43

Application.mk 44
Optimizing For (Almost) All Devices 46
Supporting All Devices 47
Android.mk 50
Performance Improvements With C/C++ 53
More About JNI 57
Native Activity 62
Building the Missing Library 64
Alternative 70
Summary 71
■ Chapter 3: Advanced NDK 73
Assembly 73
Greatest Common Divisor 74
Color Conversion 79
Parallel Computation of Average 82
ARM Instructions 87
ARM NEON 96
CPU Features 97
C Extensions 98
Built-in Functions 99
Vector Instructions 99
Tips 103
Inlining Functions 104
Unrolling Loops 104
Preloading Memory 105
LDM/STM Instead Of LDR/STD 106
Summary 107
■ Chapter 4: Using Memory Efficiently 109
A Word On Memory 109
Data Types 111
Comparing Values 113
Other Algorithms 115
Sorting Arrays 116
Defining Your Own Classes 117
Accessing Memory 118
The Cache's Line Size 119
Laying Out Your Data 120
Garbage Collection 125
Memory Leaks 125
References 127
APIs 131
Low Memory 131
Summary 132
■ Chapter 5: Multithreading and Synchronization 133
Threads 134
AsyncTask 137
Handlers and Loopers 140

Handlers 140
Loopers 142
Data Types 143
Synchronized, Volatile, Memory Model 143
Concurrency 147
Multicore 148
Modifying Algorithm For Multicore 149
Using Concurrent Cache 152
Activity Lifecycle 154
Passing Information 156
Remembering State 158
Summary 161
■ Chapter 6: Benchmarking And Profiling 163
Measuring Time 163
System.nanoTime() 164
Debug.threadCpuTimeNanos() 165
Tracing 167
Debug.startMethodTracing() 167
Using the Traceview Tool 168
Traceview in DDMS 170
Native Tracing 172
Logging 174
Summary 176
■ Chapter 7: Maximizing Battery Life 177
Batteries 177
Measuring Battery Usage 180
Disabling Broadcast Receivers 183
Disabling and Enabling the Broadcast Receiver 186
Networking 187
Background Data 188
Data Transfer 189
Location 191
Unregistering a Listener 193
Frequency of Updates 193
Multiple Providers 194
Filtering Providers 196
Last Known Location 198
Sensors 199
Graphics 200
Alarms 201
Scheduling Alarms 203
WakeLocks 204
Preventing Issues 205
Summary 206
■ Chapter 8: Graphics 207
Optimizing Layouts 207
RelativeLayout 209

Merging Layouts 213
Reusing Layouts....................214
View Stubs 215
Layout Tools....................217
Hierarchy Viewer....................218
layoutopt 218
OpenGL ES 218
Extensions....................219
Texture Compression 221
Mipmaps 226
Multiple APKs....................228
Shaders....................228
Scene Complexity 229
Culling....................229
Render Mode....................229
Power Consumption....................229
Summary 230
■ Chapter 9: RenderScript 231
Overview 231
Hello World....................233
Hello Rendering 236
Creating a Rendering Script....................237
Creating a RenderScriptGL Context 238
Extending RSSurfaceView....................239
Setting the Content View 239
Adding Variables to Script 240
HelloCompute....................243
Allocations 244
rsForEach....................245
Performance 248
Native RenderScript APIs....................249
rs_types.rsh....................250
rs_core.rsh....................253
rs_cl.rsh....................255
rs_math.rsh 259
rs_graphics.rsh....................260
rs_time.rsh 261
rs_atomic.rsh....................262
RenderScript vs. NDK....................263
Summary 263
Index 265

About the Author

Hervé Guihot started learning about computers more than 20 years ago with an Amstrad CPC464. Although the CPC464 is most likely the reason why he still appreciates green-screened devices (ask him about his phone), Hervé started working with Android as it became a popular platform for application development. It was also was the only platform that combined two of his main passions: software and pastries. After many years working in the world of interactive and digital television, he is focused on bringing Android to more devices to encourage more developers to leverage the power of Android and more people to have access to the technology. Hervé is currently a software engineering manager in MediaTek (`www.mediatek.com`), a leading fabless semiconductor company for wireless communications and digital multimedia solutions. He holds an engineering degree from the Institut de Formation Supérieure en Informatique et Télécommunication in Rennes, Brittany, and you can sometimes find him waiting in line for an éclair on 18th and Guerrero.

About the Technical Reviewers

Charles Cruz is a mobile application developer for the Android, iOS, and Windows Phone platforms. He graduated from Stanford University with B.S. and M.S. degrees in Engineering. He lives in Southern California and, when not doing technical things, plays lead guitar in an original metal band (`www.taintedsociety.com`) and a classic rock tribute band. Charles can be reached at `cruzcj@soundandcodecreations.com` and `@CodingNPicking` on Twitter.

Shane Kirk earned his B.S. in Computer Science from the University of Kentucky in 2000. He's currently a software engineer for DeLorme, a mapping and GPS technology company based in Yarmouth, Maine, where he spends his days writing C++ and Java code for mobile and desktop applications. When Shane isn't coding, you'll usually find him making lots of noise with his guitar or lost in the pages of a good book.

Eric Neff is an experienced technical architect with more than 14 years of overall experience in as a technical architect and senior software developer. He is an expert in full life-cycle application development, middle-ware, and n-tier application development, with specific expertise in Microsoft .NET application development. He specializes in object-oriented analysis and design in systems development with a focus on the scheduling of service personal or manufactured items and was instrumental in the design and implementation of data relation schemas for the lexicography industry. Eric was recently promoted to Director of Mobile Innovations at Kiefer Consulting, Inc, putting into practice several years of hobbyist development in the mobile space on iPhone, Android, and Windows Mobile. Eric is active in the local development community through his participation in the Sacramento Google Technology Group and as a board member of the Sacramento Dot Net User Group. He has given presentations on mobile web technologies, mobile development, and ASP.NET techniques.

Acknowledgments

I thank the team at Apress who made this book possible: Steve Anglin, Corbin Collins, Jim Markham, and Jill Steinberg. Working with all of them was a real pleasure and I can with confidence recommend them to any author.

I also want to thank the tech reviewers: Charles Cruz, Shane Kirk, and Eric Neff. They provided invaluable feedback, often catching mistakes I would not have seen even after dozens of readings.

To all my friends whom I did not get to see as often as I would have liked while I was working on this book, I give my thanks too: Marcely, Mathieu, Marilen, Maurice, Katie, Maggie, Jean-René, Ruby, Greg, Aline, Amy, and Gilles, I promise I will make up for the lost time. Last but not least, I owe a lot to three people: Eddy Derick, Fabrice Bernard, and Jean-Louis Gassée. Sometimes all it takes is an injured toe and a broken suitcase.

Introduction

Android quickly became almost ubiquitous. With the world transitioning from feature phones to smartphones, and then discovering that tablets are, after all, devices we can hardly live without, application developers today have a choice between mostly two platforms: Android and iOS. Android lowered, some may even say broke, the barrier of entry for application developers, because all you need to write Android applications is a computer (and of course some programming knowledge). Tools are free, and almost anyone can now write applications reaching millions of customers. With Android now spreading to a variety of devices, from tablets to televisions, it is important to make sure your applications can not only run well on all these devices but also run better than competing applications. After all, the barrier of entry was lowered for all application developers and you will in many cases find yourself competing for a slice of the ever-growing Android applications market. Whether you write applications to make a living, achieve stardom, or simply make the world a better place, performance will be one of the their key elements.

This book assumes you already have some familiarity with Android application development but want to go one step further and explore what can make your applications run faster. Although the Android tools and online documentation make it easy to create applications, performance optimization is sometimes more of an art than a science and is not documented as thoroughly. I wrote *Pro Android Apps Performance Optimization* to help you find easy ways to achieve good performance on virtually all Android devices, whether you are trying to optimize an existing application or are writing an application from scratch. Android allows developers to use Java, C/C++, and even assembly languages, and you can implement performance optimizations in many different ways, from taking advantage of the CPU features to simply using a different language more tailored to a specific problem.

Chapter 1 focuses on optimizing your Java code. Your first applications will most likely exclusively use the Java language, and we will see that algorithms themselves are more important than their implementation. You will also learn how to take advantage of simple techniques such as caching and minimizing memory allocations to greatly optimize your applications. In addition, you will learn how to keep your applications responsive, a very important performance indicator, and how to use databases efficiently.

Chapter 2 takes you one step further (or lower, depending on who you talk to) and introduces the Android NDK. Even though the Java code can be compiled to native code since Android 2.2, using C code to implement certain routines can yield better results. The NDK can also allow you to easily port existing code to Android without having to rewrite everything in Java.

Chapter 3 takes you to the abyss of assembly language. Albeit rarely used by most application developers, assembly language allows you to take advantage of every platform's specific instruction set and can be a great way to optimize your applications, though at the cost of increased complexity and maintenance. Though assembly code is typically limited to certain parts of an application, its benefits should not be ignored as tremendous results can be achieved thanks to carefully targeted optimizations.

Chapter 4 shows you how using less memory can improve performance. In addition to learning simple ways to use less memory in your code, you will learn how memory allocations and memory accesses have a direct impact on performance because of how CPUs are designed.

Chapter 5 teaches you how to use multi-threading in your Android applications in order to keep applications responsive and improve performance as more and more Android devices can run multiple threads simultaneously.

Chapter 6 shows you the basics of measuring your applications' performance. In addition to learning how to use the APIs to measure time, you will also learn how to use some of the Android tools to have a better view of where time is spent in your applications.

Chapter 7 teaches you how to make sure your applications use power rationally. As many Android devices are battery-powered, conserving energy is extremely important because an application that empties the battery quickly will be uninstalled quickly. This chapter shows you how to minimize power consumption without sacrificing the very things that make Android applications special.

Chapter 8 introduces some basic techniques to optimize your applications' layouts and optimize OpenGL rendering.

Chapter 9 is about RenderScript, a relatively new Android component introduced in Honeycomb. RenderScript is all about performance and has already evolved quite a bit since its first release. In this chapter you learn how to use RenderScript in your applications and also learn about the many APIs RenderScript defines.

I hope you enjoy this book and find many helpful tips in it. As you will find out, many techniques are not Android specific, and you will be able to re-use a lot of them on other platforms, for example iOS. Personally, I have a sweet tooth for assembly language and I hope the proliferation of the Android platform and support for assembly language in the Android NDK will entice many developers, if only to learn a new skill. However, I do want to emphasize that good design and good algorithms will often already take care of all performance optimizations you need. Good luck, and I am looking forward to your Android applications!

Chapter 1

Optimizing Java Code

Many Android application developers have a good practical knowledge of the Java language from previous experience. Since its debut in 1995, Java has become a very popular programming language. While some surveys show that Java lost its luster trying to compete with other languages like Objective-C or C#, some of these same surveys rank Java as the number 1 language popularity-wise. Naturally, with mobile devices outselling personal computers and the success of the Android platform (700,000 activations per day in December 2011) Java is becoming more relevant in today's market than ever before.

Developing applications for mobile devices can be quite different from developing applications for personal computers. Today's portable devices can be quite powerful, but in terms of performance, they lag behind personal computers. For example, some benchmarks show a quad-core Intel Core i7 processor running about 20 times faster than the dual-core Nvidia Tegra 2 that is found in the Samsung Galaxy Tab 10.1.

> **NOTE:** Benchmark results are to be taken with a grain of salt since they often measure only part of a system and do not necessarily represent a typical use-case.

This chapter shows you how to make sure your Java applications perform well on Android devices, whether they run the latest Android release or not. First, we take a look at how Android executes your code. Then, we review several techniques to optimize the implementation of a famous mathematical series, including how to take advantage of the latest APIs Android offers. Finally, we review a few techniques to improve your application's responsiveness and to use databases more efficiently.

Before you jump in, you should realize code optimization is not the first priority in your application development. Delivering a good user experience and focusing on code maintainability should be among your top priorities. In fact, code optimization should be one of your last priorities, and may not even be part of the process altogether. However, good practices can help you reach an acceptable level of performance without having you go back to your code, asking yourself "what did I do wrong?" and having to spend additional resources to fix it.

How Android Executes Your Code

While Android developers use Java, the Android platform does not include a Java Virtual Machine (VM) for executing code. Instead, applications are compiled into Dalvik bytecode, and Android uses its Dalvik VM to execute it. The Java code is still compiled into Java bytecode, but this Java bytecode is then compiled into Dalvik bytecode by the dex compiler, dx (an SDK tool). Ultimately, your application will contain only the Dalvik bytecode, not the Java bytecode.

For example, an implementation of a method that computes the n^{th} term of the Fibonacci series is shown in Listing 1–1 together with the class definition. The Fibonacci series is defined as follows:

$F_0 = 0$
$F_1 = 1$
$F_n = F_{n-2} + F_{n-1}$ for n greater than 1

Listing 1–1. *Naïve Recursive Implementation of Fibonacci Series*

```
public class Fibonacci {
    public static long computeRecursively (int n)
    {
        if (n > 1) return computeRecursively(n-2) + computeRecursively(n-1);
        return n;
    }
}
```

> **NOTE:** A trivial optimization was done by returning n when n equals 0 or 1 instead of adding another "if" statement to check whether n equals 0 or 1.

An Android application is referred to as an APK since applications are compiled into a file with the apk extension (for example, APress.apk), which is simply an archive file. One of the files in the archive is classes.dex, which contains the application's bytecode. The Android toolchain provides a tool, dexdump, which can convert the binary form of the code (contained in the APK's classes.dex file) into human-readable format.

> **TIP:** Because an apk file is simply a ZIP archive, you can use common archive tools such as WinZip or 7-Zip to inspect the content of an apk file..

Listing 1–2 shows the matching Dalvik bytecode.

Listing 1–2. *Human-Readable Dalvik Bytecode of Fibonacci.computeRecursively*

```
002548:                |[002548] com.apress.proandroid.Fibonacci.computeRecursively:(I)J
002558: 1212           |0000: const/4 v2, #int 1 // #1
00255a: 3724 1100      |0001: if-le v4, v2, 0012 // +0011
00255e: 1220           |0003: const/4 v0, #int 2 // #2
002560: 9100 0400      |0004: sub-int v0, v4, v0
```

```
002564: 7110 3d00 0000  |0006: invoke-static {v0},
    Lcom/apress/proandroid/Fibonacci;.computeRecursively:(I)J
00256a: 0b00            |0009: move-result-wide v0
00256c: 9102 0402       |000a: sub-int v2, v4, v2
002570: 7110 3d00 0200  |000c: invoke-static {v2},
    Lcom/apress/proandroid/Fibonacci;.computeRecursively:(I)J
002576: 0b02            |000f: move-result-wide v2
002578: bb20            |0010: add-long/2addr v0, v2
00257a: 1000            |0011: return-wide v0
00257c: 8140            |0012: int-to-long v0, v4
00257e: 28fe            |0013: goto 0011 // -0002
```

The first number on each line specifies the absolute position of the code within the file. Except on the very first line (which shows the method name), it is then followed by one or more 16-bit bytecode units, followed by the position of the code within the method itself (relative position, or label), the opcode mnemonic and finally the opcode's parameter(s). For example, the two bytecode units 3724 1100 at address 0x00255a translate to "if-le v4, v2, 0012 // +0011", which basically means "if content of virtual register v4 is less than or equal to content of virtual register v2 then go to label 0x0012 by skipping 17 bytecode units" (17_{10} equals 11_{16}). The term "virtual register" refers to the fact that these are not actual hardware registers but instead the registers used by the Dalvik virtual machine.

Typically, you would not need to look at your application's bytecode. This is especially true with Android 2.2 (codename Froyo) and later versions since a Just-In-Time (JIT) compiler was introduced in Android 2.2. The Dalvik JIT compiler compiles the Dalvik bytecode into native code, which can execute significantly faster. A JIT compiler (sometimes referred to simply as a JIT) improves performance dramatically because:

- Native code is directly executed by the CPU without having to be interpreted by a virtual machine.
- Native code can be optimized for a specific architecture.

Benchmarks done by Google showed code executes 2 to 5 times faster with Android 2.2 than Android 2.1. While the results may vary depending on what your code does, you can expect a significant increase in speed when using Android 2.2 and later versions.

The absence of a JIT compiler in Android 2.1 and earlier versions may affect your optimization strategy significantly. If you intend to target devices running Android 1.5 (codename Cupcake), 1.6 (codename Donut), or 2.1 (codename Éclair), most likely you will need to review more carefully what you want or need to provide in your application. Moreover, devices running these earlier Android versions are older devices, which are less powerful than newer ones. While the market share of Android 2.1 and earlier devices is shrinking, they still represent about 12% as of December 2011). Possible strategies are:

- Don't optimize at all. Your application could be quite slow on these older devices.

- Require minimum API level 8 in your application, which can then be installed only on Android 2.2 or later versions.
- Optimize for older devices to offer a good user experience even when no JIT compiler is present. This could mean disabling features that are too CPU-heavy.

> **TIP:** Use `android:vmSafeMode` in your application's manifest to enable or disable the JIT compiler. It is enabled by default (if it is available on the platform). This attribute was introduced in Android 2.2.

Now it is time to run the code on an actual platform and see how it performs. If you are familiar with recursion and the Fibonacci series, you might guess that it is going to be slow. And you would be right. On a Samsung Galaxy Tab 10.1, computing the thirtieth Fibonacci number takes about 370 milliseconds. With the JIT compiler disabled, it takes about 440 milliseconds. If you decide to include that function in a Calculator application, users will become frustrated because the results cannot be computed "immediately." From a user's point of view, results appear instantaneous if they can be computed in 100 milliseconds or less. Such a response time guarantees a very good user experience, so this is what we are going to target.

Optimizing Fibonacci

The first optimization we are going to perform eliminates a method call, as shown in Listing 1–3. As this implementation is recursive, removing a single call in the method dramatically reduces the total number of calls. For example, `computeRecursively(30)` generated 2,692,537 calls while computeRecursivelyWithLoop(30) generated "only" 1,346,269. However, the performance of this method is still not acceptable considering the response-time criteria defined above, 100 milliseconds or less, as `computeRecursivelyWithLoop(30)` takes about 270 milliseconds to complete.

Listing 1–3. *Optimized Recursive Implementation of Fibonacci Series*

```
public class Fibonacci {
    public static long computeRecursivelyWithLoop (int n)
    {
        if (n > 1) {
            long result = 1;
            do {
                result += computeRecursivelyWithLoop(n-2);
                n--;
            } while (n > 1);
            return result;
        }
        return n;
    }
}
```

> **NOTE:** This is not a true tail-recursion optimization.

From Recursive To Iterative

For the second optimization, we switch from a recursive implementation to an iterative one. Recursive algorithms often have a bad reputation with developers, especially on embedded systems without much memory, because they tend to consume a lot of stack space and, as we just saw, can generate too many method calls. Even when performance is acceptable, a recursive algorithm can cause a stack overflow and crash an application. An iterative implementation is therefore often preferred whenever possible. Listing 1–4 shows what is considered a textbook iterative implementation of the Fibonacci series.

Listing 1–4. *Iterative Implementation of Fibonacci Series*

```
public class Fibonacci {
    public static long computeIteratively (int n)
    {
        if (n > 1) {
            long a = 0, b = 1;
            do {
                long tmp = b;
                b += a;
                a = tmp;
            } while (--n > 1);
            return b;
        }
        return n;
    }
}
```

Because the n^{th} term of the Fibonacci series is simply the sum of the two previous terms, a simple loop can do the job. Compared to the recursive algorithms, the complexity of this iterative algorithm is also greatly reduced because it is linear. Consequently, its performance is also much better, and `computeIteratively(30)` takes less than 1 millisecond to complete. Because of its linear nature, you can use such an algorithm to compute terms beyond the 30^{th}. For example, `computeIteratively(50000)` takes only 2 milliseconds to return a result and, by extrapolation, you could guess `computeIteratively(500000)` would take between 20 and 30 milliseconds to complete.

While such performance is more than acceptable, it is possible to to achieve even faster results with a slightly modified version of the same algorithm, as showed in Listing 1–5. This new version computes two terms per iteration, and the total number of iterations is halved. Because the number of iterations in the original iterative algorithm could be odd, the initial values for a and b are modified accordingly: the series starts with a=0 and b=1 when n is odd, and it starts with a=1 and b=1 (Fib(2)=1) when n is even.

Listing 1–5. *Modified Iterative Implementation of Fibonacci Series*

```
public class Fibonacci {
    public static long computeIterativelyFaster (int n)
    {
        if (n > 1) {
            long a, b = 1;
            n--;
            a = n & 1;
            n /= 2;
            while (n-- > 0) {
                a += b;
                b += a;
            }
            return b;
        }
        return n;
    }
}
```

Results show this modified iterative version is about twice as fast as the original one.

While these iterative implementations are fast, they do have one major problem: they don't return correct results. The issue lies with the return value being stored in a long value, which is 64-bit. The largest Fibonacci number that can fit in a signed 64-bit value is 7,540,113,804,746,346,429 or, in other words, the 92nd Fibonacci number. While the methods will still return without crashing the application for values of n greater than 92, the results will be incorrect because of an overflow: the 93rd Fibonacci number would be negative! The recursive implementations actually have the same limitation, but one would have to be quite patient to eventually find out.

> **NOTE:** Java specifies the size of all primitive types (except boolean): long is 64-bit, int is 32-bit, and short is 16-bit. All integer types are signed.

BigInteger

Java offers just the right class to fix this overflow problem: `java.math.BigInteger`. A `BigInteger` object can hold a signed integer of arbitrary size and the class defines all the basic math operations (in addition to some not-so-basic ones). Listing 1–6 shows the `BigInteger` version of `computeIterativelyFaster`.

> **TIP:** The `java.math` package also defines `BigDecimal` in addition to `BigInteger`, while `java.lang.Math` provides math constant and operations. If your application does not need double precision, use Android's `FloatMath` instead of `Math` for performance (although gains may vary depending on platform).

Listing 1–6. *BigInteger Version of Fibonacci.computeIterativelyFaster*

```
public class Fibonacci {
    public static BigInteger computeIterativelyFasterUsingBigInteger (int n)
    {
        if (n > 1) {
            BigInteger a, b = BigInteger.ONE;
            n--;
            a = BigInteger.valueOf(n & 1);
            n /= 2;
            while (n-- > 0) {
                a = a.add(b);
                b = b.add(a);
            }
            return b;
        }
        return (n == 0) ? BigInteger.ZERO : BigInteger.ONE;
    }
}
```

That implementation guarantees correctness as overflows can no longer occur. However, it is not without problems because, again, it is quite slow: a call to `computeIterativelyFasterUsingBigInteger(50000)` takes about 1.3 seconds to complete. The lackluster performance can be explained by three things:

- `BigInteger` is immutable.
- `BigInteger` is implemented using `BigInt` and native code.
- The larger the numbers, the longer it takes to add them together.

Since `BigInteger` is immutable, we have to write “a = a.add(b)” instead of simply “a.add(b)”. Many would assume “a.add(b)” is the equivalent of “a += b” and many would be wrong: it is actually the equivalent of “a + b”. Therefore, we have to write “a = a.add(b)” to assign the result. That small detail is extremely significant as “a.add(b)” creates a new `BigInteger` object that holds the result of the addition.

Because of `BigInteger`’s current internal implementation, an additional `BigInt` object is created for every `BigInteger` object that is allocated. This results in twice as many objects being allocated during the execution of `computeIterativelyFasterUsingBigInteger`: about 100,000 objects are created when calling `computeIterativelyFasterUsingBigInteger (50000)` (and all of them but one will become available for garbage collection almost immediately). Also, `BigInt` is implemented using native code and calling native code from Java (using JNI) has a certain overhead.

The third reason is that very large numbers do not fit in a single, long 64-bit value. For example, the 50,000th Fibonacci number is 34,7111–bit long.

> **NOTE:** `BigInteger`’s internal implementation (`BigInteger.java`) may change in future Android releases. In fact, internal implementation of any class can change.

For performance reasons, memory allocations should be avoided whenever possible in critical paths of the code. Unfortunately, there are some cases where allocations are needed, for example when working with immutable objects like `BigInteger`. The next optimization focuses on reducing the number of allocations by switching to a different algorithm. Based on the Fibonacci *Q*-matrix, we have the following:

$$F_{2n-1} = F_n^2 + F_{n-1}^2$$

$$F_{2n} = (2F_{n-1} + F_n) * F_n$$

This can be implemented using `BigInteger` again (to guarantee correct results), as shown in Listing 1–7.

Listing 1–7. *Faster Recursive Implementation of Fibonacci Series Using BigInteger*

```
public class Fibonacci {
    public static BigInteger computeRecursivelyFasterUsingBigInteger (int n)
    {
        if (n > 1) {
            int m = (n / 2) + (n & 1); // not obvious at first – wouldn’t it be great to
have a better comment here?
            BigInteger fM = computeRecursivelyFasterUsingBigInteger(m);
            BigInteger fM_1 = computeRecursivelyFasterUsingBigInteger(m - 1);
            if ((n & 1) == 1) {
                // F(m)^2 + F(m-1)^2
                return fM.pow(2).add(fM_1.pow(2)); // three BigInteger objects created
            } else {
                // (2*F(m-1) + F(m)) * F(m)
                return fM_1.shiftLeft(1).add(fM).multiply(fM); // three BigInteger
objects created
            }
        }
        return (n == 0) ? BigInteger.ZERO : BigInteger.ONE; // no BigInteger object
created
    }

    public static long computeRecursivelyFasterUsingBigIntegerAllocations(int n)
    {
        long allocations = 0;
        if (n > 1) {
            int m = (n / 2) + (n & 1);
            allocations += computeRecursivelyFasterUsingBigIntegerAllocations(m);
            allocations += computeRecursivelyFasterUsingBigIntegerAllocations(m - 1);

            // 3 more BigInteger objects allocated
            allocations += 3;
        }
        return allocations; // approximate number of BigInteger objects allocated when
computeRecursivelyFasterUsingBigInteger(n) is called
    }
}
```

A call to `computeRecursivelyFasterUsingBigInteger(50000)` returns in about 1.6 seconds. This shows this latest implementation is actually slower than the fastest iterative implementation we have so far. Again, the number of allocations is the culprit as around 200,000 objects were allocated (and almost immediately marked as eligible for garbage collection).

> **NOTE:** The actual number of allocations is less than what `computeRecursivelyFasterUsingBigIntegerAllocations` would return. Because `BigInteger`'s implementation uses preallocated objects such as `BigInteger.ZERO`, `BigInteger.ONE`, or `BigInteger.TEN`, there may be no need to allocate a new object for some operations. You would have to look at Android's `BigInteger` implementation to know exactly how many objects are allocated.

This implementation is slower, but it is a step in the right direction nonetheless. The main thing to notice is that even though we need to use `BigInteger` to guarantee correctness, we don't have to use `BigInteger` for every value of n. Since we know the primitive type `long` can hold results for n less than or equal to 92, we can slightly modify the recursive implementation to mix `BigInteger` and primitive type, as shown in Listing 1–8.

Listing 1–8. *Faster Recursive Implementation of Fibonacci Series Using BigInteger and long Primitive Type*

```
public class Fibonacci {
    public static BigInteger computeRecursivelyFasterUsingBigIntegerAndPrimitive(int n)
    {
        if (n > 92) {
            int m = (n / 2) + (n & 1);
            BigInteger fM = computeRecursivelyFasterUsingBigIntegerAndPrimitive(m);
            BigInteger fM_1 = computeRecursivelyFasterUsingBigIntegerAndPrimitive(m -
1);
            if ((n & 1) == 1) {
                return fM.pow(2).add(fM_1.pow(2));
            } else {
                return fM_1.shiftLeft(1).add(fM).multiply(fM); // shiftLeft(1) to
multiply by 2
            }
        }
        return BigInteger.valueOf(computeIterativelyFaster(n));
    }

    private static long computeIterativelyFaster(int n)
    {
        // see Listing 1-5 for implementation
    }
}
```

A call to `computeRecursivelyFasterUsingBigIntegerAndPrimitive(50000)` returns in about 73 milliseconds and results in about 11,000 objects being allocated: a small modification in the algorithm yields results about 20 times faster and about 20 times fewer objects being allocated. Quite impressive! It is possible to improve the performance even further by reducing the number of allocations, as shown in Listing 1–9. Precomputed results can be quickly generated when the Fibonacci class is first loaded, and these results can later be used directly.

Listing 1–9. *Faster Recursive Implementation of Fibonacci Series Using BigInteger and Precomputed Results*

```
public class Fibonacci {
    static final int PRECOMPUTED_SIZE= 512;
    static BigInteger PRECOMPUTED[] = new BigInteger[PRECOMPUTED_SIZE];

    static {
        PRECOMPUTED[0] = BigInteger.ZERO;
        PRECOMPUTED[1] = BigInteger.ONE;
        for (int i = 2; i < PRECOMPUTED_SIZE; i++) {
            PRECOMPUTED[i] = PRECOMPUTED[i-1].add(PRECOMPUTED[i-2]);
        }
    }

    public static BigInteger computeRecursivelyFasterUsingBigIntegerAndTable(int n)
    {
        if (n > PRECOMPUTED_SIZE - 1) {
            int m = (n / 2) + (n & 1);
            BigInteger fM = computeRecursivelyFasterUsingBigIntegerAndTable (m);
            BigInteger fM_1 = computeRecursivelyFasterUsingBigIntegerAndTable (m - 1);
            if ((n & 1) == 1) {
                return fM.pow(2).add(fM_1.pow(2));
            } else {
                return fM_1.shiftLeft(1).add(fM).multiply(fM);
            }
        }
        return PRECOMPUTED[n];
    }
}
```

The performance of this implementation depends on `PRECOMPUTED_SIZE`: the bigger, the faster. However, memory usage may become an issue since many `BigInteger` objects will be created and remain in memory for as long as the Fibonacci class is loaded. It is possible to merge the implementations shown in Listing 1–8 and Listing 1–9, and use a combination of precomputed results and computations with primitive types. For example, terms 0 to 92 could be computed using `computeIterativelyFaster`, terms 93 to 127 using precomputed results and any other term using recursion. As a developer, you are responsible for choosing the best implementation, which may not always be the fastest. Your choice will be based on various factors, including:

- What devices and Android versions your application target
- Your resources (people and time)

As you may have already noticed, optimizations tend to make the source code harder to read, understand, and maintain, sometimes to such an extent that you would not recognize your own code weeks or months later. For this reason, it is important to carefully think about what optimizations you really need and how they will affect your application development, both in the short term and in the long term. It is always recommended you first implement a working solution before you think of optimizing it (and make sure you save a copy of the working solution). After all, you may realize optimizations are not needed at all, which could save you a lot of time. Also, make sure you include comments in your code for everything that is not obvious to a person with ordinary skill in the art. Your coworkers will thank you, and you may give yourself a pat on the back as well when you stumble on some of your old code. My poor comment in Listing 1–7 is proof.

> **NOTE:** All implementations disregard the fact that n could be negative. This was done intentionally to make a point, but your code, at least in all public APIs, should throw an IllegalArgumentException whenever appropriate.

Caching Results

When computations are expensive, it may be a good idea to remember past results to make future requests faster. Using a cache is quite simple as it typically translates to the pseudo-code shown in Listing 1–10.

Listing 1–10. *Using a Cache*

```
    result = cache.get(n); // input parameter n used as key
    if (result == null) {
        // result was not in the cache so we compute it and add it
        result = computeResult(n);
        cache.put(n, result); // n is the key, result is the value
    }
    return result;
```

The faster recursive algorithm to compute Fibonacci terms yields many duplicate calculations and could greatly benefit from memoization. For example, computing the 50,000th term requires computing the 25,000th and 24,999th terms. Computing the 25,000th term requires computing the 12,500th and 12,499th terms, while computing the 24,999th term requires computing… the same 12,500th and 12,499th terms again! Listing 1–11 shows a better implementation using a cache.

If you are familiar with Java, you may be tempted to use a `HashMap` as your cache, and it would work just fine. However, Android defines `SparseArray`, a class that is intended to be more efficient than `HashMap` when the key is an integer value: `HashMap` would require the key to be of type `java.lang.Integer`, while `SparseArray` uses the primitive type int for keys. Using `HashMap` would therefore trigger the creation of many `Integer` objects for the keys, which `SparseArray` simply avoids.

Listing 1–11. *Faster Recursive Implementation Using BigInteger, long Primitive TypeAnd Cache*

```
public class Fibonacci {
    public static BigInteger computeRecursivelyWithCache (int n)
    {
        SparseArray<BigInteger> cache = new SparseArray<BigInteger>();
        return computeRecursivelyWithCache(n, cache);
    }

    private static BigInteger computeRecursivelyWithCache (int n,
SparseArray<BigInteger> cache)
    {
        if (n > 92) {
            BigInteger fN = cache.get(n);
```

```
            if (fN == null) {
                int m = (n / 2) + (n & 1);
                BigInteger fM = computeRecursivelyWithCache(m, cache);
                BigInteger fM_1 = computeRecursivelyWithCache(m - 1, cache);
                if ((n & 1) == 1) {
                    fN = fM.pow(2).add(fM_1.pow(2));
                } else {
                    fN = fM_1.shiftLeft(1).add(fM).multiply(fM);
                }
                cache.put(n, fN);
            }
            return fN;
        }
        return BigInteger.valueOf(iterativeFaster(n));
    }

    private static long iterativeFaster (int n) { /* see Listing 1-5 for implementation
*/ }
}
```

Measurements showed `computeRecursivelyWithCache(50000)` takes about 20 milliseconds to complete, or about 50 fewer milliseconds than a call to `computeRecursivelyFasterUsingBigIntegerAndPrimitive(50000)`. Obviously, the difference is exacerbated as n grows: when n equals 200,000 the two methods complete in 50 and 330 milliseconds respectively.

Because many fewer `BigInteger` objects are allocated, the fact that `BigInteger` is immutable is not as big of a problem when using the cache. However, remember that three `BigInteger` objects are still created (two of them being very short-lived) when fN is computed, so using mutable big integers would still improve performance.

Even though using `HashMap` instead of `SparseArray` may be a little slower, it would have the benefit of making the code Android-independent, that is, you could use the exact same code in a non-Android environment (without `SparseArray`).

> **NOTE:** Android defines multiple types of sparse arrays: `SparseArray` (to map integers to objects), `SparseBooleanArray` (to map integers to booleans), and `SparseIntArray` (to map integers to integers).

android.util.LruCache<K, V>

Another class worth mentioning is `android.util.LruCache<K, V>`, introduced in Android 3.1 (codename Honeycomb MR1), which makes it easy to define the maximum size of the cache when it is allocated. Optionally, you can also override the `sizeOf()` method to change how the size of each cache entry is computed. Because it is only available in Android 3.1 and later, you may still end up having to use a different class to implement a cache in your own application if you target Android revisions older than 3.1. This is a very likely scenario considering Android 3.1 as of today represents only a very small portion of the Android devices in use. An alternative solution is to extend java.util.LinkedHashMap and override removeEldestEntry. An LRU cache (for Least Recently Used) discards the least recently used items first. In some applications, you may need exactly the opposite, that is, a cache that discards the most recently used items first. Android does not define such an MruCache class for now, which is not surprising considering MRU caches are not as commonly used.

Of course, a cache can be used to store information other than computations. A common use of a cache is to store downloaded data such as pictures and still maintain tight control over how much memory is consumed. For example, override LruCache's sizeOf method to limit the size of the cache based on a criterion other than simply the number of entries in the cache. While we briefly discussed the LRU and MRU strategies, you may want to use different replacement strategies for your own cache to maximize cache hits. For example, your cache could first discard the items that are not costly to recreate, or simply randomly discard items. Follow a pragmatic approach and design your cache accordingly. A simple replacement strategy such as LRU can yield great results and allow you to focus your resources on other, more important problems.

We've looked at several different techniques to optimize the computation of Fibonacci numbers. While each technique has its merits, no one implementation is optimal. Often the best results are achieved by combining multiple various techniques instead of relying on only one of them. For example, an even faster implementation would use precomputations, a cache mechanism, and maybe even slightly different formulas. (Hint: what happens when n is a multiple of 4?) What would it take to compute $F_{Integer.MAX\_VALUE}$ in less than 100 milliseconds?

API Levels

The LruCache class mentioned above is a good example of why you need to know what API level you are going to target. A new version of Android is released approximately every six months, with new APIs only available from that release. Any attempt to call an API that does not exist results in a crash, bringing not only frustration for the user but also shame to the developer. For example, calling Log.wtf(TAG, "Really?") on an Android 1.5 device crashes the application, as Log.wtf was introduced in Android 2.2 (API level 8). What a terrible failure indeed that would be. Table 1–1 shows the performance improvements made in the various Android versions.

Table 1–1. *Android Versions*

API level	Version	Name	Significant performance improvements
1	1.0	Base	
2	1.1	Base 1.1	
3	1.5	Cupcake	Camera start-up time, image capture time, faster acquisition of GPS location, NDK support
4	1.6	Donut	
5	2.0	Éclair	Graphics
6	2.0.1	Éclair 0.1	
7	2.1	Éclair MR1	
8	2.2	Froyo	V8 Javascript engine (browser), JIT compiler, memory management
9	2.3.0 2.3.1 2.3.2	Gingerbread	Concurrent garbage collector, event distribution, better OpenGL drivers
10	2.3.3 2.3.4	Gingerbread MR1	
11	3.0	Honeycomb	Renderscript, animations, hardware-accelerated 2D graphics, multicore support
12	3.1	Honeycomb MR1	LruCache, partial invalidates in hardware-accelerated views, new Bitmap.setHasAlpha() API
13	3.2	Honeycomb MR2	
14	4.0	Ice Cream Sandwich	Media effects (transformation filters), hardware-accelerated 2D graphics (required)

However, your decision to support a certain target should normally not be based on which API you want to use, but instead on what market you are trying to reach. For example, if your target is primarily tablets and not cell phones, then you could target Honeycomb. By doing so, you would limit your application's audience to a small subset of Android devices, because Honeycomb represents only about 2.4% as of December 2011, and not all tablets support Honeycomb. (For example, Barnes & Noble's Nook uses Android 2.2 while Amazon's Kindle Fire uses Android 2.3.) Therefore, supporting older Android versions could still make sense.

The Android team understood that problem when they released the Android Compatibility package, which is available through the SDK Updater. This package contains a static library with some of the new APIs introduced in Android 3.0, namely the fragment APIs. Unfortunately, this compatibility package contains only the fragment APIs and does not address the other APIs that were added in Honeycomb. Such a compatibility package is the exception, not the rule. Normally, an API introduced at a specific API level is not available at lower levels, and it is the developer's responsibility to choose APIs carefully.

To get the API level of the Android platform, you can use Build.VERSION.SDK_INT. Ironically, this field was introduced in Android 1.6 (API level 4), so trying to retrieve the version this way would also result in a crash on Android 1.5 or earlier. Another option is to use Build.VERSION.SDK, which has been present since API level 1. However, this field is now deprecated, and the version strings are not documented (although it would be pretty easy to understand how they have been created).

> **TIP:** Use reflection to check whether the SDK_INT field exists (that is, if the platform is Android 1.6 or later). See Class.forName("android.os.Build$VERSION").getField("SDK").

Your application's manifest file should use the `<uses-sdk>` element to specify two important things:

- The minimum API level required for the application to run (android:minSdkVersion)
- The API level the application targets (android:targetSdkVersion)

It is also possible to specify the maximum API level (android:maxSdkVersion), but using this attribute is not recommended. Specifying maxSdkVersion could even lead to applications being uninstalled automatically after Android updates. The target API level is the level at which your application has been explicitly tested.

By default, the minimum API level is set to 1 (meaning the application is compatible with all Android versions). Specifying an API level greater than 1 prevents the application from being installed on older devices. For example, android:minSdkVersion="4" guarantees Build.VERSION.SDK_INT can be used without risking any crash. The minimum API level does not have to be the highest API level you are using in your application as long as you make sure you call only a certain API when the API actually exists, as shown in Listing 1–12.

Listing 1–12. *Calling a SparseArray Method Introduced in Honeycomb (API Level 11)*

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    sparseArray.removeAt(1); // API level 11 and above
} else {
    int key = sparseArray.keyAt(1); // default implementation is slower
    sparseArray.remove(key);
}
```