Complexity of Programs

1 Mathematical Preliminaries

See also the video lecture recording: Mathematical Preliminaries on Canvas.

1.1 Laws of probability

The probability of an event A is written $\mathbb{P}(A)$. It is an element of $\{x \in \mathbb{R} \mid 0 \leqslant x \leqslant 1\}$.

The basic laws of probability are as follows:

- An impossible event has probability 0.
- A certain event has probability 1.
- For an event A, we have $\mathbb{P}(\text{not } A) = 1 - \mathbb{P}(A)$. For example, suppose the probability that it's raining is $\frac{1}{3}$. Then the probability that it's not raining is $\frac{2}{3}$.
- For events A and B that are mutually exclusive, we have $\mathbb{P}(A \text{ or } B) = \mathbb{P}(A) + \mathbb{P}(B)$. For example, suppose the probability that it's raining is $\frac{1}{3}$ and the probability that it's sunny is $\frac{1}{5}$. If these are mutually exclusive events, then the probability that it's either raining or sunny is $\frac{8}{15}$.
- For events A and B that are independent, we have $\mathbb{P}(A \text{ and } B) = \mathbb{P}(A) \times \mathbb{P}(B)$. For example, suppose the probability that it's raining is $\frac{1}{3}$ and the probability that John is happy is $\frac{1}{5}$. If these are independent events, then the probability that it's raining and John is happy is $\frac{1}{15}$.

1.2 Important summations

Here are some summations that come up again and again, so make sure you know them.

$$\begin{aligned} 0 + 1 + 2 + \cdots + (n-1) &= \frac{1}{2}n(n-1) \\ 1 + 2 + 3 + \cdots + n &= \frac{1}{2}n(n+1) \\ 1 + b + b^2 + \cdots + b^{n-1} &= \frac{b^n - 1}{b - 1} \qquad (b \neq 1) \\ 1 + b + b^2 + \cdots + b^n &= \frac{b^{n+1} - 1}{b - 1} \qquad (b \neq 1) \end{aligned}$$

1.3 Upper and lower bounds

- It will take me at least an afternoon to clear my office. *Lower bound.*
- Clearing the office will take me a week at most. *Upper bound.*
- Building the new railway will cost no more than 70 billion pounds. *Upper bound.*
- For the café to be viable, we need at least 30 customers a day, maybe more. *Lower bound.*

Note that an upper bound gives a guarantee.

2 Running time of a program

See also the video lecture recording: Running Time of a Program on Canvas.

2.1 Factors influencing the running time of a program

There are different factors that influence the running time of a program:

- processing power of computer
- algorithm used
- size of the input
- structure of the input
- amount of available memory
- implementation language

We will, however, keep our analysis simple without considering any hardware, implementation or input structure details.

2.2 Best, average and worst cases

Consider a sample problem, e.g. sorting (arranging items in order)

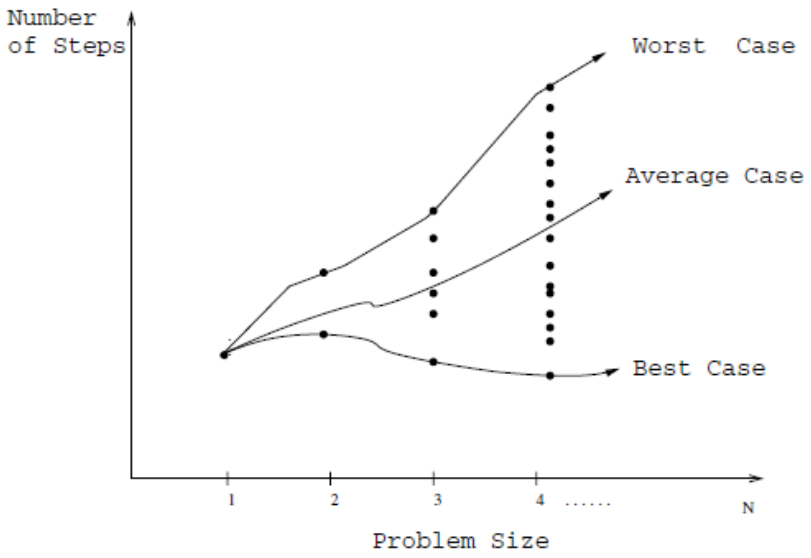


Figure 2.1: Best, worst, and average-case complexity

The above plot illustrates three functions:

- **Worst-case complexity**: It gives us an *upper bound* on the cost. It is determined by the *most difficult* input and provides a guarantee for all inputs.
- **Best-case complexity**: It gives us a *lower bound* on the cost. It is determined by the *easiest input* and provides a goal for all inputs.
- **Average-case complexity**: It gives us the *expected cost* for a random input. It requires a model for *random input* and provides a way to predict performance.

We will mainly focus input size and average/worst case complexity analysis.

Lets consider the following program that operates on an array of characters that are all `a` or `b` or `c`.

```
void f (char[] p) {
  elapse(1 second);
  for (nat i = 0; i<p.length(), i++) {
    if (p[i]=='a') {
      elapse(1 second);
    } else {
      elapse(2 seconds);
    }
    elapse (1 second);
  }
}
```

For an array of length 0, the running time is 1 second.
For an array of length 1, assuming `a`, `b`, `c` are equally likely:

Array contents	Probability	Time	Contribution
`a`	$\frac{1}{3}$	$3s$	$1s$
`b`	$\frac{1}{3}$	$4s$	$1\frac{1}{3}s$
`c`	$\frac{1}{3}$	$4s$	$1\frac{1}{3}s$
Worst case: `b`		$4s$	
Average case			$3\frac{2}{3}s$

For an array of length 2, assuming `a`, `b`, `c` are equally likely and the characters are independent:

Array contents	Probability	Time	Contribution
`aa`	$\frac{1}{9}$	$5s$	$\frac{5}{9}s$
`ab`	$\frac{1}{9}$	$6s$	$\frac{2}{3}s$
`ac`	$\frac{1}{9}$	$6s$	$\frac{2}{3}s$
`ba`	$\frac{1}{9}$	$6s$	$\frac{2}{3}s$
`bb`	$\frac{1}{9}$	$7s$	$\frac{7}{9}s$
`bc`	$\frac{1}{9}$	$7s$	$\frac{7}{9}s$
`ca`	$\frac{1}{9}$	$6s$	$\frac{2}{3}s$
`cb`	$\frac{1}{9}$	$7s$	$\frac{7}{9}s$
`cc`	$\frac{1}{9}$	$7s$	$\frac{7}{9}s$
Worst case: `bb`		$7s$	
Average case			$6\frac{1}{3}s$

Now consider an array of length n.

1. When does the worst case arise? (Just give one example.) What is its running time?

2. Assuming `a`, `b`, `c` are equally likely and the characters are independent, what is the average case running time?

Consider another example:

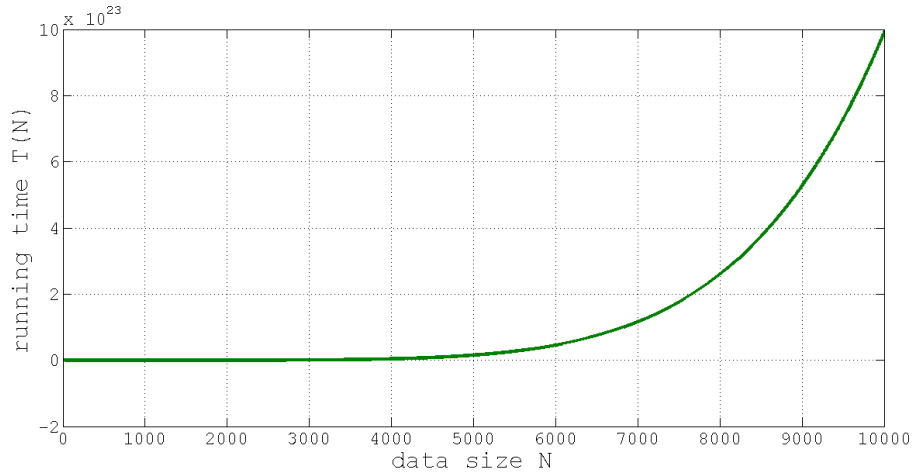
```
void g (char[] p){
  elapse(8 seconds);
  for (nat i=0; i<p.length(); i++){
    elapse(5 seconds);
    for (nat j=i; j<p.length(); j++){
      elapse(2 seconds);
    }
  }
}
```

3. What's the running time of `g` for an array of length 4?
4. What's the running time of `g` for an array of length n?

For this program, the running time for an array of given length is always the same.

2.3 Running time in terms of argument size

Running time is always expressed in terms of the *size* of the argument. Computer scientists use different definitions of size in different settings, but in this module, we always treat the argument as a word over Σ and its size is its length. (Remember that Σ is a finite set of size at least 2.) The following plot shows an example of running time $T(N)$ vs. input size N.



For example, suppose we are studying the problem of sorting a list of natural numbers. Some computer scientists would take the size to be the length of the list, and treat comparison of two numbers as a single step, ignoring the fact that comparison of large numbers takes longer. But we shall take $\Sigma = \{0, 1, [,], ,\}$, and then, for example, represent the list $[5, 2, 6]$ as the word "[101,10,110]" of length 12. Alternatively, we can use base ten; but what we cannot do is to treat each natural number as a single character, because the alphabet must be finite.

Here are two widely used algorithms that are fast in the average case but slow in the worst case.

- Quicksort, for sorting an array.
- The simplex algorithm, for solving an optimization problem called "linear programming". In the worst case it is exponential (i.e. very slow), yet it is efficient in practice.

Which is more important, the worst case or the average case? Usually it is the average case that is more important; an occasional slow run doesn't matter so much if the program runs fast on average. Nevertheless it's certainly better if you can guarantee that your program always runs quickly. And there are situations where a slow run would be catastrophic.

3 What do we care about?

See also the video lecture recording: What do we care about? on Canvas.

We don't usually work out the precise running time of a program; indeed we don't usually have the information needed to do so. Instead we work out the *complexity*, which is a kind of rough estimate of the running time that ignores two things: small arguments and constant factors.

To help us grasp the idea, let's meet four people, with different opinions on the subject of running time.

- **Little Tim** is the most discriminating. He cares about the time taken for inputs of all sizes, even small inputs.
- **Big Tim** is less discriminating than Little Tim. He cares only about the time taken for big inputs.
- **Constance** is less discriminating than Big Tim. She cares only about the time taken (for big inputs) up to a constant factor.
- **Polly** is the least discriminating. She cares only whether the time taken (for big inputs) is polynomial in the size of the input.

3.1 Small arguments

Look at the following three programs:

```
void g (char[] p){
  elapse (8 seconds);
  for (nat i=0; i<p.length(); i++){
    elapse (5 seconds);
    for (nat j=i; j<p.length(); j++){
      elapse (2 seconds);
    }
  }
}

void g2(char[] p){
  if (p.length()<1000) {
    elapse(1000000 seconds);
  } else {
    elapse (8 seconds);
    for (nat i=0; i<p.length(); i++){
      elapse (5 seconds);
      for (nat j=i; j<p.length(); j++){
        elapse (2 seconds);
      }
    }
  }
}

void g3 (char[] p){
  if (p.length()<1000) {
    elapse(1 second);
  } else {
    elapse (8 seconds);
    for (nat i=0; i<p.length(); i++){
```

```
      elapse (5 seconds);
      for (nat j=i; j<p.length(); j++){
        elapse (2 seconds);
      }
    }
  }
}
```

On arrays of size < 1000, the programs have very different running times. But since our alphabet is $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, there are only finitely many such arrays, specifically $\frac{1}{2}(3^{1000} - 1)$ of them. On all other arrays, the three programs have the same running time.

Little Tim regards `g3` as better than `g`, and `g2` as worse than `g`, but Big Tim (also Constance and Polly) regards them all as equivalent.

3.2 Constant factors

Look at the following three programs:

```
void g (char[] p){
  elapse (8 seconds);
  for (nat i=0; i<p.length(); i++){
    elapse (5 seconds);
    for (nat j=i; j<p.length(); j++){
      elapse (2 seconds);
    }
  }
}

void g4 (char[] p){
  elapse (8000 seconds);
  for (nat i=0; i<p.length(); i++){
    elapse (5000 seconds);
    for (nat j=i; j<p.length(); j++){
      elapse (2000 seconds);
    }
  }
}

void g5 (char[] p){
  elapse (0.008 seconds);
  for (nat i=0; i<p.length(); i++){
    elapse (0.005 seconds);
    for (nat j=i; j<p.length(); j++){
      elapse (0.002 seconds);
    }
  }
}
```

The running times of these programs differ by a constant factor: `g4` is a thousand times slower than `g`, and `g5` is a thousand times faster. So Big Tim regards `g5` as better than `g`, and `g4` as worse than `g`, but Constance (and also Polly) regards them all as equivalent.

3.3 Steps

Now consider the following code:

```
void g6 (char[] p){
  elapse (8 steps);
  for (nat i=0; i<p.length(); i++){
    elapse (5 steps);
    for (nat j=i; j<p.length(); j++){
      elapse (2 steps);
    }
  }
}
```

Constance regards `g6` as equivalent to `g`. Whether a step is a second, 1000 seconds or 0.001 seconds doesn't matter to her. All that matters is that a step is a fixed length of time.

Usually in complexity theory, we follow the opinion of Constance, which allows us to give the running time in steps, not seconds. This is helpful, because we can often look at a program and estimate the number of steps by making some reasonable assumptions. You will see this when studying Algorithms.

4 Big Tim examples

Let's say we have two programs. The running time (in seconds) of Program 1A on all inputs of size $n > 0$ is

$$f(n) \;=\; 12n^3 + 300n^2 - 29n + 4$$

The running time (in seconds) of Program 1B on all inputs of size $n > 0$ is

$$g(n) \;=\; 12.01n^3 - n^2 + 7n - 5$$

Which is preferable? Little Tim points out that $f(1) = 287$ and $g(1) = 13.01$, so there are some inputs for which Program 1B is faster. But for Big Tim, who doesn't care about small inputs, it's Program 1A that is faster. He says: "When the input is large, only the term of highest degree matters, and $12n^3 < 12.01n^3$."

Let's make Big Tim's argument precise. We want to show that, for sufficiently large n, we have

$$f(n) \;<\; g(n)$$

Let's note that we have

$$\begin{aligned} f(n) &\leqslant 12n^3 + 300n^2 + 4 \\ &\leqslant 12n^3 + 300n^2 + 4n^2 \\ &= 12n^3 + 304n^2 \\ \text{and } g(n) &\geqslant 12.01n^3 - n^2 - 5 \\ &\geqslant 12.01n^3 - n^2 - 5n^2 \\ &= 12.01n^3 - 6n^2 \end{aligned}$$

So we have $f(n) < g(n)$ if we have

$$\begin{aligned} 12n^3 + 304n^2 &< 12.01n^3 - 6n^2 \\ \Leftrightarrow \qquad 310n^2 &< 0.01n^3 \\ \Leftrightarrow \qquad 31000n^2 &< n^3 \\ \Leftrightarrow \qquad n &> 31000 \end{aligned}$$

And the alphabet is finite, so there are only finitely many inputs of size $\leqslant 31000$.

The same argument works for any two polynomials: all that matters for sufficiently large inputs is the term of largest degree, just as Big Tim said.

Now let's say that Program 2A has running time (in seconds) of

$$h(n) \quad = \quad n^2 + 17n + 2$$

and Program 2B has running time (in seconds) of 1.05^n. Big Tim says that Program 2A is faster, because "for large n, exponential is larger than polynomial".

Let's make this argument precise. The binomial theorem tells us that

$$\begin{aligned} 1.05^n &= 1 + n \times 0.05 + \frac{n(n-1)}{2!} \times 0.05^2 + \frac{n(n-1)(n-2)}{3!} \times 0.05^3 + \cdots \\ &\geqslant 1 + n \times 0.05 + \frac{n(n-1)}{2!} \times 0.05^2 + \frac{n(n-1)(n-2)}{3!} \times 0.05^3 \end{aligned}$$

which is cubic and therefore $> f(n)$ for sufficiently large n, as we saw before. In summary, an exponential function is always larger than a polynomial function for sufficiently large inputs, just as Big Tim said.

Now recall that $n^{0.001}$ is the thousandth root of n. Let's say that the running time (in seconds) of Program 2C is $1.05^{(n^{0.001})}$. So Program 2C is faster than Program 2B, but is it slower than Program 2A? Yes it is. To see this, put $m = n^{0.001}$. Then $f(n)$ is polynomial in m, whereas $1.05^{(n^{0.001})}$ is exponential in m, and therefore $f(n) < 1.05^{(n^{0.001})}$ for sufficiently large m. So this is also true for sufficiently large n.

Now let's say that the running time (in seconds) of Program 3A is $\log_{1.05} n$, and that of Program 3B is $n^{0.001}$. Big Tim says that Program 3A is faster because "Logarithmic is always less than polynomial". To make his argument precise, note that the statement $\log_{1.05} n < 0.001$ is equivalent to the statement $n < 1.05^{(n^{0.001})}$, which we've already seen is true for sufficiently large n.

To sum up, here are Big Tim's slogans:

- For polynomials, it's only the term of highest degree that matters.
- An exponential function is larger than every polynomial.
- A logarithmic function is smaller than every polynomial.
- All this is on the assumption that the input is sufficiently large.

5 Time Complexity using Big-O Notation

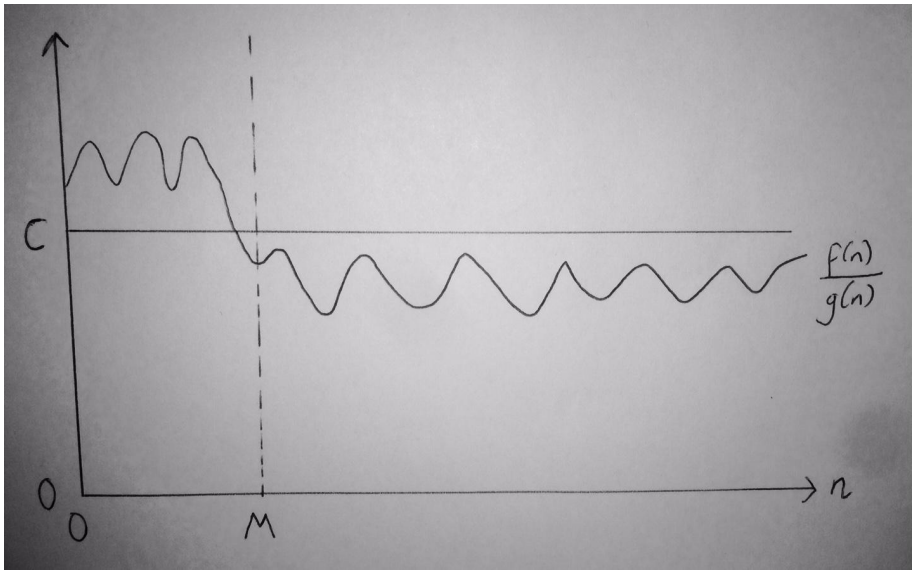
See also the video lecture recording: Big-O Notation on Canvas.

Now we are ready for the most important definition in complexity theory, **Big O** notation. Let f be a function from $\mathbb{N}$ to the positive reals. (Think of $f(n)$ as the running time for an argument of size n. It could be worst case or it could be average case.) Let g be another such function. We say that $f \in O(g)$, or more informally that $f(n)$ is $O(g(n))$, when the following condition holds: there is a natural number M and constant factor C, such that for all $n \geqslant M$, we have $\frac{f(n)}{g(n)} \leqslant C$.

In logical symbols:

$$\exists M.\, \exists C.\, \forall n \geqslant M.\, \frac{f(n)}{g(n)} \leqslant C$$

A helpful slogan to remember is "Big O means proportional or less". Here is a picture:



Note: In many cases $f(n)$ or $g(n)$ may be undefined or negative or zero (contrary to what I said), but only for small n. An example is $g(n) = \log_2 \log_2 n$, which is undefined for $n = 0$ and $n = 1$ and zero for $n = 2$ but positive for all $n \geqslant 3$. This is not a problem because, provided we take $M \geqslant 3$, the inequality makes sense.

Tip: If you want to compare two functions f and g for complexity, try dividing $f(n)$ by $g(n)$ and see what happens as n gets large.

5.1 Polynomial time

We have seen that in complexity theory we ignore small arguments and constant factors, to get a rough estimate of running time. But people like Polly take this further and ask just one question: is the running time polynomial? That is a *very* basic requirement of a program: a polynomial time program might be slow, but a program that isn't polynomial time is regarded as utterly infeasible.

Definition. Let the running time of a program be given by a function from $\mathbb{N}$ to the positive reals. (This could be worst case or average case.) The program is *polynomial time* when there is k such that $f(n)$ is $O(n^k)$. In detail, it is polynomial time when there is k and M and C such that if $n \geqslant M$ then $f(n) \leqslant C \times n^k$.

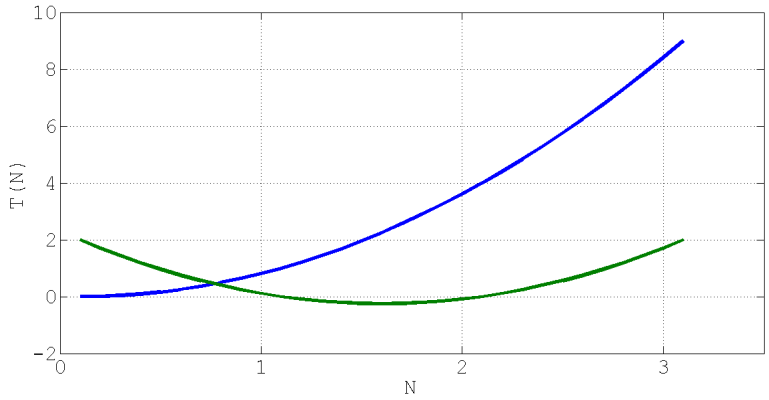
In 2002, Agrawal, Kayal and Saxena published a polynomial time algorithm for testing whether a number p is prime. Its running time is in $O(n^{12})$, where n is the length of p. This was surprising; people had previously suspected that no such algorithm existed. The result has since been improved to an algorithm whose running time is $O(n^6)$.

5.2 Algorithm analysis: quadratic growth

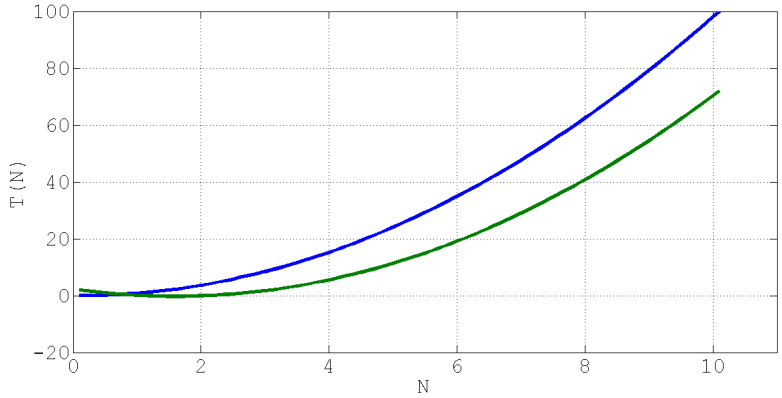
Consider the following two functions:

$$f(n) = n^2 - 3n + 2$$
$$g(n) = n^2$$

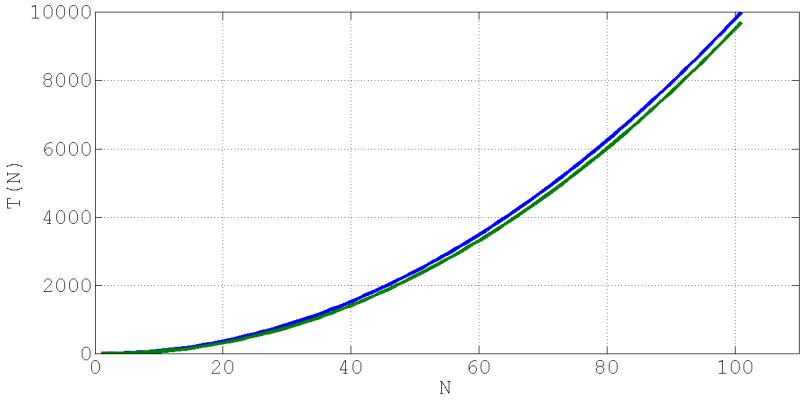
Around n = 3, they look very different



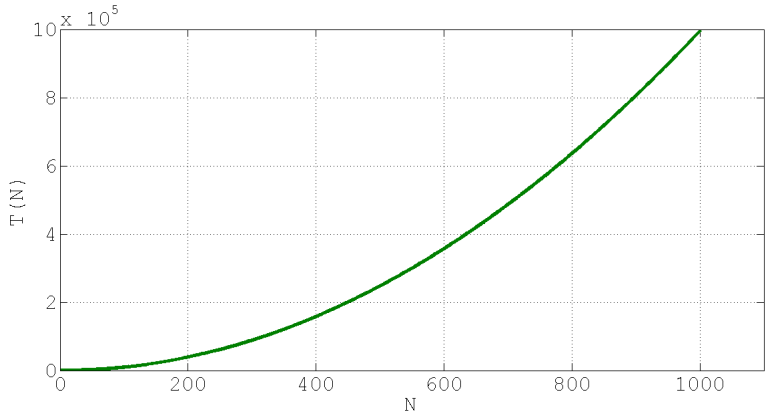
Around n = 10, they look similar.



Around n = 100, they are almost same.



Around n = 1000, the difference is indistinguishable!



The absolute difference is large, such that,

$$f(1000) = 997002$$

$$g(1000) = 1000000$$

but the relative difference is very small,

$$\left| \frac{g(1000) - f(1000)}{g(1000)} \right| = 0.002998 < 0.3\%$$

and this difference goes to zero as $n \to \infty$.

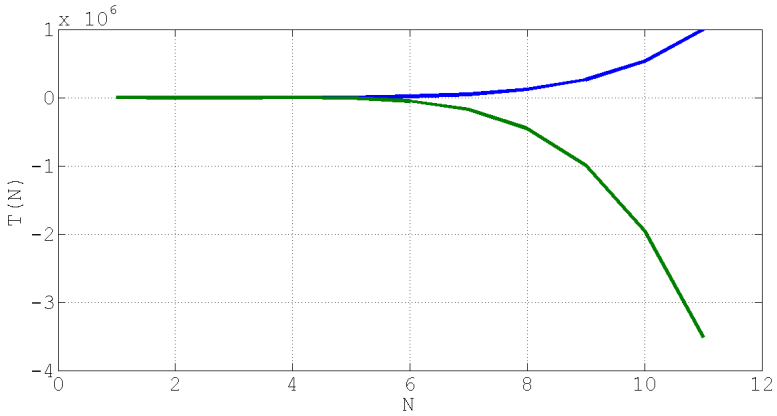
5.3 Algorithm analysis: polynomial growth

To demonstrate with another example,

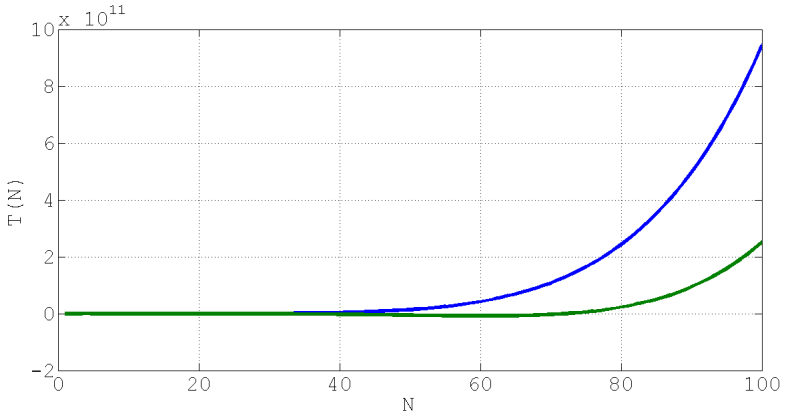
$$f(n) = n^6 - 76n^5 + 343n^4 - 345n^3 + 45n^2 - 344n$$

$$g(n) = n^6$$

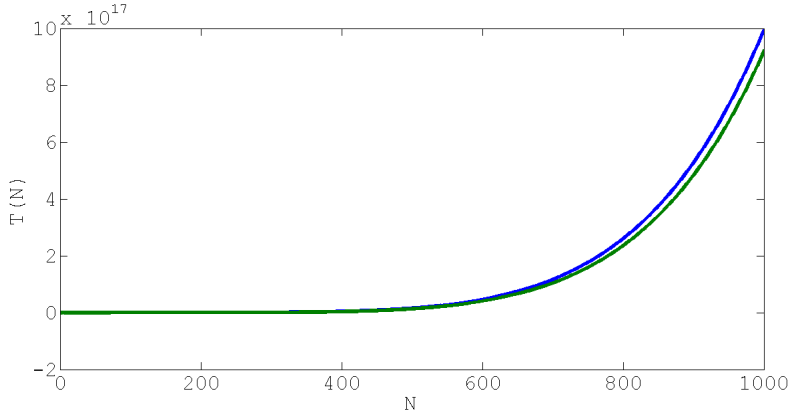
Around n = 10, they are very different:



Around n = 100, they are not similar:



Around n = 1000, they seem similar.



Both function pairs of polynomials are similar, they each had the same leading term:

- n^2 in the first case (quadratic)
- n^6 in the second case (polynomial)

In each case, both functions exhibit the same rate of growth, one is always proportionally larger than the other.

5.4 More on Big-O notation

Usually we don't need exact complexity T(n) of a given program. Generally, it suffices to know the complexity class of the program / algorithm and we ignore constant factors/overheads and lower order terms. Our focus is on performance for large input sizes (also known as "asymptotic analysis"). For example:

- $T(n) = n \Rightarrow$ complexity class $= O(n)$
- $T(n) = n + 2 \Rightarrow$ complexity class $= O(n)$
- $T(n) = 2n^2 \Rightarrow$ complexity class $= O(n^2)$
- $T(n) = n^2 - 3n + 2 \Rightarrow$ complexity class $= O(n^2)$
- $T(n) = 10n^3 + 1 \Rightarrow$ complexity class $= O(n^3)$
- $T(n) = n^6 - 76n^5 + 343n^4 - 345n^3 + 45n^2 - 344n \Rightarrow$ complexity class $= O(n^6)$
- $T(n) = 1000 \Rightarrow$ complexity class $= O(1)$

Determining the complexity class: intuitively, it suffices to count the number of loops and the number of times they are executed. Some common complexity classes are:

- $O(1) =$ "Constant"
- $O(log_2 n) =$ "Logarithmic"
- $O(n) =$ "Linear"
- $O(nlog_2 n) =$ "Log Linear"
- $O(n^2) =$ "Quadratic"
- $O(n^3) =$ "Cubic"
- $O(2^n) =$ "Exponential"

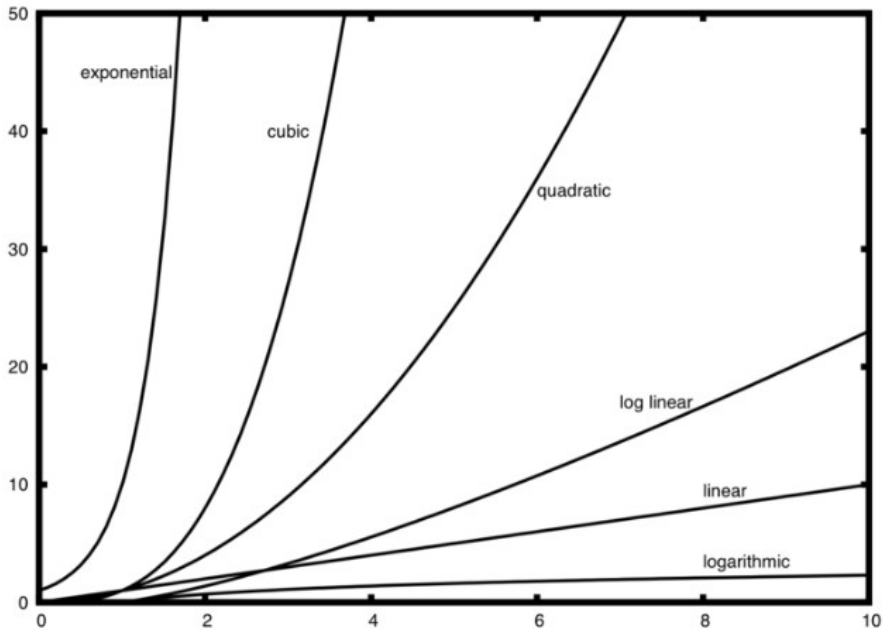
The polynomial complexities $O(n), O(n^2), O(n^3)$ are known as "tractable", indicating that the running time will be large, but it is still manageable. The exponential complexities like $O(2^n), O(C^n)$ are known as "intractable", meaning that the problem is solvable in principle, but the solution requires so much time that they can't be used in practice. To put things into perspective, lets have a look at the following table, which shows how many operations will be needed for an algorithm, with a given complexity and input size n.

f(n)	n = 4	n = 16	n = 256	n = 1024	n = 1048576
1	1	1	1	1.00×10^0	1.00×10^0
$\log_2 \log_2 n$	1	2	3	3.32×10^0	4.32×10^0
$\log_2 n$	2	4	8	1.00×10^1	2.00×10^1
n	4	16	256	1.02×10^3	1.05×10^6
$n \log_2 n$	8	64	2.05×10^3	1.02×10^4	2.10×10^7
n^2	16	256	6.55×10^4	1.05×10^6	1.10×10^{12}
n^3	64	4.10×10^3	1.68×10^7	1.07×10^9	1.15×10^{18}
2^n	16	6.55×10^4	1.16×10^{77}	1.80×10^{308}	6.74×10^{315652}

How much time will be needed for such an algorithm, assuming 1 million operations per second. The following table gives you some idea:

f(n)	n = 4	n = 16	n = 256	n = 1024	n = 1048576
1	1 usec	1 usec	1 usec	1 usec	1 usec
$\log_2 \log_2 n$	1 usec	2 usec	3 usec	3.32 usec	4.32 usec
$\log_2 n$	2 usec	4 usec	8 usec	10 usec	20 usec
n	4 usec	16 usec	256 usec	1.02 msec	1.05 sec
$n \log_2 n$	8 usec	64 usec	2.05 msec	10.2 msec	21 sec
n^2	16 usec	256 usec	65.5 msec	1.05 sec	12.72 days
n^3	64 usec	4.1 msec	16.8 sec	17.83 min	36559 yrs
2^n	16 usec	65.5 msec	3.7×10^{63} yrs	5.7×10^{294} yrs	2.14×10^{315639} yrs

In general, the following graph shows the behavior of different complexities:



5.5 How to determine the complexity of a program?

The total complexity of a program can be determined from the complexities of its components. We usually have the following two types of program compenents:

1. Sequential algorithm phases
2. Function / Method calls

Lets consider the following matrix-vector multiplication ($x = Ab$) example:

```
for i=0...n-1:
   x[i] = 0
for i=0...n-1:
   for j=0...n-1:
      x[i] = x[i] + A[i][j] * b[j]
```

We can see that it contains three *for* loops, where the second and third *for* loops are nested, therefore will be considered together. Overall, the program has two sets of loops, first one and the remaining two loops, with Big-O complexities of $O(n)$ and $O(n^2)$, respectively. In this case, we will consider these two sets of loops as *sequential* phases of the program, therefore we will take the *maximum* of the two complexities, which will be $O(n^2)$.

Lets consider another example, where we perform n array look-ups using the binary search algorithm.

```
for i=0...n-1:
   binary_search(x,v_i)
```

We *know* that the complexity of binary search is $O(log_2 n)$. In this case, the binary search function is being invoked from within the *for* loop, therefore, we will *multiply* the complexities of *for* loop and binary search function. Thus we will obtain the overall complexity as $O(n) \times O(log_2 n) \Rightarrow O(nlog_2 n)$. To summarize, the complexity of sequential algorithm/program phases is the *maximum* of all such phases and the complexity of function / method calls / nested structures is computed by *multiplication*.

Here is an interesting link for you to explore: Sorting Algorithms Animations. We are not asking you to understand these algorithms, rather as an example to show how different algorithms behave.

5.6 Test Your Understanding

1. The running time of my program, on an argument of size n, is $3n^2 + 9n + 8$. Is this $O(n^2)$? Is it $O(n)$? Is it $O(n^3)$?
2. The running time of my program, on an argument of size n, is 5^n for $n < 1000$, and $3n^2 + 9n + 8$ for $n \geqslant 1000$. Is this $O(n^2)$? Is it $O(n)$? Is it $O(n^3)$?
3. On an argument of size n, I first run a program whose running time is in $O(n^2)$, and then run a program whose running time is in $O(n^3)$. Show that the total running time is in $O(n^3)$.
4. Consider you have two algorithms to solve a given problem. The first algorithm has a running time of $3n^2 + 2n + 33$ while the second algorithm has a running time of $2^n - 5n + 5$. Which one will you prefer and why?

6 Space Complexity

We can study the space (memory) usage of a program in a similar way. For example, suppose my program, on an argument of size n, uses $8n^2 + 5$ bytes of memory. We then say that the space usage is quadratic.