Chapter 0

Prologue

In the following, "the book" means

Dasgupta, Papadimitriou, Vazirani: Algorithms. McGraw Hill, 2008.

Please have Chapter 0 of the book ready, and read along with these notes, which contain additional remarks.

0.1 Books and algorithms

The book tells the story of the German goldsmith

Johannes Gutenberg

who in the 15th century invented printing books with movable letters cast from metal, and with a printing press. `https://en.wikipedia.org/wiki/Johannes_Gutenberg`
(By the way, his picture in the book is fantasy, or rather, fake.)

The book also mentions the Roman number system used in Europe at that time.

$$\text{MCMLXXXIV} = 1000 + (1000 - 100) + 50 + 10 + 10 + 10 + (5 - 1) \text{ means } 1984.$$

Adding and subtracting numbers in this representation is difficult.

Multiplying and dividing numbers in this representation is very difficult.

The revolution that resolved these problems had already happened many centuries earlier, not known in Europe for a long time!

The decimal number system, with digit 0, had been invented in India, around 600 CE (i.e. 600 AD):

`https://en.wikipedia.org/wiki/Hindu-Arabic_numeral_system#History`

`https://en.wikipedia.org/wiki/Hindu-Arabic_numeral_system#/media/File:The_Brahmi_numeral_system_and_its_descendants.png`

For some strange reason, we nowadays call it the *Arabic* numeral system.

It became known to Abu Jafar Muhammad ibn Musa "al-Khwarizmi" ("the man from Khwarezm", about 780–850), a Persian mathematician and polymath,

`https://www.wikiwand.com/de/Al-Chwarizmi`

who wrote influential books on methods for doing calculations with this number system. One of these books was, around 820, "The Compendious Book on Calculation by Completion and Balancing", "al-Kitab al-mukhtasar fi hisab **al-jabr** wal-Muqabala", whose title contained the root from which the word **algebra** evolved! See `https://en.wikipedia.org/wiki/The_Compendious_Book_on_Calculation_by_Completion_and_Balancing` .

Al-Khwarizmi's book described methods for **Addition**, **Subtraction**, **Multiplication**, **Division**, things kids have to study in school all over the world today.

More advanced methods were also described, like extracting **square roots**, calculating digits of $\pi = 3.14159\ldots$, and so on.

Procedures described by al-Khwarizmi were precise, step-by-step, mechanical, correct, applicable to arbitrarily long numbers; they were

algorithms.

The etymology of "*algorithm*" is a mess (see `https://www.etymonline.com/word/algorithm`), but it is connected to "al-Khwarizmi".

The German mathematician **Adam Ries**, who lived in Erfurt for a while, also wrote textbooks on arithmetic, for school children and for merchants, but around 1520, many centuries later.

`https://en.wikipedia.org/wiki/Adam_Ries`

`https://en.wikipedia.org/wiki/Adam_Ries#/media/File:Rechnung_auff_der_linihen_1525_Adam_Ries.PNG`

(Haus zum Schwarzen Horn, Erfurt, was the printer's shop.)

0.2 Enter Fibonacci

Much later, in Italy, there was Leonardo of Pisa (1170–1250), called Fibonacci.

`https://fr.wikipedia.org/wiki/Leonardo_Fibonacci`

(Another fake picture in the book. Seemingly the name was not used by Leonardo himself. See `https://en.wikipedia.org/wiki/Fibonacci` .)

Fibonacci popularized the Hindu-Arabic numeral system in Europe, also by writing a book (but of course hand-copied at this early time):
Liber Abaci (Book of Calculation).

Today you all know the famous sequence of *Fibonacci numbers* $F_0, F_1, F_2, \ldots$, which is

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

The sequence is defined recursively by:

$$F_n = \begin{cases} 0, & \text{for } n = 0 \\ 1, & \text{for } n = 1 \\ F_{n-2} + F_{n-1}, & \text{for } n \geq 2 \end{cases}$$

It has a myriad of applications, in biology, demography, art, architecture, computer science, to name only a few. Actually, it was known long before Fibonacci lived, like in the study of poetic meters in Sanskrit – ancient India again!
See `https://en.wikipedia.org/wiki/Fibonacci_number` .

We want to play around with Fibonacci numbers a little bit, so it is good to know somewhat more.

The sequence grows very fast. An important number in connection with it is the golden ratio

$$\Phi = \frac{1}{2}(\sqrt{5} + 1) \approx 1.618034,$$

which is the unique positive number that satisfies $\Phi^2 = \Phi + 1$ (This is easily checked.)

Claim 1: $F_n \geq \Phi^{n-2}$, for $n \geq 1$.

The *proof* is by induction on n. First the **basis**: The claim is true for $n = 1$, since $F_1 = 1 > \Phi^{-1}$, and for $n = 2$, since $F_2 = 2 > 1 = \Phi^0$. Now assume (as **induction hypothesis**) that $n \geq 3$ and that the claim is true for all numbers $n' < n$. Then (this is the **induction step**):

$$F_n = F_{n-2} + F_{n-1} \geq \Phi^{n-4} + \Phi^{n-3} = (1 + \Phi)\Phi^{n-4} = \Phi^2\Phi^{n-4} = \Phi^{n-2}.$$

Claim 2: $F_n \leq \Phi^{n-1}$, for $n \geq 0$.

Again, the *proof* is by induction on n. Basis: The claim is true for $n = 0$, since $F_0 = 0 < \Phi^{-1}$, and for $n = 1$, since $F_1 = 1 = \Phi^0$. Further if $n \geq 2$ and the claim is true for all $n' < n$ (the induction hypothesis), we get (the induction step)

$$F_n = F_{n-2} + F_{n-1} \leq \Phi^{n-3} + \Phi^{n-2} = (1 + \Phi)\Phi^{n-3} = \Phi^2\Phi^{n-3} = \Phi^{n-1}.$$

The claims taken together say that F_n deviates from Φ^{n-1} only by a constant factor, which is at most Φ.

We say the Fibonacci sequence grows "expontially", with basis Φ.

Note that up to constant factors

$$\Phi^{n-1} \approx \Phi^n \approx (2^{\log_2 \Phi})^n = 2^{(\log_2 \Phi)n} \approx 2^{0.694n},$$

since $\log_2 \Phi \approx 0.694241$.

If you want to know even more details, look up that F_n can be given by the "closed formula"

$$F_n = \frac{1}{\sqrt{5}} \left(\Phi^n - (1 - \Phi)^n\right).$$

Here Φ and $1 - \Phi = \frac{1}{2}(1 - \sqrt{5}) \approx -0.618034$ are the two roots of the quadratic equation $x^2 = x + 1$. This formula allows us to estimate F_n as the closest integer to $\frac{1}{\sqrt{5}}\Phi^n$, for $n \geq 0$. From this one sees the exponential growth even more clearly.

The sequence

$$1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, \ldots$$

of the powers of 2 grows exponentially with basis 2.

We want to calculate the exact value of F_n, given n. The definition shows us a method.

Algorithm 1: fib1

```
function fib1($n$):
    if $n = 0$ then return 0
    if $n = 1$ then return 1
    return fib1($n - 1$) + fib1($n - 2$)        // recursion!
```

The algorithm is written in "pseudocode". (A little different from the way it is in the book, but you will understand.) Note it is *recursive*, i.e. that for executing the algorithm on input $n \geq 2$ it calls the same algorithm on smaller inputs $n - 2$ and $n - 1$.

We ask three questions, which we will ask over and over again, about all our algorithms:

- Is the algorithm correct?
- How much time does it take? (As a function of the “input size” n.)
- Can we do better?

Correctness: Yes, it is correct, since we just use the formula from the definition of the sequence.

Running time:
Let us say the tests in lines **2** and **3** and the addition in line **4** each cost one unit of time. (Just define time units so that they accommodate these operations. We assume that organizing the recursive calls and returning the result is subsumed in these costs.)

Let $T(n) :=$ number of time units for calculating F_n by **fib1**.
(We don't know the time, but we can still give it a name!)

Claim: $T(n) \leq F_{n+5} - 3$, for all $n \geq 0$.

Proof: By induction on n. Note that $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, F_6 = 8$.
Basis: $n = 0$ and $n = 1$: $T(0) = 1 \leq 5 - 3 = F_5 - 3$ and $T(1) = 2 \leq 8 - 3 = F_6 - 3$.
Now assume $n \geq 2$. The **induction hypothesis** is that the claim is true for all $n' < n$.
Induction step: We add the cost for lines **2** and **3**, the two recursive calls, and the addition in line **4**. We get:
$T(n) \leq 1 + 1 + T(n-1) + T(n-2) + 1$.
Using the induction hypothesis for $n-1$ and $n-2$ and the definition of the Fibonacci sequence we get
$T(n) \leq (F_{(n-1)+5} - 3) + (F_{(n-2)+5} - 3) + 3 = F_{n+4} + F_{n+3} - 3 = F_{n+5} - 3$,
which is the induction claim. □

The claim gives an “upper bound” on the running time, which is exponential in n.

The bad news is that function **fib1** really needs exponential running time. We just count recursive calls to **fib1**. Let

$$C(n) := \text{number of recursive calls for calculating } F_n.$$

Note that $C(0) = 1 = F_1$ and $C(1) = 1 = F_2$ and $C(n) \geq C(n-2) + C(n-1)$ for all $n \geq 2$. From this it follows directly (by induction) that $C(n) \geq F_{n+1} \approx \frac{1}{\sqrt{5}}\Phi^{n+1} \approx \frac{1}{\sqrt{5}}2^{0.694n}$.

So it seems our first algorithm is not really good. Let's try a quite small number, like $n = 200$. One can see that $F_{200} \approx \frac{1}{\sqrt{5}}2^{0.694 \cdot 200} \approx 2^{138}$, up to a small constant factor.

How long does it take to carry out the 2^{138} calls needed for F_{200}?

Here's a useful estimate (please memorize): $2^{10} = 1024 > 1000 = 10^3$.

So $2^{138} > 10^{3 \cdot 13.8} > 10^{41}$.

Now 10^{41} is quite a lot. Assume our super-supercomputer (for a list see `https://en.wikipedia.org/wiki/Supercomputer`) carries out $5 \cdot 10^{17}$ recursive calls per second (actually, the fastest supercomputers of the year 2020 carry out somewhat less than 500,000 TeraFLOPs, meaning $500{,}000 \cdot 10^{12} = 5 \cdot 10^{17}$ floating-point operations (FLOPs), but it is reasonable to assume a recursive call is slower than a FLOP). This also is a lot. But then the call **fib1**(200) takes at least

$$10^{41}/(5 \cdot 10^{17}) = 2 \cdot 10^{23} \text{ seconds, which is about } 6.34 \cdot 10^{15} \text{ years.}$$

Astrophysicists estimate that the age of the universe is about $13.8 \cdot 10^9 < 1.5 \cdot 10^{10}$ years (`https://en.wikipedia.org/wiki/Age_of_the_universe`). This means our calculation would take more than 400 000 times the age of the universe. So unreachable are calculations with exponential running times even for not too large inputs!

Can we do better?

What is wrong with **fib1**? As shown in Figure 0.1 in the book, it repeats the same calculations over and over again. Namely, **fib1**(n) is called once (F_1 times), **fib1**($n-1$) is called once (F_2 times), **fib1**($n-2$) is called twice ($F_1 + F_2 = F_3$ times), **fib1**($n-3$) is called three ($F_2 + F_3 = F_4$) times, **fib1**($n-4$) is called five ($F_3 + F_4 = F_5$) times, ..., **fib1**($n-i$) is called $F_{i-1} + F_i = F_{i+1}$ times, ..., and **fib1**(0) is called F_{n+1} times.

Fortunately, the naive (or stupid) procedure used in **fib1** can be improved easily. Of course, one should calculate the Fibonacci numbers in the order $F_0, F_1, F_2, \ldots, F_n$, and *store* the results one has already calculated. This leads to the following second attempt.

Algorithm 2: fib2

```
function fib2($n$):
    generate array $F[0..n]$ of integers
    $F[0] \leftarrow 0$; $F[1] \leftarrow 1$;
    for $i$ from 2 to $n$ do        // iteration, with memory
        $F[i] \leftarrow F[i-1] + F[i-2]$;
    return $F[n]$.
```

This is quite nice, we only have n additions, for $n = 200$ this means 200 additions. One has to take into account that the largest numbers that we add will be of size about 2^{138}. This means they need 138 bits to represent in a computer, or five 32-bit words. Still, adding two such numbers is no problem at all. (You can even do that by hand. The number of decimal digits is not more than 42.)

Our algorithm **fib2** needs an array of size n. This is not really necessary. Observe:

To calculate $F[i]$, only $F[i-2]$ and $F[i-1]$ are needed, we can forget older values.

So we require only two variables `Fi_1` and `Fi` for the two most recent values F_{i-1} and F_i, and one auxiliary variable `g` for their sum. The new algorithm is:

Algorithm 3: fib2a

```
1 function fib2a(n):
2   Fi_1 ← 0; Fi ← 1;        // Fi_1 contains F_0, Fi contains F_1
3   for i from 2 to n do
4     g ← Fi_1 + Fi;
5     Fi_1 ← Fi;
6     Fi ← g;       //  Invariant: Fi_1 and Fi contain F_{i-1} and F_i
7   return Fi.
```

With this algorithm we can calculate calculate F_{200} by hand (although it takes a little patience) and F_n for really large n. As before, we have n additions, but now also very little storage space.

How long does the calculation take? Is the running time "linear" (like the number of additions), i.e., proportional to n (we'll later call this $O(n)$ or $\Theta(n)$)?

More careful analysis

There is a catch. The numbers we are dealing with are possibly large: In round i we add F_{i-2} and F_{i-1}, they are about $2^{0.694i}$, and have about $0.694i$ bits.

How do we represent such long integers in a computer? O.k., if you need them you use a package like GNU Multiple Precision Arithmetic Library (or another one, see `https://en.wikipedia.org/wiki/List_of_C++_multiple_precision_arithmetic_libraries` for an incomplete list). But what is happening in these libraries if we are dealing with integers of thousands or millions of bits, say?

Assume you want to store an integer x with ℓ bits in a computer with word length w. (Nowadays, standard word lengths are $w = 32$ bits or $w = 64$ bits.) So create an array $A[0..k-1]$ of long unsigned integers, for[1] $k \geq \lceil \ell/w \rceil$. Then chop the binary representation of x into w-bit pieces $x_0, x_1, \ldots, x_{k-1}$, with x_0 the w lowest-order bits, $\ldots$, x_{k-1} the highest order bits. Then store x_i in $A[i]$, for $0 \leq i < k$. Note that we have

$$x = \sum_{0 \leq j < k} x_i \cdot (2^w)^j,$$

[1]For any real number α, $\lceil \alpha \rceil$ ("ceiling", "rounded up") is the smallest integer $\geq \alpha$, and $\lfloor \alpha \rfloor$ ("floor", "rounded down") is the largest integer $\leq \alpha$.

so this can be regarded as representing x in a digital number system with basis 2^w.

If we add two numbers represented in this way on a computer, we need $k \approx \ell/w$ machine instructions for adding the single words.

For our Fibonacci numbers, the bitlength of the numbers in the addition to get F_i is about $0.694i$, giving about $0.694i/w$ machine instructions. Overall, we have about[2]

$$\sum_{2\le i\le n} 0.694i/w = \frac{0.694}{w}\sum_{2\le i\le n} i = \frac{0.694}{w}\Big(\frac{n(n+1)}{2} - 1\Big) > \frac{0.347}{w}\cdot n^2$$

machine instructions.

Now assume $w = 64$. Then $0.347/w \approx 0.00542$, and the running time behaviour of **fib2a** is about $0.00542n^2$ additions. This is "quadratic", proportional to n^2 (we'll later call this $O(n^2)$ or $\Theta(n^2)$).

If $n = 10^{12}$, which is not entirely unreasonable, we will have about $0.00542 \cdot 10^{24} \approx 5 \cdot 10^{21}$ word additions. On our super-supercomputer from above this will take about 10 000 seconds, which is about $2\frac{3}{4}$ hours. (On a notebook it takes longer; for notebooks or desktop computers $n = 10^9$ is a more reasonable input size.)

Can we do better than that? It may be surprising, but yes, see exercises.

0.3 O-Notation

We want to be able to estimate the number of computer steps without getting bogged down in the details of architecture and even of implementation details for algorithms. For this, we introduce notation that allows us to just note the rough behaviour of functions (in particular running time bounds) for large input sizes n.

In the following, one should imagine $g(n)$ is a complicated, detailed bound on the running time or the number of operations carried out by an algorithm, and $f(n)$ is a smoother function used to estimate $g(n)$. (The definition is more general.)

Definition 0.3.1. *Let $f\colon \mathbb{N} \to \mathbb{R}^+$ and $g\colon \mathbb{N} \to \mathbb{R}^+$ be functions. (If $g(n)$ is negative or undefined for a finite number of n, we do not care.) We say "$g(n) = O(f(n))$" if*

$$\text{there are } c > 0 \text{ and } n_0 \in \mathbb{N} \text{ such that } g(n) \le c \cdot f(n) \text{ for all } n \ge n_0.$$

(See Figure 1.)

We read: $g(n)$ "is (big-)Oh of $f(n)$".

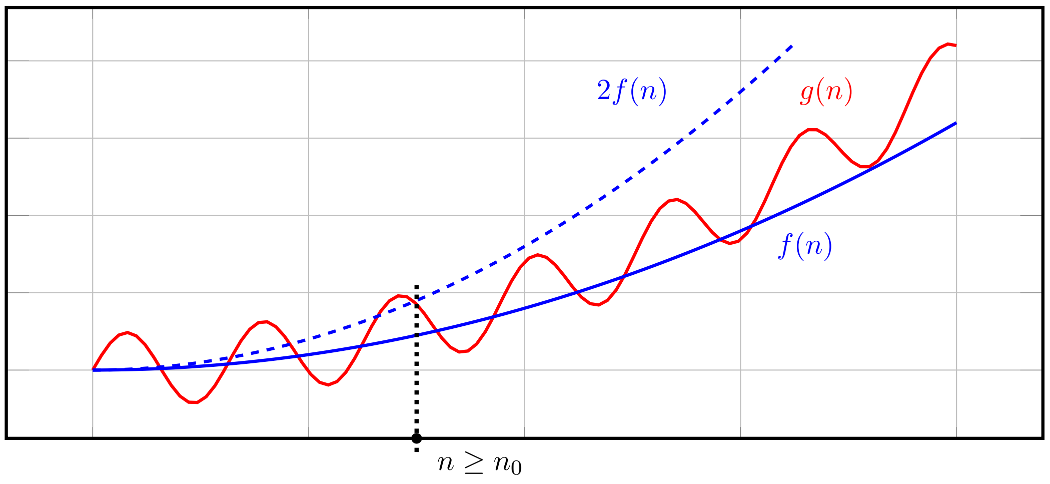

Figure 1: $g(n) = O(f(n))$, since $g(n) \leq 2 \cdot f(n)$ for $n \geq n_0$.

Sometimes one sees the notation "$g \in O(f)$" for the same idea.[3]

Example: $4n^2 + 2n + 5 = O(n^2)$, since $4n^2 + 2n + 5 \leq 5n^2$ for all $n \geq 3$. Similarly one sees $4n^2 + 2n + 5 = O(n^3)$: overestimating is not forbidden. Note that in O-notation the "=" does not mean the two things are *equal*, rather that $g(n)$ is somehow "smaller than or equal to" $f(n)$.

Informal interpretation: "*Asymptotically*" (i.e., for n getting larger and larger), $g(n)$ grows at most as fast as $f(n)$.

Some rules. (We will use these rules, and you will get used to them. For proofs see more detailed textbooks on algorithms.)

Assume $f\colon \mathbb{N} \to \mathbb{R}^+$ and $g\colon \mathbb{N} \to \mathbb{R}^+$ are functions. It does not matter if these functions are not defined for some small values of n.

Limit rule. If $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ for some $c \geq 0$, then $g(n) = O(f(n))$.

Example: $\lim_{n\to\infty} \frac{4n^2+5n+6}{n^2} = 4$, hence $4n^2+5n+6 = O(n^2)$. The generalization for arbitrary polynomials should be clear: it is the leading term that counts.

This rule is the easiest way of establishing O-relations among functions. When asked to prove $g(n) = O(f(n))$, the first approach is to try to determine the

[2] We use the well-known formula $1 + 2 + 3 + \cdots + n = n(n+1)/2$.

[3] The book uses the notation "$g = O(f)$", but this is not common. A few other authors write "$g(n) \in O(f(n))$", but this is also rare. It's better not to use these variants.

limit of the quotient.

Addition rule. If $g_1(n) = O(f_1(n))$ and $g_2(n) = O(f_2(n))$, then $g_1(n) + g_2(n) = O(f_1(n) + f_2(n)) = O(\max\{f_1(n), f_2(n)\})$.

This rule is used for analyzing the running time of algorithms carried out sequentially.

Example: If we have an algorithm A_1 with running time $t_1(n) = O(n^2)$ and an algorithm A_2 with running time $t_2(n) = O(n \log n)$, and these algorithms are carried out one after the other, the total running time will be $O(n^2 + n \log n)$, which is $O(n^2)$.

Multiplication rule. If $g_1(n) = O(f_1(n))$ and $g_2(n) = O(f_2(n))$, then $g_1(n)g_2(n) = O(f_1(n)f_2(n))$.

This rule is used for analyzing the running time of algorithms consisting of a loop.

Example: If algorithm A involves a loop that is carried out $t_1(n) = O(n)$ many times on inputs of size n, and each execution of the body of the loop takes time $t_2(n) = O(n \log n)$, the total time will be $O(n \cdot n \log n)$, which is $O(n^2 \log n)$.

Transitivity rule. If $h(n) = O(g(n))$ and $g(n) = O(f(n))$, then $h(n) = O(f(n))$.

This rule allows us to argue in a chain, like this: If $t_1(n) = O(n^2)$ and $t_2(n) = O(n^2 \log n)$, then $t_1(n) + t_2(n) = O(n^2 + n^2 \log n) = O(n^2 \log n) = O(n^{11/5})$.

Domination rule. ("Omit lower order terms.") If $\lim_{n\to\infty} \frac{h(n)}{g(n)} = 0$, then $g(n) \pm h(n) = O(g(n))$.

Constant factor rule. ("Omit constant factors.") Constant factors inside a $O(\dots)$ expression are irrelevant, so they can and should be omitted: If $f(n)$ is a "simple" function, write $O(f(n))$, not $O(c \cdot f(n))$ for a constant c.

Examples: Write $O(n^2)$, not $O(0.1n^2)$, write $O(1)$, not $O(3)$.

Write $O(n \log n)$ (short for $O(n \log_2 n)$), not $O(n \log_{10} n)$, because $n \log_{10} n = \frac{1}{\log_2 10} \cdot n \log_2 n$.

Some concrete rules are listed in the book:

1. n^a dominates n^b if $a > b > 0$. This means that $n^b = O(n^a)$ and even $n^a + n^b = O(n^a)$. The reason is that $\lim_{n\to\infty} \frac{n^b}{n^a} = \lim_{n\to\infty} \frac{1}{n^{a-b}} = 0$, and the domination rule.

2. Any polynomial dominates any logarithm, e.g. $\sqrt{n}$ dominates $(\log n)^3$. That $\lim_{n\to\infty} \frac{(\log n)^3}{\sqrt{n}} = 0$ is a fact from calculus.[4]

3. Any exponential function dominates any polynomial, e.g., 2^n dominates n^5. The reason is that $\lim_{n\to\infty} \frac{n^5}{2^n} = \lim_{n\to\infty} 2^{-(n-5\log n)} = 0$. (One could also use the quotient rule.)

Definition 0.3.2. *We say "$g(n) = \Omega(f(n))$" if*

$$\textit{there are } c > 0 \textit{ and } n_0 \in \mathbb{N} \textit{ such that } \; g(n) \geq c \cdot f(n) \textit{ for all } n \geq n_0.$$

(See Figure 2.)

We read: $g(n)$ "is (big-)Omega of $f(n)$". Informal interpretation: Asymptotically, $g(n)$ grows at least as fast as $f(n)$.

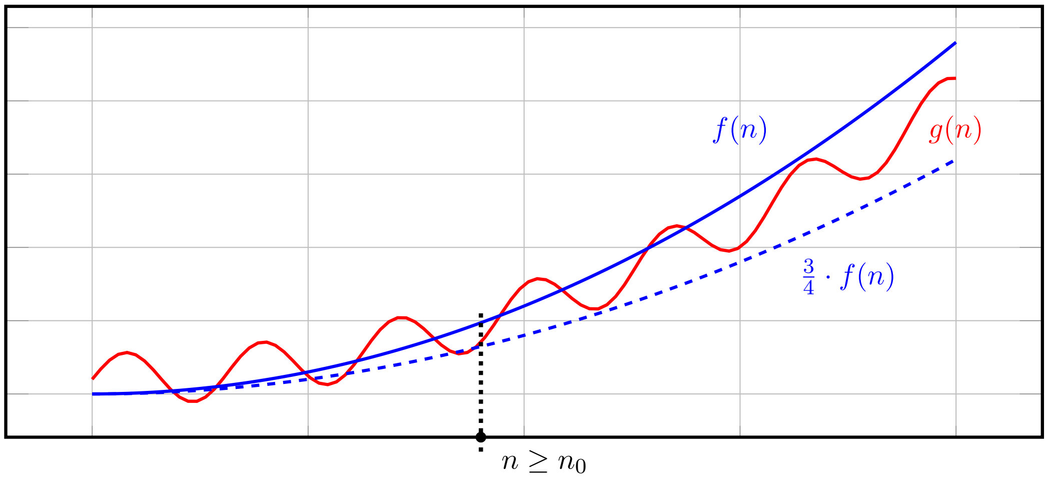

Figure 2: $g(n) = \Omega(f(n))$, since $g(n) \geq \frac{3}{4} \cdot f(n)$ for $n \geq n_0$.

In comparison to "O" only the comparison symbol switches from $\leq$ to $\geq$. Equivalently, one can say that $g(n) = \Omega(f(n))$ is the same as $f(n) = O(g(n))$.

Finally, we say $g(n) = \Theta(f(n))$ if $g(n) = O(f(n))$ *and* $g(n) = \Omega(f(n))$. (Apart from constant factors, $g(n)$ grows just as fast as $f(n)$.) For an illustration see Figure 3. We read: $g(n)$ "is (big-)Theta of $f(n)$".

[4]One possible proof: $\lim_{n\to\infty} \frac{(\log n)^3}{\sqrt{n}} = \lim_{x\to\infty} \frac{(\log x)^3}{\sqrt{x}}$ (for real numbers), and this is equal to $\lim_{t\to\infty} \frac{t^3}{\sqrt{2^t}} = \lim_{t\to\infty} \frac{t^3}{\sqrt{2}^t} = \lim_{n\to\infty} \frac{n^3}{\sqrt{2}^n}$. The latter limit is 0, by the quotient criterion.

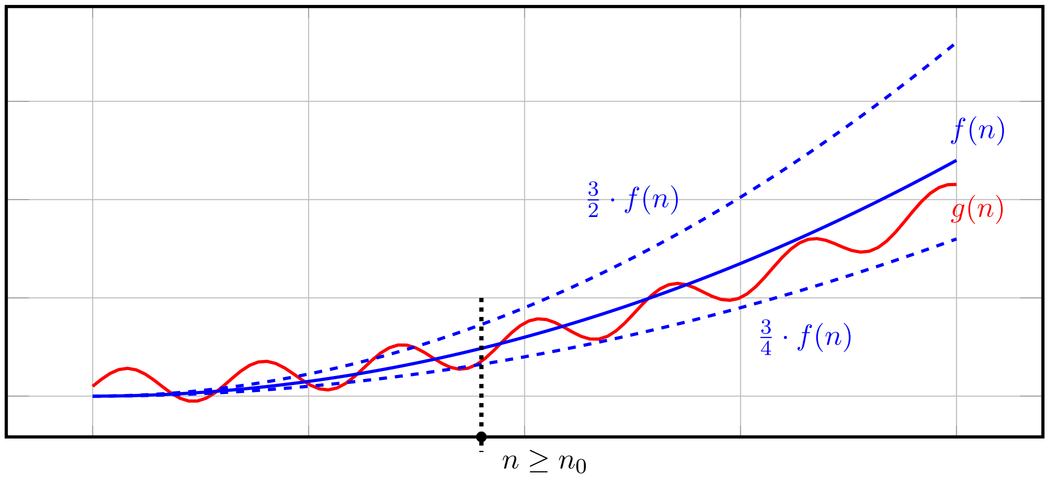

Figure 3: $g(n) = \Theta(f(n))$, since $\frac{3}{4}f(n) \leq g(n) \leq \frac{3}{2} \cdot f(n)$ for $n \geq n_0$.

To finish the discussion of estimations using the O-notation we give a list of some functions that might occur as $f(n)$ in such bounds. They are given in increasing order (column by column). Recall that $\log n$ means $\log_2 n$.

$\log n$	$n \log n$	$n^{\log n}$
$(\log n)^2$	$n(\log n)^2$	$2^{0.67n}$
$\sqrt{n}$	$n^{1.1}$	2^n
$n^{2/3}$	$n^{3/2}$	3^n
$\frac{n}{\log n}$	n^2	$n!$
n	n^5	n^n

First column: “sublinear” and “linear” growth. Second column: “polynomial” growth. Third column: “superlinear” and “exponential” growth.

A closing example: We have seen that **fib2a** calculates F_n in time $O(n^2)$, more exactly in $0.00542n^2$ additions. Exercise 0.4 in the book describes an algorithm **fib3** that calculates F_n in time $O(M(0.694n))$, where $M(\ell)$ is the time for *multiplying* two ℓ-bit numbers. We shall see later that some algorithm gives us $M(\ell) = O(\ell^{1.59})$.[5] Assume **fib3** takes $\leq 10n^{1.59}$ single word additions to calculate F_n. Then it is clear that **fib2a** will be faster than **fib3** for smaller n, but finally, for sufficiently large n,

[5]And this is not even the best exponent known. Actually, in 2019 mathematicians proved that there is a multiplication algorithm with running time $O(\ell \log \ell)$ on ℓ-bit integers. Unfortunately, this algorithm is not practically useful, as yet.

the algorithm with the "'asmptotically better"' running time $O(n^{1.59})$ will take over. Can you calculate what the crossover point is?

(Answer: $10n^{1.59} \leq 0.00542n^2$ means $n^{0.41} \geq 10/0.00542 \approx 1845$. This means $n \geq 1845^{1/0.41} \approx 9.25 \cdot 10^7$, or 92.5 million.)