Extract from “Data Structures and Algorithms Using Visual Basic.NET” by Michael McMillan p6-12

# TIMING TESTS

Because this book takes a practical approach to the analysis of the data structures and algorithms examined, we eschew the use of BigO analysis, preferring instead to run simple benchmark tests that will tell us how long in seconds (or whatever time unit) it takes for a code segment to run.

Our benchmarks will be timing tests that measure the amount of time it takes an algorithm to run to completion. Benchmarking is as much of an art as a science and you have to be careful how you time a code segment to get an accurate analysis. Let’s examine this in more detail.

# An Oversimplified Timing Test

First, we need some code to time. For simplicity’s sake, we will time a subroutine that writes the contents of an array to the console. Here’s the code:

Sub DisplayNums(ByVal arr() As Integer)

Dim index As Integer

For index = 0 To arr.GetUpperBound(0)

Console.Write(arr(index))

Next

End Sub

The array is initialized in another part of the program, which we’ll examine later.

To time this subroutine, we need to create a variable that is assigned the system time just as the subroutine is called, and we need a variable to store the time when the subroutine returns. Here’s how we wrote this code:

Dim startTime As DateTime

Dim endTime As TimeSpan

startTime = DateTime.Now

DisplayNums(nums)

endTime = DateTime.Now.Subtract(startTime)

Running this code on a laptop (running at 1.4 MHz on Windows XP Professional) takes about 5 seconds (4.9917 seconds to be exact). Whereas this code segment seems reasonable for performing a timing test, it is completely inadequate for timing code running in the .NET environment. Why?

First, this code measures the elapsed time from when the subroutine was called until the subroutine returns to the main program. The time used by other processes running at the same time as the VB.NET program adds to the time being measured by the test.

Second, the timing code used here doesn’t take into account garbage collection performed in the .NET environment. In a runtime environment such as .NET, the system can pause at any time to perform garbage collection. The sample timing code does nothing to acknowledge garbage collection and the resulting time can be affected quite easily by garbage collection. So what do we do about this?

# Timing Tests for the .NET Environment

In the .NET environment, we need to take into account the thread in which our program is running and the fact that garbage collection can occur at any time. We need to design our timing code to take these facts into consideration.

Let’s start by looking at how to handle garbage collection. First, let’s discuss what garbage collection is used for. In VB.NET, reference types (such as strings, arrays, and class instance objects) are allocated memory on something called the *heap*. The heap is an area of memory reserved for data items (the types previously mentioned). Value types, such as normal variables, are stored on the *stack*. References to reference data are also stored on the stack, but the actual data stored in a reference type are stored on the heap.

Variables that are stored on the stack are freed when the subprogram in which the variables are declared completes its execution. Variables stored on the heap, in contrast, are held on the heap until the garbage collection process is called. Heap data are only removed via garbage collection when there is not an active reference to those data.

Garbage collection can, and will, occur at arbitrary times during the execution of a program. However, we want to be as sure as we can that the garbage collector is not run while the code we are timing is executing. We can head off arbitrary garbage collection by calling the garbage collector explicitly. The .NET environment provides a special object for making garbage collection calls, GC. To tell the system to perform garbage collection, we simply write the following:

GC.Collect()

That’s not all we have to do, though. Every object stored on the heap has a special method called a finalizer. The finalizer method is executed as the last step before deleting the object. The problem with finalizer methods is that they are not run in a systematic way. In fact, you can’t even be sure an object’s finalizer method will run at all, but we know that before we can be certain an object is deleted, its finalizer method must execute. To ensure this, we add a line of code that tells the program to wait until all the finalizer methods of the objects on the heap have run before continuing. The line of code is as follows:

GC.WaitForPendingFinalizers()

We have cleared one hurdle but one remains: using the proper thread. In the .NET environment, a program is run inside a process, also called an *application domain*. This allows the operating system to separate each different program running on it at the same time. Within a process, a program or a part of a program is run inside a *thread*. Execution time for a program is allocated by the operating system via threads. When we are timing the code for a program, we want to make sure that we’re timing just the code inside the process allocated for our program and not other tasks being performed by the operating system.

We can do this by using the Process class in the .NET Framework. The Process class has methods for allowing us to pick the current process (the process in which our program is running), the thread in which the program is running, and a timer to store the time the thread starts executing. Each of these methods can be combined into one call, which assigns its return value to a variable to store the starting time (a TimeSpan object). Here’s the code:

Dim startingTime As TimeSpan

startingTime = Process.GetCurrentProcess.Threads(0).UserProcessorTime

All we have left to do is capture the time when the code segment we’re timing stops. Here’s how it’s done:

duration = Process.GetCurrentProcess.Threads(0).UserProcessorTime.Subtract(startingTime)

Now let’s combine all this into one program that times the same code we tested earlier:

Module Module1

Sub Main()

Dim nums(99999) As Integer

BuildArray(nums)

Dim startTime As TimeSpan

Dim duration As TimeSpan

startTime = Process.GetCurrentProcess.Threads(0). \_

UserProcessorTime

DisplayNums(nums)

duration = Process.GetCurrentProcess.Threads(0). \_

UserProcessorTime.Subtract(startTime)

Console.WriteLine("Time: " & duration.TotalSeconds)

End Sub

Sub BuildArray(ByVal arr() As Integer)

Dim index As Integer

For index = 0 To 99999

arr(index) = index

Next

End Sub

End Module

Using the new-and-improved timing code, the program returns in just 0.2526 seconds. This compares with the approximately 5 seconds return time using the first timing code. Clearly, a major discrepancy between these two timing techniques exists and you should use the .NET techniques when timing code in the .NET environment.

# A Timing Test Class

Although we don’t need a class to run our timing code, it makes sense to rewrite the code as a class, primarily because we’ll keep our code clear if we can reduce the number of lines in the code we test.

A Timing class needs the following data members:

* startingTime—to store the starting time of the code we are testing,
* duration—the ending time of the code we are testing,

The starting time and the duration members store times and we chose to use the TimeSpan data type for these data members.We’ll use just one constructor method, a default constructor that sets both the data members to 0.

We’ll need methods for telling a Timing object when to start timing code and when to stop timing.We also need a method for returning the data stored in the duration data member.

As you can see, the Timing class is quite small, needing just a few methods. Here’s the definition:

Public Class Timing

Private startingTime As TimeSpan

Private duration As TimeSpan

Public Sub New()

startingTime = New TimeSpan(0)

duration = New TimeSpan(0)

End Sub

Public Sub stopTime()

duration = Process.GetCurrentProcess.Threads(0). \_

UserProcessorTime.Subtract(startingTime)

End Sub

Public Sub startTime()

GC.Collect()

GC.WaitForPendingFinalizers()

startingTime = Process.GetCurrentProcess. \_

Threads(0).UserProcessorTime

End Sub

Public ReadOnly Property Result() As TimeSpan

Get

Return duration

End Get

End Property

End Class

Here’s the program to test the DisplayNums subroutine, rewritten with the Timing class:

Option Strict On

Imports Timing

Module Module1

Sub Main()

Dim nums(99999) As Integer

BuildArray(nums)

Dim tObj As New Timing()

tObj.startTime()

DisplayNums(nums)

tObj.stopTime()

Console.WriteLine("time (.NET): " & \_

tObj.Result.TotalSeconds)

Console.Read()

End Sub

Sub BuildArray(ByVal arr() As Integer)

Dim index As Integer

For index = 0 To 99999

arr(index) = index

Next

End Sub

End Module

By moving the timing code into a class, we’ve reduced the number of lines in the main program from 13 to 8. Admittedly, that’s not a lot of code to cut out of a program, but more important than the number of lines we cut is the reduction in the amount of clutter in the main program. Without the class, assigning the starting time to a variable looks like this:

startTime = Process.GetCurrentProcess.Threads(0).UserProcessorTime

With the Timing class, assigning the starting time to the class data member looks like this:

tObj.startTime()

Encapsulating the long assignment statement into a class method makes our code easier to read and less likely to have bugs.

# SUMMARY

The timing methods we develop in the Timing class make our benchmarks more realistic because they take into the account the environment with which VB.NET programs run. Simply measuring starting and stopping times using the system clock does not account for the time the operating system uses to run other processes or the time the .NET runtime uses to perform garbage collection.