Complexity of programs and Introduction to Turing Machines

▼ ***Basic probabilities and proofs***

Warning:

- If we take a real number uniformly at random between 0 and 1, $\frac{1}{2}$ $P(\frac{1}{2}) = 0$
- This is because there are infinite numbers between 0 and 1 $\therefore \frac{1}{\infty} \rightarrow 0$

Complexity of Programs

1 Mathematical Preliminaries

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.

Important summations to know:

- $0 + 1 + 2 + \cdots + (n-1) = \frac{1}{2}n(n-1)$
- $1 + 2 + 3 + \cdots + n = \frac{1}{2}n(n+1)$

- $1 + b + b^2 + \cdots + b^{n-1} = \frac{b^n - 1}{b-1} \quad (b \neq 1)$
- $1 + b + b^2 + \cdots + b^n = \frac{b^{n+1} - 1}{b-1} b \quad (b \neq 1)$

Proof:

$$S = 1 + b + b^2 + \cdots + b^{n-1}$$

Multiply both sides by b:

$$bS = b + b^2 + b^3 + \cdots + b^n$$

Now subtract S from bS:

$$bS - S = (b + b^2 + \cdots + b^n) - (1 + b + b^2 + \cdots + b^{n-1})$$

This simplifies to:

$$(b-1)S = b^n - 1$$

Solving for S:

$$S = \frac{b^n - 1}{b-1}, \quad \text{for } b \neq 1$$

$$S = 1 + b + b^2 + \cdots + b^n$$

This is just the previous formula plus b^n, so we can use the previous result:

$$S = \text{previous formula} + b^n = \frac{b^n - 1}{b-1} + b^n$$

Rewriting b^n in terms of the fraction:

$$S = \frac{b^n - 1}{b-1} + \frac{b^n(b-1)}{b-1}$$

Simplifying:

$$S = \frac{b^n - 1 + b^{n+1} - b^n}{b-1} = \frac{b^{n+1} - 1}{b-1}$$

▼ ***Running times of programs***

2 Running time of a program

2.1 Best, average and worst cases

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

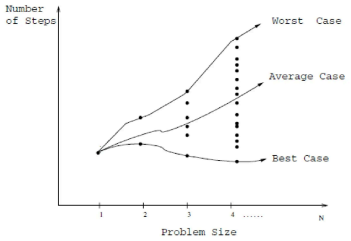


Figure 2.1: Best, worst, and average-case complexity

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 on worst/average case complexity analysis.

Let's consider the following program that operates on an array of character 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	Probability × time
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	Probability × time
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			$6\frac{1}{3}s$

1. ***When does the worst case arise? (Just give one example), what is its running time?***
    a. $b^n/c^n = (3n+1)s$
2. ***Assuming a, b, c are equally likely and the characters are independent, what is the average case running time?***
    a. Average case:
        i. Assuming each character is sampled for a uniform distribution and they're independent
            1. For $n = 0$:
                a. Time: 1
            2. For $n = 1$:
                a. Time = $3\frac{2}{3} = \frac{11}{3}$
            3. For $n = 2$:
                a. Time = $6\frac{1}{3} = \frac{19}{3}$
            4. Recognising pattern
                a. $1, \frac{11}{3}, \frac{19}{3}$
                b. Adding $2\frac{2}{3}$ each time
            5. Assuming $2\frac{2}{3}n$ is the formula, reinput back into $n = 0, 1, 2$

6. You get $0, 2\frac{2}{3}, 5\frac{1}{3}$
7. If we do the pattern minus our subtituted answers
    a. $\frac{3}{3}, \frac{3}{3}, \frac{3}{3} = 1, 1, 1$
8. Therefore if there is a difference of 1 between every subtitued answer, the formula should be modified.
9. $\therefore$
10. We take $2\frac{2}{3}n + 1$ seconds

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.

2.2 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.)

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. $8 + [5 + 4 * 2] + [5 + 3 * 2] + [5 + 2 * 2] + [5 + 1 * 2] = 50$
    a. So the first time $i = 0$ therefore the inner loop is the same structure as the outside loop so also runs 2 seconds 4 times.
    b. The following iterations the $i\ value$ increases by one so the number of 2 seconds the inner loop runs decreases by 1
4. $8 + (5 + 2n) + (5 + 2(n - 1)) + \cdots + (5 + 2 + 1)$
   $= 8 + 5n + 2[\sum_{i=1}^{n} n]$
   $= 8 + 5n + 2(\frac{1}{2}n(n + 1) =$
   $8 + 5n + n^2 + n =$
   $n^2 + 6n + 8$

💡 **Be careful sorting a list of natural numbers**

Can use $\{0, 1, [,], ,\}$ writing k numbers in binary

Size ALWAYS must be over a finite alphabet

Example algorithms:

- Quicksort
  - Sorting algorithm
- Simplex algorithm
  - Optimisation

Whether this is acceptable depends on the situation

▼ ***Intro to Tims, Constance and Polly***

💡 What do we care about?

1. ***We don't usually work out the precise running time of a program***
    a. ***We don't usually have the information needed to do so.***
2. ***Instead we work out the complexity***
    a. ***Which is a kind of rough estimate of the running time that ignores two things:***
        i. ***Small arguments***
        ii. ***Constant factors***

1. ***To help us grasp the idea, let's meet four people, with different opinions on the subject of running time.***
    a. ***Little Tim***
        i. The most discriminating
        ii. He cares about the time taken for inputs of all sizes, even small inputs.
    b. **Big Tim**
        i. Less discriminating than Little Tim
        ii. He cares only about the time taken for big inputs.
    c. **Constance**
        i. Less discriminating than Big Tim.
        ii. She cares only about the time taken (for big inputs) up to a constant factor.
    d. **Polly**
        i. The least discriminating
        ii. She cares only whether the time taken (for big inputs) is polynomial in the size of the input

4 Small arguments

4.1 Basic examples

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++){
```

🔥 Little Tim:

$g3$ better than g

$g2$ worse than g

Big Tim:

all the same

According to Big Tim:

$$2^n > 3.05n^2 > 3n^2 > 3.05n > 3n > \log n$$

Constance

g and $g2$ same

$g3$ better than g and $g2$

Polly

all the same

💡

- On array of $size < 1000$, the programs have very different running times
- Since out alphabet is a, b, c, there are only finitely many such arrays
  - Specifically $\frac{1}{2}(3^{1000} - 1)$ of them, using the summation formula $\frac{1}{2}(2a + (n-1)d)$
- On all other arrays the 3 programs have the same running time

▼ ***Binomial Theorem***

$$(1+x)^n = 1 + nx + \frac{n(n-1)}{2!}x^2 + \frac{n(n-1)(n-2)}{3!}x^3 + \ldots + \frac{n(n-1)(n-2)\ldots(n-r+1)}{r!}x^r + \ldots \infty$$

$$\begin{aligned}(a+b)^3 &= \sum_{k=0}^{3} \binom{3}{k} a^{3-k} b^k \\ &= \binom{3}{0} a^{3-0} b^0 + \binom{3}{1} a^{3-1} b^1 + \binom{3}{2} a^{3-2} b^2 + \binom{3}{3} a^{3-3} b^3 \\ &= 1 \cdot a^3 b^0 + 3 \cdot a^2 b^1 + 3 \cdot a^1 b^2 + 1 \cdot a^0 b^3 \\ &= a^3 + 3a^2 b + 3ab^2 + b^3\end{aligned}$$

4.2 Comparing functions for sufficiently large n

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

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

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

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

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 $> h(n)$ 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 $h(n)$ is polynomial in m, whereas $1.05^{(n^{0.001})}$ is exponential in m, and therefore $h(n) < 1.05^{(n^{0.001})}$ 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 $\log_{1.05} n < n^{0.001}$ 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.

🔥 $1.05^n = 1 + (0.05n) + (\frac{n(n-1)}{2!} * 0.05^2) + (\frac{n(n-1)(n-2)}{3!} * 0.05^3) + \dots$
$\geq 1 + (0.05n) + (\frac{n(n-1)}{2!} * 0.05^2) + (\frac{n(n-1)(n-2)}{3!} * 0.05^3)$

- This is cubic and $\therefore > h(n)$ for sufficiently large n, as seen previously
- 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 it is slower than Program $2A$?

- To see this, put $m = n^{0.001}$
  - Then $h(n)$ is polynomial in m, whereas $1.05^{(n^{0.001})}$ is exponential in m
    - Therefore $h(n) < 1.05^{(n^{0.001})}$ for sufficiently large m

- *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 $\log_{1.05} n < n^{0.001}$ 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 the highest degree that matters*

- *An exponential function is large than every polynomial*
- *A logarithmic function is smaller than every polynomial*
- *All this send on the assumption is sufficiently large*

▼ *Constant factor examples and complexity in Big O Notation*

5 Constant factors

5.1 Basic examples

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.

5.2 Steps

Now consider the following code:

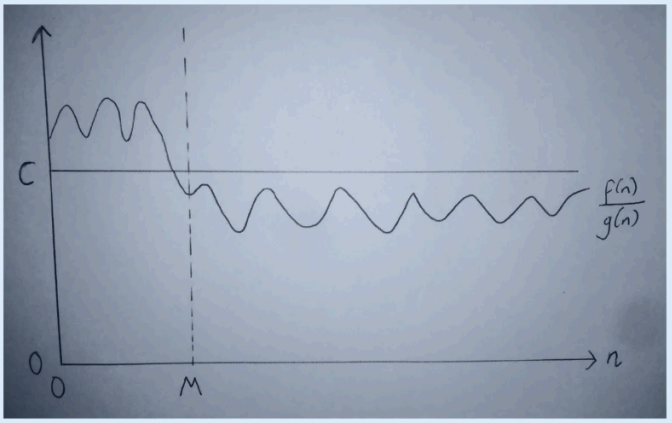
```
void g6 (char[] p){
  elapse (8 steps);
  for (nat i=0; i<p.length(); i++){
    elapse (5 steps);
    for (nat j=i; j<p.length(); j++){
      elapse (2 steps);
    }
  }
}
```

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.

5.3 Time Complexity using Big-O Notation

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 $\frac{f(n)}{g(n)} \leqslant C$.



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.

🔥 ***Ok so $f(n)$ is the running time for an argument of size n***

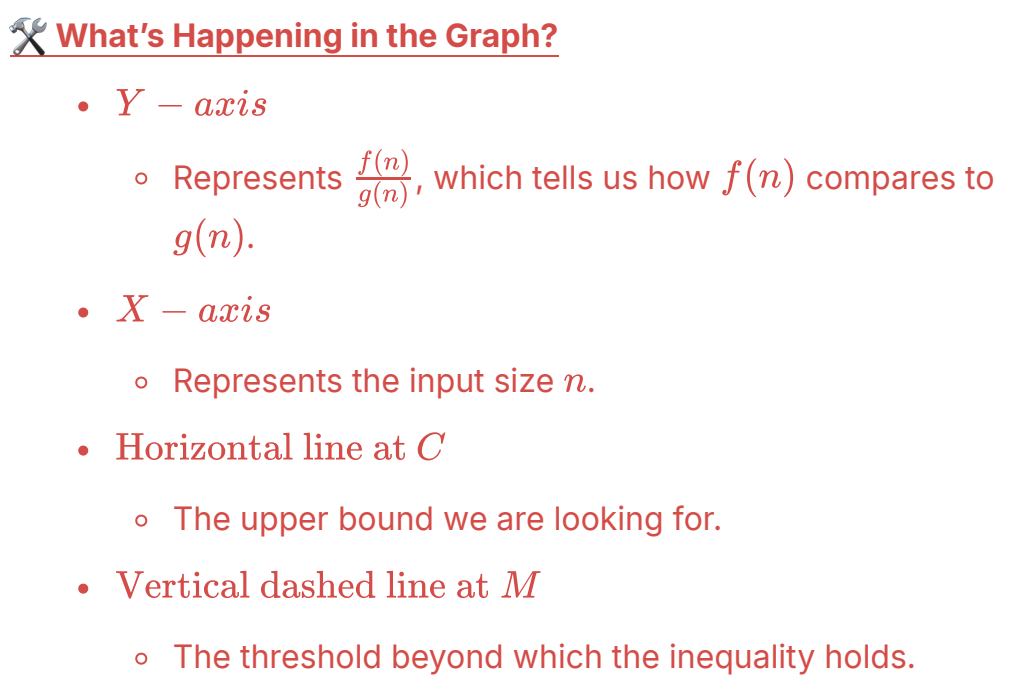
and $g(m)$ is the running time for an argument of size m

Now we say $\{f \in O(g)\}/\{f(n) = O(g(n))\}$

> ***Saying $f(n) = O(g(n))$ means that for large enough n, $f(n)$ is at most some constant multiple of $g(n)$***
>
> ***When $\exists M \in \mathbb{N}.\exists C \in \mathbb{Z}.\forall n \geq, \frac{f(n)}{g(n)} \leq C$***
>
> - ***M : The threshold after which the inequality holds.***
> - ***C: A constant multiplier that ensures $f(n)$ doesn't grow faster than $g(n)$***

A helpful slogan to remember is "Big O means proportional or less"

Here's a picture:



🛠️ **What's Happening in the Graph?**

- $Y - axis$
  - Represents $\frac{f(n)}{g(n)}$, which tells us how $f(n)$ compares to $g(n)$.
- $X - axis$
  - Represents the input size n.
- $\text{Horizontal line at } C$
  - The upper bound we are looking for.
- $\text{Vertical dashed line at } M$
  - The threshold beyond which the inequality holds.

🔥 **Key Observations:**

✅ **For small n (before M):**

- The ratio $\frac{f(n)}{g(n)}$ fluctuates wildly.

- This means the behavior of $f(n)$ is unpredictable or inconsistent in the beginning.

✅ **For large n (after M):**

- The ratio stabilizes below or at the constant C.
- This means $f(n)$ is at most a constant multiple of $g(n)$.

✅ **Big-O ignores small values:**

- The fluctuations before M **don't matter** because Big-O only cares about large n.

5.4 Complexity Notations

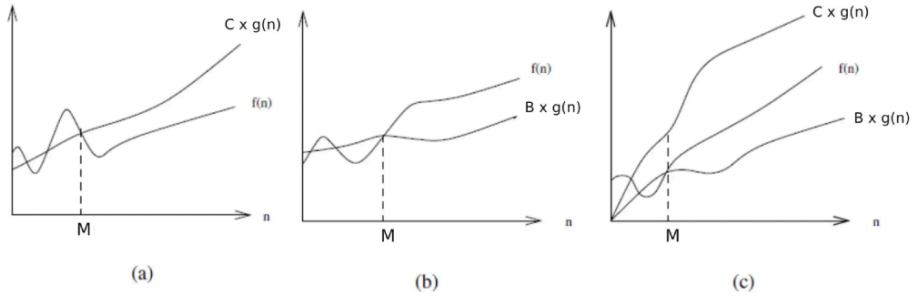
Let f and g be functions from $\mathbb{N}$ to the nonnegative reals.

(a) We say that $f \in O(g)$, or informally "$f(n)$ is $O(g(n))$", when g is an upper bound for f up to a constant factor. That is: there are numbers M and C such that for $n \geqslant M$ we have $f(n) \leqslant C \times g(n)$.

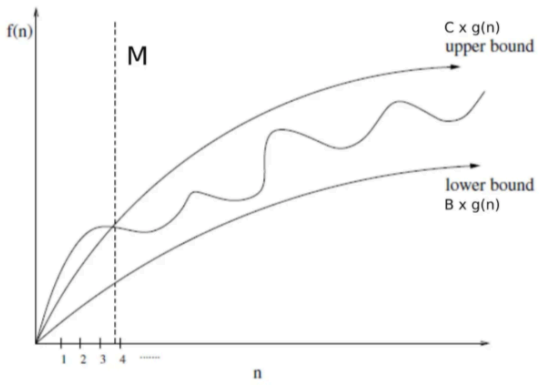
(b) We say that $f \in \Omega(g)$, or informally "$f(n)$ is $\Omega(g(n))$", when g is a lower bound for f up to a constant factor. That is: there are numbers M and $B > 0$ such that for $n \geqslant M$ we have $B \times g(n) \leqslant f(n)$.

(c) We say that $f \in \theta(g)$, or informally "$f(n)$ is $\theta(g(n))$", when both of the above conditions hold. That is: there are numbers M and C and $B > 0$ such that for $n \geqslant M$ we have $B \times g(n) \leqslant f(n) \leqslant C \times g(n)$.

The following figure illustrates the above complexity notations.



With these notations, we can be more precise about complexity. For example, if we say that the worst case running time is $O(n^2)$, it might in fact be linear, but if we know that it is $\theta(n^2)$ then it really is no better than quadratic, because it will be within the lower and upper bounds of complexity. The upper and lower bounds that are valid for $n > M$ smooth-out the behavior of complex functions, we would like to have a tight-bound to ensure our estimations are precise, as shown below:



▼ *Question on polynomial time*

6 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 $O(n^{13})$, 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 $O(n^7)$.

6.1 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 $O(n^3)$.

🔥

1. It's $O(n^2)$, since the most dominant term is $3n^2$
2. It's $O(n^2)$, since Big-O notation only focuses on the result of the function when $x \to \infty$
3. Since it doesn't look like they're dependent on each other in a loop-like format I can form the total running time equation as, $\text{Total running time} = O(n^2) + O(n^3)$. Since Big-O notation only focuses on the result of the function when $x \to \infty$, the $O(n^3)$ dominates, therefore the total running time is $O(n^3)$

7 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.