Contents

1 Basic definitions and maths review

1.1 Algorithms

1.1.1 Definitions

Definition 1 (Algorithm)**.** An *algorithm* is a finite list of instructions, often used to solve a set of problems or perform a computation.

Algorithms are unambiguous procedures for performing a calculation, processing data, automated reasoning and many other tasks.

Algorithms as a concept have existed since at least the ancient Greek mathematicians. For example, the sieve of Eratosthenes for determining which of the first n numbers are prime. The Euclidean algorithm for finding the greatest common divisor of two numbers is an algorithm you are probably already familiar with from earlier courses.

Algorithms are the cornerstone of computer science, without algorithms, computer programs could not exist. As demonstrated in the previous paragraph, algorithms have existed long before computers, although they were mostly expressed informally until relatively recently. Studying algorithm design and development will improve your analytical and problem solving skills, while developing a deeper understanding of algorithms you are already familiar with, and being exposed to algorithms that perform the same job as algorithms you are familiar with more efficiently, and even what it means to be efficient.

Algorithms typically have the following components:

- Input
- Output
- Determinism
- Finiteness
- Correctness

Now we should define these terms so we can talk about them in a way that we all agree upon. We all have some kind of intuitive understanding of each of these terms, but we need to make sure when we say "correctness" we are all working with the same definition.

Definition 2 (Input)**.** The object or objects that an algorithm requires.

Definition 3 (Output)**.** The result of running the algorithm on the inputs.

Definition 4 (Determinism)**.** Given the same input, an algorithm will give the same output. Furthermore, the underlying machine will pass through the same sequence of states.

Deterministic algorithms are the most studied algorithms. Their counterpart, non deterministic algorithms, are much more complex in general, and harder to define. For example, is a random number generator that uses a source of entropy from the physical world deterministic or non deterministic? Sometimes it makes sense to classify algorithms in that class as deterministic, other times it makes sense to classify them as non deterministic.

Definition 5 (Finiteness)**.** The algorithm will terminate after a finite number of steps

Not all algorithms are necessarily finite, for example, an operating system can be thought of as an online algorithm that never terminates. It's not even possible to tell if a given algorithm will terminate in general, which is a famous problem called the *Halting Problem*, which we will mention later in the course. For example, even the following relatively simple algorithm, we don't know for what inputs it terminates.

```
/**
Algorithm Collatz
Input: A positive integer num
Output: Unsure
*/

int Collatz(int num) {
        int steps = 0;

        while(num > 1) {
            if(num % 2 == 0) {
                num /= 2;
                steps++;
            }
            else {
                num = 3 * num + 1;
                steps++;
            }
        }

        return steps;
}
```

Definition 6 (Correctness)**.** Correctness refers to the input-output behaviour of the algorithm, that is, for each input it produces the expected output.

We should note that there is a distinction between *partial correctness*, which requires that if an answer is returned it is correct for the given inputs, and *total correctness* which further requires the algorithm terminates. It is easy to write a partially correct algorithm, see the Collatz algorithm under the finiteness definition. But it is not possible to determine whether that algorithm is a total correct algorithm as we currently don't know if it terminates for every input.

Sometimes we use approximation algorithms as well, which are not correct in the sense that they give the correct answer to the question being modeled. For example, for difficult problems, such as the traveling salesman problem, where it is hard to get the shortest possible Hamiltonian cycle in a graph, we have approximation algorithms which get close to the best possible cycle.

1.1.2 Expressing Algorithms

As with almost everything, we need to communicate effectively when we describe an algorithm to other people. It would not be effective to describe an algorithm to someone but then have them interpret the algorithm in a way that differs to the intended list of instructions.

Representations of algorithms can be classed into three levels of Turing machine description.

1. High level description. That is, prose to describe an algorithm, ignoring implementation details. At this level, we do not need to mention how the machine actually works.

2. Implementation description. That is, we use a programming language to describe the way the machine carries out the algorithm.

3. Formal description. That is, we write out the state function and transition function of the Turing machine.

We don't often use the last level of representation as it is often not useful in practice, but it is important in the understanding of computation, comparison of problems, finding problems in algorithms, and other situations where a low level description is useful.

1.1.3 Expressing Algorithms Example

A simple algorithm that should be familiar is to find the largest number in a list of numbers in random order.

Algorithm 1 (High Level Description:)**.**

1. If there are no numbers in the set, then there is no highest number.
2. Assume the first number in the set is the largest in the set.
3. For each remaining number in the set, if this number is larger than the current largest number, consider this to be the largest number in the set.
4. When there are no numbers left in the set, consider the current largest number to be the largest number in the set.

As we can see, this is all in prose and gets across the general idea of the algorithm, that is, assume the first element is the largest, then compare it to every other element in the set, updating what we think the largest number is until we have finished looking at every number in the set.

Algorithm 2 (Implementation description:)**.**

```
1  /**
2  Algorithm LargestNumber
3  Input: An array of numbers list.
4  Output: The largest number in the array list if list is non empty and Integer.
       MIN_VALUE otherwise.
5  */
6
7  int LargestNumber(int[] list) {
8          if(list != null && list.length == 0){
9              return Integer.MIN_VALUE;
10         }
11         int largest = list[0];
12         for(int i = 0; i < list.length; i++) {
13             if(list[i] > largest) {
14                 largest = list[i];
15             }
16         }
17
18         return largest;
19 }
```

This is clearly a much more precise way to talk about the algorithm than the high level description above. The caveat to this method of presenting an algorithm is the readability by a human, it is much harder to parse what is happening (especially without comments!).

1.1.4 Algorithm Design

In a somewhat cruel twist of fate, there is a general approach to designing an algorithm that solves a problem, or an algorithm for designing an algorithm if you will.

There are four stages to the algorithm design process, and while they are presented in a linear fashion, there is a lot of going back and forth between the four stages. The four stages are as follows:

1. Understand the problem!

2. Design an algorithm
3. Analyse the algorithm
4. Implement the algorithm

For the most part we will be stuck in the first three stages for this course, although some implementation will be required. Lets look at these steps in more detail.

Understanding the problem can come in many forms. It may be obvious to you what the problem is, for example, it is probably fairly obvious what the problem is when you are asked to factor a number. It is generally much more complex than that though, say you want to design an algorithm that inserts an element into an AVL tree, this is much more complex to even understand what that means, so you will have to experiment with AVL trees by pen and paper, read what other people have tried, and any other ideas you can come up with to try to understand the problem.

After you think you understand the problem, try **designing an algorithm** to solve the problem. Generally you'll quickly realise you didn't quite understand the problem and you will have to go back to the first step again. This is normal! Don't be discouraged! To consider our two examples from the previous paragraph again, you'll quickly come to the algorithm of just trying to divide a given number by every number smaller than it to find it's factors. For the AVL tree problem though, you might struggle to really get every aspect included the first time, and you may not even realise it.

Analyse the algorithm can mean many things, how much space does it require? How long does it take to run? We will cover what both of these mean later on in the course, but intuitively, we want algorithms to not use much space, and run quickly. The algorithm you came up with for finding the factors of a number probably has a pretty bad theoretical run time though, but the AVL tree insertion algorithm is probably fairly quick, probably even faster than sorting. So you may not understand the factoring problem as well as you initially thought, so you'll have to go back to the first step again to try and understand it better. You probably also understand the AVL trees better than you initially thought, because it's very likely you've got a fairly fast algorithm.

If we're happy so far with our theoretical algorithm, we should **implement the algorithm** and see how it works in practice. Some algorithms, despite having a good theoretical speed, are bad in practice, and some that seem awful in theory actually aren't that bad practice for the problems you actually want to solve. So if you've got to this point following the examples, you'll want to cry about how slow your factoring algorithm is, despite how easy it seems at first, but you can insert into an AVL tree very quickly, much quicker than factoring, so we need to go back and look at the factoring some more to try and understand why it is so slow, and how to speed it up!

So in summary, algorithm design is an ad hoc process, that generally has four key components to it, which somewhat act like an algorithm.

1.2 Mathematics Review

When talking about algorithms, we will have to talk about various components of them, such as the amount of space they require, how long they take to run, or even to try to develop better algorithms. All of this requires some understanding of maths to talk about in a precise, unambiguous way.

This section should be a review of mathematics you have learned in previous courses, but once again, it is necessary to make sure when communicating that everyone involved is on the same page with the definitions and notation used. This will act as a reference to almost all notation used throughout the course. It is common to not write either $\times$ or $\cdot$ when talking about multiplication, and instead just write ab for $a \times b$.

1.2.1 Theorems

Before beginning our discussion of mathematics, we need to know a few terms that will occur from time to time.

Definition 7 (Theorem)**.** A *theorem* is a statement that has been proven true on the basis of previously established true statements, such as other theorems, lemmas, or corollaries.

A famous example of a theorem is the four colour theorem, that says for any map you can colour the map in such a way that no two regions that share a boarder also share a colour using only four colours. There is a lot of history on this theorem and well worth a read for those interested, but we definitely won't be proving it in this course.

Definition 8 (Lemma)**.** A *lemma* is generally a small "helping theorem" which is typically only stated and proved to aid in proving a much more difficult theorem, generally lemmas can only be applied to the theorem they are designed for.

Despite lemmas being designed to prove a theorem, there are still some important lemmas which are used a lot. One you might have heard of is the handshaking lemma, often called the handshaking theorem, that is how important it is in graph theory proofs.

Definition 9 (Corollary)**.** A *corollary* is a statement that follows with very little proof from another theorem. It is often just a special case of a theorem.

If these three terms all seem basically the same, there is a good reason for that, they all basically are the same! Don't get hung up on what you call something, just name it something to get your meaning across.

Definition 10 (Proof)**.** The *proof* of a theorem is a logical argument for the theorem statement that follows a deductive list of steps that are justified by other theorems, lemmas, and corollaries.

We will have plenty of examples of different proof techniques throughout the course, proofs are the reason we know why algorithms will always work!

Example 1. For example, is $n^2 - n + 41$ always prime when n is a positive integer? If you try small numbers for n, then it certainly seems to be the case. But checking any amount of numbers does not prove anything, it just lends credibility to idea. Take the Collatz algorithm in section 1.1.1, it has been checked for just over the first 10^{20} numbers, that's bigger than a 64 bit integer! But time and again in mathematics, checking the first n numbers is not sufficient and when you get to very big numbers a conjecture may fall apart. A famous example was Skewes's number, roughly 10^{316}, which was used as an upper bound to a problem in number theory, and at the time, was the largest number used in serious mathematics, Skewes showed that a function changed from being positive to negative before 10^{316}, which until Skewes the community assumed it was always positive because it had been for every number checked, up to about 10^{19}. So, in summary, we need to prove theorems!!

1.2.2 Exponents

Being able to work with exponents will be vital in our analysis of run time and space complexity of algorithms.

The following are the laws of exponents.

Theorem 1 (Exponent Laws)**.** Any values in the denominator in the following equations are taken to be non zero.

1. $x^{-n} = \frac{1}{x^n}$.
2. $x^m x^n = x^{m+n}$.
3. $\frac{x^m}{x^n} = x^{m-n}$. This can be seen by combining the first two exponent laws.
4. $(x^m)^n = x^{mn}$.
5. $(xy)^m = x^m y^m$.
6. $\left(\frac{x}{y}\right)^m = \frac{x^m}{y^m}$.
7. $\left(\frac{x}{y}\right)^{-m} = \left(\frac{y}{x}\right)^m$. This can be seen by combining the first and sixth rule.

The following example demonstrates how to use a few of the exponent laws.

Example 2 (Exponent Example)**.** To simplify

$$\frac{x^2 x^3}{x^6}$$

we will start by simplifying the numerator using rule 2, that is, add the exponents together, to get

$$\frac{x^5}{x^6}.$$

We are now at a point where we can use rule 3, that is, subtract the exponents, to get

$$x^{-1}.$$

And finally we can use the first rule to get

$$\frac{x^2 x^3}{x^6} = \frac{1}{x}.$$

1.2.3 Logarithms

A convention often adopted is to just use $\log(x)$ if the base of the log is clear from context, and often it does not even matter what the base of the log is as $\log_a$ and $\log_b$ both grow at the same rate, which can be seen from the change of base formula. A small note, we will often use lg to denote $\log_2$ though, as this base shows up often in computer science.

Theorem 2 (Logarithmic Laws)**.** Unless otherwise stated, the base of the log does not matter for the following, as long as they are consistent.

1. $\log(1) = 0$.
2. $\log_b(b) = 1$.
3. $\log_b(b^x) = x$. This states that the exponential and logarithm cancel each other out.
4. $\log(xy) = \log(x) + \log(y)$.

5. $\log\left(\frac{x}{y}\right) = \log(x) - \log(y)$
6. $\log\left(x^d\right) = d\log(x)$
7. $\log_b(a) = \frac{\log_d(a)}{\log_d(b)}$. This law is often called the change of base formula.

You should notice a lot of overlap between the logarithm laws and the exponential laws. They both convert multiplication to addition, and division to subtraction, for example.

Example 3. We wish to simplify

$$\log(x^2) + \log(x^3) - \log(x^6).$$

First we simplify the addition by using rule 4 to give

$$\log(x^2x^3) - \log(x^6).$$

We now have a subtraction, so we can use rule 5 to give

$$\log\left(\frac{x^2x^3}{x^6}\right).$$

Now we can simplify the term we are taking the logarithm of by using our exponential rules, to give

$$\log(x^{-1}),$$

and finally, bringing the -1 out side of the log we will get

$$\log(x^2) + \log(x^3) - \log(x^6) = -\log(x).$$

1.2.4 Sequences

Definition 11 (Finite Sequence)**.** A finite sequence a is a function from the set $\{1, 2, 3, \ldots, n\}$ to a set X. We denote the ith element in the sequence by a_i or $a[i]$.

Definition 12 (Infinite Sequence)**.** An infinite sequence a is a function from the set $\{1, 2, 3, \ldots\}$ to a set X. We denote the ith element in the sequence by a_i or $a[i]$.

Usually the set X is either $\mathbb{N}$ or $\mathbb{R}$, the natural numbers, and the real numbers respectively. Unlike a set, a sequence can have more than one occurrence of the same value, for example $(1, 1, 1, 1, 1, 1, \ldots)$ is a valid sequence.

An important property of a sequence is *convergence*. If a sequence converges to a particular value, that value is known as the *limit*.

Definition 13 (Informal Limit)**.** A sequence has a limit if the elements of a sequence become closer and closer to some value L, called the limit of the sequence, and they become and remain *arbitrarily* close to L.

While this definition is informal, it also gets across the main idea of a limit, and is generally more useful than the formal definition of a limit, as that is rarely required to be used due to a large number of theorems being built upon the formal definition, abstracting away from the technicalities of it. But for completeness, the formal definition follows.

Definition 14 (Formal Limit)**.** A sequence of real number (a_n) converges to a real number L if, $\forall\varepsilon > 0 \exists N \in \mathbb{N}$ such that $\forall n > N$ we have $|a_n - L| < \varepsilon$.

Theorem 3 (Limit Laws)**.** If (a_n) and (b_n) are convergent sequences, the the following limits exist and can be computed using the following.

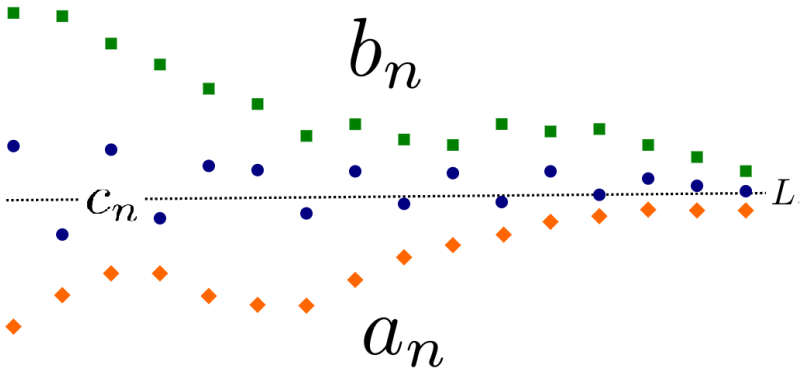
- $\lim_{n\to\infty}(a_n + b_n) = \lim_{n\to\infty} a_n + \lim_{n\to\infty} b_n$.
- $\lim_{n\to\infty} ca_n = c \lim_{n\to\infty} a_n$ for all constant $c \in \mathbb{R}$.
- $\lim_{n\to\infty} a_n b_n = \left(\lim_{n\to\infty} a_n\right)\left(\lim_{n\to\infty} b_n\right)$
- $\lim_{n\to\infty} \frac{a_n}{b_n} = \frac{\lim_{n\to\infty} a_n}{\lim_{n\to\infty} b_n}$ provided that $\lim_{n\to\infty} b_n \neq 0$

The other major theorem for sequences is called the squeeze theorem, if we know that one sequence is always in between two other sequences, and those other sequences go to the same limit, then the sequence in between them also needs to go to that limit.

Theorem 4 (Squeeze Theorem)**.** If (c_n) is a sequence such that $a_n \leq c_n \leq b_n$ for all $n > N$ and $\lim_{n\to\infty} a_n = \lim_{n\to\infty} b_n = L$ then

$$\lim_{n\to\infty} c_n = L.$$

The following picture hopefully demonstrates this idea, the green sequence represents terms in the sequence (a_n), the orange sequence represents terms in the sequence (b_n), and the blue sequence represents terms in the sequence (c_n). Notice that the terms in the blue sequence are always between the green and orange sequence, and the green and orange sequence both go to L, so (c_n) is forced to go to L as well.



Example 4. To give a concrete example of an infinite sequence, the Fibonacci sequence, which will be making an appearance at numerous points in this course is

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

which is generated by adding the two previous terms to get the current term. That is, $F_n = F_{n-1} + F_{n-2}$.

We require a couple of more definitions which are all closely related.

Definition 15 (Increasing/Decreasing Sequences)**.** We give the following names to a sequence if they exhibit the corresponding property for all i.

- *Increasing* if $a_i < a_{i+1}$, that is, it only gets bigger.
- *Decreasing* if $a_i > a_{i+1}$, that is, it only gets smaller.
- *Non increasing* if $a_i \geq a_{i+1}$, that is, it never gets bigger, but can stay the same from one term to the next.

- *Non decreasing* if $a_i \leq a_{i+1}$, that is, it never gets smaller, but can stay the same from one term to the next.

Example 5. The Fibonacci sequence is non decreasing as, except for the second term, every term after it is bigger than the previous, and the second term is the same as the first.

We also have a slightly more obtuse definition as well, which may not seem to have much use at first glance.

Definition 16 (Eventually Increasing/Decreasing Sequences)**.** We say a sequence is *eventually* one of the terms from definition 15 if for some $N \in \mathbb{N}$, every term after N satisfies the definition for the term in 15.

Example 6. The Fibonacci sequence is also eventually increasing. If we pick $N = 2$, then every term after N is bigger than the previous term, that is, the sequence $1, 2, 3, 5, 8, 13, 21, \ldots$ is increasing.

1.2.5 Series

A series is a sum of an infinite number of terms. We often denote the terms of a series as a_i, and we add together all of the a_i terms. We denote this as

$$\sum_{i=1}^{\infty} a_i.$$

Formally, the infinite series is the limit of the sequence of partial sums of the series, that number found when adding the first n elements of the sum together. We denote that as

$$\sum_{i=1}^{\infty} a_i = \lim_{n\to\infty} \sum_{i=1}^{n} a_i,$$

but don't get too hung up on this. Some of the series we come across we will have a nice way to find the sum for. Note though, that we don't need to start our sums from $i = 1$, we could start them from anywhere and we would still be adding an infinite number of terms together. Another common starting point is $i = 0$, it is just dependent on the context you need to add an infinite number of terms together.

Definition 17 (Converge)**.** A series is said to *converge* if the limit of the partial sums exist.

One way to think about a series is a loop and an array. For example, if a_i was the ith element in an array, then we could add up the first n terms in this series by the following chunk of code. Note that the array is 0 indexed, so the first element would be `a[0]`, corresponding to a_1, so we need to adjust our start and end point of our loop appropriately.

```
int sum = 0;
for(int i = 0; i < n; i++) {
    sum += a[i];
}
```

The most common type of series we will see will be geometric series. A geometric series is a series with a constant ratio between consecutive terms. For example

$$S_1 = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots$$

is a geometric series because the term a_{i+1} is found by multiplying the a_i term by the constant $\frac{1}{2}$.

The series

$$S_2 = 3 + 27 + 81 + 243 + \cdots$$

is also a geometric series, and this time the constant is 3.

Now clearly one of these series does not converge, that is, have a finite answer, S_2 increases without bound. But does S_1 have a finite answer? Conveniently, there is an easy way to tell if a geometric series converges.

First, a geometric series in general looks like

$$\sum_{n=0}^{\infty} ar^n = a + ar + ar^2 + ar^3 + ar^4 + \cdots$$

where a is our starting term, and r is the common ratio between the two consecutive terms.

Now, using the same notation we can write the following theorem.

Theorem 5 (Geometric Series Theorem)**.** Given a geometric series

$$S = \sum_{n=0}^{\infty} ar^n,$$

S will converge if and only if $|r| < 1$. Furthermore, if $|r| < 1$, then the sum is given by

$$S = \frac{a}{1-r}.$$

So now we know that as S_2 has a value of $r = 3$, we know from the geometric series theorem that S_2 diverges. S_1 has a value of $r = \frac{1}{2}$, so it converges, and we can find what it converges to using the formula given in the geometric series theorem.

$$\begin{aligned} S_1 = \sum_{n=0}^{\infty} \frac{1}{2}\left(\frac{1}{2}\right)^n &= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots \\ &= \frac{\frac{1}{2}}{1 - \frac{1}{2}} \\ &= \frac{\frac{1}{2}}{\frac{1}{2}} \\ &= 1 \end{aligned}$$

For other type of series we generally need to use induction to find what a series will converge to.

1.2.6 Proofs

Recall that the definition of a proof was the following.

Definition (Proof)**.** The *proof* of a theorem is a logical argument for the theorem statement that follows a deductive list of steps that are justified by other theorems, lemmas, and corollaries.

We will now delve deeper into types of proofs. There are four main types of proof

- Proof by construction.
- Non constructive proof.
- Proof by contradiction.
- Proof by Induction.

First, lets look deeper at proof by construction.

We wish to prove the following lemma.

Lemma 1. $2^{99} + 1$ is a composite number.

We can approach proving this theorem by finding factors of $2^{99} + 1$, that is, show that it is not prime by constructing the list of primes that combine to make it

Proof.

$$\begin{aligned} 2^{99} + 1 &= \left(2^{33}\right)^3 + 1 \\ &= (2^{33} + 1)\left(\left(2^{33}\right)^2 - 2^{33} + 1\right) \end{aligned}$$

□

Ignoring how the factorisation actually happens, but knowing that there is indeed a way to find the factors of $2^{99} + 1$, and then explicitly finding them is a constructive way to prove that $2^{99} + 1$ is composite, hence this is a proof by construction.

Now, lets see a non constructive proof.

We wish to prove the following lemma.

Lemma 2. There exist irrational x, y such that x^y is rational.

This is a hard theorem to prove constructively because we would have to prove that x and y are both irrational for some x and y we pick, which is a hard objective in general, and also prove that x^y is rational for the same given x and y. But in comparison, it is easy to prove that such an x and y must exist, as long as we don't actually care about what they are. All we need to know is $\sqrt{2}$ is irrational.

Proof. If $\sqrt{2}^{\sqrt{2}}$ is rational, then we are done, take $x = y = \sqrt{2}$.

Otherwise $\sqrt{2}^{\sqrt{2}}$ is irrational, so take $x = \sqrt{2}^{\sqrt{2}}$ and take $y = \sqrt{2}$. Now we have

$$\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}}.$$

Using the exponent laws we get

$$\sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2$$

and 2 is rational, so we are done in this case as well. □

Now we have no idea if $\sqrt{2}^{\sqrt{2}}$ is rational or irrational, but we have shown that either way we can show there exists a rational number of the form x^y where both x and y are irrational, despite not knowing what that number is.

Lets now have a look at proof by contradiction.

In the last proof, we needed to assume that $\sqrt{2}$ is irrational, so lets prove that it is!

Lemma 3. $\sqrt{2}$ is irrational.

The general approach to a proof by contradiction is to assume the opposite of the statement we want to prove, and showing that is impossible. So in this case, the opposite would be $\sqrt{2}$ is rational.

Proof. Assume that $\sqrt{2} = \frac{a}{b}$ for some integer a and some positive integer b, and the fraction $\frac{a}{b}$ is reduced, that is, a and b have no common factors.

Now we can square both sides to get

$$2 = \frac{a^2}{b^2}.$$

Now we can solve this equation for a^2 to get

$$a^2 = 2b^2,$$

and this shows that 2 is a factor of a^2, so a^2 is even, and it is not hard to show that this means a is even. That is, $a = 2k$ for some k in the integers, so we can rewrite the left hand side of that equation to get

$$(2k)^2 = 2b^2.$$

Expanding and solving for b^2 gives

$$b^2 = 2k^2,$$

which by the same reasoning as earlier means b is even, that is, $b = 2m$ for some m in the positive integers.

So, both a and b have a factor of 2, but we assumed that a and b had no common factors, so this is a contradiction, so our original assumption must be wrong, and we can't write $\sqrt{2}$ as a reduced fraction, so it can't be written as any other fraction either, so it is not rational. □

The last method of proof we will discuss is proof by induction.

This is the most confusing proof technique. It requires showing that a base case is true, typically by a constructive approach, then assuming that given an arbitrary term in a sequence is true, then the very next term must be true as well. So then we can put the base case together with the proof that given we can show that if at least one term in the sequence is true, then the next term is true as well, to get that the second term is true. Then because we know the second term is true, we know the third is as well, and so on.

Lemma 4.

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + 4 + 5 + 6 + 7 + \cdots + n = \frac{n(n+1)}{2}$$

We will approach this with induction. Our base case will be when $n = 1$, and our assumption will be that the formula is true when $n = k$, and we will show that under that assumption the formula is true when $n = k + 1$ as well.

Proof. First we show our base case is true. Set $n = 1$ to get the left hand side to be 1, and the right hand side is

$$\frac{1(1+1)}{2} = 1,$$

so the left hand side equals the right hand side, so our base case is true.

Now we wish to do the inductive component of the proof. We will assume that when $n = k$ we have

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2},$$

then we will use that in the next part. We now wish to show that given the previous formula is true, we can show that

$$\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2},$$

which is what we would expect if we substituted $n = k + 1$ into the formula of the theorem.

To show this, we start with our left hand side, so we are only working with

$$\sum_{i=1}^{k+1} i.$$

Expanding this out we get

$$\sum_{i=1}^{k+1} i = 1 + 2 + 3 + \cdots + k + (k + 1).$$

The brackets around the last $k + 1$ are purely for grouping purposes, the entire contents of the brackets came from the same term, so it helps to see where each component came from in this case. Now we can group this expression in a clever way to use our inductive assumption. If we group the terms as follows

$$(1 + 2 + 3 + \cdots + k) + (k + 1)$$

the first set of brackets are exactly the case where $n = k$, so we can replace the expression in the first set of brackets by

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

to get

$$\frac{k(k + 1)}{2} + (k + 1).$$

Now we wish to simplify this expression to hopefully get what we expect. First, put both terms over a common denominator of 2, so we get

$$\frac{k(k + 1)}{2} + \frac{2(k + 1)}{2}.$$

Now we add together these fractions to get

$$\frac{k(k + 1) + 2(k + 1)}{2}.$$

Finally, factor out a $k + 1$ from the numerator to get

$$\frac{(k + 1)(k + 2)}{2},$$

which is exactly what our expected right hand side was. □

We showed that given the first case was true, and given that if we know one case is true, then the next case is also true, we have shown that no matter what entry in the sequence we pick, it has to be true, because either the one before it is true because the one before that is true, or the one before it is true because it was our first case.

1.2.7 Polynomials

Polynomials are some of the most used structures in mathematics and computer science. Polynomials of one variable are well studied, and crop up all the time in analysis of algorithms.

Definition 18 (Polynomial)**.** A *polynomial* is an expression consisting of variables and coefficients that involve only the operations of addition, subtraction, multiplication and non negative integer exponents of variables.

For example

$$x^2 - 4x + 7$$

is a polynomial in one variable, x. $x^3 + 2xyz^2 - yz + 1$ is an example in three variables, but these are less understood and are far more complex to work with, so single variable polynomials will be our focus. An example that is **not** a polynomial is

$$x^{-1} + 3 + x + 3x^2$$

as it contains a negative power.

Definition 19 (Polynomial Degree)**.** The degree of a polynomial is the largest exponent. The degree of a polynomial is denoted $\deg(p)$.

For example, the degree of $x^2 + 2x + 1$ is 2, and the degree of $x^7 + 8x + 1$ is 7.

Theorem 6 (Polynomial Laws)**.** If $p(x)$ of degree m and $q(x)$ is a polynomial of degree n, then the following holds.

- $p(x) + q(x)$ is a polynomial of degree $\max(\deg(p), \deg(q))$.
- $p(x) - q(x)$ is a polynomial of degree $\max(\deg(p), \deg(q))$.
- $p(x)q(x)$ is a polynomial of degree $\deg(p) + \deg(q)$.

1.2.8 Logic Notation

We need to recall a small amount of logic notation for this course, so we will define all our terms in this section.

Definition 20 (Boolean variable)**.** A boolean variable p can be either *true* or *false*

We can combine boolean variables in a number of ways, we will let p and q be boolean variables for the rest of this paragraph. The first is called OR denoted as $\vee$. $p \vee q$ is true if either p or q is true. The next is called AND and denoted as $\wedge$. $p \wedge q$ is true if both p and q are true. If at least one of them is false, then $p \wedge q$ is false. We can also negate a boolean variable, that is, change it from true to false, or false to true, denoted by $\neg$. If p is true, then $\neg p$ is false, and if p is false, then $\neg p$ is true. We often communicate simple expressions in a truth table. For example, the following truth table summarises OR and AND, where 1 is used as an abbreviation for true, and 0 is used as an abbreviation for false.

p	q	$p \vee q$	$p \wedge q$
0	0	0	0
1	0	1	0
0	1	1	0
1	1	1	1

We will expand upon logic notation as the need arises later in the course.

1.2.9 Binomial Coefficients

The way to choose k objects from a set of n objects is given by a binomial coefficient. Binomial coefficients show up in a vast number of areas as counting the number of steps an algorithm takes to run often involves counting subsets of a fixed size.

Definition 21 (Binomial Coefficient)**.** The binomial coefficient is denoted as $\binom{n}{k}$ and is defined to be

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}.$$

This generally looks like a straight forward calculation, but care must be taken when calculating it, as $n!$ may be too large to store correctly, so it is often safer to simplify the expression first.

Example 7. Lets try to calculate $\binom{10}{5}$ without the aid of a calculator or computer.

$$\binom{10}{5} = \frac{10!}{(10-5)!5!}$$

First, lets simplify the brackets in the denominator to get

$$\frac{10!}{5!5!}.$$

Now we will expand out the factorials to see what we can cancel from the numerator and denominator.

$$\frac{10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1}{5 \times 4 \times 3 \times 2 \times 1 \times 5 \times 4 \times 3 \times 2 \times 1}$$

We can cancel off $5 \times 4 \times 3 \times 2 \times 1$ in both the numerator and denominator to get

$$\frac{10 \times 9 \times 8 \times 7 \times 6}{5 \times 4 \times 3 \times 2 \times 1}.$$

Now we can pair the 10 in the numerator and the 5 in the denominator and cancel those to get a 2 in the numerator and a 1 in the denominator, likewise with the 8 and 4, and 6 and 3, so we get

$$\frac{2 \times 9 \times 2 \times 7 \times 2}{1 \times 1 \times 1 \times 2 \times 1}.$$

Simplifying the denominator we get

$$\frac{2 \times 9 \times 2 \times 7 \times 2}{2}.$$

Finally, we can cancel the 2 in the denominator with a 2 from the numerator to be left with

$$9 \times 2 \times 7 \times 2 = 252,$$

so

$$\binom{10}{5} = 252.$$

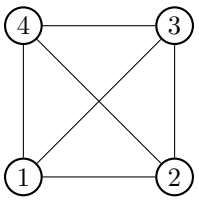
Now in this case we could have used a calculator to find this answer, but already at $n = 21$ we run into problems, we would need to calculate $n!$, but that is a 66 bit number, so can not be stored in a `long` variable, but a few lines of thinking and working our way through simplifications solves that problem.

1.2.10 Graphs

Graphs are an important tool, both to analysis problems, and to store data and relationship between data. But first, we need to define what a graph is.

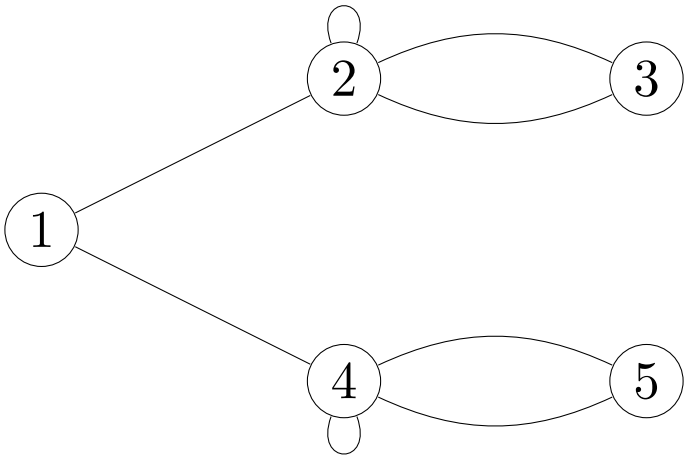
Definition 22 (Undirected Graph)**.** An undirected graph $G = (V, E)$ consists of a set V of vertices, and a set E of edges. An edge $e \in E$ is an *unordered* pair of vertices.

Example 8. The following is a graph, it has four vertices and six edges. The vertex set is $V = \{1, 2, 3, 4\}$ and the edge set is $E = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$.



A graph without a vertex that is connected to itself, that is, doesn't contain the edge (a, a) for a vertex a, and doesn't have multiple copies of the same edge is called a *simple* graph.

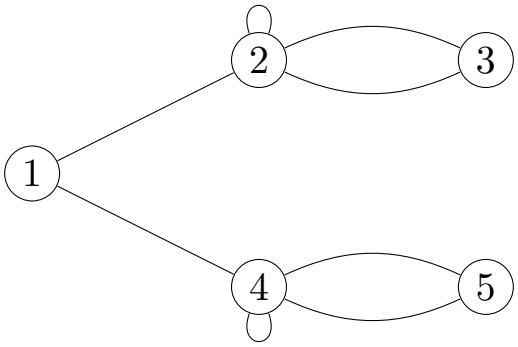
Example 9. For example, the following graph is not simple because it contains a loop on vertex 2 and 4, and the edge $(2, 3)$ and $(4, 5)$ both occur twice.



Definition 23 (Vertex Degree)**.** The *degree* of a vertex v is the number of edges incident on v.

Example 10. The degree of the vertices in the following graph are

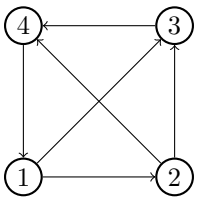
Vertex	1	2	3	4	5
Degree	2	5	2	5	2



So far, all the graphs we have seen have been undirected graphs, a directed graph, as one might guess from the name, has directions associated with the edges.

Definition 24 (Directed graphs)**.** A directed graph G consists of a set of vertices V and a set of directed edges E where each $e \in E$ is an *ordered* pair of vertices.

Example 11. The following is a graph, it has four vertices and six edges. The vertex set is $V = \{1, 2, 3, 4\}$ and the edge set is $E = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$.



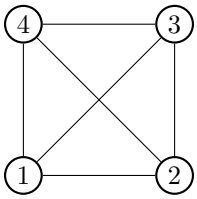
One way to represent graphs is by using an adjacency matrix.

Definition 25 (Adjacency Matrix)**.** An *adjacency matrix* is a square $|V| \times |V|$ matrix A, such that A_{ij} is 1 if there is an edge from vertex i to vertex j and a 0 if there is no edge.

While this definition seems abstract, it's simpler than it appears.

Example 12. The adjacency matrix for the following graph is

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$



The graph in example 12 is a special type of graph, called a *complete* graph.

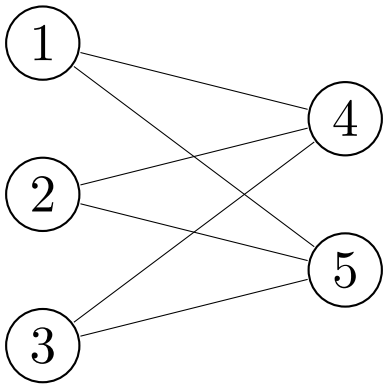
Definition 26 (Complete Graph)**.** A *complete* graph is a simple graph where every vertex is connected to every other vertex. The complete graph with n vertices is denoted as K_n.

Another important type of graph is the complete bipartite graph.

Definition 27 (Bipartite Graph)**.** A bipartite graph is a graph whose vertices can be divided in two disjoint sets U and V, such that every edge connects a vertex from U to a vertex in V. There are no edges from a vertex in U to a vertex in U, likewise there are no edges from a vertex in V to a vertex in V.

Definition 28 (Complete Bipartite Graph)**.** A *complete bipartite* graph is a bipartite graph that has an edge from every vertex in U to every vertex in V. The complete bipartite graph where $|U| = m$ and $|V| = n$ is denoted as $K_{m,n}$.

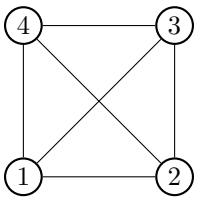
Example 13. The following is the complete bipartite graph $K_{3,2}$. $U = \{1, 2, 3\}$ and the set V is $V = \{4, 5\}$.



Definition 29 (Path)**.** A *path* of length n from a vertex v_0 to a vertex v_n in a graph G is an alternating sequence of $n + 1$ vertices and n edges, starting with v_0 and ending with v_n. When writing paths from simple graphs it is typical to leave out the edges and only list the vertices as there is only one path available to go from vertex u to vertex v.

Definition 30 (Simple Path)**.** A *simple* path from u to v is a path from u to v with no repeated vertices.

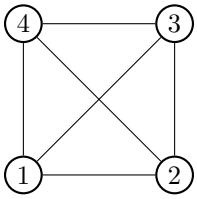
Example 14. $(1, 2, 4)$ is a simple path of length 2 in the following graph.



Definition 31 (Cycle)**.** A *cycle* is a path of length greater than 0 from u to u with no repeated edges.

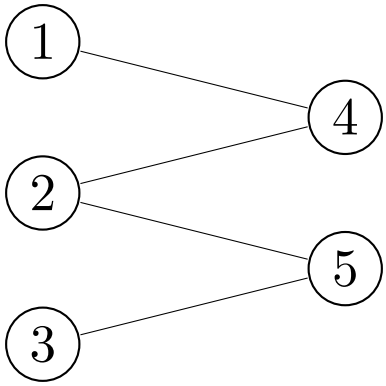
Definition 32 (Simple cycle)**.** A *simple* cycle is a cycle without repeated vertices except for the beginning and ending vertex.

Example 15. $(1, 2, 3, 1)$ is a simple cycle in the following graph.



Definition 33 (Acyclic graph)**.** A graph without any cycles is called *acyclic*

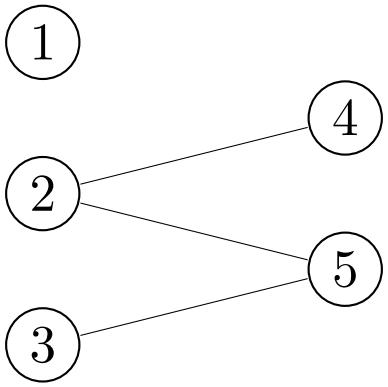
Example 16. The following is an acyclic graph.



Definition 34 (Connected Graph)**.** A graph is *connected* if there exists a path between u and v for every u and v in V

The graph in example 14 is an example of a connected graph.

Example 17. The following graph is not a connected graph, as there is no path from vertex 1 to any other vertex.



Now that we have defined connected graphs, we can circle back to cycles because there are many properties of cycles that we care about.

Definition 35 (Euler Cycle)**.** An *Euler* cycle in a graph G is a path from vertex u to itself with no repeated edges that contain all the edges and all of the vertices of G.

We know exactly when a graph has a Euler cycle from the following theorem.

Theorem 7 (Eulerian Graph)**.** A graph G has an Euler cycle if and only if G is connected and the degree of every vertex is even.

We can ask a similar question about using all of the vertices in a graph exactly once.

Definition 36 (Hamiltonian cycle)**.** A *Hamiltonian* cycle in a graph G is a cycle that contains each vertex in G exactly once.

Contrast the ease of determining if a graph has an Euler cycle to a Hamiltonian cycle. There is no efficient algorithm known how to tell if a graph has a Hamiltonian cycle. What it means to be *efficient* will be discussed later in the course.

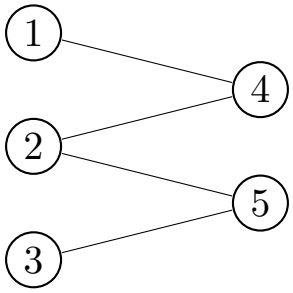
The distance between any two vertices is often important to know, both when modeling real world problems, such as trying to find the shortest distance when driving, or more abstract problems, such as determining the fewest number of coins required to give the correct amount of change (yes, this can be modeled with graphs and solved using dynamic programming!!).

Definition 37 (Distance)**.** The distance between a vertex u and a vertex v is the shortest path from u to v if it exists, and ∞ if there is no path from u to v. We denote the distance from u to v by $\text{dist}(u, v)$.

The last special class of graph we're going to go over here are trees.

Definition 38 (Tree)**.** A tree T is a simple graph such that for any two vertices u and v in T there is a unique simple path from u to v.

Example 18. The following graph is a tree as there is only a single path from any vertex to any other vertex.



Fortunately, there are many equivalent ways to determine if a graph is a tree, and they all have some use. The following theorem states the equivalence of some of these tests.

Theorem 8. The following statements about a graph G with n vertices are equivalent.

- T is a tree.
- T is connected and acyclic.
- T is connected and has $n-1$.
- T is acyclic and has $n-1$ edges.

There are special types of trees called *rooted trees*

Definition 39 (Rooted Tree)**.** In a *rooted* tree a particular vertex is designated as a root.

Definition 40 (Depth of a Vertex in a Rooted Tree)**.** The *Depth* of a vertex v in a rooted tree is the length of a simple path from the root to v.

Definition 41 (Height of a rooted tree)**.** The height of a rooted tree is the maximum depth that exists in the tree.

Definition 42 (Parent Vertex)**.** In a rooted tree, the *parent* vertex of a vertex v is the vertex connected to v on the path to the root. Every vertex has a unique parent, except the root which has no parent.

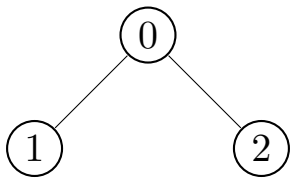
Definition 43 (Child Vertex)**.** In a rooted tree, a *child* of vertex v is a vertex of which v is the parent.

Definition 44 (Leaf Vertex)**.** A *leaf* is a vertex with no children.

Definition 45 (Internal Vertex)**.** An *internal* vertex is a vertex that is not a leaf.

Definition 46 (Binary Tree)**.** A *binary* tree is a tree such that the maximum number of children of any vertex is 2.

Example 19. The following is a rooted binary tree, where vertex 0 is the root, 1 and 2 are children of 0, 0 is the parent of 1 and 2. The only internal vertex is 0. 1 and 2 are both leaf vertices. The height of the tree is 1, and vertex 1 and 2 both have depth 1.



1.2.11 Recurrence Relations

Recurrence relations allow us to build up solutions to an instance of a problem, by using smaller instances of the same problem.

Definition 47 (Recurrence Relation)**.** A *recurrence relation* is a sequence $a_0, a_1, \ldots$ that has an associated equation for generating a_n from earlier terms in the sequence. That is

$$a_n = f(a_{n-1}, a_{n-2}, \ldots, a_0),$$

for some function f.

Definition 48 (Initial Conditions)**.** *Initial conditions* for the sequence $a_0, a_1, \ldots$ are explicitly given values for a finite number of terms in the sequence.

Again, don't be scared by formal definitions, lets look at an example.

Example 20. The classic example is the Fibonacci sequence.

$$f_n = f_{n-1} + f_{n-2}$$

with initial conditions $f_1 = 1$ and $f_2 = 1$.

This produces the sequence $1, 1, 2, 3, 5, 8, 13, 21, \ldots$.

There are generally two ways to find a closed form for a term in a recurrence relation, rather than calculating all smaller terms. We can either guess at the form and prove that it is the case using induction, or use theorems developed for solving recurrence relations.

Before we look at this theorems though, we need to understand what a linear recurrence relation is, a homogeneous and nonhomogenous recurrence relation, and what the degree of a recurrence relation all are.

Definition 49 (Linear Recurrence Relation)**.** A *linear* recurrence relation is a recurrence relation where the function f is of the form

$$\alpha + \sum_{k=1}^{n} c_k a_k$$

where α is a real number, c_k are real numbers, and a_k are terms in the sequence.

Example 21. $a_n = a_{n-1} + 2a_{n-2} + 3$ is linear as there are only sums of constants multiplied by previous terms and a constant without any previous terms.

$b_n = a_{n-1}a_{n-2}$ is *not* linear there are two previous terms multiplied together.

Definition 50 (Homogeneous Recurrence Relations)**.** A *homogeneous* recurrence relation is a recurrence relation where the function f is composed of only terms that involve a_k terms.

Example 22. $a_n = a_{n-1} + 2a_{n-2}$ is homogeneous as every term contains an a_k term.

$a_n = a_{n-1} + 2a_{n-2} + 3$ is *nonhomogeneous* as the term 3 does not contain an a_k term.

Definition 51 (Degree of a Recurrence Relation)**.** The *degree* of a recurrence relation is the largest difference between indices of the terms in the recurrence relation.

Example 23. The degree of $a_n = a_{n-1} + a_{n-2}$ is 2 as n is the largest index, and $n - 2$ is the smallest, and $n - (n - 2) = n - n + 2 = 2$.

Now we are at a point where we can start talking about solving linear, homogeneous recurrence relations.

We will need to factor polynomials, although generally this is impossible, we will only deal with simple polynomials here. The polynomial we will need to factor is called the *characteristic polynomial* and it is constructed from the recurrence relation we are trying to solve.

Definition 52 (Characteristic Polynomial)**.** The *characteristic polynomial* of an associated linear, homogeneous recurrence relation, $a_n = c_{n-1}a_{n-1} + \cdots + c_{n-k}a_{n-k}$ is

$$r^n = c_{n-1}r^{n-1} + \cdots + c_{n-k}r^{n-k}$$

which we would typically rearrange to

$$r^n - c_{n-1}r^{n-1} - \cdots - c_{n-k}r^{n-k} = 0.$$

Example 24. The characteristic polynomial for $a_n = a_{n-1} + 2a_{n-2}$ is

$$r^2 - r - 2 = 0.$$

Now we can solve all linear, homogeneous recurrence relations using the characteristic polynomial.

Theorem 9 (Solving Linear, Homogeneous Recurrence Relations)**.** Given a linear, homogeneous recurrence relation, $a_n = f(a_{n-1}, \ldots, a_{n-k})$, construct the associated characteristic polynomial, then we can write

$$a_n = \sum_{i=1}^{k} \sum_{j=0}^{m_i - 1} c_{ij} n^j r_i^n,$$

where c_{ij} is a constant which can be found using initial conditions, r_i is the ith root of the characteristic polynomial, m_i is the multiplicity of the root r_i.

This looks incredibly intimidating! Don't be intimidated though, remember how sums can be thought of as loops. Nested sums can just be thought of as nested loops. Lets look at an example though!

Example 25. First, lets look at a recurrence relation that has two distinct roots.

$$a_n = 3a_{n-1} - 2a_{n-2}$$

with initial conditions $a_1 = 2$ and $a_2 = 3$

The associated characteristic polynomial is then

$$r^2 = 3r - 2$$

which can be rearranged to

$$r^2 - 3r + 2 = 0.$$

Now we wish to factor this polynomial to find the roots.

Because this is a quadratic we can use the quadratic formula to find the roots are $r_1 = 1$ and $r_2 = 2$. Note that the order we write the roots doesn't matter, we just as easily could have wrote them the other way.

Now we just have to use the double sum from theorem 9. Seeing as the multiplicity of both roots is 1 our $m_i = 1$ in both cases, so $m_i - 1 = 0$ in both cases, so the second sum is only looked at when $j = 0$.

So we have

$$a_n = c_{1,0} n^0 r_1^n + c_{2,0} n^0 r_2^n.$$

Any number to the power of 0 is just 1, so both the n^0 terms simplify to 1, seeing as there are only two c terms, lets label them nicer and just call them c_1 and c_2, and then plug in the values for our roots to get

$$a_n = c_1 1^n + c_2 2^n.$$

We can further simplify the 1^n because that will always be 1 to get

$$a_n = c_1 + c_2 2^n.$$

This is the general solution to our recurrence relation

Now we use the initial conditions of $a_1 = 2$ and $a_2 = 3$ to find c_1 and c_2 by setting up two simultaneous equations, one when $n = 1$ and the other when $n = 2$ to get

$$2 = c_1 + c_2 2^1$$
$$3 = c_1 + c_2 2^2$$

Solving these gives $c_1 = 1$ and $c_2 = \frac{1}{2}$, so our recurrence relation is

$$a_n = 1 + \frac{1}{2} 2^n$$

which we can simplify to

$$a_n = 1 + 2^{n-1}.$$

After we find the solution using the particular initial conditions, this is called the *particular solution.*

Example 26. We will find the general solution to the following recurrence relation.

$$a_n = 7a_{n-1} - 16a_{n-2} + 12a_{n-3}.$$

We will approach this problem the same as the last, despite it being a recurrence relation of order 3.

The associated characteristic polynomial is

$$r^3 - 7r^2 + 16r - 12 = 0.$$

This is a cubic, which is not as simple to solve, so we will take for granted that it factors to

$$r^3 - 7r^2 + 16r - 12 = (x-3)(x-2)^2,$$

which you can check by expanding the right hand side. So the roots of the characteristic equation are $r_1 = 3$ and $r_2 = 2$, and r_2 has multiplicity 2.

Now we have to use the double sum from theorem 9. So $m_1 = 1$, so $m_1 - 1 = 0$ so the internal sum is only ran when $j = 0$. But $m_2 = 2$, so $m_2 - 1 = 1$, hence the internal sum is ran when $j = 0$ and $j = 1$. So our nested sums becomes

$$a_n = c_{1,0}n^0 3^n + c_{2,0}n^0 2^n + c_{2,1}n^1 2^n.$$

Simplifying the n^0 terms again gives

$$a_n = c_{1,0}3^n + c_{2,0}2^n + c_{2,1}n2^n.$$

And seeing as writing $c_{i,j}$ is tedious, we will rename them to c_1, c_2, and c_3 to get

$$a_n = c_1 3^n + c_2 2^n + c_3 n 2^n.$$

This is our general solution to the recurrence relation. To find a particular solution to it would require three initial conditions, and we would have to set up three simultaneous equations and solve them.