Chapter 11

Asymptotic Analysis

> *We could, of course, use any notation we want; do not laugh at notations; invent them, they are powerful. In fact, mathematics is, to a large extent, invention of better notations.*
>
> — Richard Feynman, *The Feynman Lectures on Physics, Vol. 1*

In this chapter, our goal is to compare a function $f(n)$ with some simple function $g(n)$ so that we can understand its order of growth as n approaches infinity. We discuss asymptotic equality $\sim$, asymptotic tightness Θ, asymptotic upper bounds O and o, and asymptotic lower bounds Ω and ω. These asymptotic notations are widely used in computer science, mathematics, and related disciplines. These notations allow us to quickly determine the order of growth of a function without forcing us to become imprecise. We briefly illustrate the use of asymptotic notations in the analysis of algorithms.

11.1 Asymptotic Equality

Let f and g be functions from the set of positive integers to the set of real numbers. We write $f \sim g$ and say that f is **asymptotically equal** to g if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

holds. By definition of the limit this means that for each real number $\epsilon > 0$ there exists a positive integer n_ϵ such that

$$\left| \frac{f(n)}{g(n)} - 1 \right| < \epsilon \tag{11.1}$$

holds for all $n \geqslant n_\epsilon$. As the notation suggests, the positive integer n_ϵ depends on the choice of ϵ. Informally, this means that for large n, the fraction $f(n)/g(n)$ gets arbitrarily close to 1.

Another way to interpret the inequality (11.1) is that two functions f and g are asymptotically equal if and only if the relative error $(f(n) - g(n))/g(n)$ between these functions vanishes for large n. Essentially, this means that the functions f and g have the same growth for large n.

Let us see why this notation is popular, especially in analysis, combinatorics, and number theory.

Example 11.1. Suppose that we are given the function

$$f(n) = \frac{n \ln n + \sqrt{n} + \ln n + 1517 \ln \ln n}{n}.$$

If we are only interested in the growth of the function f for large n, then it is important to realize that most of the terms of f do not have a significiant contribution! Indeed, the three terms

$$\lim_{n\to\infty} \frac{\sqrt{n}}{n} = 0, \quad \lim_{n\to\infty} \frac{\ln n}{n} = 0, \quad \lim_{n\to\infty} \frac{1517 \ln \ln n}{n} = 0$$

all vanish for large n, so only the term $n \ln n/n = \ln n$ has an impact on the growth of the function $f(n)$ for large n. The asymptotic equality

$$f(n) \sim \ln n$$

expresses that $f(n)$ essentially grows like the natural logarithm $\ln n$. Since $\ln n$ is shorter and more familiar than $f(n)$, it is the preferred way to describe the behavior of $f(n)$ for large arguments n.

Another example is the Harmonic number H_n that is defined by a sum that does not have a simple closed form. The asymptotic equality nevertheless allows us to describe the growth of H_n by comparing it to a familiar function.

Example 11.2. The n-th Harmonic number $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$ is asymptotically equal to the natural logarithm $\ln n$,

$$H_n \sim \ln n.$$

Indeed, since the inequalities $\ln(n+1) \leqslant H_n \leqslant 1 + \ln n$ hold, dividing by $\ln n$ and taking the limit yields for the logarithmic terms

$$\lim_{n\to\infty} \frac{\ln(n+1)}{\ln n} = \lim_{n\to\infty} \frac{n}{n+1} = 1 \quad \text{and} \quad \lim_{n\to\infty} \frac{1 + \ln n}{\ln n} = 1,$$

where we used l'Hôpital's rule in the calculation of the first limit. Thus, it follows from the squeeze theorem for limits that

$$\lim_{n\to\infty} \frac{H_n}{\ln n} = 1,$$

which proves that $H_n \sim \ln n$. In other words, the Harmonic numbers grow like the natural logarithm for large n. Relating a function such as H_n to the more familiar logarithmic function $\ln n$ certainly helps one to understand the growth of H_n as n approaches infinity.

Perhaps the best known example is given by Stirling's approximation, which relates the function $n!$ to a function that allows one to understand the asymptotic growth much better.

Example 11.3. The Stirling approximation yields

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n,$$

see Chapter 10.5.

One advantage of the asymptotic equality $\sim$ is that the expression can be simplified quite a bit. The next proposition illustrates this in the case of polynomials.

Proposition 11.4. *Let $p(x) = \sum_{k=0}^{m} a_k x^k$ be a nonzero polynomial of degree m with real coefficients. Then $p(x)$ is asymptotically equal to its leading term,*

$$p(x) \sim a_m x^m.$$

Proof. By assumption, the leading coefficient $a_m \neq 0$. By expanding terms, we get

$$\lim_{x\to\infty} \frac{p(x)}{a_m x^m} = \lim_{x\to\infty} \left(1 + \frac{a_{m-1}}{a_m x} + \cdots + \frac{a_1}{a_m x^{m-1}} + \frac{a_0}{a_m x^m}\right) = 1,$$

which proves the claim. □

The leading term of a polynomial dominates the growth. The advantage of the asymptotic notation $\sim$ is that we can dramatically simplify expressions such as $p(x) = x^3 + 3x^2 + 2x + 1$ when considering large x, since the relation $x^3 + 3x^2 + 2x + 1 \sim x^3$ tells us that the polynomial $p(x)$ grows like x^3. The notation brings clarity and gives us more compact expressions. Naturally, one can derive similar propositions for many other classes of functions.

The definition of $f \sim g$ by a limit means that all the techniques that you have learned in calculus can be applied. You can apply the knowledge that you have gained in calculus to deduce asymptotic equality in situations that are not entirely obvious. As an example, we give a convenient criterion for asymptotic equality of expressions that are of the form $f(n + k) - f(n)$ for some constant k. It is a slight generalization of a lemma given in Koecher [54, Chap. II.1.7]. The use of the mean value theorem in the proof is particularly noteworthy, since you can apply similar methods in other contexts as well.

Proposition 11.5. *Let c be a positive real number. Let f be a continuously differentiable function from the set of positive real numbers to the set of real numbers such that its derivative f' is monotonic, nonzero, and satisfies*

$$\lim_{n\to\infty} f'(n + c)/f'(n) = 1.$$

Then

$$f(n + c) - f(n) \sim cf'(n).$$

Proof. By the mean value theorem of calculus, there exists a real number θ in the range $0 \leqslant \theta \leqslant c$ such that

$$f(n+c) - f(n) = (n+c-n)f'(n+\theta) = cf'(n+\theta).$$

If f' is monotonically increasing (or monotonically decreasing), then

$$cf'(n) \underset{(\geqslant)}{\leqslant} f(n+c) - f(n) \underset{(\geqslant)}{\leqslant} cf'(n+c).$$

Dividing by $cf'(n)$ yields by assumption

$$\lim_{n\to\infty} \frac{cf'(n)}{cf'(n)} = 1 \quad \text{and} \quad \lim_{n\to\infty} \frac{cf'(n+c)}{cf'(n)} = 1.$$

Therefore, by the squeeze theorem for limits, we have

$$\lim_{n\to\infty} \frac{f(n+c) - f(n)}{cf'(n)} = 1,$$

which proves our claim. □

Example 11.6. Let c be a positive constant. Then

$$\sqrt{n+c} - \sqrt{n} \sim \frac{c}{2\sqrt{n}}.$$

Indeed, if we set $f(x) = \sqrt{x}$, then f is a continuously differentiable function on the positive real numbers. Its derivative $f'(x) = 1/(2\sqrt{x})$ is nonzero, monotonically decreasing, and satisfies $\lim_{n\to\infty} f'(n+c)/f'(n) = 1$. The claim follows from Proposition 11.5.

Example 11.7. Let k be a positive real number. Then

$$\log\left(1 + \frac{1}{n}\right)^k \sim \frac{k}{n}.$$

Indeed, choose $f(x) = k\log(x)$. Then $f'(x) = k/x$. As $f'(x) = k/x$ is monotonically decreasing and nonzero, and

$$\lim_{n\to\infty} \frac{f'(n+1)}{f'(n)} = \lim_{n\to\infty} \frac{n}{n+1} = 1,$$

it follows from Proposition 11.5 that

$$\log\left(1 + \frac{1}{n}\right)^k = k\log(n+1) - k\log(n) \sim \frac{k}{n}.$$

We conclude this section by illustrating how the asymptotic equality can simplify the calculation of limits.

Proposition 11.8. *If $f_1 \sim g_1$ and $f_2 \sim g_2$, then*

$$\lim_{n\to\infty} \frac{f_1(n)}{f_2(n)} = \lim_{n\to\infty} \frac{g_1(n)}{g_2(n)}$$

provided either of these limits exist (the proviso is important!).

Proof. We multiply the limit of f_1/f_2 by 1, here written in the form $1 = (g_1 g_2)/(g_1 g_2)$. This yields

$$\begin{aligned}\lim_{n\to\infty} \frac{f_1(n)}{f_2(n)} &= \lim_{n\to\infty} \frac{f_1(n)}{g_1(n)} \frac{g_1(n)}{g_2(n)} \frac{g_2(n)}{f_2(n)} \\ &= \left(\lim_{n\to\infty} \frac{f_1(n)}{g_1(n)}\right) \left(\lim_{n\to\infty} \frac{g_1(n)}{g_2(n)}\right) \left(\lim_{n\to\infty} \frac{g_2(n)}{f_2(n)}\right),\end{aligned}$$

where we have used in the last equality that the limit of a product is the product of the limits of its factors. Since the first and last limit is 1 by hypothesis, we can conclude that

$$\lim_{n\to\infty} \frac{f_1(n)}{f_2(n)} = \lim_{n\to\infty} \frac{g_1(n)}{g_2(n)},$$

which proves the claim. □

Example 11.9. Let us try to calculate the limit

$$\lim_{n\to\infty} \frac{3n^{15} + 7n^{14} + 8n^2 + \ln\sqrt{n}}{4n^{15} + 2n^7 + 2n\ln\sqrt{n}}.$$

Since the numerator and the denominator are both unbounded, we could apply l'Hôpital's rule 15 times[1] and calculate the limit. If we observe that $3n^{15} + 7n^{14} + 8n^2 + \ln\sqrt{n} \sim 3n^{15}$ and $4n^{15} + 2n^7 + 2n\ln\sqrt{n} \sim 4n^{15}$, then the previous proposition yields the limit

$$\lim_{n\to\infty} \frac{3n^{15} + 7n^{14} + 8n^2 + \ln\sqrt{n}}{4n^{15} + 2n^7 + 2n\ln\sqrt{n}} = \lim_{n\to\infty} \frac{3n^{15}}{4n^{15}} = \frac{3}{4}$$

in a straightforward fashion.

We can use similar tricks when the numerator and denominators contain radicals.

[1] Of course, there are also other ways to solve this problem. Focus on the explanation that follows!

Example 11.38. The function n^2 is in $\omega(n)$, since for a given positive constant c, the inequality $cn = c|n| \leqslant |n^2| = n^2$ holds for all positive integer $n \geqslant c$. On the other hand, n is not in $\omega(n)$, since there does not exist any positive integer n for which $2n = 2|n| \leqslant |n| = n$ holds.

Proposition 11.39. *Let f and g be functions from the set of positive integers to the set of real numbers, and assume that g is eventually nonzero. Then we have $f \in \omega(g)$ if and only if*

$$\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = \infty.$$

Proof. We have $f \in \omega(g)$ if and only if for all positive constants c there exists a positive integer n_0 such that $c \leqslant |f(n)|/|g(n)|$ holds for all $n \geqslant n_0$, so $|f(n)|/|g(n)|$ grows without bound. By definition of the limit, this is equivalent to

$$\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = \infty,$$

which proves the claim. □

11.6 Analysis of Algorithms

In this section, we will give a brief exposition of the analysis of algorithms. The purpose of the analysis is to get some insights into the running time or the space requirements of the computation. Since too many details of the compiler and computer architecture affect the precise time or space requirements, one has to settle for the more modest goal of obtaining reasonable bounds on these quantities. We will focus on the analysis of the running time of an algorithm.

Since we are settling for bounds on the running time rather than a precise estimate, we can use a simplified model of computation such as the random access machine. A random access machine is a very simple single-processor machine that executes instructions one after another without pipelining, speculative execution, branch prediction, or parallelism. It even lacks a memory hierarchy. We could now precisely define the instructions of the random access machine and their costs, as in Aho, Hopcroft, and Ullman [1]. However, we follow the approach taken in Cormen, Leiserson, Rivest, and Stein [17] and assume that elementary operations take a constant – yet unspecified – amount of time. Then there is no need to translate the pseudocode into random access machine instructions, which simplifies matters.

The running time of an algorithm is measured as a function of the size of the input. What is the size of the input? Unfortunately, the answer to this question depends on the application. We assume that we are given a function s that associates to an input x its **size** $s(x)$. For instance, if the input to a sorting algorithm is an array x with n elements, then $s(x) = n$ would be a natural measure for the size of the input. If the input x is an integer, then $s(x) = \lfloor \log_{10} x \rfloor + 1$ may be a good measure of input size for algorithms that deal

with long integer arithmetic. If the integers remain bounded, then $s(x) = 1$ would be a reasonable choice.

Let I denote the set of all possible inputs to an algorithm A. If we denote by $t_A(x)$ the running time of the algorithm A on an input x, then the **worst-case running time** of the algorithm A is defined as

$$\sup \{t_A(x) \mid x \in I, s(x) = n\}.$$

This measures the worst-case performance of the algorithm for all inputs of length n. We will use asymptotic notations to simplify the resulting worst-case running time.

So how do we determine the worst-case running time of an algorithm in practice? The answer is that we simply analyze the algorithm line-by-line and essentially count operations. The main complications are conditional statements, loops, and function or procedure calls. We now put forth the axioms that govern the run-time analysis. The list is incomplete, but you will see the pattern and it should be easy to extend this list by further language constructs.

A1. Elementary operations have constant running time.

Examples of elementary operations include assignments, arithmetic with machine size words, Boolean operations, comparisons, and array access, among others. The axioms says that they all have $\Theta(1)$ running time.

The running time of a compound statement is bounded by the sum of the running times of the individual component statements.

A2. If the statements S_1 and S_2 respectively have worst case running time $T(S_1)$ and $T(S_2)$ on an input of size n, then the compound statement $S_1; S_2$ has worst-case running time

$$T(S_1; S_2) \leqslant T(S_1) + T(S_2)$$

The next axiom gives a coarse-grained view of the running-time of a conditional statement. We should add that sometimes it makes sense to distinguish cases depending on the value of the condition C.

A3. The conditional statement

if C **then** S_1 **else** S_2

has worst-case running time $T(C) + \max\{T(S_1), T(S_2)\}$ when the conditional statement C and the statements S_1 and S_2 have worst case running time $T(C)$, $T(S_1)$ and $T(S_2)$ on an input of size n, respectively.

The next axiom gives the estimate for the running-time of a for loop, which iterates through all integers k in the range $a \leqslant k \leqslant b$, where a and b are integers. The notation $(a..b)$

A4. The running-time of a for-loop statement S given by

for k **in** $(a..b)$ **do**
S_1
end

has worst-case running time $T(S) \leqslant c + \sum_{k=a}^{b}(T(S_1, k) + c)$, where $T(S_1, k)$ is the worst-case running time of statement S_1 during iteration k and c is the cost of evaluating the loop condition.

In the worst case, such a for loop iterates $b-a+1$ times and checks the loop condition $b - a + 2$ times (as the loop verifies that k is out of bounds before it stops the iteration). One special case is worth noting. If $T(S_1) = T(S_1, k))$ is nonzero and independent of k, then this simplifies to

$$T(S) \leqslant (b - a + 1)\Theta(T(S_1)) + (b - a + 2)c.$$

There exist numerous variations of for loops. We will leave it to the reader to modify the axiom for these variations.

The running-time of a do-while loop (sometimes also called a repeat-until loop) is given in the next axiom. This style of loop executes the statement S_1 at least once before checking the condition C. It will repeat while the condition C is met.

A5. The running-time of a do-while loop statement S given by

begin
S_1
end while C

has worst-case running time $T(S) \leqslant \sum_{k=1}^{m(n)}(T(C)+T(S_1, k))$, where $m(n)$ is the maximal number of loop iterations on an input of size n and $T(C)$ and $T(S_1)$ are the worst-case running times of the conditional statement C and the statement S_1, respectively.

We illustrate how to obtain an estimate for the worst-case running-time with a small example.

Example 11.40. Suppose that you want to write a program to evaluate a polynomial

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0,$$

so for a given input x, you would like to know the value $y = p(x)$. You can save multiplications by using the Horner scheme

$$p(x) = (\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots a_1)x + a_0.$$

The following algorithm is based on the Horner scheme:

```
horner(a, n, x)                          cost   times
  res = a[n]                             c_1    1
  for  k = n−1  down to 0 do             c_2    n+1
    res = res * x + a[k]                 c_3    n
  end                                    c_4    n
  return  res                            c_5    1
end                                      c_6    1
```

The worst-case running time of the algorithm is given by

$$T(n) = c_1 + c_2(n+1) + c_3 n + c_4 n + c_5 + c_6 = \Theta(n).$$

This compares favorably with the naive $\Theta(n^2)$ algorithm that evaluates the polynomial $p(x)$ by computing each power independently by repeated multiplication.

EXERCISES

11.37. Analyze the running time of the following algorithm using a step count analysis (as in the Horner scheme).

```
// search a key in an array a[1..n] of length n
search(a, n, key)                        cost   times
  for k in (1..n) do
    if a[k]=key then
      return k
  end
  return  false
```

Determine the worst-case complexity of this algorithm.

11.38. Analyze the running time of the following algorithm using a step count analysis (as in the Horner scheme).

```
// determine the number of digits of an integer num
binary_digits(num)                       cost   times
  int cnt = 1
  while (num>1) do
    cnt = cnt + 1
    num = floor(num/2.0)
  end
  return cnt
```

(a) Give an axiom for the running time of a while loop.
(b) Determine the asymptotic running-time of this algorithm.

11.39. Determine how many times the block `Block(a,b)` is executed in the following code fragment:

```
n = 100
for a  in  (0..n-1) do
  for b  in  (0..n) do
    Block(a,b)
  end
end
```

You can assume that `Block(a,b)` does not exit the for loops, nor does it manipulate the loop variables `a` and `b`.

11.40. Determine how many times the block `Block(a,b)` is executed in the following code fragment:

```
n = 100
for a  in  (0..n) do
  for b  in  (0..a) do
    Block(a,b)
  end
end
```

You can assume that `Block(a,b)` does not exit the encapsulating for loops, and does not manipulate the loop variables `a` and `b`.

11.7 Notes

Asymptotic notations were introduced by Bachmann [7] and further popularized by Landau [57]. Graham, Knuth, and Patashnik [31] give a more in-depth treatment of asymptotic notations. Spencer [74] dedicates an entire book to asymptotic notations and gives numerous surprising applications. Books on the analysis of algorithms contain numerous examples of asymptotic estimates of the running time, see for example Aho, Hopcroft, and Ullman [1] or Cormen, Leiserson, Rivest, and Stein [17].