Microbenchmarks in Java, C# and C

Peter Sestoft (sestoft@itu.dk)

IT University of Copenhagen, Denmark

Version 0.9.9 of 2026-08-10

Abstract: Sometimes one wants to measure the speed of software, for instance, to determine whether a new way to solve a problem is faster than the old one. Making such time measurements and microbenchmarks requires considerable care, especially on managed platforms like the Java Virtual Machine and Microsoft's .NET, or else the results may be arbitrary and misleading.

Here we give some advice on running microbenchmarks. Most examples are in Java but the advice applies to any language executed on a managed platform, including C#, Scala, F# and Javascript. All example code in Java, C# and C is freely available from

1 The challenge of managed platforms

Measuring the execution time of a piece of software is an experimental activity, involving the software and a computer system, itself consisting of much systems software hardware. Whereas biological experiments, such as measuring bacterial growth, are influenced by natural variation and many unknown circumstances, software performance measurements may seem straightforward: in principle, everything is man-made and under the experimenter's control. However, in practice, software performance measurements are influenced by so many factors, and modern computer systems are growing so complex, that software experiments increasingly resemble biological experiments.

One source of complications is the widespread use of managed execution platforms, such as the Java Virtual Machine (JVM), Microsoft's .NET, and high-performance Javascript implementations such as Google's v8 engine. These managed execution platforms typically accept software in intermediate form (bytecode or even source code) and compile that intermediate form to real machine code at runtime, by so-called just-in-time compilation. This seriously affects the execution time, for several reasons.

First, the just-in-time compilation process itself takes some time, contributing start-up overhead. Second, just-in-time compilation typically involves adaptive optimizations, based on the execution counts actually observed at runtime. If the code is executed only a few times, the just-in-time compiler may quickly generate slow code; if it is executed many times, the just-in-time compiler may spend more time generating faster code. Hence the speed of a piece of code is likely to depend on input parameters in complex ways, making it dubious to extrapolate from measurements with short execution times. Third, the just-in-time compiler, or its designers, will try to avoid time-consuming code analyses. This means that code that gets optimized well in a simple context may not be optimized at all in a more complex context that is harder to analyse. Fourth, managed platforms typically have automatic memory management, which in practice means that the garbage collector may decide to run at any time, affecting the code execution measurements in unpredictable ways.

In addition, all this software typically runs on top of complex operating systems (MacOS, Linux, Windows), complex processors (Intel i9, AMD EPYC, Apple M3), and complex memory management systems (caches, memory management hardware), adding further uncontrollable or unknown parameters.

2 Measuring execution time

The time spent executing some software may be measured in many different ways: wall-clock time or elapsed time (that is, the total amount of real time that has passed), CPU time spent by the client code only (measured by the operating system), CPU time spent by the client code and any system calls it made, CPU time plus time spent doing input and output, and more.

Here we shall measure simply wall-clock time, to enable comparison between platforms (Java, C#, C, Linux, MacOS, Windows). Measuring wall-clock time, not just CPU time, means that the measurements are easily affected by other processes running on the same machine, so one should terminate as many of those processes as possible. See advice in section 7.

2.1 A simple timer class for Java

This Java class Timer measures wall-clock time and is portable over most Java execution platforms (in particular Linux, MacOS and Windows):

```
public class Timer {
  private long start, spent = 0;
  public Timer() { play(); }
  public double check() { return (System.nanoTime()-start+spent)/1e9; }
  public void pause() { spent += System.nanoTime()-start; }
  public void play() { start = System.nanoTime(); }
}
```

The timer is started when the Timer object is created. The elapsed time (in seconds) is returned whenever `check()` is called. The methods `pause` and `play` suspend and resume the timer; see section 4.4.

2.2 A simple timer class for C#

A similar Timer class for C#, measuring wall-clock time in seconds, can be defined like this:

```
public class Timer {
  private readonly System.Diagnostics.Stopwatch stopwatch
    = new System.Diagnostics.Stopwatch();
  public Timer() { Play(); }
  public double Check() { return stopwatch.ElapsedMilliseconds/1000.0; }
  public void Pause() { stopwatch.Stop(); }
  public void Play() { stopwatch.Start(); }
}
```

3 Developing basic microbenchmark code in Java

Our microbenchmark code is intended to measure CPU time, memory access time and any associated systems overhead. This may be used to compare different data structures or algorithms, to compare different implementations of methods, or to compare speed of access to different levels of the memory hierarchy.

In this section we shall develop, step by step, minimal basic microbenchmark methods in Java. To this end, assume that you want to measure the time to call and execute this little method:

```
private static double multiply(long i) {
  double x = 1.1 * (double)(i & 0xFF);
   return x * x * x * x * x * x * x * x * x * x * x * x
        * x * x * x * x * x * x * x * x * x * x * x * x;
}
```

Each call to the method performs 20 floating-point multiplications, as well as an integer bitwise "and" (`&`) and a conversion of a 64-bit `long` to a 64-bit `double`. Performing these operations should take 10–50 CPU cycles, or 2.5–20 nanoseconds (ns), on a contemporary laptop or server such a MacBook with an Apple M3 pro with a clock frequency of 4 GHz, since 10 / 4 GHz equals 2.5 ns.

The purpose of the `(double)(i & 0xFF)` part is to make the computation dependent on the unknown argument `i` instead of a constant. The `(i & 0xFF)` part just makes sure that `x` is between 0 and 255, in other words, it does not grow very large. The dependency on the argument `i` prevents the compiler from simply performing the multiplications once and for all (before the method is called), either when the program is compiled to bytecode by `javac`, or when the bytecode is compiled to native machine code by the `java` just-in-time compiler.

We shall now consider a sequence of increasingly meaningful ways to measure the time it takes to perform a call of method `multiply`.

3.1 Mark0: measure one operation (useless)

The simplest attempt is to start the timer, call the function, and measure and print the elapsed time in nanoseconds (ns):

```
public static void Mark0() {          // USELESS
  Timer t = new Timer();
  double dummy = multiply(10);
  double time = t.check() * 1e9;
  System.out.printf("%6.1f ns%n", time);
}
```

However, this is useless for many reasons. Running a Java program that calls `Mark0` from the command line a couple of times may produce output like this:

```
2375.0 ns
2250.0 ns
2916.0 ns
```

First, this is far too slow for 20 multiplications, and secondly, the numbers vary widely. Possibly we are measuring the time it takes to compile the `multiply` method (and maybe the Timer's `check` method) from bytecode to native machine code; it is hard to know, but the results are certainly wrong. Moreover, on some platforms the timer resolution is so low that the result is always 0.0 ns, which is equally wrong. From this measurement you can conclude nothing at all.

3.2 Mark1: measure many operations (nearly useless)

A much better approach is to start the timer, execute the function many times, here 1 million times, and print the elapsed time divided by 1 million:

```
public static void Mark1() {          // NEARLY USELESS
  Timer t = new Timer();
  long count = 1_000_000;
  for (long i=0; i<count; i++) {
    double dummy = multiply(i);
  }
  double time = t.check() * 1e9 / count;
  System.out.printf("%6.1f ns%n", time);
}
```

The CPU now also has to perform the test on loop counter `i`, the increment `i++` and so on, which affects the time spent. On a modern CPU such as Intel i9 or Apple M3, hardware instruction-level parallelism means that these operations are likely to be executed in parallel with the multiplication. In fact, the results of a couple of runs seem rather reasonable:

```
4.0 ns
3.9 ns
4.4 ns
```

However, there is still a little variation from run to run. Worse, if you increase `count` to 100 million in an attempt to reduce variation between runs, then the execution time drops to 0.3 ns, which is highly implausible:

```
0.3 ns
0.3 ns
0.3 ns
```

3.3 Mark2: avoid dead code elimination

What happened in `Mark1` probably was that the JIT compiler realized that the result of `multiply` was never used, so the loop has no effect at all — it is dead code — and therefore the loop was removed completely. Hence the implausibly low execution times.

To avoid this, we change the benchmark loop to pretend that the result is used, by adding it to a dummy variable whose value is returned by the method:

```
public static double Mark2() {
  Timer t = new Timer();
  long count = 100_000_000;
  double dummy = 0.0;
  for (long i=0; i<count; i++)
    dummy += multiply(i);
  double time = t.check() * 1e9 / count;
  System.out.printf("%6.1f ns%n", time);
  return dummy;
}
```

Running this program multiple times we now get more consistent and plausible results, like these:

```
1.9 ns
1.9 ns
1.9 ns
```

These execution times are for a laptop (Apple M3 pro 4 GHz) and somewhat reasonable for 20 floating-point multiplications, one floating-point addition, and a few integer operations, though faster than our initial estimate of 2.5–20 ns. On a server (AMD EPYC 7313 3 GHz) one gets 3.8 ns reproducibly.

3.4 Mark3: automate multiple runs

Our next step is a simple convenience to automate the execution of multiple runs, here `n = 10` times, instead of having to run the entire program multiple times:

```
public static double Mark3() {
  long n = 10;
  long count = 100_000_000;
  double dummy = 0.0;
  for (long j=0; j<n; j++) {
    Timer t = new Timer();
    for (long i=0; i<count; i++)
      dummy += multiply(i);
    double time = t.check() * 1e9 / count;
    System.out.printf("%6.1f ns%n", time);
  }
  return dummy / n;
}
```

The output gives a good idea of the variation, which here is small:

```
1.9 ns
1.9 ns
1.8 ns
1.8 ns
1.8 ns
1.9 ns
1.8 ns
1.8 ns
1.8 ns
1.8 ns
```

This is much more satisfactory. All the iterations take nearly the same amount of time, and the results are quite consistent with the `Mark2` measurements.

3.5 Mark4: compute standard deviation

Instead of printing all the measurements we could compute and print the empirical mean and standard deviation of the measurements:

```
public static double Mark4() {
  long n = 10;
  long count = 100_000_000;
  double dummy = 0.0;
  double st = 0.0, sst = 0.0;
  for (long j=0; j<n; j++) {
    Timer t = new Timer();
    for (long i=0; i<count; i++)
      dummy += multiply(i);
    double time = t.check() * 1e9 / count;
    st += time;
    sst += time * time;
  }
  double mean = st/n, sdev = Math.sqrt((sst - mean*mean*n)/(n-1));
  System.out.printf("%6.1f ns +/- %6.3f%n", mean, sdev);
  return dummy / n;
}
```

This executes the same benchmark runs but reduces the amount of output:

```
1.8 ns +/-  0.023
```

This result is pleasantly consistent with those obtained from `Mark3` above. However, be warned that in general, one should not expect such consistency for very short methods.

The standard deviation summarizes the variation between the 10 measurements, removing the need to inspect them all manually. Here the standard deviation is 78 times smaller than the mean 1.8, so the estimate of the mean is quite reliable. A statistics course would make this statement more precise.

3.6 Mark5: choosing the iteration count automatically

The number of iterations (`count` equals 100 million) used above was rather arbitrarily chosen. While it probably is sensible for the small `multiply` function, it may be far too large when measuring a more time-consuming function. To automate the choice of `count`, and to see how the mean and standard deviation are affected by the number of iterations, we add an outer loop. In this outer loop we double the `count` until the execution time is at least 0.25 seconds:

```
public static double Mark5() {
  long n = 10, count = 1, totalCount = 0;
  double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
  do {
    count *= 2;
    st = sst = 0.0;
    for (long j=0; j<n; j++) {
      Timer t = new Timer();
      for (long i=0; i<count; i++)
        dummy += multiply(i);
      runningTime = t.check();
      double time = runningTime * 1e9 / count;
      st += time;
      sst += time * time;
      totalCount += count;
    }
    double mean = st/n, sdev = Math.sqrt((sst - mean*mean*n)/(n-1));
    System.out.printf("%6.1f ns +/- %8.2f %10d%n", mean, sdev, count);
  } while (runningTime < 0.25);
  return dummy / totalCount;
}
```

This might produce output like this:

```
239.5 ns +/-   457.91          2
 83.4 ns +/-    30.29          4
 75.5 ns +/-    19.51          8
 90.6 ns +/-    44.45         16
 76.2 ns +/-    27.74         32
 66.1 ns +/-     8.95         64
 60.2 ns +/-    12.24        128
 55.3 ns +/-     2.02        256
 54.1 ns +/-     0.25        512
 54.2 ns +/-     0.53       1024
 33.0 ns +/-    14.74       2048
 19.4 ns +/-     0.19       4096
  4.1 ns +/-     0.21       8192
  4.1 ns +/-     0.17      16384
  4.1 ns +/-     0.03      32768
  3.2 ns +/-     0.98      65536
```

```
2.3 ns +/-     0.01     131072
2.2 ns +/-     0.00     262144
2.1 ns +/-     0.05     524288
1.9 ns +/-     0.07    1048576
1.8 ns +/-     0.01    2097152
1.8 ns +/-     0.02    4194304
1.8 ns +/-     0.01    8388608
1.8 ns +/-     0.01   16777216
1.9 ns +/-     0.21   33554432
1.8 ns +/-     0.01   67108864
1.9 ns +/-     0.01  134217728
```

As one might hope, the mean (first column) converges to around 1.8 ns, and the standard deviation (second column) becomes smaller, as `count` (third column) grows. The end result is also reasonably consistent with that of `Mark4`.

At count 2048, the standard deviation 14.74 is large compared to the mean 33.0 ns, so we cannot have confidence in that particular result. Probably this is caused by the Java just-in-time compiler deciding to further optimize the code at that point. This makes one execution of `multiply` very slow and the subsequent ones faster, thereby causing a high standard deviation.

3.7 Summary of basic benchmarking functions

Let us summarize the steps that led to the design of our benchmarking function `Mark5`:

- `Mark0` measured a single function call, but that included some unknown virtual machine start up costs, and also ran for too short a time to be measured accurately, and hence was useless.
- `Mark1` improved on `Mark0` by measuring a large number of function calls, hence distributing the start up costs on these. Unfortunately, the just-in-time compiler might decide to optimize away all the function calls because their results were not used.
- `Mark2` improved on `Mark1` by accumulating the function results in the `dummy` variable and returning its value, thereby preventing the just-in-time compiler from optimizing away the calls. Unfortunately, it made just one timing measurement (of `count` function calls), thus not allowing us to estimate the reliability of the reported mean.
- `Mark3` improved on `Mark2` by making a set of 10 measurements, so one can inspect the variation of the mean. Unfortunately, visual inspection of the variation is cumbersome when making many sets of measurements, for instance of many different functions.
- `Mark4` improved on `Mark3` by additionally computing the standard deviation and printing only the mean and the standard deviation for each set of 10 measurements. Unfortunately, the number `count` of function calls in each set of measurements was fixed but chosen arbitrarily, and might be far too large for some functions we want to benchmark.
- `Mark5` improved on `Mark4` by choosing `count` to be 2 initially, and doubling it until the running time of the last measurement (in a set) was at least 0.25 seconds, long enough to avoid problems with virtual machine startup and clock resolution.

Hence we have arrived at a somewhat reliable benchmarking function, `Mark5`. We further develop it into actually useful general Java benchmarking functions in the next section — and corresponding ones for C# in section 5 and for C in section 6.

4 General microbenchmark code in Java

Now that we are happy with the basic benchmarking code, we generalize it so that we can measure many different functions (not just `multiply`) more easily, for instance to compare alternative algorithms or implementations.

4.1 Mark6: generalize to any function

The functional interface java.util.function.LongToDoubleFunction describes method `applyAsDouble`:

```
public interface LongToDoubleFunction {
  double applyAsDouble(long value);
}
```

Any function from `long` to `double` can be represented by this interface. A general benchmarking function `Mark6` would simply take a LongToDoubleFunction as argument and hence work for any such function, not just `multiply`:

```
public static double Mark6(char tunit, String msg, LongToDoubleFunction f) {
  long n = 10, count = 1, totalCount = 0;
  double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
  do {
    count *= 2;
    st = sst = 0.0;
    for (long j=0; j<n; j++) {
      Timer t = new Timer();
      for (long i=0; i<count; i++)
        dummy += f.applyAsDouble(i);
      runningTime = t.check();
      double time = runningTime * scale(tunit) / count;
      st += time;
      sst += time * time;
      totalCount += count;
    }
    double mean = st/n, sdev = Math.sqrt((sst - mean*mean*n)/(n-1));
    System.out.printf("%-25s %15.1f %cs %10.2f %10d%n", msg, mean,
                      tunit, sdev, count);
  } while (runningTime < 0.25);
  return dummy / totalCount;
}
```

In addition to the method to be benchmarked, the benchmarking function takes two new arguments. The first one is the time unit parameter `tunit`, which must be `'n'` or `'u'` or `'m'` for ns = nanosecond, us = microsecond and ms = millisecond, respectively. It affects only the printing of the measured time and allows `Mark6` to be used for both very fast (nanosecond) functions and slower (microsecond or millisecond) functions. The second new parameter is a string `msg` that describes the function `f` being measured.

The function `scale` converts the time unit parameter `tunit` to a scaling factor:

```
private static double scale(char tunit) {
  return tunit=='n' ? 1e9 : tunit=='u' ? 1e6 : tunit=='m' ? 1e3 : 0;
}
```

The printed output has been simplified so that it is easier to further process with other software, for instance to make plots with MS Excel (section 8.2) or gnuplot (section 8.3) or Python Matplotlib (section 8.4).

We can now apply the general benchmarking method `Mark6` to `Benchmark::multiply`, which is a method reference denoting the `multiply` method:

```
Mark6('n', "multiply", Benchmark::multiply);
```

The result may look like this, which says that function `multiply` used on average 1.8 ns per call when called 268 million times, and that the standard deviation between 10 such runs was 0.00:

```
multiply                              275.1 ns     482.04           2
multiply                              126.1 ns      44.04           4
multiply                               90.1 ns      21.26           8
multiply                              137.0 ns      61.64          16
multiply                              104.9 ns      21.43          32
multiply                               81.0 ns       9.16          64
multiply                               58.6 ns      94.60         128
multiply                               27.6 ns       5.14         256
multiply                               27.2 ns       6.72         512
multiply                               22.2 ns       0.45        1024
multiply                               21.2 ns       0.21        2048
multiply                               21.1 ns       0.31        4096
multiply                                5.5 ns       0.81        8192
multiply                                5.3 ns       0.12       16384
multiply                                5.2 ns       0.05       32768
multiply                                5.0 ns       0.09       65536
multiply                                2.4 ns       0.62      131072
multiply                                2.1 ns       0.00      262144
multiply                                2.0 ns       0.03      524288
multiply                                1.9 ns       0.05     1048576
multiply                                1.8 ns       0.01     2097152
multiply                                1.8 ns       0.02     4194304
multiply                                1.8 ns       0.01     8388608
multiply                                1.8 ns       0.01    16777216
multiply                                1.8 ns       0.00    33554432
multiply                                1.8 ns       0.00    67108864
multiply                                1.8 ns       0.01   134217728
multiply                                1.8 ns       0.00   268435456
```

This is neat and consistent with the previous measurements. In any case, it seems that the extra call through the functional interface LongToDoubleFunction does not in itself slow down the program, presumably because the just-in-time (JIT) compiler inlines the called function (`multiply`) and thereby avoids the cost of the call.

4.2 Mark7: print only the final measurement

When the mean converges and the standard deviation decreases, as in the `multiply` example just shown, it suffices to print the results for the final value of `count`, which can be achieved simply by moving the computation and printing of `mean` and `sdev` out of the `do-while` loop in `Mark6`. This is the only difference between `Mark6` and the new `Mark7`:

```
public static double Mark7(char tunit, String msg, LongToDoubleFunction f) {
  ...
  do {
    ...
  } while (runningTime < 0.25);
  double mean = st/n, sdev = Math.sqrt((sst - mean*mean*n)/(n-1));
  System.out.printf("%-25s %15.1f %cs %10.2f %10d%n", msg, mean,
                    tunit, sdev, count);
  return dummy / totalCount;
}
```

Printing just the final value makes sense if one assumes that the optimizations performed by the just-in-time compiler and runtime system stabilize eventually, but not if the just-in-time compiler indefinitely tries and retries new optimizations and generates new machine code.

Using `Mark7` on `multiply` is easy:

```
Mark7('n', "multiply", Benchmark::multiply);
```

and produces a result consistent with the final result of Mark6:

```
multiply                                1.8 ns       0.00  268435456
```

When benchmarking many different functions, the `Mark7` simplification of the output considerably improves the overview, and we can now benchmark many functions in one go. Here we pass a lambda expression (of type LongToDoubleFunction) in each call:

```
Mark7('n', "pow",  i -> Math.pow(10.0, 0.1 * (i & 0xFF)));
Mark7('n', "exp",  i -> Math.exp(0.1 * (i & 0xFF)));
Mark7('n', "log",  i -> Math.log(0.1 + 0.1 * (i & 0xFF)));
Mark7('n', "sin",  i -> Math.sin(0.1 * (i & 0xFF)));
Mark7('n', "cos",  i -> Math.cos(0.1 * (i & 0xFF)));
Mark7('n', "tan",  i -> Math.tan(0.1 * (i & 0xFF)));
Mark7('n', "asin", i -> Math.asin(1.0/256.0 * (i & 0xFF)));
Mark7('n', "acos", i -> Math.acos(1.0/256.0 * (i & 0xFF)));
Mark7('n', "atan", i -> Math.atan(1.0/256.0 * (i & 0xFF)));
```

This may produce results such as the following:

```
pow                                    29.1 ns       0.05   16777216
exp                                     6.6 ns       0.14   67108864
log                                     7.3 ns       0.05   67108864
sin                                     6.2 ns       0.01   67108864
cos                                     6.2 ns       0.04   67108864
tan                                    16.4 ns       0.11   16777216
asin                                    5.8 ns       0.01   67108864
acos                                    5.9 ns       0.03   67108864
atan                                    6.2 ns       0.02   67108864
```

A word of warning: Whereas these results are quite reliable for more complicated functions such as `pow`, `exp` and trigonometry, the result for method `multiply` or even simpler ones may not be. What is measured in that case is really the just-in-time compiler's ability to inline function calls and schedule machine instructions in a particular context. For instance, on other hardware and with a different Java implementation, one might get rather different execution times for `multiply`, all apparently with low standard deviation.

4.3 Mark8: problem size parameters

Often, the execution time of a method depends on the problem size. For instance, we may want to measure the execution time of a binary array search method:

```
static int binarySearch(int x, int[] arr) { ... }
```

The method returns the index of item `x` in sorted array `arr`, or returns -1 if `x` is not in the array. Clearly the execution time depends on the size of the array, so we would want to measure it for a range of such array sizes. Moreover, we would like the array size to be printed along with the measured mean execution time and the standard deviation.

Thus we define yet another benchmarking method that has an additional `info` parameter, and prints it after the left-justified `msg` parameter but right-justified, as is suitable for a string representing a numerical value:

```
public static double Mark8(char tunit, String msg, String info,
                           LongToDoubleFunction f) {
  long n = 10, count = 1, totalCount = 0;
  double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
  do {
    count *= 2;
    st = sst = 0.0;
    for (long j=0; j<n; j++) {
      Timer t = new Timer();
      for (long i=0; i<count; i++)
        dummy += f.applyAsDouble(i);
      runningTime = t.check();
      double time = runningTime * scale(tunit) / count;
      st += time;
      sst += time * time;
      totalCount += count;
    }
  } while (runningTime < 0.25);
  double mean = st/n, sdev = Math.sqrt((sst - mean*mean*n)/(n-1));
  System.out.printf("%-25s %s%15.1f %cs %10.2f %10d%n", msg, info, mean,
                    tunit, sdev, count);
  return dummy / totalCount;
}
```

Note that the repeat count `n` (here 10) and minimal execution time (here 0.25 s) are rather arbitrarily chosen, and could be made into parameters instead, but we will not do that.

To measure the binary search execution time for a range of array sizes, we run this benchmark:

```
for (int size = 100; size <= 10_000_000; size *= 2) {
  final int[] intArray = SearchAndSort.fillIntArray(size);
  final int successItem = (int)(0.49 * size);
  Mark8('n', "binary_search_success",
        String.format("%8d", size),
        i -> SearchAndSort.binarySearch(successItem, intArray));
}
```

Notice the use of `String.format` to make sure that the `info` string always has the same length, so that the ASCII output continues to line up in columns. Indeed, a possible result from running this benchmark may be:

```
binary_search_success          100            1.3 ns       0.00  268435456
binary_search_success          200            5.9 ns       0.05   67108864
binary_search_success          400            6.8 ns       0.08   67108864
binary_search_success          800            5.0 ns       0.02   67108864
binary_search_success         1600            7.8 ns       0.01   33554432
binary_search_success         3200            8.8 ns       0.13   33554432
binary_search_success         6400            6.8 ns       0.02   67108864
binary_search_success        12800           11.0 ns       0.01   33554432
binary_search_success        25600           13.4 ns       0.01   33554432
binary_search_success        51200           14.3 ns       0.02   33554432
binary_search_success       102400           15.4 ns       0.02   16777216
binary_search_success       204800           14.2 ns       0.03   33554432
binary_search_success       409600           17.7 ns       0.03   16777216
binary_search_success       819200           16.5 ns       0.01   16777216
binary_search_success      1638400           20.1 ns       0.03   16777216
binary_search_success      3276800           21.2 ns       0.04   16777216
binary_search_success      6553600           20.1 ns       0.06   16777216
```

This looks neat, but these results probably seriously underestimate realistic execution times. The problem is not that we measure in the wrong way, but that we measure the wrong thing: it is not realistic to search for the same item (`successItem` above) again and again in the millions of repetitions performed for each row in the result. See section 9.1 for a discussion and more realistic results.

4.4 Mark8Setup: Benchmarks that require setup

Sometimes a method that we want to benchmark will modify or destroy the data it works on, so the data must be restored before each execution of the code. For instance, an array sort procedure will leave the argument array sorted. Since there is little point in measuring how long it takes to sort an already-sorted array, the array disorder must be reestablished before the next benchmark run, for instance by randomly shuffling the array.

We define new method `Mark8Setup` that differs from `Mark8` only in taking an extra argument `setup` of type Runnable, a method that takes no argument and returns no result. It will be called before each measurement of `f`, but the reported time for `f` will not include the setup time, because the timer is suspended (section 2.1) before the call to `setup` and resumed after it:

```
static double Mark8Setup(char tunit, String msg, String info,
                         Runnable setup, LongToDoubleFunction f) {
  long n = 10, count = 1, totalCount = 0;
  double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
  do {
    count *= 2;
    st = sst = 0.0;
    for (long j=0; j<n; j++) {
      Timer t = new Timer();
      for (long i=0; i<count; i++) {
        t.pause();
        setup.run();
        t.play();
        dummy += f.applyAsDouble(i);
      }
      runningTime = t.check();
      double time = runningTime * scale(tunit) / count;
      st += time; sst += time * time; totalCount += count;
    }
```

```
  } while (runningTime < 0.25);
  double mean = st/n, sdev = Math.sqrt((sst - mean*mean*n)/(n-1));
  System.out.printf("%-25s %s%15.1f %cs %10.2f %10d%n", msg, info,
                    mean, tunit, sdev, count);
  return dummy / totalCount;
}
```

This method should be used only to benchmark functions that take some substantial amount of time (say, at least 10 ns/call) to execute, otherwise the benchmark will mostly measure the amount of time it takes to suspend and resume the timer.

Let us use it to compare three algorithms for sorting an array `intArray` of 10,000 integers integers: selection sort, quicksort, and heap sort. We use a random permutation (shuffle) of the `intArray` as the `setup` function, and set the time unit to `'u'` for microseconds to avoid excess digits in the output:

```
final int[] intArray = SearchAndSort.fillIntArray(10_000);
Mark7('u', "shuffle int",
      i -> { SearchAndSort.shuffle(intArray); return 0.0; });
Mark8Setup('u', "shuffle", "",
           () -> { },
           i -> { SearchAndSort.shuffle(intArray); return 0.0; });
Mark8Setup('u', "selection_sort", "",
           () -> SearchAndSort.shuffle(intArray),
           i -> { SearchAndSort.selsort(intArray); return 0.0; });
Mark8Setup('u', "quicksort", "",
           () -> SearchAndSort.shuffle(intArray),
           i -> { SearchAndSort.quicksort(intArray); return 0.0; });
Mark8Setup('u', "heapsort", "",
           () -> SearchAndSort.shuffle(intArray),
           i -> { SearchAndSort.heapsort(intArray); return 0.0; });
```

Here are the results from one execution. They show that selection sort is 36 times slower than quicksort, and that heap sort is 1.1 times slower than quicksort, on random-ordered arrays with 10000 elements:

```
selection_sort                    15567.6 us      55.21         32
quicksort                           433.6 us       1.17       1024
heapsort                            478.9 us       1.04       1024
```

4.5 Execution time as a function of problem size

We can further use `Mark8Setup` to measure the time to sort arrays of size 100, 200, 400, and so on:

```
for (int size = 100; size <= 2_000_000; size *= 2) {
  final int[] intArray = SearchAndSort.fillIntArray(size);
  Mark8Setup('u', "quicksort",
             String.format("%8d", size),
             () -> SearchAndSort.shuffle(intArray),
             i -> { SearchAndSort.quicksort(intArray); return 0.0; });
}
```

The result may look like this:

```
quicksort                      100           2.6 us       0.00     131072
quicksort                      200           5.8 us       0.04      65536
quicksort                      400          12.8 us       0.05      32768
quicksort                      800          28.0 us       0.08      16384
```

```
quicksort                    1600          60.9 us       0.13       4096
quicksort                    3200         130.0 us       0.43       2048
quicksort                    6400         280.4 us       1.11       1024
quicksort                   12800         598.2 us       1.57        512
quicksort                   25600        1271.1 us       3.85        256
quicksort                   51200        2683.5 us      50.21        128
quicksort                  102400        5630.9 us      11.90         64
quicksort                  204800       12004.4 us     200.55         32
quicksort                  409600       25111.6 us     309.36         16
quicksort                  819200       52137.8 us     470.08          8
quicksort                 1638400      108483.7 us     325.22          4
```

Note that the standard deviation is now larger than previously, though never more than 2 % of the mean.

The Java benchmarking functions `Mark6`, `Mark7`, `Mark8` and `Mark8Setup` developed in this section and the examples can be found in file `Benchmark.java` and `SearchAndSort.java`. They are quite robust and good enough for our purposes. Sections 5 and 6 present C# and C versions of them. Sections 9 and 10 present more applications.

5 Benchmarking in C#

File `Benchmark.cs` contains C# versions of the general Java benchmarking methods shown in section 4. They are identical apart from small differences caused by the language, such as using C#'s type Action instead of Java's type Runnable:

```
double Mark6(char tunit, String msg, Func<long,double> f)
double Mark7(char tunit, String msg, Func<long,double> f)
double Mark8(char tunit, String msg, String info, Func<long,double> f)
double Mark8Setup(char tunit, String msg, String info,
                  Action setup, Func<long,double> f)
```

The C# methods produce output in precisely the same columnar text format as the Java methods. File `Benchmark.cs` also contains many of the same examples. For instance, a sorting scalability benchmark corresponding to the Java one shown in section 4.5 can be written just like the Java one:

```
for (int size = 100; size <= 2000000; size *= 2) {
  int[] intArray = SearchAndSort.FillIntArray(size);
  Mark8Setup('u', "quicksort",
             String.Format("{0,8:D}", size),
             () => { SearchAndSort.Shuffle(intArray); },
             (long i) => { SearchAndSort.Quicksort(intArray); return 0.0; });
}
```

6 Benchmarking in C

Since C code is always compiled ahead of time by `clang` or `gcc` or another C compiler, not at run-time by a just-in-time compiler, benchmarking of C code has fewer pitfalls than benchmarking of Java or C#. Nevertheless, some of the concerns are the same: dead code elimination, the need to measure repeated execution of fast functions to obtain reliable results, and so on.

To permit comparison of implementations in different languages, in file `benchmark.c` we provide C versions of some of the Java benchmarking functions from section 4, specifically:

```
double Mark6(char tunit, char* msg, double (*f)(int64_t))
double Mark7(char tunit, char* msg, double (*f)(int64_t))
double Mark8Setup(char tunit, char* msg,
                  void (*setup)(), double (*f)(int64_t))
```

The function parameters have the same meaning as in the Java and C# versions. The complicated C notation `double (*f)(int64_t)` says that `f` is a pointer to a function that takes an argument of type `int64_t` and returns a result of type `double`, where `int64_t` is C's version of the 64-bit integer type `long` in Java and C#. Recall that C has a `long` type but that it has different sizes on different platforms. Similarly, `void (*setup)()` means that `setup` is a pointer to a function that takes no arguments and returns no result (`void`).

These C functions produce output in precisely the same columnar text format as the Java and `C#` versions, so the output can be processed by the same plotting scripts from section 8.

Unlike Java and C#'s function closures, a C function pointer refers only to the function's code, not to an environment for the function's free variables. This makes it more cumbersome to pass function-type arguments such as `f` and `setup`. For instance, in Java, one can easily call `Mark8Setup` with a setup function and a sorting function as argument, as we did in section 4.5:

```
Mark8Setup('u', "quicksort",
           String.format("%8d", size),
           () -> SearchAndSort.shuffle(intArray),
           i -> { SearchAndSort.quicksort(intArray); return 0.0; });
```

Both of the functions `() -> ...` and `i -> ...` have `intArray`, the array to be shuffled or sorted, as a free variable. Doing this in a modular manner in C involves complications that we want to avoid. Instead, we take a simpler approach, where we make `intarray` a global variable along with its size `arraysize` and along with functions to allocate the array, fill it with numbers, shuffle it, and so on:

```
int arraysize;
int *intarray;       // Array with arraysize elements
void intarrayAlloc(int size) {
  arraysize = size;
  intarray = (int*)malloc(arraysize * sizeof(int));
}
void fillOrdered() {
  for (int i=0; i<arraysize; i++)
    intarray[i] = i;
}
void shuffle(int* arr, int n) {
  for (int i = n-1; i > 0; i--)
    swap(arr, i, random() % (i+1));
}
```

Then we can define auxiliary functions `fillShuffle` and `callquicksort` that refer to global variables and functions, yet have precisely the types required for `setup` and `f` above:

```
void fillShuffle() {
  fillOrdered();
  shuffle(intarray, arraysize);
}
double callquicksort(int64_t i) {
  myquicksort(intarray, arraysize);
  return intarray[0];
}
```

These latter C functions are a form of "poor man's closures", referring to global variables instead of local ones. Now we can use these auxiliary functions to do sorting scalability experiments as in section 4.5, this time in C:

```
for (int size = 100; size <= 2000000; size *= 2) {
  intarrayAlloc(size);
  char buffer[64];
  snprintf(buffer, sizeof(buffer), "%-15s %8d", "quicksort", size);
  dummy += Mark8Setup('u', buffer, fillShuffle, callquicksort);
  free(intarray);
}
```

There are certainly more general and modular ways to simulate function closures in C, for instance with a struct containing a function pointer as well as data fields that represent the closure's environment. However, these extraneous complications would obscure the basic simplicity of the benchmarking framework, so we prefer the simpler approach.

The C benchmark implementation contains an `#ifdef` to handle the Windows platform specially for platform identification, wall-clock timing and random number generation. In particular, Windows C does not have the standard C `random` function, and the Windows `rand` function produces only 15 bits of randomness which ruins some benchmark results. Hence, under Windows `benchmark.c` defines a simple but better 64-bit linear congruential `random` function used in the musl project [10].

7 Advice and warnings

We have already warned against some pitfalls of performance measurements, and offered some advice. Here is a more complete list, but we recommend to read also the advice from the JVM authors in [13].

- To minimize interference from other software during benchmark runs, you should shut down as many external programs as possible, and in particular web browsers, Teams, Zoom, MS Office, mail clients, virus checkers, web servers, database servers and AI inference processes running on the same machine. The Windows Update mechanism seems particularly prone to seriously slow down the machine at inopportune times. Although this probably affects the disk subsystem more than CPU times and memory accesses, it appears to have some effect on those too.

- Remove or disable all logging and debugging messages in the code being benchmarked. Generating such messages can easily spend more than 90 per cent of the execution time, completely distorting the measurements, and hiding even major speed-ups to the actual data processing.

- Never measure performance from inside an integrated development environment (IDE) such as Eclipse, IntelliJ or VS Code; use the command line. Even in "release builds" and similar, the software may be slower due to injected debugging code, some form of execution supervision, or just because the IDE itself consumes a lot of CPU time.

- Turn off power savings schemes in your operating system, to prevent them from reducing CPU speed in the middle of the benchmark run. In particular, some laptops by default reduce CPU speed when running on battery instead of mains power.

- Compile and run code with relevant optimizations options, where available. For instance, C# code should be built and run with option `-c Release`, as in `dotnet run -c Release`. Without this option, some optimizations may be disabled to accommodate debugging. Similarly, for C code one should use relevant optimization levels, such as `clang -O3` or `gcc -O3`, or at the very least `-O1`.

- Different implementations, versions and builds of the Java Virtual Machine (Apple, Eclipse, Microsoft, Oracle), different implementations of .NET, and different Javascript engines (Apple, Google, Mozilla) may have very different performance characteristics. General statements about "the speed of Java" or "the speed of C#" or "the speed of Javascript" should be taken with large spoonfuls of salt.

- Be aware that the Java or .NET virtual machines incorporate several different garbage collectors, which may heavily influence performance in functional and object-oriented programming.

- Be aware that different C compilers (`clang`, Microsoft MSVC, `gcc`) generate code of very different quality, and that the same compiler will generate very different code depending on the optimization level (`-O1`, `-O2`, and so on). Also, most of these compilers accept a large number of other optimization-related options, allowing detailed control over code generation. Hence statements about "the speed of C" should be taken with large spoonfuls of salt.

- Be aware that different CPU types (Intel, AMD, Arm64) and different CPU architectures (for instance Apple Arm64 for MacBooks or Broadcom Arm64 for Raspberry Pi) and different generations of these, may have different performance characteristics. Confusingly, different microarchitectures may be marketed under the same name, and the same microarchitecture may be marketed under different names. Hence, if this matters to you, take note of the processor model number. Also, the rest of the system around the CPU matters: the speed of RAM, the size and organization of caches, the latency and bandwidth of the memory bus, and so on.

- If you measure the time to execute threads, input-output, GPU calls or other asynchronous activities, make sure that you wait for these activities to actually compute and terminate, for instance by calling `thread.join()`, before `timer.check()` is called.

- Reflect on your results. If they are unreasonably fast or unreasonably slow, you may have overlooked something important, and there may be an opportunity to learn something.

  For example, once I found that array read access in C code on an AMD processor was unreasonably fast. It turned out that the array was never actually allocated in RAM; the memory pages were just entered into a CPU memory management table. By first writing to the array elements, actual allocation of the memory pages to RAM was forced, giving more realistic read access times.

  For a second example, some early Scala code seemed unreasonably slow. It turned out that since the code was written in a Scala Application trait, it was compiled to Java Virtual Machine (JVM) bytecode inside a constructor, which the JVM at the time did not optimize. Putting the Scala code into an ordinary Scala function made it 12 times faster.

  For a third example, random read access in C code seemed suspiciously fast for a large array. It turned out that a "random" permutation was generated with the Windows `rand()` function, which has only 32768 different values, so most of the array access was sequential, not random. With better randomness the access time became 18 times higher and more realistic.

8 Plotting benchmark results

Showing the benchmark results in tabular form is all well and good, but if the overall purpose is to compare scalability of multiple algorithms in multiple programming languages or on multiple machines, then a graphical presentation, or plot, is much better.

8.1 Collecting and further processing benchmark results

The output of a benchmark run, as shown for instance at the end of section 4.3, can be piped into a file for further processing by other software, such as plotting:

```
$ java Benchmark > binary-search.txt
```

8.2 Simple plotting with Excel

To import a data file such as that created in section 4.3 into MS Excel, choose `File > Import` and select the file. A Text Import Wizard dialog appears, where you should choose `Fixed width`, choose `Start import at row 5` to ignore the system information at the top of the file, then click `Next`. The wizard will usually guess the file's columns correctly, in which case you just click `Next` again, then `Finish`. Hint: If Excel uses decimal comma and the text file uses decimal point, or vice versa, then open the Advanced subdialog and change Excel's expectations; otherwise the import goes wrong.

The data appears in an Excel sheet, as shown in figure 1.

To display this kind of data, one should almost always use an xy plot (scatter plot), not any of the other fancy but more business-oriented chart types. Select the data in columns B and C, that is, range B1:C17, as shown in the figure, choose `Insert > Chart > X Y (Scatter)`, and choose the chart subtype with straight lines and markers. Smooth lines may look neater, but they distort the data. The result will be a chart as shown in figure 2. The appearance of such charts can be improved interactively by right-clicking on relevant chart elements, such as the axes, the plot lines, the data point markers, chart background, and so on.

8.3 Simple plotting with gnuplot

Using `gnuplot` one can plot the binary search data produced in section 4.3:

```
set term pdfcairo
set output "binary-search-plot.pdf"
plot "binary-search-1.txt" using 2:3:5 with errorlines
```

This says that the data (x, y, dy) are in columns 2, 3 and 5 of text file `binary-search-1.txt`, thus ignoring the descriptive text in column 1 and the `ns` in column 4, and that dy should be used for plotting y-value error bars at each data point (x, y). The plot is saved as a `PDF` file, but many other file formats are possible; type `set term` in a `gnuplot` session to see which ones. The plot is shown in figure 3.

Similarly, we can plot the running times of the three sorting algorithms benchmarked in section 4.5, using logarithmic axes because selection sort is so much slower than quicksort and heap sort:

```
set term pdfcairo
set output "sort-scalability-plot.pdf"
set logscale xy
plot "sort-scalability.txt" using 2:3:5 with errorlines
```

The resulting plot is shown in figure 4. Gnuplot has many options and features, see its manual [7].

	A	B	C	D	E	F
1	binary_search_success	100	1.3	ns	0	268435456
2	binary_search_success	200	5.9	ns	0.05	67108864
3	binary_search_success	400	6.8	ns	0.08	67108864
4	binary_search_success	800	5	ns	0.02	67108864
5	binary_search_success	1600	7.8	ns	0.01	33554432
6	binary_search_success	3200	8.8	ns	0.13	33554432
7	binary_search_success	6400	6.8	ns	0.02	67108864
8	binary_search_success	12800	11	ns	0.01	33554432
9	binary_search_success	25600	13.4	ns	0.01	33554432
10	binary_search_success	51200	14.3	ns	0.02	33554432
11	binary_search_success	102400	15.4	ns	0.02	16777216
12	binary_search_success	204800	14.2	ns	0.03	33554432
13	binary_search_success	409600	17.7	ns	0.03	16777216
14	binary_search_success	819200	16.5	ns	0.01	16777216
15	binary_search_success	1638400	20.1	ns	0.03	16777216
16	binary_search_success	3276800	21.2	ns	0.04	16777216
17	binary_search_success	6553600	20.1	ns	0.06	16777216

Figure 1: Binary search execution time data imported into Excel.

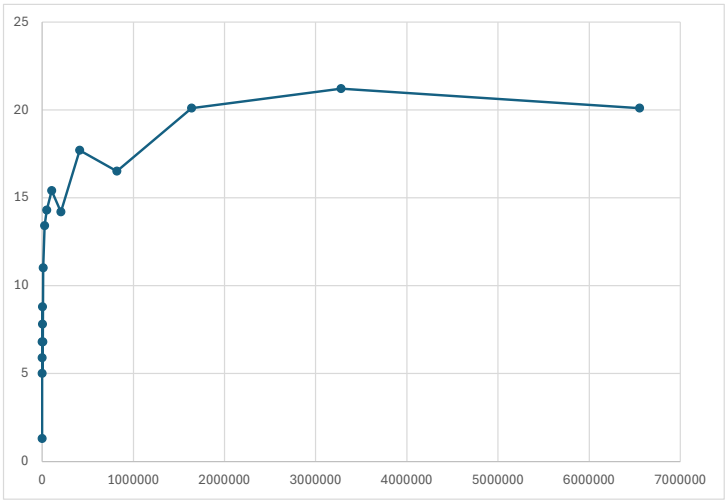

Figure 2: Excel scatter chart showing the binary search data from figure 1.

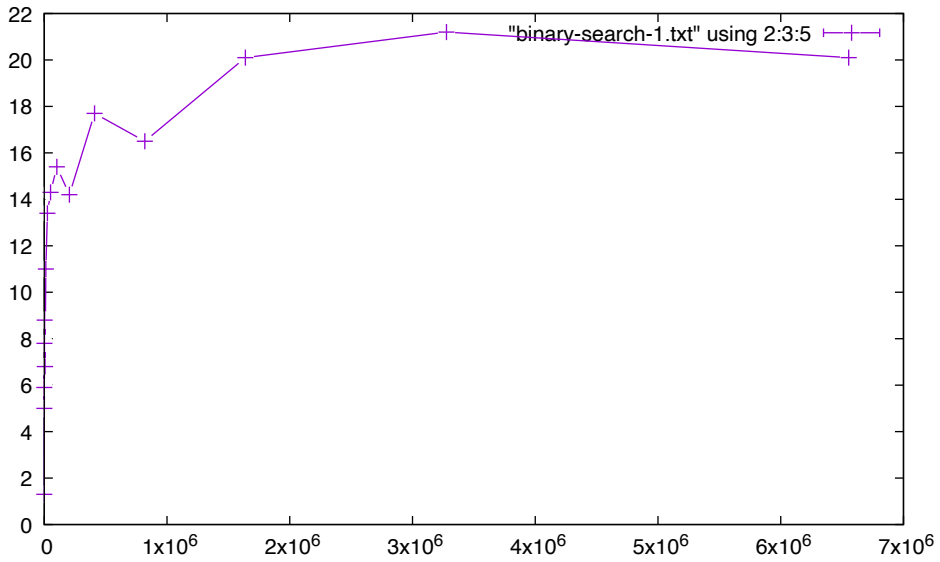

Figure 3: Binary search execution time as a function of array size. Graph produced by `gnuplot`. It displays the suspicious results from section 4.3, not the good ones from section 9.2.

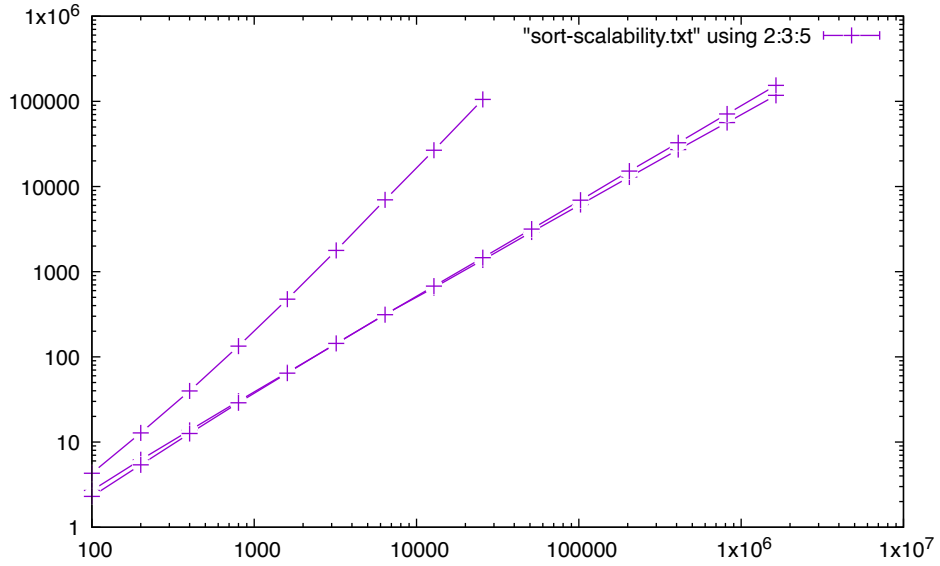

Figure 4: Three sorting algorithms, execution time as a function of array size. Graph produced by `gnuplot`. Note the logarithmic scale on both axes.

8.4 Simple plotting with Python Matplotlib

The binary search data can also be plotted using Matplotlib [9] and the Python programming language. Create a script file `plotbinsearch.py` that imports the `pandas` and `matplotlib` packages , and then uses the `pandas` function `read_fwf` to read the file `binary-search-2.txt` in fixed-width format (fwf) while specifying the names of the columns, like this:

```
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_fwf("binary-search-2.txt",
                   colspecs="infer",
                   names=["algo", "size", "mean", "skip1", "sdev", "skip2"],
                   header=None,
                   skiprows=4)
```

Now we have a dataframe object called `data` with columns `data['algo']`, `data['size']` and so on. Then the Matplotlib function `figure` can create a figure and function `plot` can plot the data on it, here with the `size` values on the `x` axis and the `mean` values on the `y` axis:

```
plt.figure(figsize=(10, 6))
plt.plot(data["size"], data["mean"], marker="o")
```

A bunch of other function calls set up the plot's title, axis descriptions and a start value for the `y` axis, adds grid lines, and finally saves the plot in `PDF` format to a file:

```
plt.title("Binary Search Time vs Array Size")
plt.xlabel("Number of array elements")
plt.ylabel("Mean wall-clock time (ns)")
plt.ylim(bottom=0)
plt.grid(True)
filename = f"binary-search-py.pdf"
plt.savefig(filename, format="pdf")
plt.close()
```

The script can be run from the command line using a Python system:

```
$ python3 plotbinsearch.py
```

The resulting plot is shown in figure 5. The Python approach may look somewhat cumbersome, but scripting is great for automation and can conveniently generate many plots (for instance, for many combinations of algorithms, datasets, programming languages and execution platforms) in one go.

Similarly, we can plot the running times of the three sorting algorithms benchmarked in section 4.5. Since a Python script `plotsortscalability.py` to do so is very similar to the previous script, we show only the actual plotting code below. As before we create a single figure. The expression `data.groupby("algo")` groups the observations by their `algo` value, so that all selection sort observations go together, all quicksort observations go together, and so on; the result is three groups each containing a dataframe corresponding to an `algo` label. Then we use a Python `for` loop to iterate over the groups, plotting an xy-curve for each group. The `label` parameter to the `plot` function creates a legend showing which colour line represents which sorting algorithm. We make both the `x` and `y` axes logarithmic, as in the `gnuplot` version:

```
plt.figure(figsize=(10, 6))
for algo, group in data.groupby("algo"):
    plt.plot(group["size"], group["mean"], marker="o", label=algo)
    plt.xscale("log")
    plt.yscale("log")
```

The result of running `python3 plotsortscalability.py` is shown in figure 6.

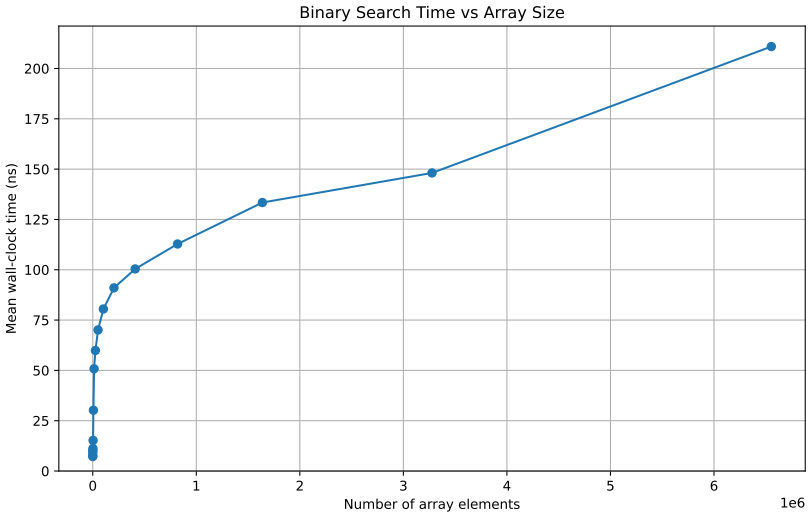

Figure 5: Binary search execution time as a function of array size. Graph produced by Python Matplotlib. Unlike figure 3, it displays the good experimental results from section 9.2.

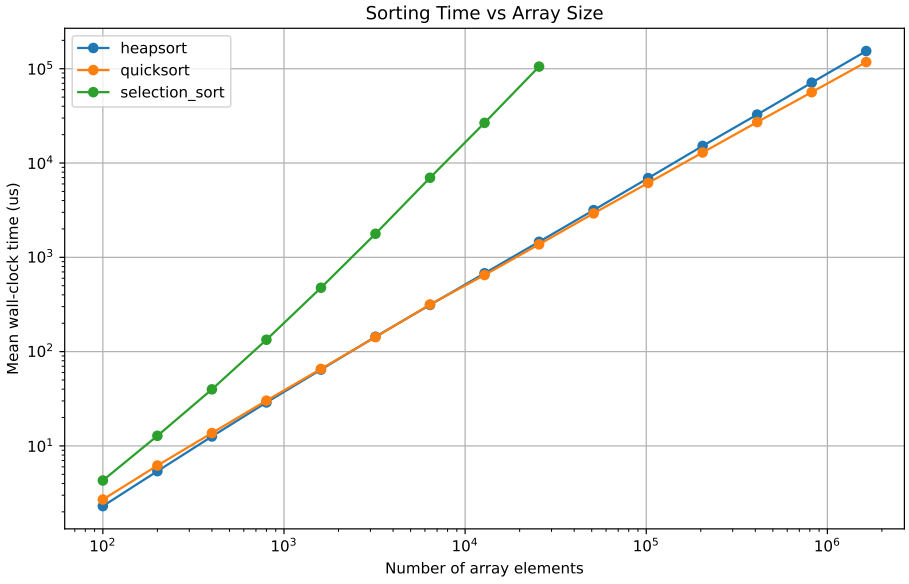

Figure 6: Three sorting algorithms, execution time as a function of array size; graph produced by Python Matplotlib and showing the same data as figure 4. Note the logarithmic scale on both axes.

9 Benchmark design considerations

Until now we have mostly been concerned with *how* to measure execution time. We also need to think about *what* to measure: on which inputs should we run the code when measuring execution time? The answer of course is highly problem dependent, but for inspiration, let us consider again the binary search benchmark.

9.1 Benchmark design for binary search

The binary search routine in section 4.3 has two input parameters whose values we need to choose: the item `x` to search for, and the sorted array `arr` of items to search in.

- Let us consider first the array `arr`. It is clear that the execution time depends on the *size* of the array, so we want to call the method on a range of such sizes. Indeed we did that, by virtue of the outer for loop, in section 4.3.

  The array should be sorted in non-decreasing order (or else the method does not work at all), but still there are many possible choices of array contents: all items are equal, some items are equal, or no items are equal (that is, all items are distinct). For binary search, the latter choice forces the worst case execution time, so that is what we choose. Moreover, when all items are distinct, they may be dense as in $0, 1, 2, 3, \ldots$ or have gaps as in $1, 17, 18, 23, \ldots$. For the execution time of binary search in a sorted array, this makes no difference, but for other representations of sets or bags it may. We choose a dense set of items.

- Next, what items `x` should we search for? The most basic decision is whether we want successful search (for an item that is in the array) or an unsuccessful search (for an item not in the array). The latter forces the highest execution time, and therefore might be a reasonable choice, but precisely for binary search the difference is modest. In general one should choose a representative mix of successful and unsuccessful searches.

  In the example from section 4.3, we ran only successful searches, and always for the same item `successItem`, at a position just before the middle of the array. In retrospect, this was a dubious choice, because the same search is performed repeatedly and each one will hit the same array positions, which are therefore almost guaranteed to be cached in all CPU caches. This unrealistic scenario may severely underestimate the typical execution time for a successful binary search.

  There are several ways to remedy this situation. For an n-element array, we could either search for the items $0, 1, ..., n-1$ in order, or we could repeatedly generate a pseudorandom number in the range $[0, n-1]$ and search for that. However, the first approach is "too predictable" and likely to use the caches well, and hence will give unrealistically low execution times. The second approach suffers from another problem: the cost of generating a pseudorandom item to search for can be high, comparable to that of the search itself. We will investigate this and a possible solution in the next section.

9.2 Binary search for pseudorandom items

We can generate a pseudorandom item in the range $[0, n-1]$ by `rnd.nextInt(n)`. Using the machinery already developed we can measure the cost of this operation:

```
final java.util.Random rnd = new java.util.Random();
final int n = 1638400;
Mark8('n', "random_index", "", i -> rnd.nextInt(n));
```

Running this code tells us that the cost of generating a pseudorandom item is around 4.0 ns, a considerable fraction of the cost of the binary search itself as measured in section 4.3; thus it may distort the measurements.

What can be done? One possibility is to pre-generate a large enough array `items` of pseudorandom items *before* performing the benchmark, and sequentially retrieve and search for these items in the benchmark itself. Then the time measurements will include the time to access the `items` array, but this access will be sequential and cache-friendly, and hence hopefully not contribute too much to the execution time:

```
for (int size = 100; size <= 10_000_000; size *= 2) {
  final int[] intArray = SearchAndSort.fillIntArray(size);  // [0,1,...]
  final int[] items = SearchAndSort.fillIntArray(size);
  final int n = size;
  SearchAndSort.shuffle(items);
  Mark8("binary_search_success",
        String.format("%8d", size),
        i -> SearchAndSort.binarySearch(items[i % n], intArray));
}
```

Now the results are as follows:

```
binary_search_success          100            5.7 ns       0.02   67108864
binary_search_success          200            6.0 ns       0.22   67108864
binary_search_success          400            7.1 ns       0.21   67108864
binary_search_success          800            8.5 ns       0.02   33554432
binary_search_success         1600            9.8 ns       0.02   33554432
binary_search_success         3200           10.9 ns       0.02   33554432
binary_search_success         6400           29.6 ns       5.42    8388608
binary_search_success        12800           46.9 ns       4.06    8388608
binary_search_success        25600           52.1 ns       2.07    8388608
binary_search_success        51200           64.2 ns       1.15    4194304
binary_search_success       102400           70.6 ns       0.07    4194304
binary_search_success       204800           83.5 ns       3.64    4194304
binary_search_success       409600           88.0 ns       0.09    4194304
binary_search_success       819200          100.8 ns       2.36    4194304
binary_search_success      1638400          120.9 ns       3.47    4194304
binary_search_success      3276800          144.4 ns       0.58    2097152
binary_search_success      6553600          169.2 ns       4.89    2097152
```

These execution times are up to 9 times higher than those measured in section 4.3. Whether they are more correct is a question of which usage scenario is more likely in practice: repeatedly searching for the same item, or repeatedly searching for an item that is completely unrelated to the one previously searched for. Certainly the latter gives a more conservative execution time estimate, and one that fits the textbook complexity $O(\log n)$ for binary search better.

Note that we still perform only successful searches, which may be unrealistic. To improve on this situation we could make the `intArray` contain only odd numbers $1, 3, 5, \ldots$ and have both even and odd numbers in the array of `items` to search for. It is left as an exercise to see what results this would give.

Another possible concern is that the new higher execution times include the time to compute the remainder operator (`%`) and retrieve the pre-generated pseudorandom items from the array. If we suspect that this overhead is significant, we can measure it:

```
for (int size = 100; size <= 10_000_000; size *= 2) {
  final int[] items = SearchAndSort.fillIntArray(size);
  final int n = size;
  SearchAndSort.shuffle(items);
  Mark8('n', "get_pseudorandom_items",
        String.format("%8d", size),
        i -> items[(int)(i % n)]);
}
```

The results below show that the time to access the pseudorandom items is negligible and completely constant over very different `items` array sizes, confirming that the sequential retrieval of items to search for is cache-friendly:

```
get_pseudorandom_items          100            0.7 ns        0.01  536870912
get_pseudorandom_items          200            0.7 ns        0.02  536870912
get_pseudorandom_items          400            0.7 ns        0.00  536870912
get_pseudorandom_items          800            0.7 ns        0.00  536870912
get_pseudorandom_items         1600            0.7 ns        0.00  536870912
get_pseudorandom_items         3200            0.7 ns        0.00  536870912
get_pseudorandom_items         6400            0.7 ns        0.00  536870912
get_pseudorandom_items        12800            0.7 ns        0.00  536870912
get_pseudorandom_items        25600            0.7 ns        0.00  536870912
get_pseudorandom_items        51200            0.7 ns        0.00  536870912
get_pseudorandom_items       102400            0.7 ns        0.00  536870912
get_pseudorandom_items       204800            0.7 ns        0.00  536870912
get_pseudorandom_items       409600            0.7 ns        0.00  536870912
get_pseudorandom_items       819200            0.7 ns        0.00  536870912
get_pseudorandom_items      1638400            0.7 ns        0.00  536870912
get_pseudorandom_items      3276800            0.7 ns        0.00  536870912
get_pseudorandom_items      6553600            0.7 ns        0.01  536870912
```

10 Measuring memory read access times

A modern laptop or server CPU has a memory architecture with multiple levels of caches [6, section 3] as illustrated by Figure 7, which also gives some indicative numbers for size and speed of the memory components.

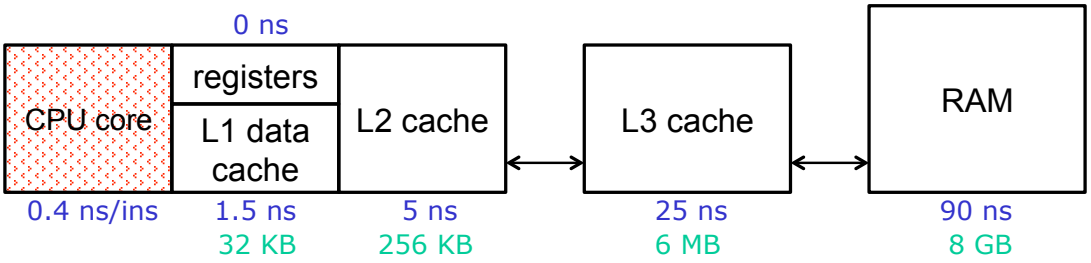

Figure 7: The memory components (caches) of an Intel i7-4870HQ CPU. The L1 and L2 caches are local to a core; the L3 cache and RAM are shared between cores.

In this section we present a simple approach to measuring memory read access times (or access latency), assuming that arrays are stored as contiguous sequences of bytes in memory. For garbage collection reasons this is not necessarily true in Java, but our experimental results fit well with available documentation, and the approach can also be used in C which has a simple and predictable array layout.

10.1 Core measurement loop

The core piece of code to measure memory read times is this method, containing a simple loop:

```
private static double jump(int[] arr) {
  int k = 0;
  for (int j=0; j<134_217_728; j++)
    k = arr[k];
  return k;
}
```

In the setups we consider, the loop always performs 134 million (2^{27}) iterations and array accesses. However, its execution time varies by a factor of up to 100 depending on the contents of the array `arr`. Namely, if the array contains a short cycle of indexes such as 6, 0, 3, 5, 7, 1, 4, 2 then only the first 8 elements of the array will be accessed, repeatedly. Since the 8 elements require only 32 bytes of memory, the array elements will always be in L1 cache, so the loop will effectively measure the L1 cache read access time (plus some overhead for manipulating `j` and possibly array bounds checking). Figure 8 illustrates this situation.

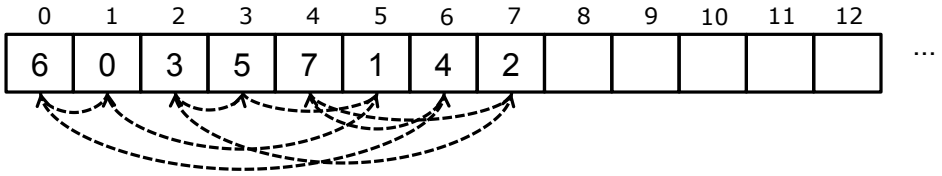

Figure 8: Integer array containing a cyclic index chain of length 8, covering 32 bytes of memory.

If on the other hand the array contains a cyclic chain of indexes that range over 1 million array elements (4 MB of memory) then the loop will effectively measure the L3 cache read access time, provided the indexes do not appear consecutively 1, 2, 3, ... on the same cache lines; see section 10.2.

The code used to measure the `jump` method's execution time looks like this:

```
final int K = 1 << 10, KB = K / 4, MB = K * KB, maxSize = 512 * MB;
final int[] arr = new int[maxSize];
final double factor = Math.sqrt(2.0);
for (int size = 8*KB; size <= maxSize+1;
         size = (int)(Math.round(size * factor))) {
  fillCyclic(arr, size);
  Mark7('u', String.format("memory cycle %11d B", size*4),
        (long i) -> jump(arr));
}
```

The code defines `K` as $2^{10} = 1024$ bytes, `KB` as $K/4 = 256$ array elements, `MB` as 1024 times `KB`, and `maxSize` as enough array elements to cover 512 MB of space, and allocates an integer array `arr` with that many elements. Then, for memory cycle sizes covering from 8 KB up to 512 MB, it calls `fillCyclic` (see section 10.2) to fill the preallocated array with a cycle of that length, and then measures the time to execute the `jump` method, which performs $2^{27} = 134$ million array lookups.

Instead of increasing `size` by a factor of 2 in each iteration of the outer loop, it increases it by a factor of $\sqrt{2}$ in each step for better resolution.

Figure 9 shows the measured time in nanoseconds per iteration of the above `jump` method's loop, as a function of the amount of memory covered, for six different CPUs.

Memory cover (KiB)	M3pro	Core i9	Xeon E5	AMD 7742	AMD 7402P	AMD 7313
8	1.0	1.2	1.5	1.8	1.8	1.6
11	1.0	1.2	1.5	1.8	1.8	1.6
16	1.0	1.2	1.5	1.8	1.8	1.6
23	1.0	1.1	1.5	1.8	1.8	1.6
32	1.0	1.1	1.5	1.8	1.8	1.6
45	1.0	1.6	2.2	2.5	2.6	2.3
64	1.0	2.0	2.6	3.0	3.1	2.8
91	1.0	2.2	2.9	3.4	3.4	3.1
128	1.0	2.3	3.2	3.6	3.6	3.3
181	2.7	2.6	3.4	3.8	3.8	3.4
256	3.8	3.4	5.9	3.9	3.9	3.5
362	4.5	5.3	8.3	5.1	4.8	4.6
512	5.0	7.0	11.3	6.6	6.3	6.0
724	5.4	8.2	13.2	8.2	7.8	8.3
1024	5.7	9.0	14.1	9.4	9.0	10.2
1448	5.8	9.6	14.8	10.2	9.9	11.4
2048	5.9	10.6	15.2	10.7	10.6	12.0
2897	6.2	11.3	15.5	11.1	11.1	12.4
4097	7.3	11.1	15.8	11.4	11.4	12.7
5793	7.9	11.4	15.9	11.6	11.6	12.9
8193	8.1	20.4	16.0	11.7	11.8	13.0
11587	8.4	30.8	16.1	11.8	11.9	13.2
16386	9.2	44.5	16.2	14.7	14.6	13.5
23174	19.7	63.9	16.7	44.6	43.9	13.5
32773	40.7	78.5	32.5	66.7	65.5	16.0
46348	58.7	88.8	47.3	82.1	79.8	43.1
65545	74.5	95.6	57.4	89.8	88.2	61.9
92695	85.2	101.8	65.1	96.7	94.6	75.3
131091	91.9	106.5	70.6	102.0	99.7	84.9
185391	97.2	109.7	74.4	106.1	103.9	91.9
262182	101.0	110.8	77.1	109.1	107.0	97.1
370781	104.2	112.7	79.0	111.2	109.1	100.6

Figure 9: Memory access times in ns per access `k = arr[k]` in method `jump`, for several CPUs. Measured using C; results in Java and C# are similar but may include array bounds checking. It is clear that the Apple M3pro has at least 128 KB L1 cache whereas the other CPUs have 32 KB L1 cache. All machines have RAM access times around 100 ns, except the old Intel Xeon E5 2660v3 at 80 ns.

The table shows that despite the `jump` loop always performing exactly the same operation `k = arr[k]` exactly the same number of times, the execution time per iteration may vary from 1.0 to 104.2 ns depending on the contents of `arr`, that is, by a factor of 104. This depends only on the contents stored in the array `arr`, which gives rise to different memory access times.

The Apple M3pro has 128 KB L1 data cache and therefore has constant 1.0 ns access times for memory cover 8–128 KB, beyond which the access time increases. It has 16 MB L2 cache, giving gently increasing access times for memory cover 181–16386 KB, beyond which the access time increases considerably. The size of the L3 cache is unknown, but it cannot be much larger than 32 MB. All three AMD EPYC machines have fairly similar access times.

10.2 How to populate the array of indexes

The array `arr` of integers used in method `jump` is filled by method `fillCyclic(arr, S)` which ensures that `arr[0..S-1]` contains a random cyclic chain of indexes of length `S`, starting and ending at `arr[0]`:

```
private static void fillCyclic(int[] arr, int S) {
  ArrayList<Integer> indexes = new ArrayList<>();
  for (int k=0; k<S; k++)
    indexes.add(k);
  Collections.shuffle(indexes);
  // Make indexes[0] == 0.
  int shift = S - indexes.get(0);
  for (int k=0; k<S; k++)
    indexes.set(k, (indexes.get(k) + shift) % S);
  // Make arr[0] = indexes[1], arr[arr[0]] = indexes[2],
  // ..., arr^k[0] = indexes[k], ..., arr^S[0] = 0;
  // the cyclic chain starting at arr[0] will have length S.
  int i = 0;
  for (int k=0; k<S; k++)
    i = arr[i] = indexes.get((k+1) % S);
}
```

The `fillCyclic` code is a little intricate, so here is an explanation of the design. The idea is to force the `jump` loop to read ever larger ranges of memory, thus observing the difference between those ranges that fit in L1, L2 and L3 caches, or only in RAM.

Some possibilities and their drawbacks or advantages are:

1. Sequentially reading the elements of an array would not work because reading one element on a cache line would fetch the entire cache line (typically 64 bytes) and hence also the other 15 integer elements on that cache line.

2. Instead one could use a constant stride > 1 as in

   ```
   final int S = arr.length;
   for (int k=0; k<S; k++)
     arr[k] = (k + stride) % S;
   ```

   This would cover the entire array provided `arr.length` and `stride` are mutually prime, but it is too friendly towards the hardware cache prefetch mechanism, which will recognize the constant stride and prefetch the next item if the `stride` is small enough.

3. Instead one could initialize the array with random indexes like this:

```
for (int k=0; k<S; k++)
  arr[k] = rnd.nextInt(S);
```

and then follow the chain of indexes `for (...) k = arr[k]`, but one would likely get short chains thanks to the birthday paradox and so visit only a small part of the array, maybe held in cache.

4. Instead one could initialize the array with elements `0,1,...,S-1` and randomly shuffle these by `Collection.shuffle()`, but one could still get very short chains; for instance, there is a non-zero probability that `arr[0]` equals `0`. One could mitigate that risk by reshuffling the arrays until the reference chain is at least 90 percent of the array length, but this is very inefficient.

5. An even better approach is to obtain a random permutation of an array `indexes` containing `0,1,...,S-1`, normalize it so that `indexes[0]` equals `0` (by subtracting `indexes[0]` from each element, modulo `S`), and fill the array `arr` with next-indexes from a sequential scan of `indexes`. This will give a cyclic index chain of length `S`.

Method `fillCyclic` shown above implements the last approach.

11 Platform identification

Sometimes the purpose of a performance test is to compare the speed of several different platforms (virtual machines, processors, and operating systems) or several different versions of such platforms. Then it is useful to let each platform identify itself as part of the test output.

Printing this information before any benchmarking is performed has the beneficial side effect of forcing the virtual machine to load (and possibly compile) library classes involved in console input-output, string manipulation, and string formatting, so that this does not happen in the middle of the benchmark runs.

11.1 Platform identification in Java

In Java, the following method in `Benchmark.java` can be used to print relevant system information:

```
public static void SystemInfo() {
  System.out.printf("# OS:   %s; %s; %s%n",
                    System.getProperty("os.name"),
                    System.getProperty("os.version"),
                    System.getProperty("os.arch"));
  System.out.printf("# JVM:  %s; %s%n",
                    System.getProperty("java.vendor"),
                    System.getProperty("java.version"));
  // The processor identifier works only on MS Windows:
  System.out.printf("# CPU:  %s; %d \"cores\"%n",
                    System.getenv("PROCESSOR_IDENTIFIER"),
                    Runtime.getRuntime().availableProcessors());
  java.util.Date now = new java.util.Date();
  System.out.printf("# Date: %s%n",
    new java.text.SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ").format(now));
}
```

The hash mark (`#`) at the beginning of each line marks it as a non-data line in input to eg. `gnuplot`. The output may look like this on a Windows 11 machine with Arm64 processor and Java 25:

```
# OS:   Windows 11; 10.0; aarch64
# JVM:  Microsoft; 25.0.2
# CPU:  ARMv8 (64-bit) [...], Qualcomm Technologies Inc; 8 "cores"
# Date: 2026-08-06T10:47:12+0200
```

and like this on MacOS with an Apple M3 pro (Arm64) CPU and Java 25:

```
# OS:   Mac OS X; 26.6; aarch64
# JVM:  Oracle Corporation; 25
# CPU:  null; 12 "cores"
# Date: 2026-08-04T13:32:18+0200
```

and like this on Linux with an AMD EPYC CPU and Java 24:

```
# OS:   Linux; 4.18.0-513.5.1.el8_9.x86_64; amd64
# JVM:  Eclipse Adoptium; 24.0.1
# CPU:  null; 64 "cores"
# Date: 2026-08-04T10:58:52+0200
```

The Date stamp in the very last line says 4 August 2026 at 10:58:52 UTC plus 2 hours, that is, east of Greenwich. There seems to be no portable way, across operating systems, to obtain more detailed CPU data, such as brand, family, version, speed (in GHz), cache sizes, and so on. The processor count `"cores"` is as reported by the Java library, and sometimes includes hyperthreading (Intel) or simultaneous multithreading (AMD), sometimes not.

11.2 Platform identification in C#

The C# code in `Benchmark.cs` also provides a method `SystemInfo` to print basic system information similar to that for Java above. When run under MacOS, which is a Unix variant, with an Apple M3pro Arm64 processor and .NET 10 it might print this:

```
# OS:   Unix 26.6.0
# .NET: 10.0.6
# CPU:  Arm64; 12 "cores"
# Date: 2026-08-05T10:46:41
```

11.3 Platform identification in C

The C code in `benchmark.c` also provides a method `SystemInfo`. When run under MacOS, whose core is called Darwin, with an Apple M3pro Arm64 processor it might print this:

```
# OS:   Darwin; 25.6.0; arm64
#
# CPU:  (identifier not available); 12 "cores"
# Date: 2026-08-05T13:10:19+0200
```

The empty second line ensures that system identification takes four lines on all platforms, so that the same plotting scripts can be used with all the benchmarking code.

12 Inspecting machine code

Sometimes a benchmarking result is so surprising — unexpectedly fast or unexpectedly slow — that one wants to study the actual machine code that runs on the hardware. Luckily this is possible in a fairly portable manner nowadays. It is useful to have a table of the relevant instruction set, such as [14] or [3] for Arm64 machines, or [1] for Intel 64 (`x86_64`) machines. Or else one may have luck with an AI chatbot explaining fragments of machine code.

12.1 Inspecting machine code generated from Java

To inspect the machine code generated by the just-in-time compiler in a Java Virtual Machine, one can run the `java` run-time system with these options:

```
java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly Benchmark
```

This produces a huge amount of text output, maybe several hundred thousand lines, so pipe it into a file and use a robust editor to browse it. To find the machine code generated for method `multiply` in class Benchmark, say, search for `Benchmark::multiply`. There may be multiple versions of the code for a method, both because the just-in-time compiler may have generated more and more optimized versions, and because the method may have been inlined into methods that call it, such as `Mark6` and `Mark7`. Usually, you want the last such version generated.

For the generated machine code to be readable, you need to install a "Hotspot disassembler" helper file to translate numeric machine instruction codes to their symbolic names; see [11, 12]. For Java on an Arm64-based Mac, the file is called `hsdis-aarch64.dylib`.

12.2 Inspecting machine code generated from C#

To inspect the machine code generated for a specific method by a .NET just-in-time compiler, set an environment variable to the name of the method. For instance, use the relevant one of the first three lines, then run `dotnet`:

```
export DOTNET_JitDisasm="Multiply"          # Linux or MacOS bash
$env:DOTNET_JitDisasm="Multiply"            # Windows Powershell
set DOTNET_JitDisasm=Multiply               # Windows
dotnet run -c Release
```

This will run the program and additionally print the symbolic machine code for method `Multiply` in a fairly compact format. Several versions of the machine code for the method may be printed, as the .NET just-in-time compiler optimizes the code further.

It is hard to find official documentation of these `dotnet` features, apart from [5].

12.3 Inspecting machine code generated from C

To inspect the machine code generated by a C compiler, use compiler option `-S`, for instance:

```
clang -S -O3 benchmark.c -o benchmark.s
```

Then text file `benchmark.s` will contain the symbolic machine code generated for all functions in the source file.

13 Professional microbenchmarking frameworks

In the preceding sections we have developed our own benchmarking methods. This served both to understand the challenges encountered in modern performance measurements and to produce some actual tools that we can use and adapt to our needs. In industrial practice, it is better to use existing professional tools that have been developed and tested over many years, and that are designed for inclusion in build and deployment pipelines.

13.1 Java Microbenchmark Harness (JMH)

The Java Microbenchmark Harness (JMH) [8] is a widely used benchmarking framework for Java. The recommended way to use JMH is through a Maven project, so install Maven [2] first. Then follow the instructions on the JMH homepage to install it; this will create a new directory `test`. The Java code that you want to benchmark should be in directory `test/src/main/java/org/sample/`, for instance in a file `MyBenchmark.java` with this contents:

```
package org.sample;
import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.Blackhole;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Fork(1)
public class MyBenchmark {
  private long value = 0L;
  @Benchmark
  public void testMethod(Blackhole bh) {
    bh.consume(multiply(value++));
  }
  private static double multiply(long i) {
    double x = 1.1 * (double)(i & 0xFF);
    return x * x * x * x * x * x * x * x * x * x * x * x
      * x * x * x * x * x * x * x * x * x * x * x * x;
  }
}
```

The package `org.sample` of the class must match the file path `org/sample/`. The annotations, such as `@Benchmark`, are read by JMH to generate code that performs the benchmarking. The purpose of the Blackhole object `bh` is to consume the result of the `multiply` method, in the hope that the just-in-time compiler will not optimize away the call to the method — that is, it plays the same role as the `dummy` variable in our benchmarking methods in sections 3.3 and 4.

You can now build the project and run the benchmark by giving these commands in directory `test`:

```
$ mvn clean verify
$ java -jar target/test-1.0.jar
```

This will run the benchmark warmup, iterations, and so on, and produce a rather verbose output (here with some deletions `[...]`):

```
# JMH version: 1.37
# VM version: JDK 25, Java HotSpot(TM) 64-Bit Server VM, 25+37-LTS-3491
```

```
# VM invoker: [...]/jdk-25.jdk/Contents/Home/bin/java
# VM options: <none>
# Blackhole mode: compiler [...]
# Warmup: 5 iterations, 10 s each
# Measurement: 5 iterations, 10 s each
[...]
# Fork: 1 of 1
# Warmup Iteration   1: 1.754 ns/op
# Warmup Iteration   2: 1.747 ns/op
# Warmup Iteration   3: 1.747 ns/op
# Warmup Iteration   4: 1.757 ns/op
# Warmup Iteration   5: 1.812 ns/op
Iteration   1: 1.819 ns/op
Iteration   2: 1.806 ns/op
Iteration   3: 1.775 ns/op
Iteration   4: 1.849 ns/op
Iteration   5: 1.755 ns/op

Result "org.sample.MyBenchmark.testMethod":
  1.801 +- (99.9%) 0.141 ns/op [Average]
  (min, avg, max) = (1.755, 1.801, 1.849), stdev = 0.037
  CI (99.9%): [1.659, 1.942] (assumes normal distribution)

[...]
Benchmark               Mode  Cnt  Score    Error  Units
MyBenchmark.testMethod  avgt    5  1.801 +- 0.141  ns/op
```

The measurements seem to confirm our `multiply` results from sections 4.1 and 4.2.

At the time of writing (August 2026), the default JMH installation needs some adaptation if you use Java version 21 or later. Otherwise JMH's code generation based on annotations will silently fail and the directory `test/target/generated-sources/annotations/` will be empty. Specifically, the `test/pom.xml` file needs to be updated for the Java version (here `25`), for a modern Maven plugin (`3.13.0`), and to force annotation processing to execute (`-proc:full`). These are the important fragments:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.release>25</maven.compiler.release>
  <jmh.version>1.37</jmh.version>
</properties>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.13.0</version>
      <configuration>
        <compilerArgs>
          <arg>-proc:full</arg>
        </compilerArgs>
      </configuration>
    </plugin>
    [...]
  </plugins>
</build>
```

13.2 BenchmarkDotNet (BDN)

BenchmarkDotNet (BDN) [4] is a widely used benchmarking framework for .NET, including C#. First create a source file `MyBenchmark.cs` similar to the Java one in section 13.1:

```
using System;
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;
using BenchmarkDotNet.Engines;
namespace MyBenchmark {
  public class MultiplyBenchmark {
    private long value = 0L;
    private readonly Consumer _consumer = new();
    [Benchmark]
    public void testMethod() {
      _consumer.Consume(Multiply(value++));
    }
    public double Multiply(long i) {
      double x = 1.1 * (double)(i & 0xFF);
      return x * x * x * x * x * x * x * x * x * x * x
        * x * x * x * x * x * x * x * x * x * x * x;
    }
  }
  public class Program {
    public static void Main(string[] args) {
      var summary = BenchmarkRunner.Run<MultiplyBenchmark>();
    }
  }
}
```

The Consumer plays the same role as the Blackhole in JMH (section 13.1) or the variable `dummy` in our benchmarking methods: it hopefully prevents the just-in-time compiler from eliminating the call to the measured method.

You also need a rudimentary `mybenchmark.csproj` file, maybe:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
  </PropertyGroup>
</Project>
```

Now get `dotnet` to download and install the BenchmarkDotNet framework and add a package reference to the `.csproj` file:

```
$ dotnet add package BenchmarkDotNet
```

Then simply build and run in Release mode:

```
$ dotnet run -c Release
```

This will produce multiple files and much output on the terminal, ending with this summary (here with some deletions `[...]`):

```
BenchmarkDotNet v0.15.8, macOS Tahoe 26.6.1 (25G76) [Darwin 25.6.0]
Apple M3 Pro, 1 CPU, 12 logical and 12 physical cores
.NET SDK 10.0.202
  [Host]     : .NET 10.0.6 ([...]), Arm64 RyuJIT armv8.0-a
  DefaultJob : .NET 10.0.6 ([...]), Arm64 RyuJIT armv8.0-a

| Method     | Mean     | Error     | StdDev    | Median   |
|----------- |---------:|----------:|----------:|---------:|
| testMethod | 1.158 ns | 0.0422 ns | 0.0605 ns | 1.199 ns |

[...]
```

The measured execution time for `Multiply` here comes out as 1.2 ns, which is somewhat less than previously measured, but it is not clear which of these results is more correct. Note that BDN does not provide all the platform identification recommended in section 11, but more than JMH does.

14 Source code for examples

The complete source code presented here, including the examples, can be found (in Java, C# and C) from this webpage:

```
https://raspi.itu.dk/people/sestoft/benchmarking/
```

The hope is that the reader can adapt and further develop the benchmarking code for particular use contexts.

15 Acknowledgements

Thanks to the many students whose unexpected benchmark results originally inspired this note; to Finn Schiermer Andersen and David Christiansen for helping explain unreasonable results in C and Scala; and to Claus Brabrand, Kasper Østerbye and Michael Reichhardt Hansen for constructive comments.

16 Exercises

Exercise 1: Run the `Mark1` through `Mark6` measurements yourself, on one or more computers. Use the `SystemInfo` method to record basic system identification, and supplement with whatever other information you can find about the execution platform; see section 11.

Exercise 2: Use `Mark7` to measure the execution time for the mathematical functions `pow`, `exp`, and so on, as in section 4.2. Record the results along with appropriate system identification. Preferably do this on at least two different computers, eg. your own computer and one at the university.

Exercise 3: Run the measurements of sorting algorithm performance as in section 4.5. Record the results along with appropriate system identification. Use Excel or gnuplot or Matplotlib or some other plotting package to make graphs of the execution time as function of the problem size, as in figure 6. Preferably do this on at least two different computers, eg. your own computer and one at the university.

References

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual, volumes 2A and 2B*, September 2016.

[2] Apache. Maven. Homepage. At `https://maven.apache.org/`.

[3] Arm Ltd. A64 instruction set architecture guide. At `https://developer.arm.com/documentation/102374/latest/`.

[4] BenchmarkDotNet. Homepage. At `https://benchmarkdotnet.org/`.

[5] Dotnet Github. Viewing JIT disassembly and dumps. Webpage. At `https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/jit/viewing-jit-dumps.md`.

[6] U. Drepper. What every programmer should know about memory. Technical report, Redhat, 2007. At `http://www.akkadia.org/drepper/cpumemory.pdf`.

[7] Gnuplot. Homepage. At `http://www.gnuplot.info/`.

[8] Java Microbenchmark Harness. Homepage. At `https://github.com/openjdk/jmh/`.

[9] Matplotlib. Homepage. At `https://matplotlib.org/`.

[10] musl libc. Homepage. At `https://musl.libc.org/`.

[11] C. Newland. Openjdk hsdis (hotspot disassembly plugin) downloads. Web page. At `https://chriswhocodes.com/hsdis/`.

[12] OpenJDK. hsdis - a hotspot plugin for disassembling dynamically generated code. Web page. At `https://github.com/openjdk/jdk/tree/master/src/utils/hsdis`.

[13] J. Rose. MicroBenchmarks. Homepage. At `https://wiki.openjdk.org/spaces/HotSpot/pages/11829255/MicroBenchmarks`.

[14] UW course CSE469. ARMv8 A64 Quick Reference. At `https://courses.cs.washington.edu/courses/cse469/20wi/arm64.pdf`, University of Washington course CSE469 Computer Architecture I.