Mathematical Background

> *"When you have mastered numbers, you will in fact no longer be reading numbers, any more than you read words when reading books. You will be reading meanings."*, W. E. B. Du Bois

In this chapter, we review some of the mathematical concepts that we will use in this course. Most of these are not very complicated, but do require some practice and exercise to get comfortable with. If you have not previously encountered some of these concepts, there are several excellent freely-available resources online for them. In particular, the CS 121 webpage contains a program for self study of all the needed notions using the lecture notes, videos, and assignments of MIT course 6.042j Mathematics for Computer science. (The MIT lecture notes are also used by Harvard CS 20.)

0.4 A mathematician's apology

Before explaining the math background, perhaps I should explain why does this course is so "mathematically heavy". After all, this is supposed to be a course about *computation*; one might think we should be talking mostly about *programs*, rather than more "mathematical" objects such as *sets*, *functions*, and *graphs*, and doing more *coding* on an actual computer than writing mathematical proofs with pen and paper. So, why are we doing so much math in this course? Is it just some form of hazing? Perhaps a revenge of the "math nerds" against the "hackers"?

At the end of the day, mathematics is simply a language for modelling concepts in a precise and unambiguous way. In this course, we will be mostly interested in the concept of *computation*. For example, we will look at questions such as *"is there an efficient algorithm to find the prime factors of a given integer?"*.[1] To even *phrase* such a question,

[1] Actually, scientists currently do not know the answer to this question, but we will see that settling it in either direction has very interesting applications touching on areas as far apart as Internet security and quantum mechanics.

we need to give a precise *definition* of the notion of an *algorithm*, and of what it means for an algorithm to be *efficient*. Also, if the answer to this or similar questions turns out to be *negative*, then this cannot be shown by simply writing and executing some code. After all, there is no empirical experiment that will prove the *non existence* of an algorithm. Thus, our only way to show this type of *negative results* is to use *mathematical proofs*. So you can see why our main tools in this course will be mathematical proofs and definitions.

0.5 A quick overview of mathematical prerequisites

The main notions we will use in this course are the following:

- **Proofs:** First and foremost, this course will involve a heavy dose of formal mathematical reasoning, which includes mathematical *definitions*, *statements*, and *proofs*.
- **Sets:** Including notation such as membership ($\in$), containment ($\subseteq$), and set operations such as union, intersection, set difference and Cartesian product ($\cup, \cap, \setminus$ and $\times$).
- **Functions:** Including the notions of the *domain* and *range* of a function, properties such being *one-to-one* (also known as *injective*) or *onto* (also known as *surjective*) functions, as well as *partial functions* (that, unlike standard or "total" functions, are not necessarily defined on all elements of their domain).
- **Logical operations:** The operations AND, OR, and NOT ($\wedge, \vee, \neg$) and the quantifiers "exists" and "forall" ($\exists$,$\forall$).
- **Tuples and strings:** The notation Σ^k and Σ^* where Σ is some finite set which is called the *alphabet* (quite often $\Sigma = \{0,1\}$).
- **Basic combinatorics:** Notions such as $\binom{n}{k}$ (the number of k-sized subset of a set of size n).
- **Graphs:** Undirected and directed graphs, connectivity, paths, and cycles.
- **Big Oh notation:** $O, o, \Omega, \omega, \Theta$ notation for analyzing asymptotics of functions.
- **Discrete probability:** Later on in this course we will use *probability theory*, and specifically probability over *finite* samples spaces such as tossing n coins. We will only use probability theory in the second half of this course, and will review it beforehand. However,

probabilistic reasoning is a subtle (and extremely useful!) skill, and it's always good to start early in acquiring it.

While I highly recommend the resources linked above, in the rest of this section we briefly review these notions. This is partially to remind the reader and reinforce material that might not be fresh in your mind, and partially to introduce our notation and conventions which might occasionally differ from those you've encountered before.

0.6 Basic discrete math objects

We now quickly review some of the mathematical objects and definitions we in this course.

0.6.1 Sets

A *set* is an unordered collection of objects. For example, when we write $S = \{2,4,7\}$, we mean that S denotes the set that contains the numbers 2, 4, and 7. (We use the notation "$2 \in S$" to denote that 2 is an element of S.) Note that the set $\{2,4,7\}$ and $\{7,4,2\}$ are identical, since they contain the same elements. Also, a set either contains an element or does not contain it -there is no notion of containing it "twice"- and so we could even write the same set S as $\{2,2,4,7\}$ (though that would be a little weird). The *cardinality* of a finite set S, denoted by $|S|$, is the number of distinct elements it contains.[2] So, in the example above, $|S| = 3$. A set S is a *subset* of a set T, denoted by $S \subseteq T$, if every element of S is also an element of T. (We can also describe this by saying that T is a *superset* of S.) For example, $\{2,7\} \subseteq \{2,4,7\}$. The set that contains no elements is known as the *empty set* and it is denoted by $\emptyset$.

[2] Later in this course we will discuss how to extend the notion of cardinality to *infinite* sets.

We can define sets by either listing all their elements or by writing down a rule that they satisfy such as

$$EVEN = \{x \ : \ x = 2y \text{ for some non-negative integer } y\} \ . \tag{1}$$

Of course there is more than one way to write the same set, and often we will use intuitive notation listing a few examples that illustrate the rule. For example, we can also define $EVEN$ as

$$EVEN = \{0,2,4,\ldots\} \ . \tag{2}$$

Note that a set can be either finite (such as the set $\{2,4,7\}$) or infinite (such as the set $EVEN$). Also, the elements of a set don't have to be numbers. We can talk about the sets such as the set $\{a,e,i,o,u\}$ of all the vowels in the English language, or the set { `New York`, `Los Angeles`, `Chicago`, `Houston`, `Philadelphia`, `Phoenix`, `San Antonio`, `San Diego`, `Dallas` } of all cities in the U.S. with population more than one million per the 2010 census. A set can even have other sets as elements, such as the set $\{\emptyset,\{1,2\},\{2,3\},\{1,3\}\}$ of all even-sized subsets of $\{1,2,3\}$.

Operations on sets: The *union* of two sets S,T, denoted by $S \cup T$, is the set that contains all elements that are either in S *or* in T. The *intersection* of S and T, denoted by $S \cap T$, is the set of elements that are both in S *and* in T. The *set difference* of S and T, denoted by $S \setminus T$ (and in some texts also by $S - T$), is the set of elements that are in S but *not* in T.

Tuples, lists, strings, sequences: A *tuple* is an *ordered* collection of items. For example $(1,5,2,1)$ is a tuple with four elements (also known as a 4-tuple or quadruple). Since order matters, this is not the same tuple as the 4-tuple $(1,1,5,2)$ or the 3-tuple $(1,5,2)$. A 2-tuple is also known as a *pair*. We use the terms *tuples* and *lists* interchangeably. A tuple where every element comes from some finite set Σ (such as $\{0,1\}$) is also known as a *string*. Analogously to sets, we denote the *length* of a tuple T by $|T|$. Just like sets, we can also think of an infinite analogs of tuples, such as the ordered collection $(1,2,4,9,\ldots)$ of all perfect squares. Infinite ordered collections are known as *sequences*; we might sometimes use the term "infinite sequence" to emphasize this, and use "finite sequence" as a synonym for a tuple.[3]

[3] We can identify a sequence $(a_0,a_1,a_2,\ldots)$ of elements in some set S with a *function* $A : \mathbb{N} \to S$ (where $a_n = A(n)$ for every $n \in \mathbb{N}$). Similarly, we can identify a k-tuple $(a_0,\ldots,a_{k-1})$ of elements in S with a function $A : [k] \to S$.

Cartesian product: If S and T are sets, then their *Cartesian product*, denoted by $S \times T$, is the set of all ordered pairs (s,t) where $s \in S$ and $t \in T$. For example, if $S = \{1,2,3\}$ and $T = \{10,12\}$, then $S \times T$ contains the 6 elements $(1,10),(2,10),(3,10),(1,12),(2,12),(3,12)$. Similarly if S,T,U are sets then $S \times T \times U$ is the set of all ordered triples (s,t,u) where $s \in S$, $t \in T$, and $u \in U$. More generally, for every positive integer n and sets $S_0,\ldots,S_{n-1}$, we denote by $S_0 \times S_1 \times \cdots \times S_{n-1}$ the set of ordered n-tuples $(s_0,\ldots,s_{n-1})$ where $s_i \in S_i$ for every $i \in \{0,\ldots,n-1\}$.

For every set S, we denote the set $S \times S$ by S^2, $S \times S \times S$ by S^3, $S \times S \times S \times S$ by S^4, and so on and so forth.

0.6.2 Special sets

There are several sets that we will use in this course time and again, and so find it useful to introduce explicit notation for them. For starters we define

$$\mathbb{N} = \{0, 1, 2, \ldots\} \tag{3}$$

to be the set of all *natural numbers*, i.e., non-negative integers. For any natural number n, we define the set $[n]$ as $\{0, \ldots, n-1\} = \{k \in \mathbb{N} : k < n\}$.[4]

We will also occasionally use the set $\mathbb{Z} = \{\ldots, -2, -1, 0, +1, +2, \ldots\}$ of (negative and non-negative) *whole numbers*,[5] as well as the set $\mathbb{R}$ of *real* numbers. (This is the set that includes not just the whole numbers, but also fractional and even irrational numbers; e.g., $\mathbb{R}$ contains numbers such as $+0.5$, $-\pi$, etc.) We denote by $\mathbb{R}_+$ the set $\{x \in \mathbb{R} : x > 0\}$ of *positive* real numbers. This set is sometimes also denoted as $(0, \infty)$.

[4] We start our indexing of both $\mathbb{N}$ and $[n]$ from 0, while many other texts index those sets from 1. Starting from zero or one is simply a convention that doesn't make much difference, as long as one is consistent about it.

[5] The letter Z stands for the German word "Zahlen", which means *numbers*.

Strings: Another set we will use time and again is

$$\{0,1\}^n = \{(x_0, \ldots, x_{n-1}) \ : \ x_0, \ldots, x_{n-1} \in \{0,1\}\} \tag{4}$$

which is the set of all n-length binary strings for some natural number n. That is $\{0,1\}^n$ is the set of all n-tuples of zeroes and ones. This is consistent with our notation above: $\{0,1\}^2$ is the Cartesian product $\{0,1\} \times \{0,1\}$, $\{0,1\}^3$ is the product $\{0,1\} \times \{0,1\} \times \{0,1\}$ and so on.

We will write the string $(x_0, x_1, \ldots, x_{n-1})$ as simply $x_0 x_1 \cdots x_{n-1}$ and so for example

$$\{0,1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\} \ . \tag{5}$$

For every string $x \in \{0,1\}^n$ and $i \in [n]$, we write x_i for the i^{th} coordinate of x. If x and y are strings, then xy denotes their *concatenation*. That is, if $x \in \{0,1\}^n$ and $y \in \{0,1\}^m$, then xy is equal to the string $z \in \{0,1\}^{n+m}$ such that for $i \in [n]$, $z_i = x_i$ and for $i \in \{n, \ldots, n+m-1\}$, $z_i = y_{i-n}$.

We will also often talk about the set of binary strings of *all* lengths,

which is

$$\{0,1\}^* = \{(x_0,\ldots,x_{n-1}) \ : \ n \in \mathbb{N} \ , \ , x_0,\ldots,x_{n-1} \in \{0,1\}\} \ . \quad (6)$$

Another way to write this set is as

$$\{0,1\}^* = \{0,1\}^0 \cup \{0,1\}^1 \cup \{0,1\}^2 \cup \cdots \quad (7)$$

or more concisely as

$$\{0,1\}^* = \cup_{n\in\mathbb{N}} \{0,1\}^n \ . \quad (8)$$

The set $\{0,1\}^*$ contains also the "string of length 0" or "the empty string", which we will denote by "".[6]

[6] We follow programming languages in this notation; other texts sometimes use ϵ or λ to denote the empty string. However, this doesn't matter much since we will rarely encounter this "edge case".

Generalizing the star operation: For every set Σ, we define

$$\Sigma^* = \cup_{n\in\mathbb{N}} \Sigma^n \ . \quad (9)$$

For example, if $\Sigma = \{a,b,c,d,\ldots,z\}$ then Σ^* denotes the set of all finite length strings over the alphabet a-z.

Concatenation: The *concatenation* of two strings $x \in \Sigma^n$ and $y \in \Sigma^m$ is the $(n+m)$-length string xy obtained by writing y after x. That is, $(xy)_i$ equals x_i if $i < n$ and equals y_{i-n} if $n \leq i < n+m$.

0.6.3 Functions

If S and T are sets, a *function* F mapping S to T, denoted by $F : S \to T$, associates with every element $x \in S$ an element $F(x) \in T$. The set S is known as the *domain* of F and the set T is known as the *range* or *co-domain* of F. Just as with sets, we can write a function either by listing the table of all the values it gives for elements in S or using a rule. For example if $S = \{0,1,2,3,4,5,6,7,8,9\}$ and $T = \{0,1\}$. Then the function F defined by the input output behavior as in the table below, is the same as defining $F(x) = (x \mod 2)$.

If $F : S \to T$ satisfies that $F(x) \neq F(y)$ for all $x \neq y$ then we say that F is *one-to-one* (also known as an *injective* function or simply an *injection*).

If F satisfies that for every $y \in T$ there is some x such that $F(x) = y$ then we say that F is *onto* (also known as a *surjective* function or

Input	Output
0	0
1	1
2	0
3	1
4	0
5	1
6	0
7	1
8	0
9	1

simply a *surjection*). A function that is both one-to-one and onto is known as a *bijective* function or simply a *bijection*. If $S = T$ then a bijection from S to T is also known as a *permutation*. If $F : S \to T$ is a bijection then for every $y \in T$ there is a unique $x \in S$ s.t. $F(x) = y$. We denote this value x by $F^{-1}(y)$. Note that F^{-1} is itself a bijection from T to S (can you see why?).

Giving a bijection between two sets is often a good way to show they have the same size. In fact, the standard mathematical definition of the notion that "S and T have the same cardinality" is that there exists a bijection $f : S \to T$. In particular, the cardinality of a set S is defined n if there is a bijection from S to the set $\{0, \ldots, n-1\}$. As we will see later in this course, this is a definition that can generalizes to defining the cardinality of *infinite* sets.

Partial functions: We will sometimes be interested in *partial* functions from S to T. This is a generalization of the notion of a function to consider also F that is not necessarily defined on every element of S. For example, the partial function $F(x) = \sqrt{x}$ is only defined on non-negative real numbers. When we want to distinguish between partial functions and standard (i.e., non-partial) functions, we will call the latter *total* functions. When we say "function" without any qualifier then we mean a *total* function. That is, the notion of partial functions is a strict generalization of functions, and so a partial function *not* necessarily a function. The set of partial functions is a proper superset of the set of total functions, since a partial function is allowed to be defined on all its input elements. When we want to emphasize that a function f from A to B might not be total, we will write $f : A \to_p B$. We can think of a partial function F from S to T also as a total function from S to $T \cup \{\perp\}$ where $\perp$ is some special "failure symbol", and so instead of saying that F is undefined at x, we can say that $F(x) = \perp$.

Basic facts about functions: Verifying that you can prove the following results is an excellent way to brush up on functions:

- If $F : S \to T$ and $G : T \to U$ are one-to-one functions, then their *composition* $H : S \to U$ defined as $H(s) = G(F(s))$ is also one to one.
- If $F : S \to T$ is one to one, then there exists an onto function $G : T \to S$ such that $G(F(s)) = s$ for every $s \in S$.
- If $G : T \to S$ is onto then there exists a one-to-one function $F : S \to T$ such that $G(F(s) = s$ for every $s \in S$.
- If S and T are finite sets then the following conditions are equivalent to one another: **(a)** $|S| \leq |T|$, **(b)** there is a one-to-one function $F : S \to T$, and **(c)** there is an onto function $G : T \to S$.

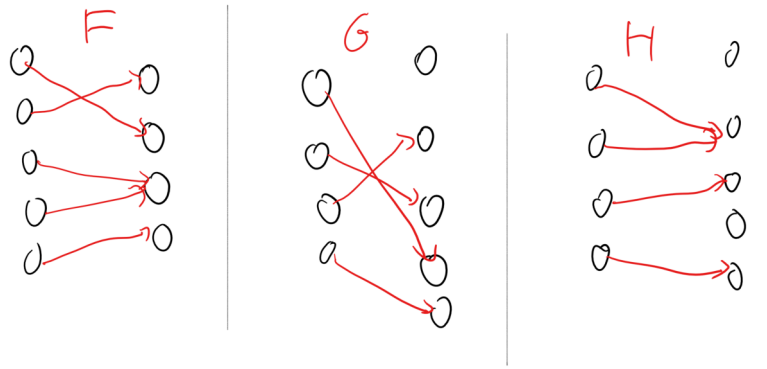

Figure 1: We can represent finite functions as a directed graph where we put an edge from x to $f(x)$. The *onto* condition corresponds to requiring that every vertex in the range of the function has in-degree *at least* one. The *one-to-one* condition corresponds to requiring that every vertex in the range of the function has in-degree *at most* one. In the examples above F is an onto function, G is one to one, and H is neither onto nor one to one.

You can find the proofs of these results in many discrete math texts, including for example, section 4.5 in the Leham-Leighton-Meyer notes. However, I strongly suggest you try to prove them on your own, or at least convince yourself that they are true by proving special cases of those for small sizes (e.g., $|S| = 3, |T| = 4, |U| = 5$).

Let us prove one of these facts as an example:

Lemma 0.1 If S, T are non-empty sets and $F : S \to T$ is one to one, then there exists an onto function $G : T \to S$ such that $G(F(s)) = s$ for every $s \in S$.

Proof. Let S, T and $F : S \to T$ be as in the Lemma's statement, and choose some $s_0 \in S$. We will define the function $G : T \to S$ as follows: for every $t \in T$, if there is some $s \in S$ such that $F(s) = t$ then set $G(t) = s$ (the choice of s is well defined since by the one-to-one property of F, there cannot be two distinct s, s' that both map to t). Otherwise, set $G(t) = s_0$. Now for every $s \in S$, by the definition of G, if $t = F(s)$ then $G(t) = G(F(s)) = s$. Moreover, this also shows that G is *onto*, since it means that for every $s \in S$ there is some t (namely $t = F(s)$) such that $G(t) = s$. ■

0.6.4 Graphs

Graphs are ubiquitous in Computer Science, and many other fields as well. They are used to model a variety of data types including social networks, road networks, deep nueral nets, gene interactions, correlations between observations, and a great many more. The formal definitions of graphs are below, but if you have not encountered them before then I urge you to read up on them in one of the sources linked above. Graphs come in two basic flavors: *undirected* and *directed*.[7]

[7] It is possible, and sometimes useful, to think of an undirected graph as simply a directed graph with the special property that for every pair u, v either both the edges $\overrightarrow{uv}$ and $\overleftarrow{uv}$ are present or neither of them is. However, in many settings there is a significant difference between undirected and directed graphs, and so it's typically best to think of them as separate categories.

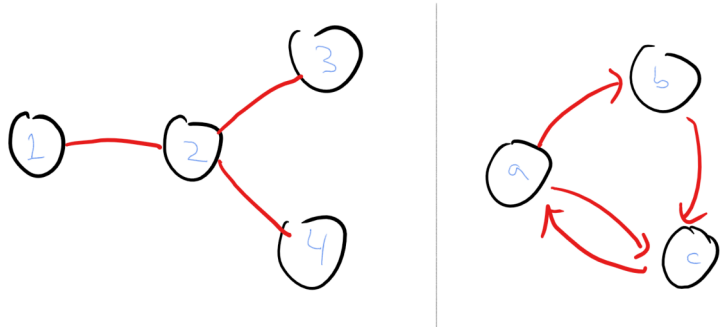

Figure 2: An example of an undirected and a directed graph. The undirected graph has vertex set $\{1,2,3,4\}$ and edge set $\{\{1,2\},\{2,3\},\{2,4\}\}$. The directed graph has vertex set $\{a,b,c\}$ and the edge set $\{(a,b),(b,c),(c,a),(a,c)\}$.

Definition 0.2 — Undirected graphs. An *undirected graph* $G = (V, E)$ consists of a set V of *vertices* and a set E of edges. Every edge is a size two subset of V. We say that two vertices $u, v \in V$ are *neighbors*, denoted by $u \sim v$, if the edge $\{u, v\}$ is in E.

Given this definition, we can define several other properties of graphs and their vertices. We define *degree* of u to be the number of neighbors u has. A *path* in the graph is a tuple $(u_0, \ldots, u_k) \in V^k$, for some $k > 0$ such that u_{i+1} is a neighbor of u_i for every $i \in [k]$. A *simple path* is a path $(u_0, \ldots, u_{k-1})$ where all the u_i's are distinct. A *cycle*

is a path $(u_0, \ldots, u_k)$ where $u_0 = u_k$. We say that two vertices $u, v \in V$ are *connected* if either $u = v$ or there is a path from $(u_0, \ldots, u_k)$ where $u_0 = u$ and $u_k = v$. We say that the graph G is *connected* if every pair of vertices in it is connected.

Here are some basic facts about undirected graphs. We give some informal arguments below, but leave the full proofs as exercises. (The proofs can also be found in most basic texts on graph theory.)

Lemma 0.3 In any undirected graph $G = (V, E)$, the sum of the degrees of all vertices is equal to twice the number of edges.

Lemma 0.3 can be shown by seeing that every edge $\{u, v\}$ contributes twice to the sum of the degrees (once for u and the second time for v.)

Lemma 0.4 The connectivity relation is *transitive*, in the sense that if u is connected to v, and v is connected to w, then u is connected to w.

Lemma 0.4 can be shown by simply attaching a path of the form $(u, u_1, u_2, \ldots, u_{k-1}, v)$ to a path of the form $(v, u'_1, \ldots, u'_{k'-1}, w)$ to obtain the path $(u, u_1, \ldots, u_{k-1}, v, u'_1, \ldots, u'_{k'-1}, w)$ that connects u to w.

Lemma 0.5 For every undirected graph $G = (V, E)$ and connected pair u, v, the shortest path from u to v is simple. In particular, for every connected pair there exists a simple path that connects them.

Lemma 0.5 can be shown by "shortcutting" any non simple path of the form $(u, u_1, \ldots, u_{i-1}, w, u_{i+1}, \ldots, u_{j-1}, w, u_{j+1}, \ldots, u_{k-1}, v)$ where the same vertex w appears in both the i-th and j-position, to obtain the shorter path $(u, u_1, \ldots, u_{i-1}, w, u_{j+1}, \ldots, u_{k-1}, v)$.

If you haven't seen these proofs before, it is indeed a great exercise to transform the above informal exercises into fully rigorous proofs.

Definition 0.6 — Directed graphs. A *directed graph* $G = (V, E)$ consists of a set V and a set $E \subseteq V \times V$ of *ordered pairs* of V. We denote the edge (u, v) also as $\overrightarrow{uv}$. If the edge $\overrightarrow{uv}$ is present in the graph then we say that v is an *out-neighbor* of u and u is an *in-neigbor* of v.

A directed graph might contain both $\overrightarrow{uv}$ and $\overrightarrow{vu}$ in which case u will be both an in-neighbor and an out-neighbor of v and vice versa. The *in-degree* of u is the number of in-neighbors it has, and the *out-degree* of v is the number of out-neighbors it has. A *path* in the graph

is a tuple $(u_0, \ldots, u_k) \in V^k$, for some $k > 0$ such that u_{i+1} is an out-neighbor of u_i for every $i \in [k]$. As in the undirected case, a *simple path* is a path $(u_0, \ldots, u_{k-1})$ where all the u_i's are distinct and a *cycle* is a path $(u_0, \ldots, u_k)$ where $u_0 = u_k$. One type of directed graphs we often care about is *directed acyclic graphs* or *DAGs*, which, as their name implies, are directed graphs without any cycles.

The lemmas we mentioned above have analogs for directed graphs. We again leave the proofs (which are essentially identical to their undirected analogs) as exercises for the reader:

Lemma 0.7 In any directed graph $G = (V, E)$, the sum of the in-degrees is equal to the sum of the out-degrees, which is equal to the number of edges.

Lemma 0.8 In any directed graph G, if there is a path from u to v and a path from v to w, then there is a path from u to w.

Lemma 0.9 For every directed graph $G = (V, E)$ and a pair u, v such that there is a path from u to v, the *shortest path* from u to v is simple.

Graph terminology The word *graph* in the sense above was coined by the mathematician Sylvester in 1878 in analogy with the chemical graphs used to visualize molecules. There is an unfortunate confusion with the more common usage of the term as a way to plot data, and in particular a plot of some function $f(x)$ as a function of x. We can merge these two meanings by thinking of a function $f : A \to B$ as a special case of a directed graph over the vertex set $V = A \cup B$ where we put the edge $\overrightarrow{xf(x)}$ for every $x \in A$. In a graph constructed in this way every vertex in A has out-degree one.

The following lecture of Berkeley CS70 provides an excellent overview of graph theory.

0.6.5 Logic operators and quantifiers.

If P and Q are some statements that can be true or false, then P AND Q (denoted as $P \wedge Q$) is the statement that is true if and only if both P *and* Q are true, and P OR Q (denoted as $P \vee Q$) is the statement that is true if and only if either P *or* Q is true. The *negation* of P, denoted as $\neg P$ or $\overline{P}$, is the statement that is true if and only if P is false.

Suppose that $P(x)$ is a statement that depends on some *parameter* x (also sometimes known as an *unbound* variable) in the sense that for

every instantiation of x with a value from some set S, $P(x)$ is either true or false. For example, $x > 7$ is a statement that is not a priori true or false, but does become true or false whenever we instantiate x with some real number. In such case we denote by $\forall_{x \in S} P(x)$ the statement that is true if and only if $P(x)$ is true *for every* $x \in S$.[8] We denote by $\exists_{x \in S} P(x)$ the statement that is true if and only if *there exists* some $x \in S$ such that $P(x)$ is true.

[8] In these notes we will place the variable that is bound by a quantifier in a subscript and so write $\forall_{x \in S} P(x)$ whereas other texts might use $\forall x \in S.P(x)$.

For example, the following is a formalization of the true statement that there exists a natural number n larger than 100 that is not divisible by 3:

$$\exists_{n \in \mathbb{N}} (n > 100) \wedge (\forall_{k \in N} k + k + k \neq n) \ . \tag{10}$$

"For sufficiently large n" One expression which comes up time and again is the claim that some statement $P(n)$ is true "for sufficiently large n". What this means is that there exists an integer N_0 such that $P(n)$ is true for every $n > N_0$. We can formalize this as $\exists_{N_0 \in \mathbb{N}} \forall_{n > N_0} P(n)$.

0.6.6 Quantifiers for summations and products

The following shorthands for summing up or taking products of several numbers are often convenient. If $S = \{s_0, \ldots, s_{n-1}\}$ is a finite set and $f : S \to \mathbb{R}$ is a function, then we write $\sum_{x \in S} f(x)$ as shorthand for

$$f(s_0) + f(s_1) + f(s_2) + \ldots + f(s_{n-1}) \ , \tag{11}$$

and $\prod_{x \in S} f(x)$ as shorthand for

$$f(s_0) \cdot f(s_1) \cdot f(s_2) \cdot \ldots \cdot f(s_{n-1}) \ . \tag{12}$$

For example, the sum of the squares of all numbers from 1 to 100 can be written as

$$\sum_{i \in \{1,\ldots,100\}} i^2 \ . \tag{13}$$

Since summing up over intervals of integers is so common, there is a special notation for it, and for every two integers $a \leq b$, $\sum_{i=a}^{b} f(i)$

denotes $\sum_{i \in S} f(i)$ where $S = \{x \in \mathbb{Z} : a \leq x \leq b\}$. Hence we can write the sum Eq. (13) as

$$\sum_{i=1}^{100} i^2 \,. \tag{14}$$

0.6.7 Parsing formulas: bound and free variables

In mathematics as in code, we often have symbolic "variables" or "parameters". It is important to be able to understand, given some formula, whether a given variable is *bound* or *free* in this formula. For example, in the following statement n is free but a, b are bound by the $\exists$ quantifier:

$$\exists_{a,b \in \mathbb{N}} (a \neq 1) \wedge (a \neq n) \wedge (n = a \times b) \tag{15}$$

Since n is free, it can be set to any value, and the truth of the statement Eq. (15) depends on the value of n. For example, if $n = 8$ then Eq. (15) is true, but for $n = 11$ it is false. (Can you see why?)

The same issue appears when parsing code. For example, in the following snippet from the C++ programming language

```
for (int i=0 ; i<n ; i=i+1) {
   printf("*");
}
```

the variable `i` is bound to the `for` operator but the variable `n` is free.

The main property of bound variables is that we can change them to a different name (as long as it doesn't conflict with another used variable) without changing the meaning of the statement. Thus for example the statement

$$\exists_{x,y \in \mathbb{N}} (x \neq 1) \wedge (x \neq n) \wedge (n = x \times y) \tag{16}$$

is equivalent to Eq. (15) in the sense that it is true for exactly the same set of n's. Similarly, the code

```
for (int j=0 ; j<n ; j=j+1) {
   printf("*");
}
```

produces the same result.

Aside: mathematical vs programming notation Mathematical notation has a lot of similarities with programming language, and for the same reasons. Both are formalisms meant to convey complex concepts in a precise way. However, there are some cultural differences. In programming languages, we often try to use meaningful variable names such as `NumberOfVertices` while in math we often use short identifiers such as n. (Part of it might have to do with the tradition of mathematical proofs as being handwritten and verbally presented, as opposed to typed up and compiled.) One consequence of that is that in mathematics we often end up reusing identifier, and also "run out" of letters and hence use greek letters too, as well as distinguish between small and capital letters. Similarly, mathematical notation tends to use quite a lot of "overloading", using operators such as $+$ for a great variety of objects (e.g., real numbers, matrices, finite field elements, etc..), and assuming that the meaning can be inferred from the context. Both fields have a notion of "types", and in math we often try to reserve certain letters for variables of a particular type. For example, variables such as i, j, k, ℓ, m, n will often denote integers, and ϵ will often denote a small positive real number. When reading or writing mathematical texts, we usually don't have the advantage of a "compiler" that will check type safety for us. Hence it is important to keep track of the type of each variable, and see that the operations that are performed on it "make sense".

0.6.8 Asymptotics and big-Oh notation

> *"$\log\log\log n$ has been proved to go to infinity, but has never been observed to do so."*, Anonymous, quoted by Carl Pomerance (2000)

It is often very cumbersome to describe precisely quantities such as running time and is also not needed, since we are typically mostly interested in the "higher order terms". That is, we want to understand the *scaling behavior* of the quantity as the input variable grows. For example, as far as running time goes, the difference between an n^5-time algorithm and an n^2-time one is much more significant than the difference between an $100n^2 + 10n$ time algorithm and an $10n^2$

time algorithm. For this purpose, Oh notation is extremely useful as a way to "declutter" our text and focus our attention on what really matters. For example, using Oh notation, we can say that both $100n^2 + 10n$ and $10n^2$ are simply $\Theta(n^2)$ (which informally means "the same up to constant factors"), while $n^2 = o(n^5)$ (which informally means that n^2 is "much smaller than" n^5).[9]

[9] While Big Oh notation is often used to analyze running time of algorithms, this is by no means the only application. At the end of the day, Big Oh notation is just a way to express asymptotic inequalities between functions on integers. It can be used regardless of whether these functions are a measure of running time, memory usage, or any other quantity that may have nothing to do with computation.

Generally (though still informally), if F, G are two functions mapping natural numbers to non-negative reals, then "$F = O(G)$" means that $F(n) \leq G(n)$ if we don't care about constant factors, while "$F = o(G)$" means that F is much smaller than G, in the sense that no matter by what constant factor we multiply F, if we take n to be large enough then G will be bigger (for this reason, sometimes $F = o(G)$ is written as $F \ll G$). We will write $F = \Theta(G)$ if $F = O(G)$ and $G = O(F)$, which one can think of as saying that F is the same as G if we don't care about constant factors. More formally, we define Big Oh notation as follows:

Definition 0.10 — Big Oh notation. For $F, G : \mathbb{N} \to \mathbb{R}_+$, we define $F = O(G)$ if there exist numbers $a, N_0 \in \mathbb{N}$ such that $F(n) \leq a \cdot G(n)$ for every $n > N_0$. We define $F = \Omega(G)$ if $G = O(F)$.

We write $F = o(G)$ if for every $\epsilon > 0$ there is some N_0 such that $F(n) < \epsilon G(n)$ for every $n > N_0$. We write $F = \omega(G)$ if $G = o(F)$. We write $F = \Theta(G)$ if $F = O(G)$ and $G = O(F)$.

We can also use the notion of *limits* to define big and little oh notation. You can verify that $F = o(G)$ (or, equivalently, $G = \omega(F)$) if and only if $\lim_{n\to\infty} \frac{F(n)}{G(n)} = 0$. Similarly, if the limit $\lim_{n\to\infty} \frac{F(n)}{G(n)}$ exists and is a finite number then $F = O(G)$. If you are familiar with the notion of *supremum*, then you can verify that $F = O(G)$ if and only if $\limsup_{n\to\infty} \frac{F(n)}{G(n)} < \infty$.

Using the equality sign for Oh notation is extremely common, but is somewhat of a misnomer, since a statement such as $F = O(G)$ really means that F is in the set $\{G' : \exists_{N,c} \text{ s.t. } \forall_{n>N} G'(n) \leq cG(n)\}$. For this reason, some texts write $F \in O(G)$ instead of $F = O(G)$. If anything, it would have made more sense use *inequalities* and write $F \leq O(G)$ and $F \geq \Omega(G)$, reserving equality for $F = \Theta(G)$, but by now the equality notation is quite firmly entrenched. Nevertheless, you should remember that a statement such as $F = O(G)$ means that F is "at most" G in some rough sense when we ignore constants, and a statement such as $F = \Omega(G)$ means that F is "at least" G in the same rough sense.

It's often convenient to use "anonymous functions" in the context

of Oh notation, and also to emphasize the input parameter to the function. For example, when we write a statement such as $F(n) = O(n^3)$, we mean that $F = O(G)$ where G is the function defined by $G(n) = n^3$. Chapter 7 in Jim Apsnes' notes on discrete math provides a good summary of Oh notation, see also this tutorial for a gentler and more programmer-oriented introduction.

0.6.9 *Some "rules of thumbs" for big Oh notation*

There are some simple heuristics that can help when trying to compare two functions F and G:

- Multiplicative constants don't matter in Oh notation, and so if $F(n) = O(G(n))$ then $100F(n) = O(G(n))$.
- When adding two functions, we only care about the larger one. For example, for the purpose of Oh notation, $n^3 + 100n^2$ is the same as n^3, and in general in any polynomial, we only care about the larger exponent.
- For every two constants $a, b > 0$, $n^a = O(n^b)$ if and only if $a \leq b$, and $n^a = o(n^b)$ if and only if $a < b$. For example, combining the two observations above, $100n^2 + 10n + 100 = o(n^3)$.
- Polynomial is always smaller than exponential: $n^a = o(2^{n^\epsilon})$ for every two constants $a > 0$ and $\epsilon > 0$ even if ϵ is much smaller than a. For example, $100n^{100} = o(2^{\sqrt{n}})$.
- Similarly, logarithmic is always smaller than polynomial: $(\log n)^a$ (which we write as $\log^a n$) is $o(n^\epsilon)$ for every two constants $a, \epsilon > 0$. For example, combining the observations above, $100n^2 \log^{100} n = o(n^3)$.

In most (though not all!) cases we use Oh notation, the constants hidden by it are not too huge and so on an intuitive level, you can think of $F = O(G)$ as saying something like $F(n) \leq 1000G(n)$ and $F = \Omega(G)$ as saying something $F(n) \geq 0.001G(n)$.

0.7 *Proofs*

Many people think of mathematical proofs as a sequence of logical deductions that starts from some axioms and ultimately arrives at a conclusion. In fact, some dictionaries define proofs that way. This is not entirely wrong, but in reality a mathematical proof of a statement

X is simply an argument that convinces the reader that X is true beyond a shadow of a doubt. To produce such a proof you need to:

1. Understand precisely what X means.
2. Convince *yourself* that X is true.
3. Write your reasoning down in plain, precise and concise English (using formulas or notation only when they help clarity).

In many cases, Step 1 is the most important one. Understanding what a statement means is often more than halfway towards understanding why it is true. In Step 3, to convince the reader beyond a shadow of a doubt, we will often want to break down the reasoning to "basic steps", where each basic step is simple enough to be "self evident". The combination of all steps yields the desired statement.

0.7.1 Proofs and programs

There is a great deal of similarity between the process of writing *proofs* and that of writing *programs*, and both require a similar set of skills. Writing a *program* involves:

1. Understanding what is the *task* we want the program to achieve.
2. Convincing *yourself* that the task can be achieved by a computer, perhaps by planning on a whiteboard or notepad how you will break it up to simpler tasks.
3. Converting this plan into code that a compiler or interpreter can understand, by breaking up each task into a sequence of the basic operations of some programming language.

In programs as in proofs, step 1 is often the most important one. A key difference is that the reader for proofs is a human being and for programs is a compiler.[10] Thus our emphasis is on *readability* and having a *clear logical flow* for the proof (which is not a bad idea for programs as well..). When writing a proof, you should think of your audience as an intelligent but highly skeptical and somewhat petty reader, that will "call foul" at every step that is not well justified.

[10] This difference might be eroding with time, as more proofs are being written in a *machine verifiable form* and progress in artificial intelligence allows expressing programs in more human friendly ways, such as "programming by example". Interestingly, much of the progress in automatic proof verification and proof assistants relies on a much deeper correspondence between *proofs* and *programs*. We *might* see this correspondence later in this course.

0.8 Extended example: graph connectivity

To illustrate these ideas, let us consider the following example of a true theorem:

Theorem 0.11 — Minimum edges for connected graphs. Every connected undirected graph of n vertices has at least $n - 1$ edges.

We are going to take our time to understand how one would come up with a proof for Theorem 0.11, and how to write such a proof down. This will not be the shortest way to prove this theorem, but hopefully following this process will give you some general insights on reading, writing, and discovering mathematical proofs.

Before trying to prove Theorem 0.11, we need to understand what it means. Let's start with the terms in the theorems. We defined undirected graphs and the notion of connectivity in **??** above. In particular, an undirected graph $G = (V, E)$ is *connected* if for every pair $u, v \in V$, there is a path $(u_0, u_1, \ldots, u_k)$ such that $u_0 = u$, $u_k = v$, and $\{u_i, u_{i+1}\} \in E$ for every $i \in [k]$.

It is crucial that at this point you pause and verify that you completely understand the definition of connectivity. Indeed, you should make a habit of pausing after any statement of a theorem, even before looking at the proof, and verifying that you understand all the terms that the theorem refers to.

To prove Theorem 0.11 we need to show that there is no 2-vertex connected graph with fewer than 1 edges, 3-vertex connected graph with fewer than 2 edges, and so on and so forth. One of the best ways to prove a theorem is to first try to *disprove it*. By trying and failing to come up with a counterexample, we often understand why the theorem can not be false. For example, if you try to draw a 4-vertex graph with only two edges, you can see that there are basically only two choices for such a graph as depicted in Fig. 3, and in both there will remain a vertex that is not connected.

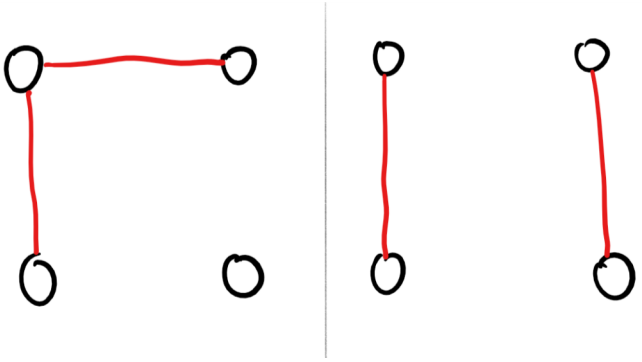

Figure 3: In a four vertex graph with two edges, either both edges have a shared vertex or they don't. In both cases the graph will not be connected.

In fact, we can see that if we have a budget of 2 edges and we choose some vertex u, we will not be able to connect to u more than two other vertices, and similarly with a budget of 3 edges we will not be able to connect to u more than three other vertices. We can keep trying to draw such examples until we convince ourselves that the theorem is probably true, at which point we want to see how we can *prove* it.

P If you have not seen the proof of this theorem before (or don't remember it), this would be an excellent point to pause and try to prove it yourself.

There are several ways to approach this proof, but one version is to start by proving it for small graphs, such as graphs with 2,3 or 4 edges, for which we can check all the cases, and then try to extend the proof for larger graphs. The technical term for this proof approach is *proof by induction*.

0.8.1 Mathematical induction

Induction is simply an application of the self-evident Modus Ponens rule that says that if **(a)** P is true and **(b)** P implies Q then Q is true. In the setting of proofs by induction we typically have a statement $Q(k)$ that is parameterized by some integer k, and we prove that **(a)** $Q(0)$ is true and **(b)** For every $k > 0$, if $Q(0), \ldots, Q(k-1)$ are all true then $Q(k)$ is true.[11] By repeatedly applying Modus Ponens, we can deduce from **(a)** and **(b)** that $Q(1)$ is true, and then from **(a)**,**(b)** and $Q(1)$ that $Q(2)$ is true, and so on and so forth to obtain that $Q(k)$ is true for every k. The statement **(a)** is called the "base case", while **(b)** is called the "inductive step". The assumption in **(b)** that $Q(i)$ holds for $i < k$ is called the "inductive hypothesis".

[11] Usually proving **(b)** is the hard part, though there are examples where the "base case" **(a)** is quite subtle.

R **Induction and recursion** Proofs by inductions are closely related to algorithms by recursion. In both cases we reduce solving a larger problem to solving a smaller instance of itself. In a recursive algorithm to solve some problem P on an input of length k we ask ourselves "what if someone handed me a way to solve P on instances smaller than k?". In an inductive proof to prove a statement Q parameterized by a number k, we ask ourselves "what if I already knew that $Q(k')$ is true for $k' < k$". Both induction and recursion are crucial concepts for this course and Computer Science at large (and even other areas

of inquiry, including not just mathematics but other sciences as well). Both can be initially (and even post-initially) confusing, but with time and practice they become clearer. For more on proofs by induction and recursion, you might find the following Stanford CS 103 handout, this MIT 6.00 lecture or this excerpt of the Lehman-Leighoton book useful.

0.8.2 *Proving the theorem by induction*

There are several ways to use induction to prove Theorem 0.11. We will do so by following our intuition above that with a budget of k edges, we cannot connect to a vertex more than k other vertices. That is, we will define the statement $Q(k)$ as follows:

> $Q(k)$ is *"For every graph $G = (V, E)$ with at most k edges and every $u \in V$, the number of vertices that are connected to u (including u itself) is at most $k+1$"*

Note that $Q(n-2)$ implies our theorem, since it means that in an n vertex graph of $n-2$ edges, there would be at most $n-1$ vertices that are connected to u, and hence in particular there would be *some* vertex that is not connected to u. More formally, if we define, given any undirected graph G and vertex u of G, the set $C_G(u)$ to contain all vertices connected to u, then the statement $Q(k)$ is that for every undirected graph $G = (V, E)$ with $|E| = k$ and $u \in V$, $|C_G(u)| \leq k+1$.

To prove that $Q(k)$ is true for every k by induction, we will first prove that **(a)** $Q(0)$ is true, and then prove **(b)** if $Q(0), \ldots, Q(k-1)$ are true then $Q(k)$ is true as well. In fact, we will prove the stronger statement **(b')** that if $Q(k-1)$ is true then $Q(k)$ is true as well. (**(b')** is a stronger statement than **(b)** because it has same conclusion with a weaker assumption.) Thus, if we show both **(a)** and **(b')** then we complete the proof of Theorem 0.11.

Proving **(a)** (i.e., the "base case") is actually quite easy. The statement $Q(0)$ says that if G has zero edges, then $|C_G(u)| = 1$, but this is clear because in a graph with zero edges, u is only connected to itself. The heart of the proof is, as typical with induction proofs, is in proving a statement such as **(b')** (or even the weaker statement **(b)**). Since we are trying to prove an *implication*, we can *assume* the so-called "inductive hypothesis" that $Q(k-1)$ is true and need to prove from this assumption that $Q(k)$ is true. So, suppose that $G = (V, E)$ is a graph of k edges, and $u \in V$. Since we can use induction, a nat-

ural approach would be to remove an edge $e \in E$ from the graph to create a new graph G' of $k-1$ edges. We can use the induction hypothesis to argue that $|C_{G'}(u)| \leq k$. Now if we could only argue that removing the edge e reduced the connected component of u by at most a single vertex, then we would be done, as we could argue that $|C_G(u)| \leq |C_{G'}(u)| + 1 \leq k + 1$.

> Please ensure that you understand why showing that $|C_G(u)| \leq |C_{G'}(u)| + 1$ completes the inductive proof.

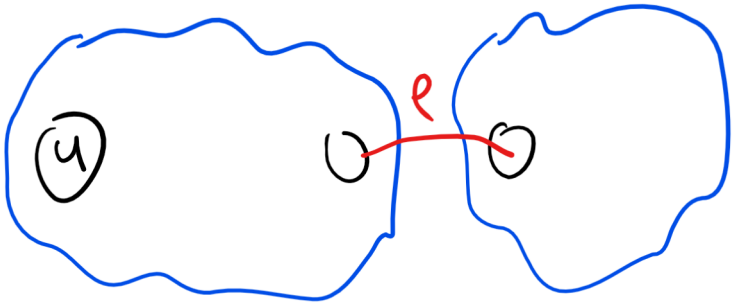

Figure 4: Removing a single edge e can greatly decrease the number of vertices that are connected to a vertex u.

Alas, this might not be the case. It could be that removing a single edge e will greatly reduce the size of $C_G(u)$. For example that edge might be a "bridge" between two large connected components; such a situation is illustrated in Fig. 4. This might seem as a real stumbling block, and at this point we might go back to the drawing board to see if perhaps the theorem is false after all. However, if we look at various concrete examples, we see that in any concrete example, there is always a "good" choice of an edge, adding which will increase the component connect to u by at most one vertex.

The crucial observation is that this always holds if we choose an edge $e = \{s, w\}$ where $w \in C_G(u)$ has degree one in the graph G, see Fig. 5. The reason is simple. Since every path from u to w must pass through s (which is w's only neighbor), removing the edge $\{s, w\}$ merely has the effect of disconnecting w from u, and hence $C_{G'}(u) = C_G(u) \setminus \{w\}$ and in particular $|C_{G'}(u)| = |C_G(u)| - 1$, which is exactly the condition we needed.

Now the question is whether there will always be a degree one vertex in $C_G(u) \setminus \{u\}$. Of course generally we are not guaranteed that a graph would have a degree one vertex, but we are not dealing

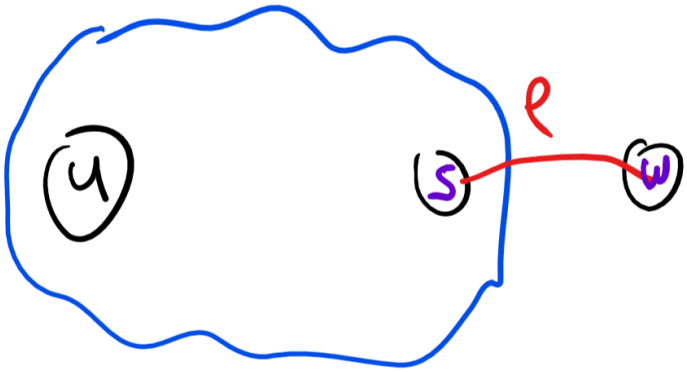

Figure 5: Removing an edge $e = \{s, w\}$ where $w \in C_G(u)$ has degree one removes only w from $C_G(u)$.

with a general graph here but rather a graph with a small number of edges. We can assume that $|C_G(u)| > k+1$ (otherwise we're done) and each vertex in $C_G(u)$ must have degree at least one (as otherwise it would not be connected to u). Thus, the only case where there is no vertex $w \in C_G(u) \setminus \{u\}$ of degree one, is when the degrees of all vertices in $C_G(u)$ are at least 2. But then by Lemma 0.3 the number of edges in the graph is at least $\frac{1}{2} \cdot 2 \cdot (k+1) > k$, which contradicts our assumption that the graph G has at most k edges. Thus we can conclude that either $|C_G(u)| \leq k+1$ (in which case we're done) or there is a degree one vertex $w \neq u$ that is connected to u. By removing the single edge e that touches w, we obtain a $k-1$ edge graph G' which (by the inductive hypothesis) satisfies $|C_{G'}(u)| \leq k$, and hence $|C_G(u)| = |C_{G'}(u) \cup \{w\}| \leq k+1$. This suffices to complete an inductive proof of statement $Q(k)$.

0.8.3 Writing down the proof

All of the above was a discussion of how we *discover* the proof, and convince *ourselves* that the statement is true. However, once we do that, we still need to write it down. When writing the proof, we use the benefit of hindsight, and try to streamline what was a messy journey into a linear and easy-to-follow flow of logic that starts with the word **"Proof:"** and ends with **"QED"** or the symbol ■.[12] All our discussions, examples and digressions can be very insightful, but we keep them outside the space delimited between these two words, where (as described by this excellent handout) "every sentence must be load bearing". Just like we do in programming, we can break the proof into little "subroutines" or "functions" (known as *lemmas* or

[12] QED stands for "quod erat demonstrandum", which is "What was to be demonstrated." or "The very thing it was required to have shown." in Latin.

claims in math language), which will be smaller statements that help us prove the main result. However, it should always be crystal-clear to the reader in what stage we are of the proof. Just like it should always be clear to which function a line of code belongs to, it should always be clear whether an individual sentence is part of a proof of some intermediate result, or is part of the argument showing that this intermediate result implies the theorem. Sometimes we highlight this partition by noting after each occurrence of **"QED"** to which lemma or claim it belongs.

Let us see how the proof of Theorem 0.11 looks in this streamlined fashion. We start by repeating the theorem statement

Theorem 0.12 — Minimum edges for connected graphs (restated). Every connected undirected graph of n vertices has at least $n-1$ edges.

Proof of Theorem 0.12. The proof will follow from the following lemma:

Lemma 0.13 For every $k \in \mathbb{N}$, undirected graph $G = (V, E)$ of at most k edges, and $u \in V$, the number of vertices connected to u in G is at most $k+1$.

We start by showing that Lemma 0.13 implies the theorem:

> **Proof of Theorem 0.12 from Lemma 0.13:** We will show that for undirected graph $G = (V, E)$ of n vertices and at most $n - 2$ edges, there is a pair u, v of vertices that are disconnected in G. let G be such a graph and u be some vertex of G. By Lemma 0.13, the number of vertices connected to u is at most $n-1$, and hence (since $|V| = n$) there is a vertex $v \in V$ that is not connected to u, thus completing the proof. **QED (Proof of Theorem 0.12 from Lemma 0.13)**

We now turn to proving Lemma 0.13. Let $G = (V, E)$ be an undirected graph of k edges and $u \in V$. We define $C_G(u)$ to be the set of vertices connected to u. To complete the proof of Lemma 0.13, we need to prove that $|C_G(u)| \leq k+1$. We will do so by induction on k.

The *base* case that $k = 0$ is true because a graph with zero edges, u is only connected to itself.

Now suppose that Lemma 0.13 is true for $k-1$ and we will prove it for k. Let $G = (V, E)$ and $u \in V$ be as above, where $|E| = k$, and suppose (towards a contradiction) that $|C_G(u)| \geq k+2$. Let

$S = C_G(u) \setminus \{u\}$. Denote by $deg(v)$ the degree of any vertex v. By Lemma 0.3, $\sum_{v \in S} deg(v) \leq \sum_{v \in V} deg(v) = 2|E| = 2k$. Hence in particular, under our assumption that $|S| + 1 = |C_G(u)| \geq k + 2$, we get that $\frac{1}{|S|} \sum_{v \in S} deg(v) \leq 2k/(k+1) < 2$. In other words, the *average* degree of a vertex in S is smaller than 2, and hence in particular there is *some* vertex $w \in S$ with degree smaller than 2. Since w is connected to u, it must have degree at least one, and hence (since w's degree is smaller than two) degree *exactly* one. In other words, w has a single neighbor which we denote by s.

Let G' be the graph obtained by removing the edge $\{s, w\}$ from G. Since G' has at most $k-1$ edges, by the inductive hypothesis we can assume that $|C_{G'}(u)| \leq k$. The proof of the lemma is concluded by showing the following claim:

> **Claim:** Under the above assumptions, $|C_G(u)| \leq |C_{G'}(u)| + 1$.

> **Proof of claim:** The claim says that $C_{G'}(u)$ has at most one fewer element than $C_G(u)$. Thus it follows from the following statement $(*)$: $C_{G'}(u) \supseteq C_G(u) \setminus \{w\}$. To prove (*) we need to show that for every $v \neq w$ that is connected to u, $v \in C_{G'}(u)$. Indeed for every such v, Lemma 0.5 implies that there must be some *simple* path $(t_0, t_1, \ldots, t_{i-1}, t_i)$ in the graph G where $t_0 = u$ and $t_i = v$. But w cannot belong to this path, since w is different from the endpoints u and v of the path and can't equal one of the intermediate points either, since it has degree one and that would make the path not simple. More formally, if $w = t_j$ for $0 < j < i$, then since w has only a single neighbor s, it would have to hold that w's neighbor s satisfies $s = t_{j-1} = t_{j+1}$, contradicting the simplicity of the path. Hence the path from u to v is also a path in the graph G', which means that $v \in C_{G'}(u)$, which is what we wanted to prove. **QED (claim)**

The claim implies Lemma 0.13 since by the inductive assumption, $|C_{G'}(u)| \leq k$, and hence by the claim $|C_G(u)| \leq k + 1$, which is what we wanted to prove. This concludes the proof of Lemma 0.13 and hence also of Theorem 0.12. **QED (Lemma 0.13), QED (Theorem 0.12)** ■

The proof above used the observation that if the *average* of some n numbers $x_0, \ldots, x_{n-1}$ is at most X, then there must *exists* at least a

single number $x_i \leq X$. (In this particular proof, the numbers were the degrees of vertices in S.) This is known as the *averaging principle*, and despite its simplicity, it is often extremely useful.

P Reading a proof is no less of an important skill than producing one. In fact, just like understanding code, it is a highly non-trivial skill in itself. Therefore I strongly suggest that you re-read the above proof, asking yourself at every sentence whether the assumption it makes are justified, and whether this sentence truly demonstrates what it purports to achieve. Another good habit is to ask yourself when reading a proof for every variable you encounter (such as u, t_i, G', etc. in the above proof) the following questions: **(1)** What *type* of variable is it? is it a number? a graph? a vertex? a function? and **(2)** What do we know about it? Is it an arbitrary member of the set? Have we shown some facts about it?, and **(3)** What are we *trying* to show about it?.

0.9 *Proof writing style*

A mathematical proof is a piece of writing, but it is a specific genre of writing with certain conventions and preferred styles. As in any writing, practice makes perfect, and it is also important to revise your drafts for clarity.

In a proof for the statement X, all the text between the words **"Proof:"** and **"QED"** should be focused on establishing that X is true. Digressions, examples, or ruminations should be kept outside these two words, so they do not confuse the reader. The proof should have a clear logical flow in the sense that every sentence or equation in it should have some purpose and it should be crystal-clear to the reader what this purpose is. When you write a proof, for every equation or sentence you include, ask yourself:

1. Is this sentence or equation stating that some statement is true?
2. If so, does this statement follow from the previous steps, or are we going to establish it in the next step?
3. What is the *role* of this sentence or equation? Is it one step towards proving the original statement, or is it a step towards proving some intermediate claim that you have stated before?
4. Finally, would the answers to questions 1-3 be clear to the reader? If not, then you should reorder, rephrase or add explanations.

Some helpful resources on mathematical writing include this handout by Lee, this handout by Hutching, as well as several of the excellent handouts in Stanford's CS 103 class.

0.9.1 Patterns in proofs

Just like in programming, there are several common patterns of proofs that occur time and again. Here are some examples:

Proofs by contradiction: One way to prove that X is true is to show that if X was false then we would get a contradiction as a result. Such proofs often start with a sentence such as "Suppose, towards a contradiction, that X is false" and end with deriving some contradiction (such as a violation of one of the assumptions in the theorem statement). Here is an example:

Lemma 0.14 There are no natural numbers a, b such that $\sqrt{2} = \frac{a}{b}$.

Proof. Suppose, towards the sake of contradiction that this is false, and so let $a \in \mathbb{N}$ be the smallest number such that there exists some $b \in \mathbb{N}$ satisfying $\sqrt{2} = \frac{a}{b}$. Squaring this equation we get that $2 = a^2/b^2$ or $a^2 = 2b^2$ $(*)$. But this means that a^2 is *even*, and since the product of two odd numbers is odd, it means that a is even as well, or in other words, $a = 2a'$ for some $a' \in \mathbb{N}$. Yet plugging this into $(*)$ shows that $4a'^2 = 2b^2$ which means $b^2 = 2a'^2$ is an even number as well. By the same considerations as above we gat that b is even and hence $a/2$ and $b/2$ are two natural numbers satisfying $\frac{a/2}{b/2} = \sqrt{2}$, contradicting the minimality of a. ■

Proofs of a universal statement: Often we want to prove a statement X of the form "Every object of type O has property P." Such proofs often start with a sentence such as "Let o be an object of type O" and end by showing that o has the property P. Here is a simple example:

Lemma 0.15 For every natural number $n \in N$, either n or $n+1$ is even.

Proof. Let $n \in N$ be some number. If $n/2$ is a whole number then we are done, since then $n = 2(n/2)$ and hence it is even. Otherwise, $n/2 + 1/2$ is a whole number, and hence $2(n/2 + 1/2) = n + 1$ is even. ■

Proofs of an implication: Another common case is that the statement X has the form "A implies B". Such proofs often start with a

sentence such as "Assume that A is true" and end with a derivation of B from A. Here is a simple example:

Lemma 0.16 If $b^2 \geq 4ac$ then there is a solution to the quadratic equation $ax^2 + bx + c = 0$.

Proof. Suppose that $b^2 \geq 4ac$. Then $d = b^2 - 4ac$ is a non-negative number and hence it has a square root s. Thus $x = (-b+s)/(2a)$ satisfies

$$ax^2 + bx + c = a(-b+s)^2/(4a^2) + b(-b+s)/(2a) + c = (b^2 - 2bs + s^2)/(4a) + (-b^2 + bs)/(2a) + c \,. \tag{17}$$

Rearranging the terms of Eq. (17) we get

$$s^2/(4a) + c - b^2/(4a) = (b^2 - 4ac)/(4a) + c - b^2/(4a) = 0 \tag{18}$$

■

Proofs of equivalence: If a statement has the form "A if and only if B" (often shortened as "A iff B") then we need to prove both that A implies B and that B implies A. We call the implication that A implies B the "only if" direction, and the implication that B implies A the "if" direction.

Proofs by combining intermediate claims: When a proof is more complex, it is often helpful to break it apart into several steps. That is, to prove the statement X, we might first prove statements X_1,X_2, and X_3 and then prove that $X_1 \wedge X_2 \wedge X_3$ implies X.[13] Our proof of Theorem 0.11 had this form.

[13] As mentioned below, $\wedge$ denotes the logical AND operator.

Proofs by case distinction: This is a special case of the above, where to prove a statement X we split into several cases $C_1, \ldots, C_k$, and prove that **(a)** the cases are *exhaustive*, in the sense that *one* of the cases C_i must happen and **(b)** go one by one and prove that each one of the cases C_i implies the result X that we are after.

"Without loss of generality (w.l.o.g)": This term can be initially quite confusing to students. It is essentially a way to shorten case distinctions such as the above. The idea is that if Case 1 is equal to Case 2 up to a change of variables or a similar transformation, then the proof of Case 1 will also imply the proof of case 2. It is always a statement that should be viewed with suspicion. Whenever you see it in a proof, ask yourself if you understand *why* the assumption made is truly without loss of generality, and when you use it, try to see if the use is indeed justified. Sometimes it might be easier to just repeat the proof of the second case (adding a remark that the proof is very similar to the first one).

Proofs by induction: We can think of such proofs as a variant of the above, where we have an unbounded number of intermediate claims $X_0, X_2, \ldots, X_k$, and we prove that X_0 is true, as well that X_0 implies X_1, and that $X_0 \wedge X_1$ implies X_2, and so on and so forth. The website for CMU course 15-251 contains a useful handout on potential pitfalls when making proofs by induction.

0.10 Non-standard notation

Most of the notation we discussed above is standard and is used in most mathematical texts. The main points where we diverge are:

- We index the natural numbers $\mathbb{N}$ starting with 0 (though many other texts, especially in computer science, do the same).
- We also index the set $[n]$ starting with 0, and hence define it as $\{0, \ldots, n-1\}$. In most texts it is defined as $\{1, \ldots, n\}$. Similarly, we index coordinates of our strings starting with 0, and hence a string $x \in \{0,1\}^n$ is written as $x_0 x_1 \cdots x_{n-1}$.
- We use *partial* functions which are functions that are not necessarily defined on all inputs. When we write $f : A \to B$ this will refer to a *total* function unless we say otherwise. When we want to emphasize that f can be a partial function, we will sometimes write $f : A \to_p B$.
- As we will see later on in the course, we will mostly describe our computational problems in the terms of computing a *Boolean function* $f : \{0,1\}^* \to \{0,1\}$. In contrast, most textbooks will refer to this as the task of *deciding a language* $L \subseteq \{0,1\}^*$. These two viewpoints are equivalent, since for every set $L \subseteq \{0,1\}^*$ there is a corresponding function $f = 1_L$ such that $f(x) = 1$ if and only if $x \in L$. Computing *partial functions* corresponds to the task known in the literature as a solving a *promise problem*.[14]
- Some other notation we use is $\lceil x \rceil$ and $\lfloor x \rfloor$ for the "ceiling" and "floor" operators that correspond to "rounding up" or "rounding down" a number to the nearest integer. We use $(x \mod y)$ to denote the "remainder" of x when divided by y. That is, $(x \bmod y) = x - y\lfloor x/y \rfloor$. In context when an integer is expected we'll typically "silently round" the quantities to an integer. For example, if we say that x is a string of length $\sqrt{n}$ then we'll typically mean that x is of length $\lceil \sqrt{n} \rceil$. (In most such cases, it will not make a difference whether we round up or down.)

[14] Because the language notation is so prevalent in textbooks, we will occasionally remind the reader of this correspondence.

- Like most Computer Science texts, we default to the logarithm in base two. Thus, $\log n$ is the same as $\log_2 n$.
- We will also use the notation $f(n) = poly(n)$ as a short hand for $f(n) = n^{O(1)}$ (i.e., as shorthand for saying that there is some constants a, b such that $f(n) \leq a \cdot n^b$ for every sufficiently large n). Similarly, we will use $f(n) = polylog(n)$ as shorthand for $f(n) = poly(\log n)$ (i.e., as shorthand for saying that there are some constant a, b such that $f(n) \leq a \cdot (\log n)^b$ for every sufficiently large n).

0.11 Exercises

Exercise 0.1 — Inclusion Exclusion. 1. Let A, B be finite sets. Prove that $|A \cup B| = |A| + |B| - |A \cap B|$.

2. Let $A_0, \ldots, A_{k-1}$ be finite sets. Prove that $|A_0 \cup \cdots \cup A_{k-1}| \geq \sum_{i=0}^{k-1} |A_i| - \sum_{0 \leq i < j < k} |A_i \cap A_j|$.

3. Let $A_0, \ldots, A_{k-1}$ be finite subsets of $\{1, \ldots, n\}$, such that $|A_i| = m$ for every $i \in [k]$. Prove that if $k > 100n$, then there exist two distinct sets A_i, A_j s.t. $|A_i \cap A_j| \geq m^2/(10n)$.

■

Exercise 0.2 Prove that if S, T are finite and $F : S \to T$ is one to one then $|S| \leq |T|$. ■

Exercise 0.3 Prove that if S, T are finite and $F : S \to T$ is onto then $|S| \geq |T|$. ■

Exercise 0.4 Prove that for every finite S, T, there are $(|T| + 1)^{|S|}$ partial functions from S to T. ■

Exercise 0.5 Suppose that $\{S_n\}_{n \in \mathbb{N}}$ is a sequence such that $S_0 \leq 10$ and for $n > 1$ $S_n \leq 5S_{\lfloor \frac{n}{5} \rfloor} + 2n$. Prove by induction that $S_n \leq 100n \log n$ for every n. ■

Exercise 0.6 Describe the following statement in English words: $\forall_{n \in \mathbb{N}} \exists_{p > n} \forall a, b \in \mathbb{N} (a \times b \neq p) \vee (a = 1)$. ■

Exercise 0.7 Prove that for every undirected graph G of 100 vertices, if every vertex has degree at most 4, then there exists a subset S of at 20 vertices such that no two vertices in S are neighbors of one another. ■

Exercise 0.8 Suppose that we toss three independent fair coins $a, b, c \in \{0, 1\}$. What is the probability that the XOR of a,b, and c is equal to 1? What is the probability that the AND of these three values is equal to 1? Are these two events independent? ■

Exercise 0.9 For every pair of functions F, G below, determine which of the following relations holds: $F = O(G)$, $F = \Omega(G)$, $F = o(G)$ or $F = \omega(G)$.

a. $F(n) = n$, $G(n) = 100n$.

b. $F(n) = n$, $G(n) = \sqrt{n}$.

c. $F(n) = n$, $G(n) = 2^{(\log(n))^2}$.

d. $F(n) = n$, $G(n) = 2^{\sqrt{\log n}}$ ■

Exercise 0.10 Give an example of a pair of functions $F, G : \mathbb{N} \to \mathbb{N}$ such that neither $F = O(G)$ nor $G = O(F)$ holds. ■

Exercise 0.11 — Topological sort. Prove that for every directed acyclic graph (DAG) $G = (V, E)$, there exists a map $f : V \to \mathbb{N}$ such that $f(u) < f(v)$ for every edge $\overrightarrow{u\,v}$ in the graph.[15] ■

[15] Hint: Use induction on the number of vertices. You might want to first prove the claim that every DAG contains a *sink*: a vertex without an outgoing edge.

0.12 *Bibliographical notes*

The section heading "A Mathematician's Apology", refers of course to Hardy's classic book. Even when Hardy is wrong, he is very much worth reading.

0.13 *Acknowledgements*