3
Analyzing and Measuring Performance

Since this is a book about writing C++ code that runs efficiently, we need to cover some basics regarding how to measure software performance and estimate algorithmic efficiency. Most of the topics in this chapter are not specific to C++ and can be used whenever you are facing a problem where performance is an issue.

You will learn how to estimate algorithmic efficiency using big O notation. This is essential knowledge when choosing algorithms and data structures from the C++ standard library. If you are new to big O notation, this part might take some time to digest. But don't give up! This is a very important topic to grasp in order to understand the rest of the book, and, more importantly, to become a performance-aware programmer. If you want a more formal or more practical introduction to these concepts, there are plenty of books and online resources dedicated to this topic. On the other hand, if you have already mastered big O notation and know what amortized time complexity is, you can skim the next section and go to the later parts of this chapter.

This chapter includes sections on:

- Estimating algorithmic efficiency using big O notation
- A suggested workflow when optimizing code so that you don't spend time fine-tuning code without good reason

- CPU profilers—what they are and why you should use them
- Microbenchmarking

Let's begin by taking a look at how to estimate algorithmic efficiency using big O notation.

Asymptotic complexity and big O notation

There is usually more than one way to solve a problem, and if efficiency is a concern, you should first focus on high-level optimizations by choosing the right algorithms and data structures. A useful way of evaluating and comparing algorithms is by analyzing their asymptotic computational complexity—that is, analyzing how the running time or memory consumption grows when the size of the input increases. In addition, the C++ standard library specifies the asymptotic complexity for all containers and algorithms, which means that a basic understanding of this topic is a must if you are using this library. If you already have a good understanding of algorithm complexity and the big O notation, you can safely skip this section.

Let's start off with an example. Suppose we want to write an algorithm that returns `true` if it finds a specific key in an array, or `false` otherwise. In order to find out how our algorithm behaves when passed different sizes of the array, we would like to analyze the running time of this algorithm as a function of its input size:

```
bool linear_search(const std::vector<int>& vals, int key) noexcept {
  for (const auto& v : vals) {
    if (v == key) {
      return true;
    }
```

```
  }
  return false;
}
```

The algorithm is straightforward. It iterates over the elements in the array and compares each element with the key. If we are lucky, we find the key at the beginning of the array and it returns immediately, but we might loop through the entire array without finding the key at all. This would be the worst case for the algorithm, and in general, that is the case we want to analyze.

But what happens with the running time when we increase the input size? Say we double the size of the array. Well, in the worst case, we need to compare all elements in the array that would double the running time. There seems to be a linear relationship between the input size and the running time. We call this a linear growth rate:

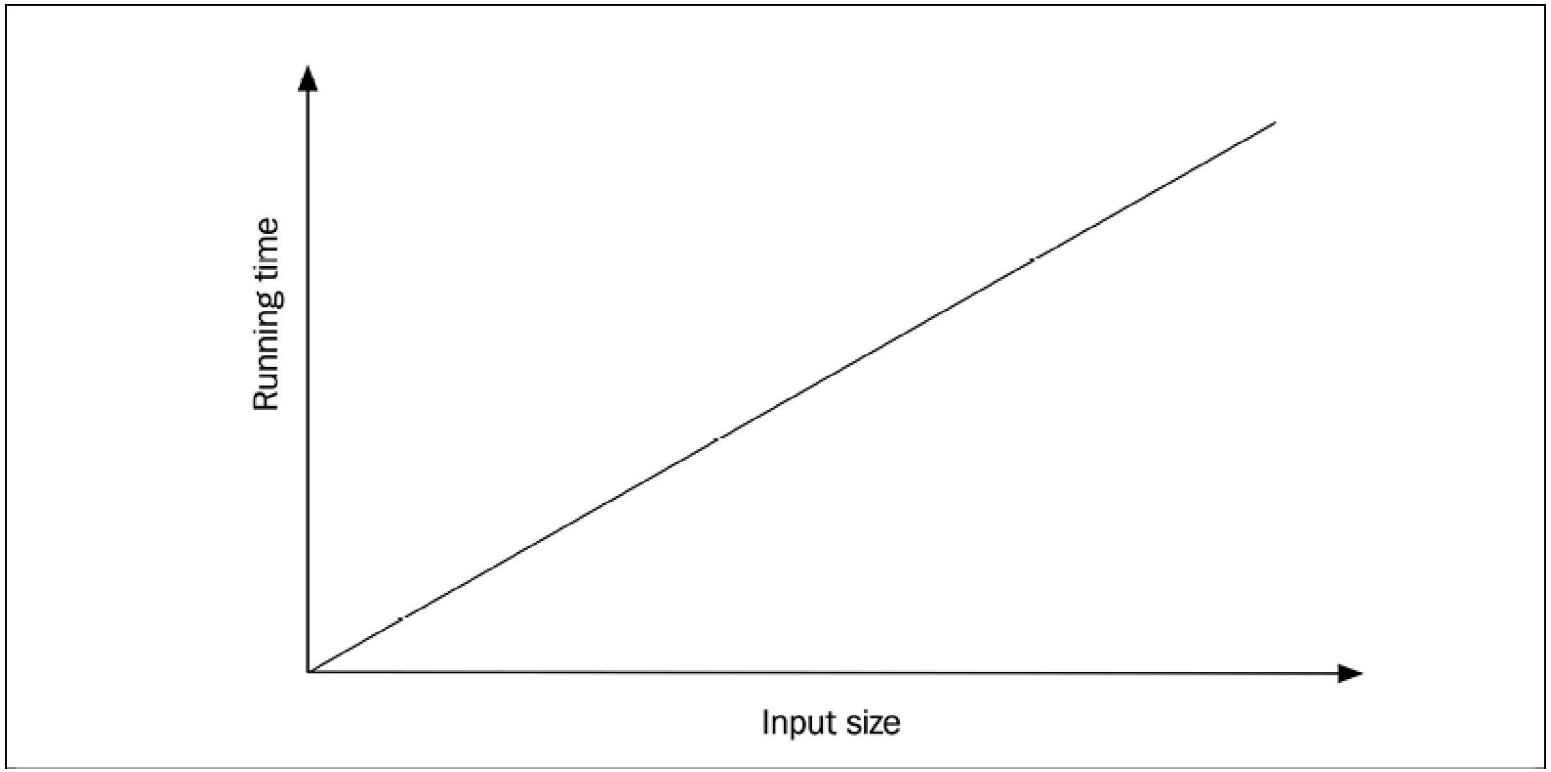

Figure 3.1: Linear growth rate

Now consider the following algorithm:

```
struct Point {
  int x_{};
  int y_{};
};

bool linear_search(const std::vector<Point>& a, const Point& key) {
  for (size_t i = 0; i < a.size(); ++i) {
    if (a[i].x_ == key.x_ && a[i].y_ == key.y_) {
```

```
      return true;
    }
  }
  return false;
}
```

We are comparing points instead of integers and we are using an index with the subscript operator to access each element. How is the running time affected by these changes? The absolute running time is probably higher compared to the first algorithm since we are doing more work—for example, the comparison of points involves two integers instead of just one for each element in the array. However, at this stage, we are interested in the growth rate the algorithm exhibits, and if we plot the running time against the input size, we will still end up with a straight line, as shown in the preceding figure.

As the last example of searching for integers, let's see whether we can find a better algorithm if we assume that the elements in the array are sorted. Our first algorithm would work regardless of the order of the elements, but if we know that they are sorted, we can use a binary search. It works by looking at the element in the middle to determine whether it should continue searching in the first or second half of the array. For simplicity, the indexes `high`, `low`, and `mid` are of type `int` and requires a `static_cast`. A better option would be to use iterators that will be covered in succeeding chapters. Here follows the algorithm:

```
bool binary_search(const std::vector<int>& a, int key) {
  auto low = 0;
  auto high = static_cast<int>(a.size()) - 1;
  while (low <= high) {
    const auto mid = std::midpoint(low, high); // C++20
    if (a[mid] < key) {
```

```
      low = mid + 1;
    } else if (a[mid] > key) {
      high = mid - 1;
    } else {
      return true;
    }
  }
  return false;
}
```

As you can see, this algorithm is harder to get correct than a simple linear scan. It looks for the specified key by *guessing* that it's in the middle of the array. If it's not, it compares the key with the element in the middle to decide which half of the array it should keep looking for the key in. So, in each iteration, it cuts the array in half.

Assume we called `binary_search()` with an array containing 64 elements. In the first iteration we reject 32 elements, in the next iteration we reject 16 elements, in the next iteration we reject 8 elements, and so on, until there are no more elements to compare or until we find the key. For an input size of 64, there will be, at most, 7 loop iterations. What if we *double the input size* to 128? Since we halve the size in each iteration, it means that we only need *one more loop iteration*. Clearly, the growth rate is no longer linear—it's actually logarithmic. If we measure the running time of `binary_search()`, we will see that the growth rate looks similar to the following:

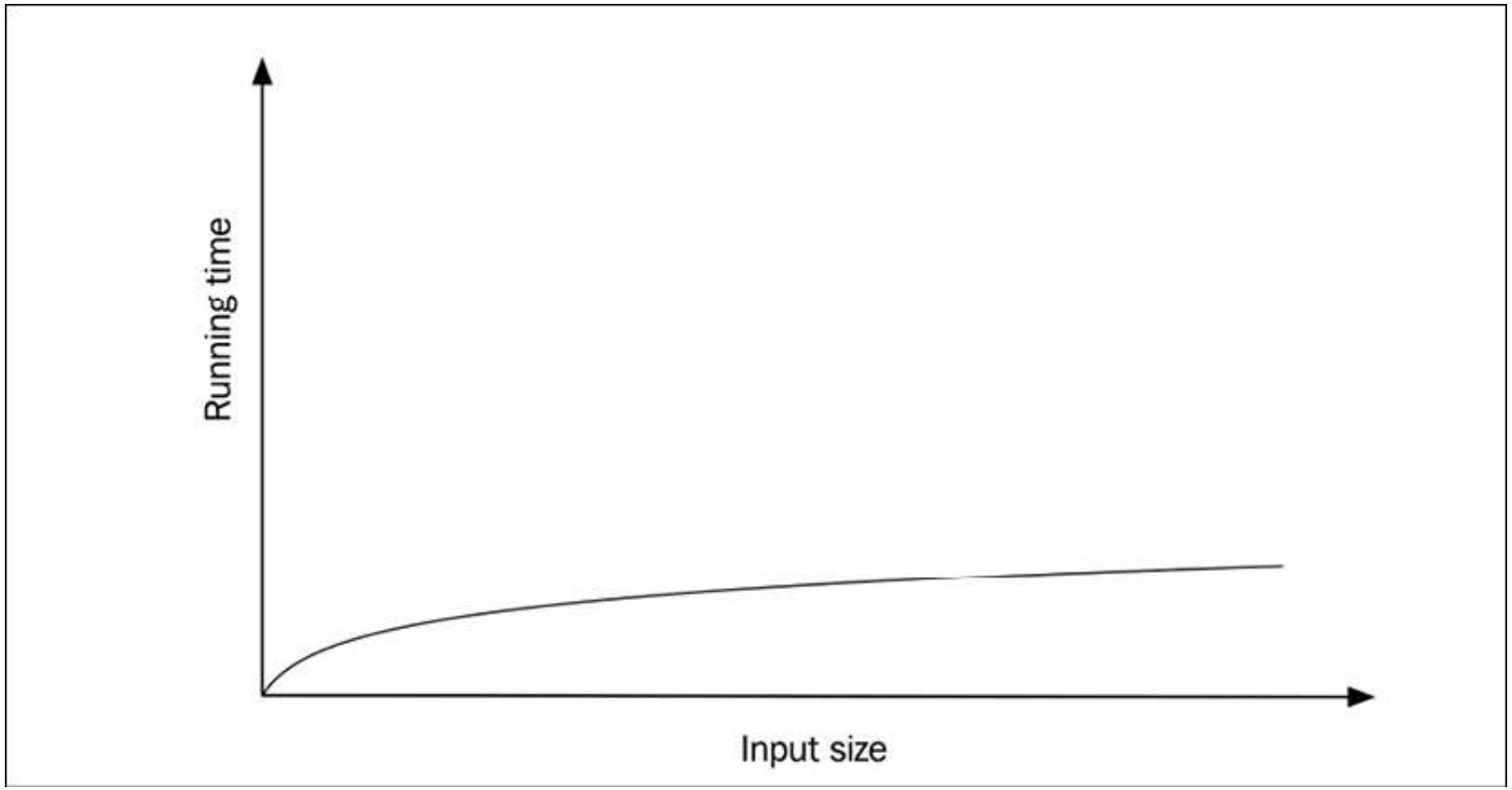

Figure 3.2: Logarithmic growth rate

On my machine, a quick timing of the three algorithms repeatedly called 10,000 times with various input sizes (*n*) produced the results shown in the following table:

Algorithm	n = 10	n = 1,000	n = 100,000
Linear search with `int`	0.04 ms	4.7 ms	458 ms

Linear search with `Point`	0.07 ms	6.7 ms	725 ms
Binary search with `int`	0.03 ms	0.08 ms	0.16 ms

Table 3.1: Comparison of different versions of search algorithms

Comparing algorithms 1 and 2, we can see that comparing points instead of integers takes more time, but they are still in the same order of magnitude even when the input size increases. However, if we compare all three algorithms when the input size increases, what really matters is the growth rate the algorithm exhibits. By exploiting the fact that the array was sorted, we could implement the search function with very few loop iterations. For large arrays, a binary search is practically free compared to linearly scanning the array.

It's usually not a good idea to spend time tuning your code before you are certain that you have chosen the correct algorithms and data structures for your problem.

Wouldn't it be nice if we could express the growth rate of algorithms in a way that would help us decide which algorithm to use? Here is where the big O notation comes in handy.

Here follows an informal definition:

If *f(n)* is a function that specifies the running time of an algorithm with input size *n*, we say that *f(n)* is *O(g(n))* if there is a constant *k* such that $f(n) \leq k * g(n)$.

This means that we could say that the time complexity of `linear_search()` is *O(n)*, for both versions (the one that operates with integers and the one that operates with points), whereas the time complexity of `binary_search()` is *O(log n)* or big O of *log n*.

In practice, when we want to find the big O of a function, we can do that by eliminating all terms except the one with the largest growth rate and then remove any constant factors. For example, if we have an algorithm with a time complexity described by $f(n) = 4n^2 + 30n + 100$, we pick out the term with the highest growth rate, $4n^2$. Next, we remove the constant factor of 4 and end up with n^2, which means that we can say that our algorithm runs in $O(n^2)$. Finding the time complexity of an algorithm can be hard, but the more you start thinking of it while writing code, the easier it will get. For the most part, it's enough to keep track of loops and recursive functions.

Let's try to find the time complexity of the following sorting algorithm:

```
void insertion_sort(std::vector<int>& a) {
  for (size_t i = 1; i < a.size(); ++i) {
    auto j = i;
    while (j > 0 && a[j-1] > a[j]) {
      std::swap(a[j], a[j-1]);
      --j;
    }
  }
}
```

The input size is the size of the array. The running time could be estimated approximately by looking at the loops that iterate over all elements. First, there is an outer loop iterating over *n - 1* elements. The in-

ner loop is different: the first time we reach the `while` -loop, `j` is 1 and the loop only runs one iteration. On the next iteration, `j` starts at 2 and decreases to 0. For each iteration in the outer `for` -loop, the inner loop needs to do more and more work. Finally, `j` starts at *n - 1*, which means that we have, in the worst case, executed `swap()` *1 + 2 + 3 + ... + (n - 1)* times. We can express this in terms of *n* by noting that this is an arithmetic series. The sum of the series is:

$$1 + 2 + \cdots + k = \frac{k(k+1)}{2}$$

So, if we set *k = (n - 1)*, the time complexity of the sorting algorithm is:

$$\frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = (1/2)\,n^2 - (1/2)\,n$$

We can now find the big O of this function by first eliminating all terms except the one with the largest growth rate, which leaves us with $(1/2)n^2$. After that, we remove the constant *1/2* and conclude that the running time of the sorting algorithm is $O(n^2)$.

Growth rates

As stated previously, the first step in finding the big O of a complexity function is to remove all terms except the one with the highest growth rate. To be able to do that, we must know the growth rate of some common functions. In the following figure, I have plotted some of the most common functions:

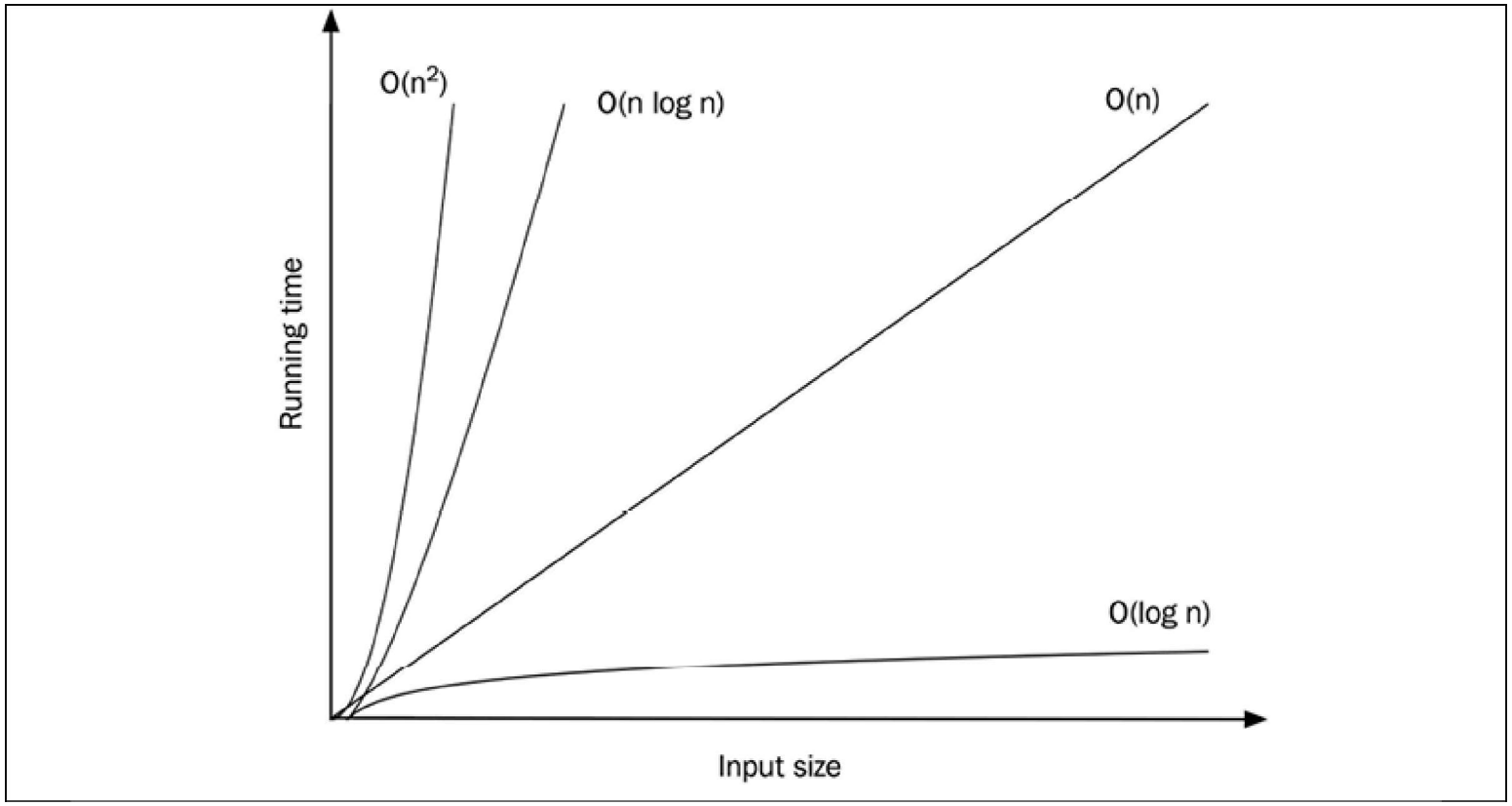

Figure 3.3: Comparison of growth rate functions

The growth rates are independent of machine or coding style and so on. When the growth rates differ between two algorithms, the one with the slowest growth rate will always win when the input size gets sufficiently large. Let's see what happens with the running time for different growth rates if we assume that it takes 1 ms to perform 1 unit of work. The following table lists the growth function, its common name, and different input sizes, *n*:

Big O	Name	n = 10	n = 50	n = 1000

$O(1)$	Constant	0.001 sec	0.001 sec	0.001 sec
$O(\log n)$	Logarithmic	0.003 sec	0.006 sec	0.01 sec
$O(n)$	Linear	0.01 sec	0.05 sec	1 sec
$O(n \log n)$	Linearithmic or *n log n*	0.03 sec	0.3 sec	10 sec
$O(n^2)$	Quadratic	0.1 sec	2.5 sec	16.7 minutes
$O(2^n)$	Exponential	1 sec	35,700 years	$3.4 * 10^{290}$ years

Table 3.2: Absolute running times for different growth rates and various values of input size

Note that the number in the bottom-right cell is a 291-digit number! Compare this with the age of the universe, $13.7 * 10^9$ years, which is only an 11-digit number.

Next, I will introduce amortized time complexity, which is frequently used in the C++ standard library.

Amortized time complexity

Usually, an algorithm behaves differently with different inputs. Going back to our algorithm that linearly searched for an element in an array, we were analyzing a case where the key was not in the array at all. For that algorithm, that was the worst case—that is, it used the *most* resources the algorithm will need.

The best case refers to the *least* amount of resources the algorithm will need, whereas the average case specifies the amount of resources the algorithm will use on average with different inputs.

The standard library usually refers to the *amortized running time* of functions that operate on containers. If an algorithm runs in constant amortized time, it means that it will run in *O(1)* in almost all cases, except very few where it will perform worse. At first sight, amortized running time can be confused with average time, but as you will see, they are not the same.

To understand amortized time complexity, we will spend some time thinking about `std::vector::push_back()`. Let's assume that the vector internally has a fixed-size array to store all its elements. If there is room for more elements in the fixed-size array when calling `push_back()`, the operation will run in constant time, *O(1)*—that is, it's not dependent on how many elements are already in the vector as long as the internal array has room for one more:

```
if (internal_array.size() > size) {
  internal_array[size] = new_element;
  ++size;
}
```

But what happens when the internal array is full? One way to handle the growing vector is to create a new empty internal array with a bigger size and then move all the elements from the old array to the new one. This is obviously not constant time anymore since we need one move per element in the array—that is, *O(n)*. If we considered this the worst case, it would mean that `push_back()` is *O(n)*. However, if we call `push_back()` many times, we know that the expensive `push_back()` can't happen very often, and so it would be pessimistic, and not very useful, to say that `push_back()` is *O(n)* if we know that `push_back()` is called many times in a row.

Amortized running time is used for analyzing a *sequence* of operations rather than a single one. We are still analyzing the worst case, but for a sequence of operations. The amortized running time can be computed by first analyzing the running time of the entire sequence and then dividing that by the length of the sequence. Suppose we are performing a sequence of *m* operations with the total running time *T(m)*:

$$T(m) = t_0 + t_1 + t_2 \ldots + t_{m-1}$$

where $t_0 = 1$, $t_1 = n$, $t_2 = 1$, $t_3 = n$, and so on. In other words, half of the operations run in constant time and the other half run in linear time. The total time *T* for all *m* operations can be expressed as follows:

$$T(m) = n \cdot \frac{m}{2} + 1 \cdot \frac{m}{2} = \frac{(n+1)m}{2}$$

The amortized complexity for each operation is the total time divided by the number of operations, which turns out to be *O(n)*:

$$T(m)/m = \frac{(n+1)m}{2m} = \frac{n+1}{2} = O(n)$$

However, if we can guarantee that the number of expensive operations differs by orders of magnitude compared to the number of constant time operations, we will achieve lower amortized running costs. For example, if we can guarantee that an expensive operation only occurs once in a sequence *T(n) + T(1) + T(1) + ...*, then the amortized running time is *O(1)*. So, depending on the frequency of the expensive operations, the amortized running time changes.

Now, back to `std::vector`. The C++ standard states that `push_back()` needs to run in amortized constant time, *O(1)*. How do the library vendors achieve this? If the capacity is increased by a fixed number of elements each time the vector becomes full, we will have a case similar to the preceding one where we had a running time of *O(n)*. Even if we use a large constant, the capacity changes would still occur at fixed intervals. The key insight is that the vector needs to grow exponentially in order to get the expensive operations to occur rarely enough. Internally, the vector uses a growth factor such that the capacity of the new array is the current size times the growth factor.

A big growth factor would potentially waste more memory but would make the expensive operation occur less frequently. To simplify the math, let's use a common strategy, namely by doubling the capacity each time the vector needs to grow. We can now estimate how often the expensive calls occur. For a vector of size *n*, we would need to grow the internal array $log_2(n)$ times since we are doubling the size all the time. Each time we grow the array, we need to move all the elements that are currently in the array. The i^{th} time we grow the array there will be 2^i elements to move. So if we perform *m* number of `push_back()` operations, the total running time of the grow operations will be:

$$T(m) = \sum_{i=1}^{log_2(m)} 2^i$$

This is a geometric series and can also be expressed as:

$$\frac{2 - 2^{log_2(m)+1}}{1 - 2} = 2m - 2 = O(m)$$

Dividing this by the length of the sequence, *m*, we end up with the amortized running time *O(1)*.

As I have already said, amortized time complexity is used a lot in the standard library, so it's good to understand the analysis. Thinking about how `push_back()` could be implemented in amortized constant time has helped me remember the simplified version of amortized constant time: It will run in *O(1)* in almost all cases, except very few where it will perform worse.

That is all we are going to cover regarding asymptotic complexity. Now we will move on to how you can tackle a performance problem and work effectively by optimizing your code.

What to measure and how?

Optimizations almost always add complexity to your code. High-level optimizations, such as choosing algorithms and data structures, can make the intention of the code clearer, but for the most part, optimizations will make the code harder to read and maintain. We therefore want to be absolutely sure that the optimizations we add have an actual impact on what we are trying to achieve in terms of performance. Do we really need to make the code faster? In what way? Does the code really use too much memory? To understand what optimizations are possible, we need to have a good understanding of the requirements, such as latency, throughput, and memory usage.

Optimizing code is fun, but it's also very easy to get lost without any measurable gains. We will start this section with a suggested workflow to follow when tuning your code:

1. **Define a goal**: It's easier to know how to optimize and when to stop optimizing if you have a well-defined, quantitative goal. For some applications, it's clear from the start what the requirements are, but in many cases it tends to be fuzzier. Even though it might be obvious that the code is running too

slow, it's important to know what would be good enough. Each domain has its own limits, so make sure you understand the ones that are relevant to your application. Here are some examples to make it more concrete:

    1. A response time for user-interactive applications of 100 ms; refer to https://www.nngroup.com/articles/response-times-3-important-limits.
    2. Graphics with 60 Frames Per Second (FPS) give you 16 ms per frame.
    1. Real-time audio with a 128 sample buffer at a 44.1 kHz sample rate means slightly less than 3 ms.

2. **Measure**: Once we know what to measure and what the limits are, we proceed by measuring how the application is performing right now. From *step 1*, it should be obvious if we are interested in average times, peaks, load, and so on. In this step, we are only concerned with measuring the goal we have set up. Depending on the application, measuring can be anything from using a stopwatch to using highly sophisticated performance analysis tools.
3. **Find the bottlenecks**: Next, we need to find the application's bottlenecks—the parts that are too slow and make the application useless. Don't trust your gut feeling at this point! Maybe you gained some insights by measuring the code at various points in *step 2*—that's fine, but you usually need to profile your code further in order to find the hot spots that matter most.
4. **Make an educated guess**: Come up with a hypothesis for how to improve the performance. Can a lookup table be used? Can we cache data to gain the overall throughput? Can we change the code so that the compiler can vectorize it? Can we decrease the number of allocations in the critical sections by reusing memory? Coming up with ideas is usually not that hard if you know that they are just educated guesses. It's okay to be wrong—you will find out later whether they had an impact or not.
5. **Optimize**: Let's implement the hypothesis we sketched in *step 4*. Don't spend too much time on this step making it perfect before you know that it actually has an effect. Be prepared to reject this optimization. It might not have the desired effect.
6. **Evaluate**: Measure again. Do the exact same test as in *step 2* and compare the results. What did we gain? If we didn't gain anything, reject the code and go back to *step 4*. If the optimization actually had

a positive effect, you need to ask yourself whether it's good enough to spend more time on. How complicated is the optimization? Is it worth the effort? Is this a general performance gain or is it highly specific to a certain case/platform? Is it maintainable? Can we encapsulate it, or does it spread out all over the code base? If you can't motivate the optimization, go back to *step 4*, otherwise continue to the final step.

7. **Refactor**: If you followed the instructions in *step 5* and didn't spend too much time writing perfect code in the first place, it's time to refactor the optimization to make it cleaner. Optimizations almost always need some comments to explain why we are doing things in an unusual way.

Following this process will ensure that you stay on the right track and don't end up with complicated optimizations that aren't motivated. The importance of spending time on defining concrete goals and measuring cannot be overestimated. In order to be successful in this area, you need to understand what performance properties are relevant for your application.

Performance properties

Before you start measuring, you must know which performance properties are important for the application you are writing. In this section, I will explain some frequently used terms when measuring performance. Depending on the application you are writing, some properties are more relevant than others. For example, throughput might be a more important property than latency if you are writing an online image converter service, whereas latency is key when writing interactive applications with real-time requirements. Below are some valuable terms and concepts that are worth becoming familiar with during performance measurement:

- **Latency/response time**: Depending on the domain, latency and response time might have very precise and different meanings. However, in this book, I mean the time between the request and the response

of an operation—for example, the time it takes for an image conversion service to process one image.
- **Throughput**: This refers to the number of transactions (operations, requests, and so on) processed per time unit—for example, the number of images that an image conversion service can process per second.
- **I/O bound or CPU bound**: A task usually spends the majority of its time computing things on the CPU or waiting for I/O (hard drives, networks, and so on). A task is said to be CPU bound if it would run faster if the CPU were faster. It's said to be I/O bound if it would run faster by making the I/O faster. Sometimes you hear about memory-bound tasks too, which means that the amount or speed of the main memory is the current bottleneck.
- **Power consumption**: This is a very important consideration for code that executes on mobile devices with batteries. In order to decrease the power usage, the application needs to use the hardware more efficiently, just as if we are optimizing for CPU usage, network efficiency, and so on. Other than that, high-frequency polling should be avoided since it prevents the CPU from going to sleep.
- **Data aggregation**: It's usually necessary to aggregate the data when collecting a lot of samples during performance measurement. Sometimes *mean values* are a good enough indicator of how the program performs, but more often the *median* tells you more about the actual performance since it's more robust against outliers. If you are interested in outliers, you can always measure *min* and *max* values (or the 10th percentile, for example).

This list is by no means exhaustive, but it's a good start. The important thing to remember here is that there are established terms and concepts that we can use when measuring performance. Spending some time on defining what we really mean by optimizing code helps us reach our goals faster.

Speedup of execution time

When we compare the relative performance between two versions of a program or function, it's customary to talk about **speedup**. Here I will give you a definition of speedup when comparing execution time (or latency). Assume we have measured the execution times of two versions of some code: an old slower version, and a new faster version. The speedup of execution time can then be computed accordingly:

$$Speedup\ of\ execution\ time = \frac{T_{old}}{T_{new}}$$

Where T_{old} is the execution time of the initial version of the code, and T_{new} is the execution time of the optimized version. This definition of speedup implies that a speedup of 1 means no speedup at all.

Let's make sure that you know how to measure the relative execution time with an example. Assume that we have a function that executes in 10 ms (T_{old} = 10 ms) and we manage to make it run in 4 ms after some optimization (T_{new} = 4 ms). We can then compute the speedup as follows:

$$Speedup = \frac{T_{old}}{T_{new}} = \frac{10\ ms}{4\ ms} = 2.5$$

In other words, our new optimized version provided a 2.5x speedup. If we want to express this improvement as a percentage, we can use the following formula to convert speedup to percentage improvement:

$$\%\ Improvement = 100\left(1 - \frac{1}{Speedup}\right) = 100\left(1 - \frac{1}{2.5}\right) = 60\%$$

We can then say that the new version of the code runs 60% faster than the old one and that this corresponds to a speedup of 2.5x. In this book, I will consistently use speedup, and not percentage improvement, when comparing execution time.

In the end, we are usually interested in execution time, but time is not always the best thing to measure. By inspecting other values on the hardware, the hardware might give us some other useful guidance toward optimizing our code.

Performance counters

Apart from the obvious properties, such as execution time and memory usage, it can sometimes be beneficial to measure other things. Either because they are more reliable or because they can give us better insights into what is causing our code to run slow.

Many CPUs are equipped with hardware performance counters that can provide us with metrics such as the number of instructions, CPU cycles, branch mispredictions, and cache misses. I haven't introduced these hardware aspects yet in this book, and we will not explore performance counters in depth. However, it's good to know that they exist and that there are ready-made tools and libraries (accessible through APIs) for all the major operating systems to collect **Performance Monitoring Counters** (**PMC**) while running a program.

The support for performance counters varies depending on the CPU and operating system. Intel provides a powerful tool called VTune, which can be used for monitoring performance counters. FreeBSD offers `pmcstat`. macOS comes with DTrace and Xcode Instruments. Microsoft Visual Studio provides support for collecting CPU counters on Windows.

Another popular tool is `perf`, which is available on GNU/Linux systems. Running the command:

```
perf stat ./your-program
```

will reveal a lot of interesting events, such as the number of context switches, page faults, mispredicted branches, and so on. Here is an example of what it output when running a small program:

```
Performance counter stats for './my-prog':
     1 129,86 msec task-clock              # 1,000 CPUs utilized
           8      context-switches        # 0,007 K/sec
           0      cpu-migrations          # 0,000 K/sec
      97 810      page-faults             # 0,087 M/sec
3 968 043 041     cycles                  # 3,512 GHz
1 250 538 491     stalled-cycles-frontend  # 31,52% frontend cycles idle
  497 225 466     stalled-cycles-backend   # 12,53% backend cycles idle
6 237 037 204     instructions            # 1,57  insn per cycle
                                          # 0,20  stalled cycles per insn
1 853 556 742     branches                # 1640,516 M/sec
    3 486 026     branch-misses           # 0,19% of all branches
  1,130355771 sec  time elapsed
  1,026068000 sec  user
  0,104210000 sec  sys
```

We will now move on to highlight some best practices when testing and evaluating performance.

Performance testing — best practices

For some reason, it's more common to see regression tests covering functional requirements than performance requirements or other non-functional requirements covered in tests. Performance testing is usu-

ally carried out more sporadically and, more often than not, way too late in the development process. My recommendation is to measure early and detect regression as soon as possible by adding performance tests to your nightly builds.

Choose algorithms and data structures wisely if they are to handle large inputs, but don't fine-tune code without good reason. It's also important to test your application with realistic test data early on. Ask questions about data sizes early in the project. How many table rows is the application supposed to handle and still be able to scroll smoothly? Don't just try it with 100 elements and hope that your code will scale—test it!

Plotting your data is a very effective way of understanding the data you have collected. There are so many good and easy-to-use plotting tools available today, so there is really no excuse for not plotting. Both RStudio and Octave provide powerful plotting capabilities. Other examples include gnuplot and Matplotlib (Python), which can be used on various platforms and require a minimal amount of scripting to produce useful plots after collecting your data. A plot does not have to look pretty in order to be useful. Once you plot your data, you are going to see the outliers and patterns that are usually hard to find in a table full of numbers.

This concludes our *What to measure and how*? section. Next, we'll now move on to exploring ways to find the critical parts of your code that waste too many resources.

Knowing your code and hot spots

The Pareto principle, or the 80/20 rule, has been applied in various fields since it was first observed by the Italian economist Vilfredo Pareto more than 100 years ago. He was able to show that 20% of the Italian population owned 80% of the land. In computer science, it has been widely used, perhaps even

overused. In software optimization, it suggests that 20% of the code is responsible for 80% of the resources that a program uses.

This is, of course, only a rule of thumb and shouldn't be taken too literally. Nevertheless, for code that has not been optimized, it's common to find some relatively small hot spots that spend the vast majority of the total resources. As a programmer, this is actually good news, because it means that we can write most of our code without tweaking it for performance reasons and instead focus on keeping the code clean. It also means that when doing optimizations, we need to know *where* to do them; otherwise, there is a good chance we will optimize code that will not have an impact on the overall performance. In this section, we will look at methods and tools for finding the 20% of your code that might be worth optimizing.

Using a profiler is usually the most efficient way of identifying hot spots in a program. Profilers analyze the execution of a program and output a statistical summary, a profile, of how often the functions or instructions in the program are being called.

In addition, profilers usually also output a call graph that shows the relationship between function calls, that is, the callers and callees for each function that was called during the profiling. In the following figure, you can see that the `sort()` function was called from `main()` (the caller) and that `sort()` called the function `swap()` (the callee):

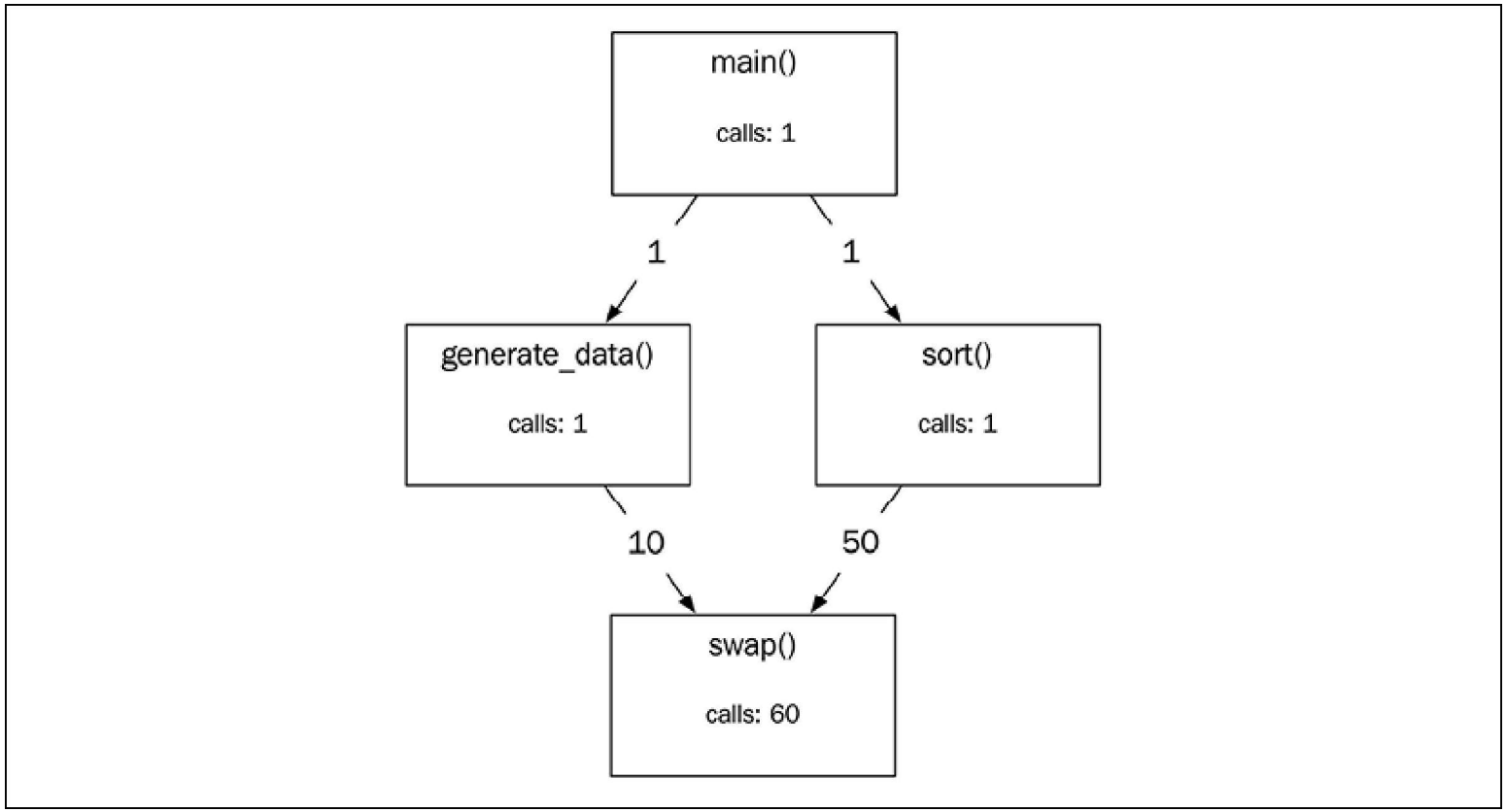

Figure 3.4: Example of a call graph. The function sort() is called once and calls swap() 50 times.

There are two main categories of profilers: sampling profilers and instrumentation profilers. The approaches can also be mixed to create a hybrid of sampling and instrumentation. `gprof`, the Unix performance analysis tool, is an example of this. The sections that follow focus on instrumentation profilers and sampling profilers.

Instrumentation profilers

By instrumentation, I mean inserting code into a program to be analyzed in order to gather information about how frequently each function is being executed. Typically, the inserted instrumentation code records each entry and exit point. You can write your own primitive instrumentation profiler by inserting the code manually yourself, or you can use a tool that automatically inserts the necessary code as a step in the build process.

A simple implementation might be good enough for your purposes, but be aware of the impact that the added code can have on performance, which can make the profile misleading. Another problem with a naive implementation like this is that it might prevent compiler optimizations or run the risk of being optimized away.

Just to give you an example of an instrumentation profiler, here is a simplified version of a timer class I have used in previous projects:

```
class ScopedTimer {
public:
  using ClockType = std::chrono::steady_clock;
  ScopedTimer(const char* func)
      : function_name_{func}, start_{ClockType::now()} {}
  ScopedTimer(const ScopedTimer&) = delete;
  ScopedTimer(ScopedTimer&&) = delete;
  auto operator=(const ScopedTimer&) -> ScopedTimer& = delete;
  auto operator=(ScopedTimer&&) -> ScopedTimer& = delete;
  ~ScopedTimer() {
    using namespace std::chrono;
    auto stop = ClockType::now();
    auto duration = (stop - start_);
    auto ms = duration_cast<milliseconds>(duration).count();
```

```
    std::cout << ms << " ms " << function_name_ << '\n';
  }

private:
  const char* function_name_{};
  const ClockType::time_point start_{};
};
```

The `ScopedTimer` class will measure the time from when it was created to the time it went out of scope, that is, destructed. We are using the class `std::chrono::steady_clock`, available since C++11, which was designed for measuring time intervals. `steady_clock` is monotonic, which means that it will never decrease between two consecutive calls to `clock_type::now()`. This is not the case for the system clock, for example, which can be adjusted at any time.

We can now use our timer class by measuring each function in a program by creating a `ScopedTimer` instance at the beginning of each function:

```
auto some_function() {
  ScopedTimer timer{"some_function"};
  // ...
}
```

Even though we don't recommend the use of preprocessor macros in general, this might be a case for using one:

```
#if USE_TIMER
#define MEASURE_FUNCTION() ScopedTimer timer{__func__}
#else
#define MEASURE_FUNCTION()
#endif
```

We are using the only predefined function-local `__func__` variable available since C++11 to get the name of the function. C++20 also introduced the handy `std::source_location` class, which provides us with the functions `function_name()`, `file_name()`, `line()`, and `column()`. If `std::source_location` is not supported by your compiler yet, there are other nonstandard predefined macros that are widely supported and can be really useful for debugging purposes, for example, `__FUNCTION__`, `__FILE__`, and `__LINE__`.

Now, our `ScopedTimer` class can be used like this:

```
auto some_function() {
  MEASURE_FUNCTION();
  // ...
}
```

Assuming that we have defined `USE_TIMER` when compiling our timer, it will produce the following output each time `some_function()` returns:

```
2.3 ms some_function
```

I have demonstrated how we can manually instrument our code by inserting code that prints the elapsed time between two points in the code. Although this is a handy tool for some scenarios, please be aware of the misleading results a simple tool like this can produce. In the next section, I will introduce a profiling method that doesn't require any modifications of the executing code.

Sampling profilers

Sampling profilers create a profile by looking at the running program's state at even intervals—typically, every 10 ms. Sampling profilers usually have a minimum impact on the program's actual performance, and it's also possible to build the program in release mode with all optimizations turned on. A drawback of sampling profilers is their inaccuracy and statistical approach, which is usually not a problem as long as you are aware of it.

The following figure shows a sampling session of a running program with five functions: `main()`, `f1()`, `f2()`, `f3()`, and `f4()`. The $\mathbf{t_1}$ - $\mathbf{t_{10}}$ labels indicate when each sample was taken. The boxes indicate the entry and exit point of each executing function:

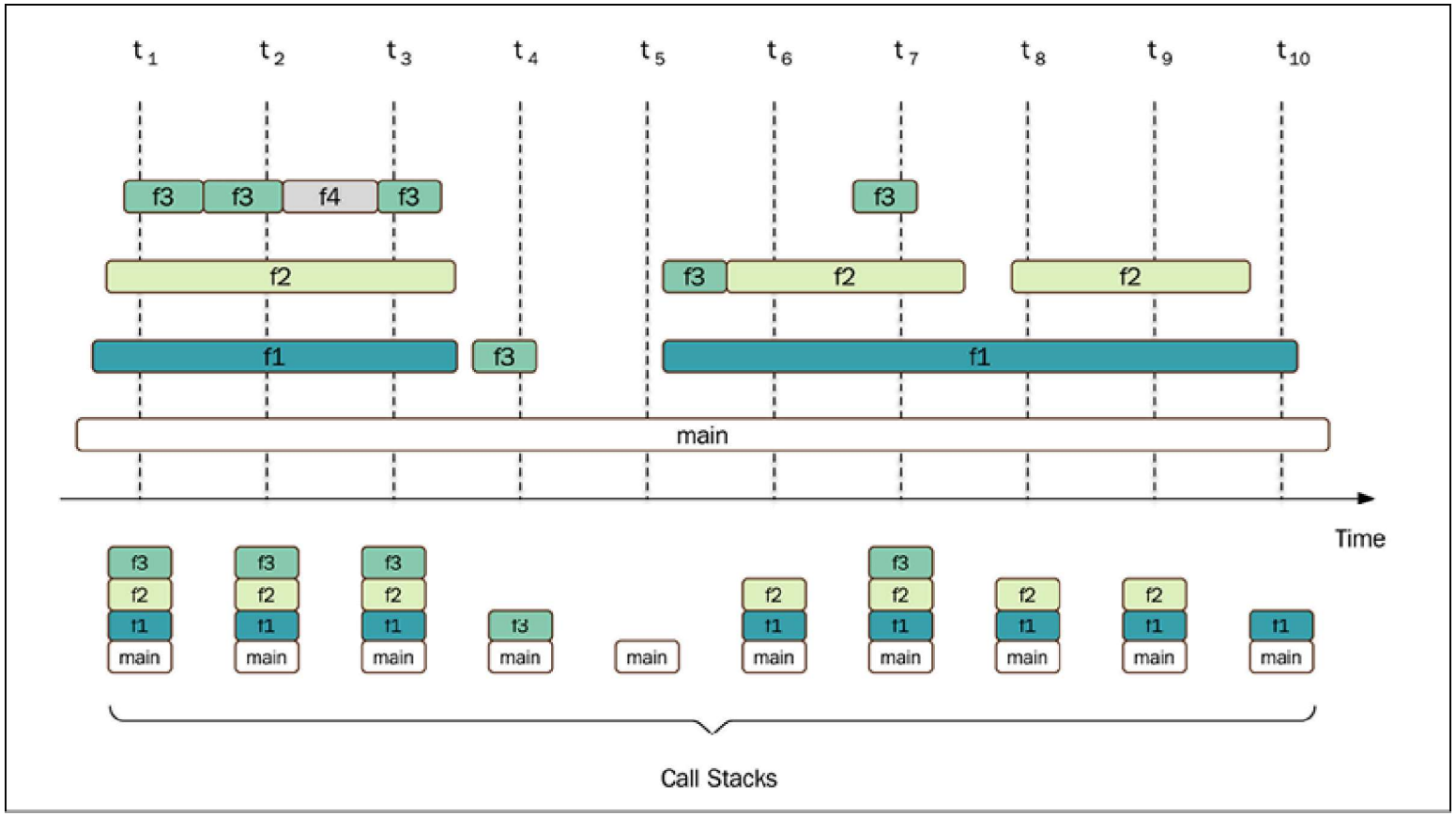

Figure 3.5: Example of a sampling profiler session

The profile is summarized in the following table:

Function	Total	Self
main()	100%	10%

`f1()`	80%	10%
`f2()`	70%	30%
`f3()`	50%	50%

Table 3.3: For each function, the profile shows the total percentage of call stacks that it appeared in (Total) and the percentage of call stacks where it occurred on top of the stack (Self).

The **Total** column in the preceding table shows the percentage of call stacks that contained a certain function. In our example, the main function was present in all 10 out of 10 call stacks (100%), whereas the `f2()` function was only detected in 7 call stacks, which corresponds to 70% of all call stacks.

The **Self** column shows, for each function, how many times it occurred on top of the call stack. The `main()` function was detected once on top of the call stack at the fifth sample, $\mathbf{t_5}$, whereas the `f2()` function was on top of the call stack at samples $\mathbf{t_6}$, $\mathbf{t_8}$, and $\mathbf{t_9}$, which corresponds to 3/10 = 30%.

The `f3()` function had the highest **Self** value (5/10) and was on top of the call stack whenever it was detected.

Conceptually, a sampling profiler stores samples of call stacks at even time intervals. It detects what is currently running on the CPU. Pure sampling profilers usually only detect functions that are currently being executed in a thread that is in a running state, since sleeping threads do not get scheduled on the CPU. This means that if a function is waiting for a lock that causes the thread to sleep, that time will not show up in the time profile. This is important because your bottlenecks might be caused by thread synchronization, which might be invisible to the sampling profiler.

What happened to the `f4()` function? According to the graph, it was called by the `f2()` function between samples two and three, but it never showed up in our statistical profile since it was never registered in any of the call stacks. This is an important property of sampling profilers. If the time between each sample is too long or the total sampling session is too short, then short and infrequently called functions will not show up in the profile. This is usually not a problem since these functions are rarely the functions you need to tune. You may note that the `f3()` function was also missed between $\mathbf{t}_5$ and $\mathbf{t}_6$, but since `f3()` was called very frequently, it had a big impact on the profile, anyway.

Make sure you understand what your time profiler actually registers. Be aware of its limitations and strengths in order to use it as effectively as possible.

Microbenchmarking

Profiling can help us find the bottlenecks in our code. If these bottlenecks are caused by inefficient data structures (see *Chapter 4*, *Data Structures*), the wrong choice of algorithm (see *Chapter 5*, *Algorithms*), or unnecessary contention (see *Chapter 11*, *Concurrency*), these bigger issues should be addressed first. But sometimes we find a small function or a small block of code that we need to optimize, and in those cases, we can use a method called **microbenchmarking**. With this process we create a microbenchmark—a program that runs a small piece of code in isolation from the rest of the program. The process of microbenchmarking consists of the following steps:

1. Find a hot spot that needs tuning, preferably using a profiler.
2. Separate it from the rest of the code and create an isolated microbenchmark.

3. Optimize the microbenchmark. Use a benchmarking framework to test and evaluate the code during optimization.
4. Integrate the newly optimized code into the program and *measure again* to see if the optimizations are relevant when the code runs in a bigger context with more relevant input.

The four steps of the process are illustrated in the following figure:

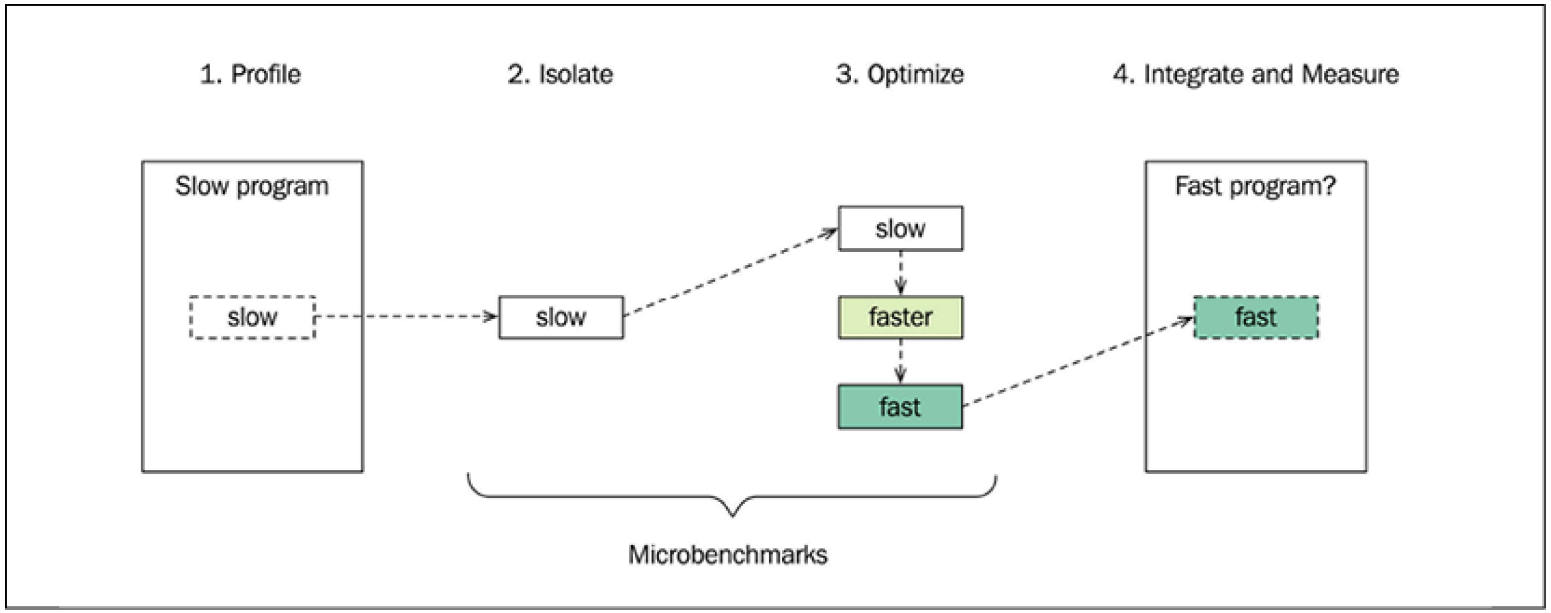

Figure 3.6: The microbenchmarking process

Microbenchmarking is fun. However, before diving into the process of trying to make a particular function faster, we should first make sure that:

- The time spent inside the function when running the program significantly affects the overall performance of the program we want to speed up. Profiling and Amdahl's law will help us understand this.

Amdahl's law will be explained below.
- We cannot easily decrease the number of times the function is being called. Eliminating calls to expensive functions is usually the most effective way of optimizing the overall performance of the program.

Optimizing code using microbenchmarking should usually be seen as the last resort. The expected overall performance gains are usually small. However, sometimes we cannot avoid the fact that we need to make a relatively small piece of code run faster by tuning its implementation, and in those cases microbenchmarks can be very effective.

Next, you will learn how the speedup of a microbenchmark can affect the overall speedup of a program.

Amdahl's law

When working with microbenchmarks, it's essential to keep in mind how big (or small) an impact the optimization of the isolated code will have on the complete program. It's our experience that it is easy to get a little too excited sometimes when improving a microbenchmark, just to realize that the overall effect was nearly negligible. This risk of going nowhere is partly addressed by using a sound profiling technique, but also by keeping the overall impact of an optimization in mind.

Say that we are optimizing an isolated part of a program in a microbenchmark. The upper limit of the overall speedup of the entire program can then be computed using Amdahl's law. We need to know two values in order to compute the overall speedup:

- First, we need to know how much execution time the isolated part accounts for in proportion to the overall execution time. We denote this value of *proportional execution* time with the letter p.

- Secondly, we need to know the speedup of the part we are optimizing—the microbenchmark that is. We denote this value of *local speedup* with the letter *s*.

Using *p* and *s*, we can now use Amdahl's law to compute the overall speedup:

$$Overall\ speedup = \frac{1}{(1-p) + \frac{p}{s}}$$

Hopefully this doesn't look too complicated, because it's very intuitive when put in to use. To get an intuition for Amdahl's law, you can see what the overall speedup becomes when using various extreme values of *p* and *s*:

- Setting *p = 0* and *s = 5x* means that the part we optimize has no impact on the overall execution time. Therefore, the overall speedup will always be 1x regardless of the value of *s*.
- Setting *p = 1*, *s = 5x* means that we optimize a part that accounts for the entire execution time of a program, and in that case, the overall speedup will always be equal to the speedup we achieve in the part we optimize—5x in this case.
- Setting *p = 0.5* and *s = ∞* means that we completely remove the part of the program that accounted for half of the execution time. The overall speedup will then be 2x.

The results are summarized in the following table:

p	s	Overall speedup

0	5x	1x
1	5x	5x
0.5	∞	2x

Table 3.4: Extreme values of p and s and the achieved overall speedup

A complete example will demonstrate how we can use Amdahl's law in practice. Say that you are optimizing a function so that the optimized version is 2 times faster than the original version, that is, a speedup of *2x (s = 2)*. Further, let's assume that this function is only responsible for 1% of the overall execution time of a program (*p = 0.01*), then the overall speedup of the entire program can be computed as follows:

$$Overall\ speedup = \frac{1}{(1-p)+\frac{p}{s}} = \frac{1}{(1-0.01)+\frac{0.01}{2}} = 1.005$$

So, even if we managed to make our isolated code 2 times faster, the overall speedup is only a factor of 1.005—not saying that this speedup is necessarily negligible, but we constantly need to go back and look at our gains in proportion to the bigger picture.

Pitfalls of microbenchmarking

There are plenty of hidden difficulties when measuring software performance in general and microbenchmarks in particular. Here, I will present a list of things to be aware of when working with

microbenchmarks:

- Results are sometimes overgeneralized and are treated as universal truths.
- The compiler might optimize isolated code differently compared to how it is optimized in the full program. For example, a function might be inlined in the microbenchmark but not when compiled in the full program. Or, the compiler might be able to precompute parts of the microbenchmark.
- Unused returned values in a benchmark might make the compiler remove the function we are trying to measure.
- Static test data provided in the microbenchmark might give the compiler an unrealistic advantage when optimizing the code. For example, if we hardcode how many times a loop will be executed, and the compiler knows that this hardcoded value happens to be a multiple of 8, it can vectorize the loop differently, skipping the prologue and epilogue of parts that might not be aligned with the SIMD register size. And then in the real code, where this hardcoded compile-time constant is replaced with a runtime value, that optimization doesn't happen.
- Unrealistic test data can have an impact on branch prediction when running the benchmark.
- Results between multiple measurements might vary because of factors such as frequency scaling, cache pollution, and the scheduling of other processes.
- The limiting factor of a code's performance could be due to cache misses, not the time it actually takes to execute instructions. Therefore, in many scenarios, an important rule of microbenchmarking is that you have to thrash the cache before you measure, otherwise you're not really measuring anything.

I wish I had a simple formula for avoiding all the pitfalls listed above, but unfortunately, I don't. However, in the next section, we will have a look at a concrete example to see how some of those pitfalls can be addressed by using a microbenchmarking support library.

A microbenchmark example

We will wrap this chapter up by going back to our initial examples with linear search and binary search from this chapter and demonstrate how they can be benchmarked using a benchmarking framework.

We began this chapter by comparing two ways of searching for an integer in a `std::vector`. If we knew that the vector was already sorted, we could use a binary search, which outperformed the simple linear search algorithm. I will not repeat the definition of the functions here, but the declaration looked like this:

```
bool linear_search(const std::vector<int>& v, int key);
bool binary_search(const std::vector<int>& v, int key);
```

The difference in the execution time of these functions is very obvious once the input is sufficiently large, but it will serve as a good enough example for our purpose. We will begin by only measuring `linear_search()`. Then, when we have a working benchmark in place, we will add `binary_search()` and compare the two versions.

In order to make a testing program, we first need a way to generate a sorted vector of integers. A simple implementation, as follows, will be sufficient for our needs:

```
auto gen_vec(int n) {
  std::vector<int> v;
  for (int i = 0; i < n; ++i) {
    v.push_back(i);
  }
```

```
  return v;
}
```

The vector that is returned will contain all integers between 0 and *n - 1*. Once we have that in place, we can create a naive test program like this:

```
int main() { // Don't do performance tests like this!
  ScopedTimer timer("linear_search");
  int n = 1024;
  auto v = gen_vec(n);
  linear_search(v, n);
}
```

We are searching for the value `n`, which we know isn't in the vector, so the algorithm will exhibit its worst-case performance using this test data. That's the good part of this test. Other than that, it is afflicted with many flaws that will make this benchmark useless:

- Compiling this code using optimizations will most likely completely remove the code because the compiler can see that the results from the functions are not being used.
- We don't want to measure the time it takes to create and fill the `std::vector`.
- By only running the `linear_search()` function once, we will not achieve a statistically stable result.
- It's cumbersome to test for different input sizes.

Let's see how these problems can be addressed by using a microbenchmarking support library. There are various tools/libraries for benchmarking, but we will use **Google Benchmark**,

https://github.com/google/benchmark, because of its widespread use, and as a bonus, it can also be easily tested online on the page **http://quick-bench.com** without the need for any installation.

Here is how a simple microbenchmark of `linear_search()` might look when using Google Benchmark:

```
#include <benchmark/benchmark.h> // Non-standard header
#include <vector>
bool linear_search(const std::vector<int>& v, int key) { /* ... */ }
auto gen_vec(int n) { /* ... */ }
static void bm_linear_search(benchmark::State& state) {
 auto n = 1024;
 auto v = gen_vec(n);
 for (auto _ : state) {
  benchmark::DoNotOptimize(linear_search(v, n));
 }
}
BENCHMARK(bm_linear_search); // Register benchmarking function
BENCHMARK_MAIN();
```

That's it! The only thing we haven't addressed yet is the fact that the input size is hardcoded to 1024. We will fix that in a while. Compiling and running this program will generate something like this:

```
---------------------------------------------------------------
Benchmark             Time   CPU         Iterations
---------------------------------------------------------------
bm_linear_search       361 ns 361 ns       1945664
```

The number of iterations reported in the rightmost column reports the number of times the loop needed to execute before a statistically stable result was achieved. The `state` object passed to our benchmarking function determines when to stop. The average time per iteration is reported in two columns: **Time** is the wall-clock time and **CPU** is the time spent on the CPU by the main thread. In this case they were the same, but if `linear_search()` had been blocked waiting for I/O (for example), the CPU time would have been lower than the wall-clock time.

Another important thing to note is that the code that generates the vector is not included in the reported time. The only code that is being measured is the code inside this loop:

```
for (auto _ : state) {   // Only this loop is measured
  benchmark::DoNotOptimize(binary_search(v, n));
}
```

The boolean value returned from our search functions is wrapped inside `benchmark::DoNotOptimize()`. This is the mechanism used to ensure that the returned value is not optimized away, which could make the entire call to `linear_search()` disappear.

Now let's make this benchmark a little more interesting by varying the input size. We can do that by passing arguments to our benchmarking function using the `state` object. Here is how to do it:

```
static void bm_linear_search(benchmark::State& state) {
  auto n = state.range(0);
  auto v = gen_vec(n);
  for (auto _ : state) {
    benchmark::DoNotOptimize(linear_search(v, n));
```

```
 }
}
BENCHMARK(bm_linear_search)->RangeMultiplier(2)->Range(64, 256);
```

This will start with an input size of 64 and double the size until it reaches 256. On my machine, the test generated the following output:

```
-------------------------------------------------------------------
Benchmark             Time    CPU        Iterations
-------------------------------------------------------------------
bm_linear_search/64     17.9 ns 17.9 ns     38143169
bm_linear_search/128    44.3 ns 44.2 ns     15521161
bm_linear_search/256    74.8 ns 74.7 ns     8836955
```

As a final example, we will benchmark both the `linear_search()` and the `binary_search()` functions using a variable input size and also try to let the framework estimate the time complexity of our functions. This can be done by providing the input size to the `state` object using the `SetComplexityN()` function. The complete microbenchmark example looks like this:

```
#include <benchmark/benchmark.h>
#include <vector>
bool linear_search(const std::vector<int>& v, int key) { /* ... */ }
bool binary_search(const std::vector<int>& v, int key) { /* ... */ }
auto gen_vec(int n) { /* ... */ }
static void bm_linear_search(benchmark::State& state) {
  auto n = state.range(0);
  auto v = gen_vec(n);
```

```
  for (auto _ : state) {
    benchmark::DoNotOptimize(linear_search(v, n));
  }
  state.SetComplexityN(n);
}
static void bm_binary_search(benchmark::State& state) {
  auto n = state.range(0);
  auto v = gen_vec(n);
  for (auto _ : state) {
    benchmark::DoNotOptimize(binary_search(v, n));
  }
  state.SetComplexityN(n);
}
BENCHMARK(bm_linear_search)->RangeMultiplier(2)->
  Range(64, 4096)->Complexity();
BENCHMARK(bm_binary_search)->RangeMultiplier(2)->
  Range(64, 4096)->Complexity();
BENCHMARK_MAIN();
```

When running the benchmark, we will get the following results printed to the console:

```
-------------------------------------------------------------------
Benchmark             Time     CPU       Iterations
-------------------------------------------------------------------
bm_linear_search/64     18.0 ns  18.0 ns     38984922
bm_linear_search/128    45.8 ns  45.8 ns     15383123
...
bm_linear_search/8192   1988 ns  1982 ns     331870
bm_linear_search_BigO   0.24 N   0.24 N
```

```
bm_linear_search_RMS        4 %   4 %
bm_binary_search/64      4.16 ns  4.15 ns     169294398
bm_binary_search/128     4.52 ns  4.52 ns     152284319
...
bm_binary_search/4096    8.27 ns  8.26 ns     80634189
bm_binary_search/8192    8.90 ns  8.90 ns     77544824
bm_binary_search_BigO    0.67 lgN 0.67 lgN

bm_binary_search_RMS        3 %   3 %
```

The output is aligned with our initial results in this chapter, where we concluded that the algorithms exhibit linear runtime and logarithmic runtime, respectively. If we plot the values in a table, we can clearly see the linear and logarithmic growth rates of the functions.

The following figure was generated using Python with Matplotlib:

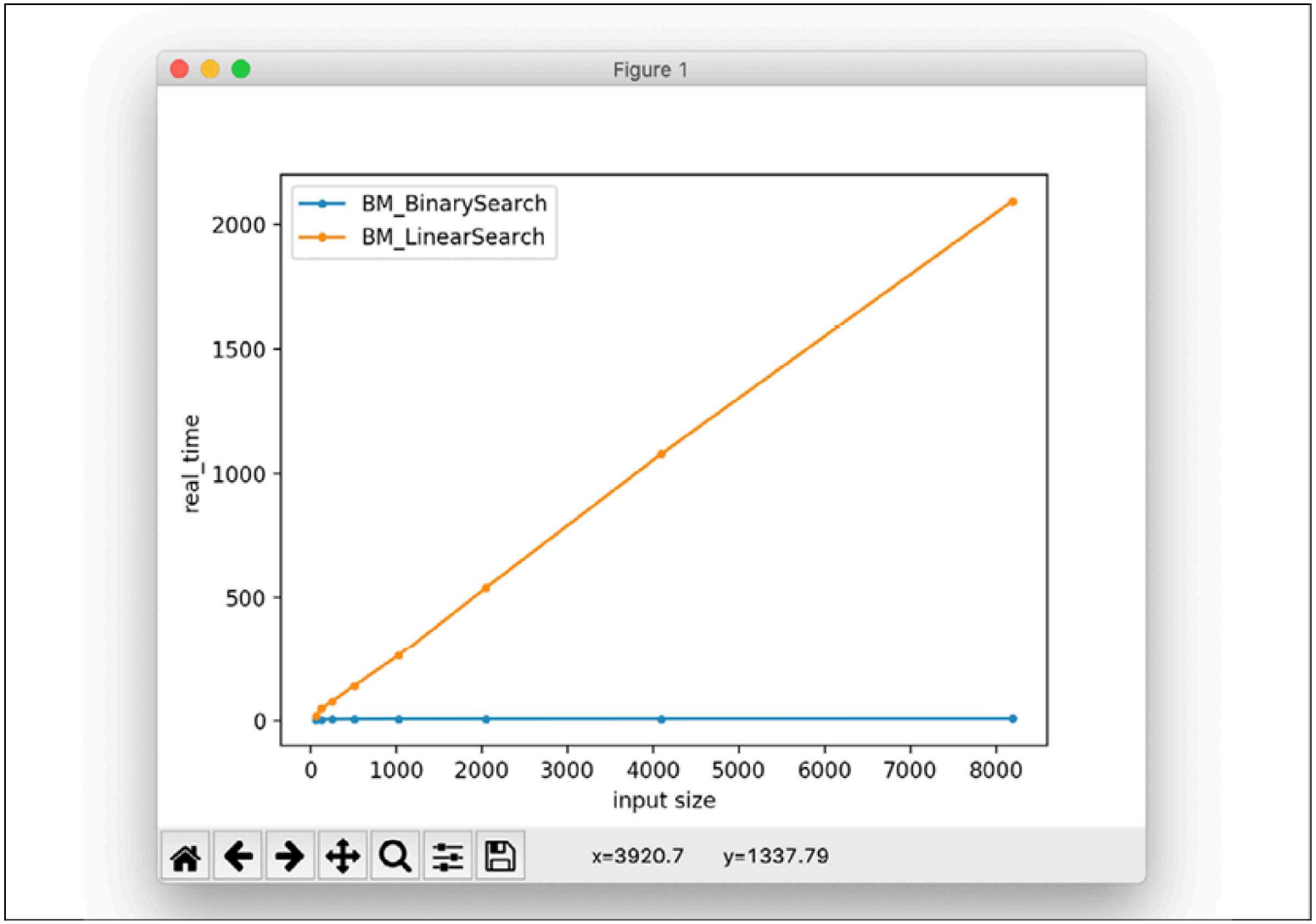

Figure 3.7: Plotting the execution time for various input sizes reveals the growth rates of the search functions

You now have a lot of tools and insights for finding and improving the performance of your code. I cannot stress enough the importance of measuring and setting goals when working with performance. A quote

from Andrei Alexandrescu will conclude this section:

> *"Measuring gives you a leg up on experts who don't need to measure."*
>
> *- Andrei Alexandrescu, 2015, Writing Fast Code I, code::dive conference 2015, https://codedive.pl/2015/writing-fast-code-part-1.*

Summary

In this chapter, you learned how to compare the efficiency of algorithms by using big O notation. You now know that the C++ standard library provides complexity guarantees for algorithms and data structures. All standard library algorithms specify their worst-case or average-case performance guarantees, whereas containers and iterators specify amortized or exact complexity.

You also discovered how to quantify software performance by measuring latency and throughput.

Lastly, you learned how to detect hot spots in your code by using CPU profilers and how to perform microbenchmarkings to improve isolated parts of your program.

In the next chapter, you will find out how to use data structures provided by the C++ standard library efficiently.