Review of Mathematical Concepts

1 Sets

1.1 What is a Set?

A ***set*** is simply a collection of objects. The objects (which we call the ***elements*** or ***members*** of the set) can be anything: numbers, people, fruits, whatever. For example, all of the following are sets:

A = {13, 11, 8, 23}
B = {8, 23, 11, 13}
C = {8, 8, 23, 23, 11, 11, 13, 13}
D = {apple, pear, banana, grape}
E = {January, February, March, April, May, June, July, August, September, October, November, December}
F = {x : x ∈ E and x has 31 days}
G = {January, March, May, July, August, October, December}
N = the nonnegative integers (We will generally call this set N, the ***natural numbers***.)
H = {i : ∃x ∈ N and i = 2x}
I = {0, 2, 4, 6, 8, …}
J = the even natural numbers
K = the syntactically valid C programs
L = {x : x ∈ K and x never gets into an infinite loop}
Z = the integers (… -3, -2, -1, 0, 1, 2, 3, …)

In the definitions of F and H, we have used the colon notation. Read it as "such that". We've also used the standard symbol ∈ for "element of". We will also use ∉ for "not an element of". So, for example, 17 ∉ A is true.

Remember that a set is simply a collection of elements. So if two sets contain precisely the same elements (regardless of the way we actually defined the sets), then they are identical. Thus F and G are the same set, as are H, I, and J.

An important aside: **Zero is an even number.** This falls out of any reasonable definition we can give for even numbers. For example, the one we used to define set H above. Or consider: 2 is even and any number that can be derived by adding or subtracting 2 from an even number is also even. In order to construct a definition for even numbers that does not include zero, we'd have to make a special case. That would make for an inelegant definition, which we hate. And, as we'll see down the road, we'd also have to make corresponding special cases for zero in a wide variety of algorithms.

Since a set is defined only by what elements it contains, it does not matter what order we list the elements in. Thus A and B are the same set.

Our definition of a set considers only whether or not an element is contained within the set. It does not consider how many times the element is mentioned. In other words, duplicates don't count. So A, B, and C are all equal.

Whenever we define a set, it would be useful if we could also specify a decision procedure for it. A ***decision procedure*** for a set S is an algorithm that, when presented with an object O, returns True if O ∈ S and False otherwise. Consider set K above (the set of all syntactically valid C programs). We can easily decide whether or not an object is an element of K. First, of course, it has to be a string. If you bring me an apple, I immediately say no. If it is a string, then I can feed it to a C compiler and let it tell me whether or not the object is in K. But now consider the set L (C programs that are guaranteed to halt on all inputs). Again, I can reject apples and anything else that isn't even in K. I can also reject some programs that clearly do loop forever. And I can accept some C programs, for example ones that don't contain any loops at all. But what about the general problem. Can I find a way to look at an arbitrary C program and tell whether or not it belongs in L. It turns out, as we'll see later, that the answer to this is no. We can prove that no program to solve this problem can exist. But that doesn't mean that the set L doesn't exist. It's a perfectly fine set. There just isn't a decision procedure for it.

The smallest set is the set that contains no elements. It is called the ***empty set***, and is written ∅ or {}.

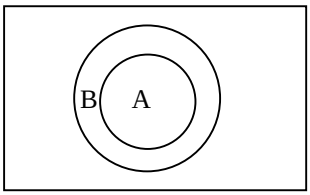
When you are working with sets, it is very important to keep in mind the difference between a set and the elements of a set. Given a set that contains more than one element, this not usually tricky. It's clear that {1, 2} is distinct from either the number 1 or the number 2. It sometimes becomes a bit trickier though with ***singleton sets*** (sets that contain only a single element). But it is equally true here. So, for example, {1} is distinct from the number 1. As another example, consider {∅}. This is a set that contains one element. That element is in turn a set that contains no elements (i.e., the empty set).

1.2 Relating Sets to Each Other

We say that A is a ***subset*** of B (which we write as $A \subseteq B$) if every element of A is also an element of B. The symbol we use for subset ($\subseteq$) looks somewhat like $\leq$. This is no accident. If $A \subseteq B$, then there is a sense in which the set A is "less than or equal to" the set B, since all the elements of A must be in B, but there may be elements of B that are not in A.

Given this definition, notice that every set is a subset of itself. This fact turns out to offer us a useful way to prove that two sets A and B are equal: First prove that A is a subset of B. Then prove that B is a subset of A. We'll have more to say about this later in Section 6.2.

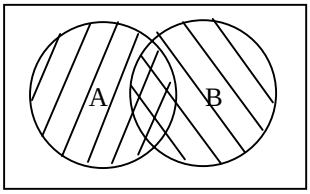
We say that A is ***proper subset*** of B (written $A \subset B$) if $A \subseteq B$ and $A \neq B$. The following Venn diagram illustrates the proper subset relationship between A and B:



Notice that the empty set is a subset of every set (since, trivially, every element of ∅, all none of them, is also an element of every other set). And the empty set is a *proper* subset of every set other than itself.

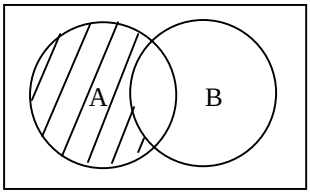
It is useful to define some basic operations that can be performed on sets:

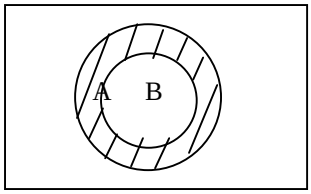
The ***union*** of two sets A and B (written $A \cup B$) contains all elements that are contained in A or B (or both). We can easily visualize union using a Venn diagram. The union of sets A and B is the entire hatched area:



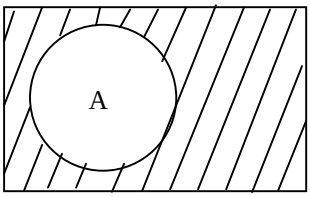
The ***intersection*** of two sets A and B (written $A \cap B$) contains all elements that are contained in both A and B. In the Venn diagram shown above, the intersection of A and B is the double hatched area in the middle.

The ***difference*** of two sets A and B (written A - B) contains all elements that are contained in A but not in B. In both of the following Venn diagrams, the hatched region represents A - B.

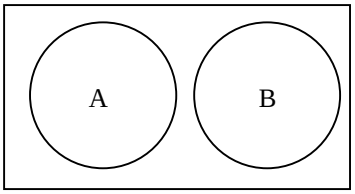




The ***complement*** of a set A with respect to a specific domain D (written as $\overline{A}$ or $\neg A$) contains all elements of D that are not contained in A (i.e, $\overline{A}$ = D - A). For example, if D is the set of residents of Austin and A is the set of Austin residents who like barbeque, then $\overline{A}$ is the set of Austin residents who don't like barbeque. The complement of A is shown as the hatched region of the following Venn diagram:



Two sets are ***disjoint*** if they have no elements in common (i.e., their intersection is empty). In the following Venn diagram, A and B are disjoint:



So far, we've talked about operations on pairs of sets. But just as we can extend binary addition and sum up a whole set of numbers, we can extend the binary operations on sets and perform then on sets of sets. Recall that for summation, we have the notation

$$\sum A_i$$

Similarly, we'll introduce

$$\bigcup A_i \qquad \text{and} \qquad \bigcap A_i$$

to indicate the union of a set of sets and the intersection of a set of sets, respectively.

Now consider a set A. For example, let A = {1, 2, 3}. Next, let's enumerate the set of all subsets of A:

{∅, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}

We call this set the ***power set*** of A, and we write it 2^A. The power set of A is interesting because, if we're working with the elements of A, we may well care about all the ways in which we can combine those elements.

Now for one final property of sets. Again consider the set A above. But this time, rather than looking for all possible subsets, let's just look for a single way to carve A up into subsets such that each element of A is in precisely one subset. For example, we might choose any of the following sets of subsets:

{{1}, {2, 3}} or {{1, 3}, 2} or {{1, 2, 3}}

We call any such set of subsets a partition of A. Partitions are very useful. For example, suppose we have a set S of students in a school. We need for every student to be assigned to precisely one lunch period. Thus we must construct a partition of S: a set of subsets, one for each lunch period, such that each student is in precisely one subset. More formally, we say that Π is a ***partition*** of a set A if and only if (a) no element of Π is empty; (b) all members of Π are disjoint (alternatively, each element of A is in only one element of Π); and (c) $\bigcup \Pi = A$ (alternatively, each element of A is in some element of Π and no element not in A is in any element of Π).

This notion of partitioning a set is fundamental to programming. Every time you analyze the set of possible inputs to your program and consider the various cases that must be dealt with, you're forming a partition of the set of inputs: each input must fall through precisely one path in your program. So it should come as no surprise that, as we build formal models of computational devices, we'll rely heavily on the idea of a partition on a set of inputs as an analytical technique.

2 Relations and Functions

In the last section, we introduced some simple relations that can hold between sets (subset and proper subset) and we defined some operations (functions) on sets (union, intersection, difference, and complement). But we haven't yet defined formally what we mean by a relation or a function. Let's do that now. (By the way, the reason we introduced relations and functions on sets in the last section is that we're going to use sets as the basis for our formal definitions of relations and functions and we will need the simple operations we just described as part of our definitions.)

2.1 Relations

An ***ordered pair*** is a sequence of two objects. Given any two objects, x and y, there are two ordered pairs that can be formed. We write them as (x, y) and (y, x). As the name implies, in an ordered pair (as opposed to in a set), order matters (unless x and y happen to be equal).

The ***Cartesian product*** of two sets A and B (written $A \times B$) is the set of all ordered pairs (a, b) such that $a \in A$ and $b \in B$. For example, let A be a set of people {Dave, Sue, Billy} and let B be a set of desserts {cake, pie, ice cream}. Then

$A \times B$ = { (Dave, cake), (Dave, pie), (Dave, ice cream),
(Sue, cake), (Sue, pie), (Sue, ice cream),
(Billy, cake), (Billy, pie), (Billy, ice cream)}

As you can see from this example, the Cartesian product of two sets contains elements that represent all the ways of pairing someone from the first set with someone from the second. Note that $A \times B$ is not the same as $B \times A$. In our example,

$B \times A$ = { (cake, Dave), (pie, Dave), (ice cream, Dave),
(cake, Sue), (pie, Sue), (ice cream, Sue),
(cake, Billy), (pie, Billy), (ice cream, Billy)}

We'll have more to say about the cardinality (size) of sets later, but for now, let's make one simple observation about the cardinality of a Cartesian product. If A and B are finite and if there are p elements in A and q elements in B, then there are p*q elements in $A \times B$ (and in $B \times A$).

We're going to use Cartesian product a lot. It's our basic tool for constructing complex objects out of simpler ones. For example, we 're going to define the class of Finite State Machines as the Cartesian product of five sets. Each individual finite state machine then will be a five tuple (K, Σ , δ, s, F) drawn from that Cartesian product. The sets will be:

1. The set of all possible sets of states: {{q1}, {q1, q2}, {q1, q2, q3}, …}. We must draw K from this set.
2. The set of all possible input alphabets: {{a}, {a, b, c}, {α, β, γ}, {1, 2, 3, 4}, {1, w, h, j, k}, {q, a, f}, {a, β, 3, j, f}…}. We must draw Σ from this set.
3. The set of all possible transition functions, which tell us how to move from one state to the next. We must draw δ from this set.
4. The set of all possible start states. We must draw s from this set.
5. The set of all possible sets of final states. (If we land in one of these when we've finished processing an input string, then we accept the string, otherwise we reject.) We must draw F from this set.

Let's return now to the simpler problem of choosing dessert. Suppose we want to define a relation that tells us, for each person, what desserts he or she likes. We might write the Dessert relation, for example as

{(Dave, cake), (Dave, ice cream), (Sue, pie), (Sue, ice cream)}

In other words, Dave likes cake and ice cream, Sue likes pie and ice cream, and Billy hates desserts.

We can now define formally what a relation is. A ***binary relation*** over two sets A and B is a subset of $A \times B$. Our dessert relation clearly satisfies this definition. So do lots of other relations, including common ones defined on the integers. For

example, Less than (written <) is a binary relation on the integers. It contains an infinite number of elements drawn from the Cartesian product of the set of integers with itself. It includes, for example:

{(1,2), (2,3), (3, 4), …}

Notice several important properties of relations as we have defined them. First, a relation may be equal to the empty set. For example, if Dave, Sue, and Billy all hate dessert, then the dessert relation would be {} or ∅.

Second, there are no constraints on how many times a particular element of A or B may occur in the relation. In the dessert example, Dave occurs twice, Sue occurs twice, Billy doesn't occur at all, cake occurs once, pie occurs once, and ice cream occurs twice.

If we have two or more binary relations, we may be able combine them via an operation we'll call composition. For example, if we knew the number of fat grams in a serving of each kind of dessert, we could ask for the number of fat grams in a particular person's dessert choices. To compute this, we first use the Dessert relation to find all the desserts each person likes. Next we get the bad news from the FatGrams relation, which probably looks something like this:

{(cake, 25), (pie, 15), (ice cream, 20)

Finally, we see that the composed relation that relates people to fat grams is {(Dave, 25), (Dave, 20), (Sue, 15), (Sue, 20)}. Of course, this only worked because when we applied the first relation, we got back desserts, and our second relation has desserts as its first component. We couldn't have composed Dessert with Less than, for example.

Formally, we say that the ***composition*** of two relations $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$, written $R_2 \circ R_1$ is $\{(a,c) : \exists (a, b) \in R_1$ and $(b, c) \in R_2\}$. Note that in this definition, we've said that to compute $R_2 \circ R_1$, we first apply R_1, then R_2. In other words we go right to left. Some definitions go the other way. Obviously we can define it either way, but it's important to check carefully what definition people are using and to be consistent in what you do. Using this notation, we'd represent the people to fat grams composition described above as FatGrams ° Dessert.

Now let's generalize a bit. An ordered pair is a sequence (where order counts) of two elements. We could also define an ordered triple as a sequence of three elements, an ordered quadruple as a sequence of four elements, and so forth. More generally, if n is any positive integer, then an ***ordered n-tuple*** is a sequence of n elements. For example, (Ann, Joe, Mark) is a 3-tuple.

We defined binary relation using our definition of an ordered pair. Now that we've extended our definition of an ordered pair to an ordered n-tuple, we can extend our notion of a relation to allow for an arbitrary number of elements to be related. We define an ***n-ary relation*** over sets $A_1, A_2, \ldots A_n$ as a subset of $A_1 \times A_2 \times \ldots \times A_n$. The n sets may be different, or they may be the same. For example, let A be a set of people:

A = {Dave, Sue, Billy, Ann, Joe, Mark, Cathy, Pete}

Now suppose that Ann and Dave are the parents of Billy, Ann and Joe are the parents of Mark, and Mark and Sue are the parents of Cathy. Then we could define a 3-ary (or ***ternary***) relation Child-of as the following subset of $A \times A \times A$:

{(Ann, Dave, Billy), (Ann, Joe, Mark), (Mark, Sue, Cathy)}

2.2 Functions

Relations are very general. They allow an object to be related to any number of other objects at the same time (as we did in the dessert example above). Sometimes, we want a more restricted notion, in which each object is related to a unique other object. For example, (at least in an ideal world without criminals or incompetent bureaucrats) each American resident is related to a unique social security number. To capture this idea we need functions. A ***function*** from a set A to a set B is a special kind of a binary relation over A and B in which each element of A occurs precisely once. The dessert relation we defined earlier is not a function since Dave and Sue each occur twice and Billy doesn't occur at all. We haven't restricted each person to precisely one dessert. A simple relation that *is* a function is the successor function Succ defined on the integers:

Succ(n) = n + 1.

Of course, we cannot write out all the elements of Succ (since there are an infinite number of them), but Succ includes:

{…, (-3, -2), (-2, -1), (-1, 0), (0, 1), (1, 2), (2, 3), …}

It's useful to define some additional terms to make it easy to talk about functions. We start by writing

$f: A \rightarrow B$,

which means that f is a function from the set A to the set B. We call A the ***domain*** of f and B the ***codomain*** or ***range***. We may also say that f is a function from A to B. If $a \in A$, then we write

f(a),

which we read as "f of a" to indicate the element of B to which a is related. We call this element the ***image*** of a under f or the ***value*** of f for a. Note that, given our definition of a function, there must be exactly one such element. We'll also call a the ***argument*** of f. For example we have that

Succ(1) = 2, Succ (2) = 3, and so forth.

Thus 2 is the image (or the value) of the argument 1 under Succ.

Succ is a ***unary function***. It maps from a single element (a number) to another number. But there are lots of interesting functions that map from ordered pairs of elements to a value. We call such functions ***binary functions***. For example, integer addition is a binary function:

$+: (Z \times Z) \rightarrow Z$

Thus + includes elements such as ((2, 3), 5), since 2 + 3 is 5. We could also write

+((2,3)) = 5

We have double parentheses here because we're using the outer set to indicate function application (as we did above without confusion for Succ) and the inner set to define the ordered pair to which the function is being applied. But this is confusing. So, generally, when the domain of a function is the Cartesian product of two or more sets, as it is here, we drop the inner set of parentheses and simply write

+(2,3) = 5.

Alternatively, many common binary functions are written in infix notation rather than the prefix notation that is standard for all kinds of function. This allows us to write

2+3 = 5

So far, we've had unary functions and binary functions. But just as we could define n-ary relations for arbitrary values of n, we can define n-ary functions. For any positive integer n, an ***n-ary function*** f is a function is defined as

$F: (D_1 \times D_2 \ldots \times D_n) \rightarrow R$

For example, let Z be the set of integers. Then

QuadraticEquation: $(Z \times Z \times Z) \rightarrow F$

is a function whose domain is an ordered triple of integers and whose domain is a set of functions. The definition of Quadratic Equation is:

QuadraticEquation(a, b, c) (x) = $ax^2 + bx + c$

What we did here is typical of function definition. First we specify the domain and the range of the function. Then we define how the function is to compute its value (an element of the range) given its arguments (an element of the domain). QuadraticEquation may seem a bit unusual since its range is a set of functions, but both the domain and the range of a function can be any set of objects, so sets of functions qualify.

Recall that in the last section we said that we could compose binary relations to derive new relations. Clearly, since functions are just special kinds of binary relations, if we can compose binary relations we can certainly compose binary functions. Because a function returns a unique value for each argument, it generally makes a lot more sense to compose functions than it does relations, and you'll see that although we rarely compose relations that aren't functions, we compose functions all the time. So, following our definition above for relations, we define the ***composition of two functions*** $F_1 \subseteq A \times B$ and $F_2 \subseteq B \times C$, written $F_2 \circ F_1$ is $\{(a,c) : \exists b\ (a, b) \in F_1$ and $(b, c) \in F_2\}$. Notice that the composition of two functions must necessarily also be a function. We mentioned above that there is sometimes confusion about the order in which relations (and now functions) should be applied when they are composed. To avoid this problem, let's introduce a new notation F(G(x)). We use the parentheses here to indicate function application, just as we did above. So this notation is clear. Apply F to the result of first applying G to x. This notation reads right to left as does our definition of the $\circ$ notation.

A function is a special kind of a relation (one in which each element of the domain occurs precisely once). There are also special kinds of functions:

A function f : D → R is ***total*** if it is defined for every element of D (i.e., every element of D is related to some element of R). The standard mathematical definition of a function requires totality. The reason we haven't done that here is that, as we pursue the idea of "computable functions", we'll see that there are total functions whose domains cannot be effectively defined (for example, the set of C programs that always halt). Thus it is useful to expand the definition of the function's domain (e.g., to the set of all C programs) and acknowledge that if the function is applied to certain elements of the domain (e.g., programs that don't halt), its value will be undefined. We call this broader class of functions (which does include the total functions as a subset) the set of ***partial*** functions. For the rest of our discussion in this introductory unit, we will consider only total functions, but be prepared for the introduction of partial functions later.

A function f : D → R is ***one to one*** if no element of the range occurs more than once. In other words, no two elements of the domain map to the same element of the range. Succ is one to one. For example, the only number to which we can apply Succ and derive 2 is 1. QuadraticEquation is also one to one. But + isn't. For example, both +(2,3) and +(4,1) equal 5.

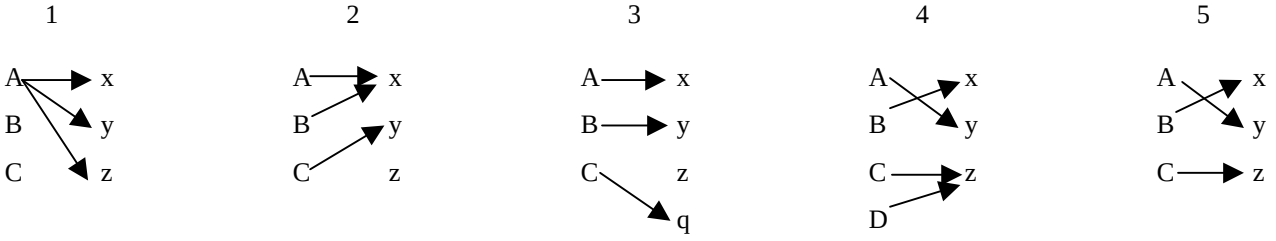
A function f : D → R is ***onto*** if every element of R is the value of some element of D. Another way to think of this is that a function is onto if all of the elements of the range are "covered" by the function. As we defined it above, Succ is onto. But let's define a different function Succ' on the natural numbers (rather than the integers). So we define

Succ' : N → N.

Succ' is not onto because there is no natural number i such that Succ'(i) = 0.

The easiest way to envision the differences between an arbitrary relation, a function, a one to one function and an onto function is to make two columns (the first for the domain and the second for the range) and think about the sort of matching problems you probably had on tests in elementary school.

Let's consider the following five matching problems and let's look at various ways of relating the elements of column 1 (the domain) to the elements of column 2 (the range):



The relationship in example 1 is a relation but it is not a function, since there are three values associated A. The second example is a function since, for each object in the first column, there is a single value in the second column. But this function is neither one to one (because x is derived from both A and B) nor onto (because z can't be derived from anything). The third example is a function that is one to one (because no element of the second column is related to more than one element of the first column). But it still isn't onto because z has been skipped: nothing in the first column derives it. The fourth example is a function that is onto (since every element of column two has an arrow coming into it), but it isn't one to one, since z is derived from both C and D. The fifth and final example is a function that is both one to one and onto. By the way, see if you can modify either example 3 or example 4 to make them both one to one and onto. You're not allowed to change the number of elements in either column, just the arrows. You'll notice that you can't do it. In order for a function to be both one to one and onto, there must be equal numbers of elements in the domain and the range.

The ***inverse*** of a binary relation R is simply the set of ordered pairs in R with the elements of each pair reversed. Formally, if $R \subseteq A \times B$, then $R^{-1} \subseteq B \times A = \{(b, a): (a, b) \in R\}$. If a relation is a way of associating with each element of A a corresponding element of B, then think of its inverse as a way of associating with elements of B their corresponding elements in A. Every relation has an inverse. Every function also has an inverse, but that inverse may not also be a function. For example, look again at example two of the matching problems above. Although it is a function, its inverse is not. Given the argument x, should we return the value A or B? Now consider example 3. Its inverse is also not a (total) function, since there is no value to be returned for the argument z. Example four has the same problem example

two does. Now look at example five. Its inverse is a function. Whenever a function is both one to one and onto, its inverse will also be a function and that function will be both one to one and onto.

Inverses are useful. When a function has an inverse, it means that we can move back and forth between columns one and two without loss of information. Look again at example five. We can think of ourselves as operating in the {A, B, C} universe or in the {x, y, z} universe interchangeably since we have a well defined way to move from one to the other. And if we move from column one to column two and then back, we'll be exactly where we started. Functions with inverses (alternatively, functions that are both one to one and onto) are called ***bijections***. And they may be used to define ***isomorphisms*** between sets, i.e., formal correspondences between the elements of two sets, often with the additional requirement that some key structure be preserved. We'll use this idea a lot. For example, there exists an isomorphism between the set of states of a finite state machine and a particular set of sets of input strings that could be fed to the machine. In this isomorphism, each state is associated with precisely the set of strings that drive the machine to that state.

3 Binary Relations on a Single Set

Although it makes sense to talk about n-ary relations, for arbitrary values of n (and we have), it turns out that the most useful relations are often binary ones -- ones where n is two. In fact, we can make a further claim about some of the most useful relations we'll work with: they involve just a single set. So instead of being subsets of $A \times B$, for arbitrary values of A and B, they are subsets of $A \times A$, for some particular set of A of interest. So let's spend some additional time looking at this restricted class of relations.

3.1 Representing Binary Relations on a Single Set

If we're going to work with some binary relation R, and, in particular, if we are going to compute with it, we need some way to represent the relation. We have several choices. We could:

1) List the elements of R.
2) Encode R as a computational procedure. There are at least two ways in which a computational procedure can define R. It may:
   a) enumerate the elements of R, or
   b) return a boolean value, when given an ordered pair. True means that the pair is in R; False means it is not.
3) Encode R as a directed graph.
4) Encode R as an ***incidence matrix***.

For example, consider the mother-of relation M in a family in which Doreen is the mother of Ann, Ann is the mother of Catherine, and Catherine is the mother of Allison. To exploit approach 1, we just write

M = {(Doreen, Ann), (Ann, Catherine), (Catherine, Allison)}.

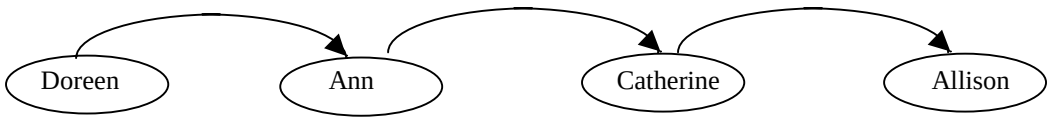
Clearly, this approach only works for finite relations.

The second approach simply requires code appropriate to the particular relation we're dealing with. One appeal of this approach is that it works for both finite and infinite relations, although, of course, a program that enumerates elements of an infinite relation will never halt.

Next we consider approach 3. Assuming that we are working with a finite relation $R \subseteq A \times A$, we can build a directed graph to represent R as follows:

1) Construct a set of nodes, one for each element of A that appears in any element of R.
2) For each ordered pair in R, draw an edge from the first element of the pair to the second.

The following directed graph represents our example relation M defined above:



And, finally, approach 4: Again assuming a finite relation $R \subseteq A \times A$, we can build an incidence matrix to represent R as follows:

1) Construct a square boolean matrix S whose number of rows and columns equals the number of elements of A that appear in any element of R.
2) Label one row and one column for each such element of A.
3) For each element (p, q) of R, set S(p, q) to 1(or True). Set all other elements of S to 0 (or False).

The following boolean matrix represents our example relation M defined above:

	Doreen	Ann	Catherine	Allison
Doreen	0	1	0	0
Ann	0	0	1	0
Catherine	0	0	0	1
Allison	0	0	0	0

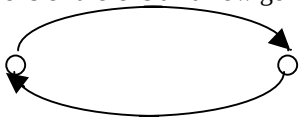
3.2 Properties of Relations

Many useful binary relations have some kind of structure. For example, it might be the case that every element of the underlying set is related to itself. Or it might happen that if a is related to b, then b must necessarily be related to a. There's one special kind of relation, called an equivalence relation that is particularly useful. But before we can define it, we need first to define each of the individual properties that equivalence relations possess.

A relation $R \subseteq A \times A$ is ***reflexive*** if, for each $a \in A$, $(a, a) \in R$. In other words a relation R on the set A is reflexive if every element of A is related to itself. For example, consider the relation Address defined as "lives at same address as". Address is a relation over a set of people. Clearly every person lives at the same address as him or herself, so Address is reflexive. So is the Less than or equal to relation on the integers. Every integer is Less than or equal to itself. But the Less than relation is not reflexive: in fact no number is Less than itself. Both the directed graph and the matrix representations make it easy to tell if a relation is reflexive. In the graph representation, every node will have an edge looping back to itself. In the graph representation, there will be ones along the major diagonal:

1		
	1	
		1

A relation $R \subseteq A \times A$ is ***symmetric*** if, whenever $(a, b) \in R$, so is (b, a). In other words, if a is related to b, then b is related to a. The Address relation we described above is symmetric. If Joe lives with Ann, then Ann lives with Joe. The Less than or equal relation is not symmetric (since, for example, $2 \leq 3$, but it is not true that $3 \leq 2$). The graph representation of a symmetric relation has the property that between any two nodes, either there is an arrow going in both directions or there is an arrow going in neither direction. So we get graphs with components that look like this:



If we choose the matrix representation, we will end up with a symmetric matrix (i.e., if you flip it on its major diagonal, you'll get the same matrix back again). In other words, if we have a matrix with 1's wherever there is a number in the following matrix, then there must also be 1's in all the squares marked with an *:

	*	*		
1				3
2				
	*			

A relation $R \subseteq A \times A$ is ***antisymmetric*** if, whenever $(a, b) \in R$ and $a \neq b$, then $(b, a) \notin R$ The Mother-of relation we described above is antisymmetric: if Ann is the mother of Catherine, then one thing we know for sure is that Catherine is not also the mother of Ann. Our Address relation is clearly not antisymmetric, since it is symmetric. There are, however, relations that are neither symmetric nor antisymmetric. For example, the Likes relation on the set of people: If Joe likes Bob, then it is possible that Bob likes Joe, but it is also possible that he doesn't.

A relation $R \subseteq A \times A$ is ***transitive*** if, whenever $(a, b) \in R$ and $(b, c) \in R$, $(a, c) \in R$. A simple example of a transitive relation is Less than. Address is another one: if Joe lives with Ann and Ann lives with Mark, then Joe lives with Mark. Mother-of is not transitive. But if we change it slightly to Ancestor-of, then we get a transitive relation. If Doreen is an ancestor of Ann and Ann is an ancestor of Catherine, then Doreen is an ancestor of Catherine.

The three properties of reflexivity, symmetry, and transitivity are almost logically independent of each other. We can find simple, possibly useful relationships with seven of the eight possible combinations of these properties:

	Domain	Example
None of the properties	people	Mother-of
Just reflexive	people	Would-recognize-picture-of
Just symmetric	people	Has-ever-been-married-to
Just transitive	people	Ancestor-of
Reflexive and symmetric	people	Hangs-out-with (assuming we can say one hangs out with oneself)
Reflexive and transitive	numbers	Less than or equal to
Symmetric and transitive		
All three	numbers	Equality
	people	Address

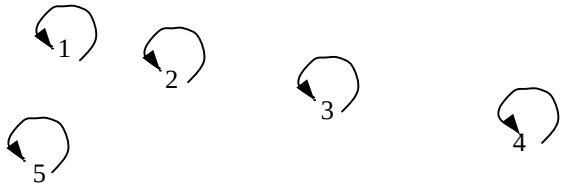
To see why we can't find a good example of a relation that is symmetric and transitive but not reflexive, consider a simple relation R on {1, 2, 3, 4}. As soon as R contains a single element that relates two unequal objects (e.g., (1, 2)), it must, for symmetry, contain the matching element (2, 1). So now we have R = {(1, 2), (2, 1)}. To make R transitive, we must add (1, 1). But that also makes R reflexive.

3.3 Equivalence Relations

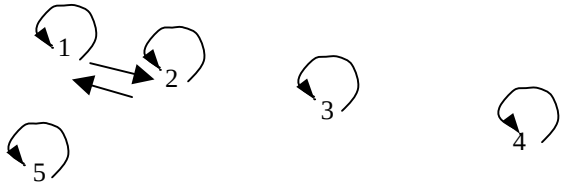
Although all the combinations we just described are possible, one combination is of such great importance that we give it a special name. A relation is an ***equivalence relation*** if it is reflexive, symmetric and transitive. Equality (for numbers, strings, or whatever) is an equivalence relation (what a surprise, given the name). So is our Address (lives at same address) relation.

Equality is a very special sort of equivalence relation because it relates an object only to itself. It doesn't help us much to carve up a large set into useful subsets. But in fact, equivalence relations are an incredibly useful way to carve up a set. Why? Let's look at a set P, with five elements, which we can draw as a set of nodes as follows:

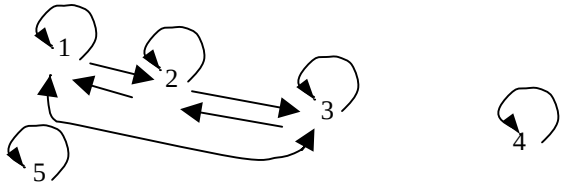
Now let's build an equivalence relation E on P. The first thing we have to do is to relate each node to itself, in order to make the relation reflexive. So we've now got:



Now let's add one additional element (1,2). As soon as we do that, we must also add (2,1), since E must be symmetric. So now we've got:



Suppose we now add (2,3). We must also add (3,2) to maintain symmetry. In addition, because we have (1, 2) and (2, 3), we must create (1,3) for transitivity. And then we need (3, 1) to restore symmetry. That gives us



Notice what happened here. As soon as we related 3 to 2, we were also forced to relate 3 to 1. If we hadn't, we would no longer have had an equivalence relation. See what happens now if you add (3, 4) to E.

What we've seen in this example is that an equivalence relation R on a set S carves S up into a set of clusters, which we'll call ***equivalence classes***. This set of equivalence classes has the following key property:

For any s, t $\in$ S, if s $\in$ $Class_i$ and (s, t) $\in$ R, then t $\in$ $Class_i$.

In other words, all elements of S that are related under R are in the same equivalence class. To describe equivalence classes, we'll use the notation [a] to mean the equivalence class to which a belongs. Or we may just write [description], where description is some clear property shared by all the members of the class. Notice that in general there may be lots of different ways to describe the same equivalence class. In our example, for instance, [1], [2], and [3] are different names for the same equivalence class, which includes the elements 1, 2, and 3. In this example, there are two other equivalence classes as well: [4] and [5].

It is possible to prove that if R is an equivalence relation on a nonempty set A then the equivalence classes of R constitute a partition of A. Recall that Π is a partition of a set A if and only if (a) no element of Π is empty; (b) all members of Π are disjoint; and (c) $\bigcup \Pi = A$. In other words, if we want to take a set A and carve it up into a set of subsets, an equivalence relation is a good way to do it.

For example, our Address relation carves up a set of people into subsets of people who live together. Let's look at some more examples:

- Let A be the set of all strings of letters. Let SameLength $\subseteq A \times A$ relate strings whose lengths are the same. SameLength is an equivalence relation that carves up the universe of all strings into a collection of subsets, one for each natural number (i.e., strings of length 0, strings of length 1, etc.).

- Let Z be the set of integers. Let EqualMod3 $\subseteq Z \times Z$ relate integers that have the same remainder when divided by 3. EqualMod3 has three equivalence classes, [0], [1], and [2]. [0] includes 0, 3, 6, etc.

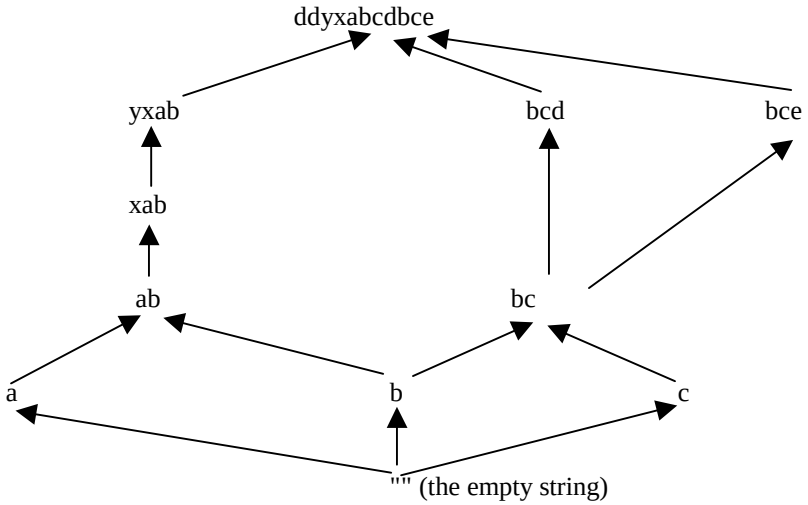
- Let CP be the set of C programs, each of which accepts an input of variable length. We'll call the length of any specific input n. Let SameComplexity $\subseteq CP \times CP$ relate two programs if their running-time complexity is the same. More specifically, $(c_1, c_2) \in$ SameComplexity precisely in case:
  $\exists m_1, m_2, k\ [\forall n > k, \text{RunningTime}(c_1) \leq m_1 * \text{RunningTime}(c_2)$ AND $\text{RunningTime}(c_2) \leq m_2 * \text{RunningTime}(c_1)]$

Not every relation that connects "similar" things is an equivalence relation. For example, consider SimilarCost(x, y), which holds if the price of x is within $1 of the price of y. Suppose A costs $10, B costs $10.50, and C costs $11.25. Then SimilarCost(A, B) and SimilarCost(B, C), but not SimilarCost(A, C). So SimilarCost is not transitive, although it is reflexive and symmetric.

3.4 Orderings

Important as equivalence relations are, they're not the only special kind of relation worth mentioning. Let's consider two more.

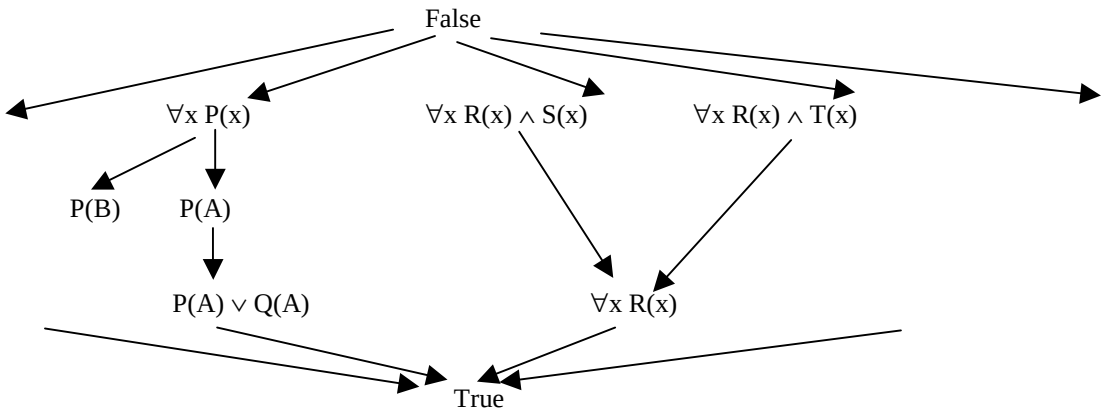
A ***partial order*** is a relation that is reflexive, antisymmetric, and transitive. If we write out any partial order as a graph, we'll see a structure like the following one for the relation SubstringOf. Notice that in order to make the graph relatively easy to read, we'll adopt the convention that we don't write in the links that are required by reflexivity and transitivity. But, of course, they are there in the relation itself:



Read an arrow from x to y as meaning that (x, y) is an element of the relation. So, in this example, "a" is a substring of "ab", which is a substring of "xab", and so forth. Note that in a partial order, it is often the case that there are some elements (such as "ab" and "bc") that are not related to each other at all (since neither is a substring of the other).

Sometimes a partial order on a domain D defines a mininal and/or a maximal element. In this example, there is a minimal element, the empty string, which is a substring of every string. There appears from this picture to be a maximal element, but that's just because we drew only a tiny piece of the graph. Every string is a substring of some longer string, so there is in fact no maximal element.

Let's consider another example based on the ***subsumption*** relation between pairs of logical expressions. A logical expression A subsumes another expression B iff (if and only if), whenever A is true B must be true regardless of the values assigned to the variables and functions of A and B. For example: $\forall x\ P(x)$ subsumes $P(A)$, since, regardless of what the predicate P is and independently of any axioms we have about it, and regardless of what object A represents, if $\forall x\ P(x)$ is true, then $P(A)$ must be true. Why is this a useful notion? Suppose we're building a theorem proving or reasoning program. If we already know $\forall x\ P(x)$, and we are then told $P(A)$, we can throw away this new fact. It doesn't add to our knowledge (except perhaps to focus our attention on the object A) since it is subsumed by something we already knew. A small piece of the subsumption relation on logical expressions is shown in the following graph. Notice that now there is a maximal element, False, which subsumes everything (in other words, if we have the assertion False in our knowledge base, we have a contradiction even if we know nothing else). There is also a minimal element, True, which tells us nothing.



A ***total order*** $R \subseteq A \times A$ is a partial order that has the additional property that $\forall a, b \in A$, either (a, b) or $(b, a) \in R$. In other words, every pair of elements must be related to each other one way or another. The classic example of a total order is $\leq$ (or $\geq$, if you prefer) on the integers. The $\leq$ relation is reflexive since every integer is equal to itself. It's antisymmetric since if $a \leq b$ and $a \neq b$, then for sure it is not also true that $b \leq a$. It's transitive: if $a \leq b$ and $b \leq c$, then $a \leq c$. And, given any two integers a and b, either $a \leq b$ or $b \leq a$. If we draw any total order as a graph, we'll get something that looks like this (again without the reflexive and transitive links shown):

This is only a tiny piece of the graph, of course. It continues infinitely in both directions. But notice that, unlike our earlier examples of partial orders, there is no splitting in this graph. For every pair of elements, one is above and one is below.

4 Important Properties of Binary Functions

Any relation that uniquely maps from all elements of its domain to elements of its range is a function. The two sets involved can be anything and the mapping can be arbitrary. However, most of the functions we actually care about behave in some sort of regular fashion. It is useful to articulate a set of properties that many of the functions that we'll study have. When these properties are true of a function, or a set of functions, they give us techniques for proving additional properties of the objects involved. In the following definitions, we'll consider an arbitrary binary function # defined over a set we'll call A with elements we'll call a, b, and c. As examples, we'll consider functions whose actual domains are sets, integers, strings, and boolean expressions.

A binary function # is ***commutative*** iff $\forall a,b \;\; a \# b = b \# a$

Examples:		
	$a + b = b + a$	integer addition
	$a \cap b = b \cap a$	set intersection
	a AND b = b AND a	boolean and

A binary function # is ***associative*** iff $\forall a,b,c \;\; (a \# b) \# c = a \# (b \# c)$

Examples:		
	$(a + b) + c = a + (b + c)$	integer addition
	$(a \cap b) \cap c = a \cap (b \cap c)$	set intersection
	(a AND b) AND c = a AND (b AND c)	boolean and
	$(a \parallel b) \parallel c = a \parallel (b \parallel c)$	string concatenation

A binary function # is ***idempotent*** iff $\forall a \;\; a \# a = a.$

Examples:		
	min(a, a) = a	integer min
	$a \cap a = a$	set intersection
	a AND a = a	boolean and

The ***distributivity*** property relates two binary functions: A function # distributes over another function ! iff

$\forall a,b,c \;\; a \# (b \,!\, c) = (a \# b) \,!\, (a \# c)$ and $(b \,!\, c) \# a = (b \# a) \,!\, (c \# a)$

Examples:		
	$a * (b + c) = (a * b) + (a * c)$	integer multiplication over addition
	$a \cup (b \cap c) = (a \cup b) \cap (a \cup c)$	set union over intersection
	a AND (b OR c)=(a AND b) OR (a AND c)	boolean AND over OR

The ***absorption laws*** also relate two binary functions to each other: A function # absorbs another function ! iff

$\forall a,b \;\; a \# (a \,!\, b) = a$

Examples:		
	$a \cap (a \cup b) = a$	set intersection absorbs union
	a OR (a AND b) = a	boolean OR absorbs AND

It is often the case that when a function is defined over some set A, there are special elements of A that have particular properties with respect to that function. In particular, it is worth defining what it means to be an identity and to be a zero:

An element a is an ***identity*** for the function # iff $\forall x \in A$, $x \# a = x$ and $a \# x = x$

Examples:		
	$b * 1 = b$	1 is an identity for integer multiplication
	$b + 0 = b$	0 is an identity for integer addition
	$b \cup \varnothing = b$	$\varnothing$ is an identity for set union
	b OR False = b	False is an identity for boolean OR
	b ‖ "" = b	"" is an identity for string concatenation

Sometimes it is useful to differentiate between a right identity (one that satisfies the first requirement above) and a left identity (one that satisfies the second requirement above). But for all the functions we'll be concerned with, if there is a left identity, it is also a right identity and vice versa, so we will talk simply about an identity.

An element a is a ***zero*** for the function # iff $\forall x \in A$, $x \# a = a$ and $a \# x = a$

Examples:		
	$b * 0 = 0$	0 is a zero for integer multiplication
	$b \cap \varnothing = \varnothing$	$\varnothing$ is a zero for set intersection
	b AND FALSE = FALSE	FALSE is a zero for boolean AND

Just as with identities, it is sometimes useful to distinguish between left and right zeros, but we won't need to.

Although we're focusing here on binary functions, there's one important property that unary functions may have that is worth mentioning here:

A unary function % is a ***self inverse*** iff $\forall x \;\; \%(\%x)) = x$. In other words, if we compose the function with itself (apply it twice), we get back the original argument. Note that this is not the same as saying that the function is its own inverse. In most of the cases we'll consider (including the examples given here), it is not. A single application of the function produces a new value, but if we apply the function a second time, we get back to where we started.

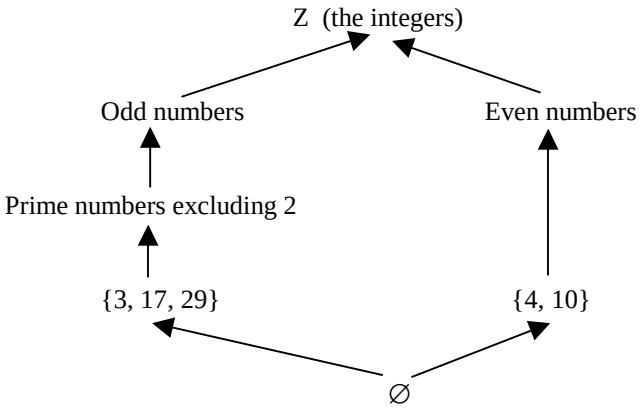
Examples:	$-(-(a)) = a$	Multiplying by -1 is a self inverse for integers
	$\frac{1}{(1/a)}$	Dividing into 1 is a self inverse for integers
	$\overline{\overline{a}} = a$	Complement is a self inverse for sets
	$\neg(\neg\, a) = a$	Negation is a self inverse for booleans
	$(a^R)^R = a$	Reversal is a self inverse for strings

5 Relations and Functions on Sets

In the last two sections, we explored various useful properties of relations and functions. With those tools in hand, let's revisit the basic relations and functions on sets.

5.1 Relations

We have defined two relations on sets: subset and proper subset. What can we say about them? Subset is a partial order, since it is reflexive (every set is a subset of itself), transitive (if $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$) and antisymmetric (if $A \subseteq B$ and $A \neq B$, then it must not be true that $B \subseteq A$). For example, we see that the subset relation imposes the following partial order if you read each arrow as "is a subset of":



What about proper subset? It is not a partial order since it is not reflexive.

5.2 Functions

All of the functional properties we defined above apply in one way or another to the functions we have defined on sets. Further, as we saw above, there some set functions have a zero or an identity. We'll summarize here (without proof) the most useful properties that hold for the functions we have defined on sets:

Commutativity	$A \cup B = B \cup A$
	$A \cap B = B \cap A$
Associativity	$(A \cup B) \cup C = A \cup (B \cup C)$
	$(A \cap B) \cap C = A \cap (B \cap C)$

Idempotency	$A \cup A = A$ $A \cap A = A$
Distributivity	$(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$ $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$
Absorption	$(A \cup B) \cap A = A$ $(A \cap B) \cup A = A$
Identity	$A \cup \varnothing = A$
Zero	$A \cap \varnothing = \varnothing$
Self Inverse	$\bar{\bar{A}} = A$

In addition, we will want to make use of the following theorems that can be proven to apply specifically to sets and their operations (as well as to boolean expressions):

De Morgan's laws	$\overline{A \cup B} = \bar{A} \cap \bar{B}$ $\overline{A \cap B} = \bar{A} \cup \bar{B}$

6 Proving Properties of Sets

A great deal of what we do when we build a theory about some domain is to prove that various sets of objects in that domain are equal. For example, in our study of automata theory, we are going to want to prove assertions such as these:

- The set of strings defined by some regular expression E is identical to the set of strings defined by some second regular expression E'.
- The set of strings that will be accepted by some given finite state automaton M is the same as the set of strings that will be accepted by some new finite state automaton M' that is smaller than M.
- The set of languages that can be defined using regular expressions is the same as the set of languages that can be accepted by a finite state automaton.
- The set of problems that can be solved by a Turing Machine with a single tape is the same as the set of problems that can be solved by a Turing Machine with any finite number of tapes.

So we become very interested in the question, "How does one prove that two sets are identical"? There are lots of ways and many of them require special techniques that apply in specific domains. But it's worth mentioning two very general approaches here.

6.1 Using Set Identities and the Definitions of the Functions on Sets

Sometimes we want to compare apples to apples. We may, for example, want to prove that two sets of strings are identical, even though they may have been derived differently. In this case, one approach is to use the set identity theorems that we enumerated in the last section. Suppose, for example, that we want to prove that

$$A \cup (B \cap (A \cap C)) = A$$

We can prove this as follows:

$A \cup (B \cap (A \cap C))$	$= (A \cup B) \cap (A \cup (A \cap C))$	Distributivity
	$= (A \cup B) \cap ((A \cap C) \cup A)$	Commutativity
	$= (A \cup B) \cap A$	Absorption
	$= A$	Absorption

Sometimes, even when we're comparing apples to apples, the theorems we've listed aren't enough. In these cases, we need to use the definitions of the operators. Suppose, for example, that we want to prove that

$$A - B = A \cap \bar{B}$$

We can prove this as follows (where U stands for the Universe with respect to which we take complement):

$$
\begin{aligned}
A - B &= \{x : x \in A \text{ and } x \notin B\} \\
&= \{x : x \in A \text{ and } (x \in U \text{ and } x \notin B)\} \\
&= \{x : x \in A \text{ and } x \in U - B\} \\
&= \{x : x \in A \text{ and } x \in \overline{B}\} \\
&= A \cap \overline{B}
\end{aligned}
$$

6.2 Showing Two Sets are the Same by Showing that Each is a Subset of the Other

Sometimes, though, our problem is more complex. We may need to compare apples to oranges, by which I mean that we are comparing sets that aren't even defined in the same terms. For example, we will want to be able to prove that A: the set of languages that can be defined using regular expressions is the same as B: the set of languages that can be accepted by a finite state automaton. This seems very hard: Regular expressions look like

$$a^* (b \cup ba)^*$$

Finite state machines are a collection of states and rules for moving from one state to another. How can we possibly prove that these A and B are the same set? The answer is that we can show that the two sets are equal by showing that each is a subset of the other. For example, in the case of the regular expressions and the finite state machines, we will show first that, given a regular expression, we can construct a finite state machine that accepts exactly the strings that the regular expression describes. That gives us $A \subseteq B$. But there might still be some finite state machines that don't correspond to any regular expressions. So we then show that, given a finite state machine, we can construct a regular expression that defines exactly the same strings that the machine accepts. That gives us $B \subseteq A$. The final step is to exploit the fact that

$$A \subseteq B \text{ and } B \subseteq A \Rightarrow A = B$$

7 Cardinality of Sets

It seems natural to ask, given some set A, "What is the size of A?" or "How many elements does A contain?" In fact, we've been doing that informally. We'll now introduce formal techniques for discussing exactly what we mean by the size of a set. We'll use the term ***cardinality*** to describe the way we answer such questions. So we'll reply that the cardinality of A is X, for some appropriate value of X. For simple cases, determining the value of X is straightforward. In other cases, it can get quite complicated. For our purposes, however, we can get by with three different kinds of answers: a natural number (if A is finite), "countably infinite" (if A has the same number of elements as there are integers), and "uncountably infinite" (if A has more elements than there are integers).

We write the cardinality of a set A as $|A|$.

A set A is ***finite*** and has cardinality $n \in N$ (the natural numbers) if either $A = \varnothing$ or there is a bijection from A to $\{1, 2, \ldots n\}$, for some value of n. In other words, a set is finite if either it is empty or there exists a one-to-one and onto mapping from it to a subset of the positive integers. Or, alternatively, a set is finite if we can count its elements and finish. The cardinality of a finite set is simply a natural number whose value is the number of elements in the set.

A set is ***infinite*** if it is not finite. The question now is, "Are all infinite sets the same size?" The answer is no. And we don't have to venture far to find examples of infinite sets that are not the same size. So we need some way to describe the cardinality of infinite sets. To do this, we need to define a set of numbers we'll call the cardinal numbers. We'll use these numbers as our measure of the size of sets. Initially, we'll define all the natural numbers to be cardinal numbers. That lets us describe the cardinality of finite sets. Now we need to add new cardinal numbers to describe the cardinality of infinite sets.

Let's start with a simple infinite set N, the natural numbers. We need a new cardinal number to describe the (infinite) number of natural numbers that there are. Following Cantor, we'll call this number $\aleph_0$. (Read this as "aleph null". Aleph is the first symbol of the Hebrew alphabet.)

Next, we'll say that any other set that contains the same number of members as N does also has cardinality $\aleph_0$. We'll also call a set with cardinality $\aleph_0$ ***countably infinite***. And one more definition: A set is ***countable*** if it is either finite or countably infinite.

To show that a set has cardinality $\aleph_0$, we need to show that there is a bijection between it and N. The existence of such a bijection proves that the two sets have the same number of elements. For example, the set E of even natural numbers has cardinality $\aleph_0$. To prove this, we offer the bijection:

Even : E $\rightarrow$ N
Even(x) = x/2

So we have the following mapping from E to N:

E	N
0	0
2	1
4	2
6	3
…	…

This one was easy. The bijection was obvious. Sometimes it's less so. In harder cases, a good way to think about the problem of finding a bijection from some set A to N is that we need to find an enumeration of A. An ***enumeration*** of A is simply a list of the elements of A in some order. Of course, if A is infinite, the list will be infinite, but as long as we can guarantee that every element of A will show up eventually, we have an enumeration. But what is an enumeration? It is in fact a bijection from A to the positive integers, since there is a first element, a second one, a third one, and so forth. Of course, what we need is a bijection to N, so we just subtract one. Thus if we can devise a technique for enumerating the elements of A, then our bijection to N is simply

Enum : A $\rightarrow$ N
Enum(x) = x's position in the enumeration - 1

Let's consider an example of this technique:

Theorem: The union of a countably infinite number of countably infinite sets is countably infinite.

To prove this theorem, we need a way to enumerate all the elements of the union. The simplest thing to do would be to start by dumping in all the elements of the first set, then all the elements of the second, etc. But, since the first set is infinite, we'll never get around to considering any of the elements of the other sets. So we need another technique. If we had a finite number of sets to consider, we could take the first element from each, then the second element from each, and so forth. But we also have an infinite number of sets, so if we try that approach, we'll never get to the second element of any of the sets. So we follow the arrows as shown below. The numbers in the squares indicate the order in which we select elements for the enumeration. This process goes on forever, but it is systematic and it guarantees that, if we wait long enough, any element of any of the sets will eventually be enumerated.

	Set 1	Set 2	Set 3	Set 4	…
Element 1	1	3	4		
Element 2	2	5			
Element 3	6	8			
…	7				

It turns out that a lot of sets have cardinality $\aleph_0$. Some of them, like the even natural numbers, appear at first to contain fewer elements. Some of them, like the union of a countable number of countable sets, appear at first to be bigger. But in both cases there is a bijection between the elements of the set and the natural numbers, so the cardinality is $\aleph_0$.

However, this isn't true for every set. There are sets with more than $\aleph_0$ elements. There are more than $\aleph_0$ real numbers, for example. As another case, consider an arbitrary set S with cardinality $\aleph_0$. Now consider the power set of S (the set of all subsets of S). This set has cardinality greater than $\aleph_0$. To prove this, we need to show that there exists no bijection

between the power set of S and the integers. To do this, we will use a technique called ***diagonalization***. Diagonalization is a kind of proof by contradiction. It works as follows:

Let's start with the original countably infinite set S. We can enumerate the elements of S (since it's countable), so there's a first one, a second one, etc. Now we can represent each subset SS of S as a binary vector that contains one element for each element of the original set S. If SS contains element 1 of S, then the first element of its vector will be 1, otherwise 0. Similarly for all the other elements of S. Of course, since S is countably infinite, the length of each vector will also be countably infinite. Thus we might represent a particular subset SS of S as the vector:

Elem 1 of S	Elem 2 of S	Elem 3 of S	Elem 4 of S	Elem 5 of S	Elem 6 of S	……
1	0	0	1	1	0	……

Next, we observe that if the power set P of S were countably infinite, then there would be an enumeration of it that put its elements in one to one correspondence with the natural numbers. Suppose that enumeration were the following (where each row represents one element of P as described above. Ignore for the moment the numbers enclosed in parentheses.):

	Elem 1 of S	Elem 2 of S	Elem 3 of S	Elem 4 of S	Elem 5 of S	Elem 6 of S	…….
Elem 1 of P	1 (1)	0	0	0	0	0	…..
Elem 2 of P	0	1 (2)	0	0	0	0	…..
Elem 3 of P	1	1	0 (3)	0	0	0	…..
Elem 4 of P	0	0	1	0 (4)	0	0	…..
Elem 5 of P	1	0	1	0	0 (5)	0	…..
Elem 6 of P	1	1	1	0	0	0 (6)	…..

If this really is an enumeration of P, then it must contain all elements of P. But it doesn't. To prove that it doesn't, we will construct an element L ∈ P that is not on the list. To do this, consider the numbers in parentheses in the matrix above. Using them, we can construct L:

¬(1)	¬(2)	¬(3)	¬(4)	¬(5)	¬(6)	…..

What we mean by ¬(1) is that if (1) is a 1 then 0; if (1) is a 0, then 1. So we've constructed the representation for an element of P. It must be an element of P since it describes a possible subset of S. But we've built it so that it differs from the first element in the list above by whether or not it includes element 1 of S. It differs from the second element in the list above by whether or not it includes element 2 of S. And so forth. In the end, it must differ from every element in the list above in at least one place. Yet it is clearly an element of P. Thus we have a contradiction. The list above was not an enumeration of P. But since we made no assumptions about it, no enumeration of P can exist. In particular, if we try to fix the problem by simply adding our new element to the list, we can just turn around and do the same thing again and create yet another element that 's not on the list. Thus there are more than $\aleph_0$ elements in P. We'll say that sets with more than $\aleph_0$ elements are ***uncountably infinite***.

The real numbers are uncountably infinite. The proof that they are is very similar to the one we just did for the power set except that it's a bit tricky because, when we write out each number as an infinite sequence of digits (like we wrote out each set above as an infinite sequence of 0's and 1's), we have to consider the fact that several distinct sequences may represent the same number.

Not all uncountably infinite sets have the same cardinality. There is an infinite number of cardinal numbers. But we won't need any more. All the uncountably infinite sets we'll deal with (and probably all the ones you can even think of unless you keep taking power sets) have the same cardinality as the reals and the power set of a countably infinite set.

Thus to describe the cardinality of all the sets we'll consider, we will use one of the following:

- The natural numbers, which we'll use to count the number of elements of finite sets,

- $\aleph_0$, which is the cardinality of all countably infinite sets, and
- uncountable, which is the cardinality of any set with more than $\aleph_0$ members.

8 Closures

Imagine some set A and some property P. If we care about making sure that A has property P, we are likely to do the following:

1. Examine A for P. If it has property P, we're happy and we quit.
2. If it doesn't, then add to A the smallest number of additional elements required to satisfy P.

Let's consider some examples:

- Let A be a set of friends you're planning to invite to a party. Let P be "A should include everyone who is likely to find out about the party" (since we don't want to offend anyone). Let's assume that if you invite Bill and Bill has a friend Bob, then Bill may tell Bob about the party. This means that if you want A to satisfy P, then you have to invite not only your friends, but your friends' friends, and their friends, and so forth. If you move in a fairly closed circle, you may be able to satisfy P by adding a few people to the guest list. On the other hand, it's possible that you'd have to invite the whole city before P would be satisfied. It depends on the connectivity of the FriendsOf relation in your social setting. The problem is that whenever you add a new person to A, you have to turn around and look at that person's friends and consider whether there are any of them who are not already in A. If there are, they must be added, and so forth. There's one positive feature of this problem, however. Notice that there is a unique set that does satisfy P, given the initial set A. There aren't any choices to be made.
- Let A be a set of 6 people. Let P be "A can enter a baseball tournament". This problem is different from the last in two important ways. First, there is a clear limit to how many elements we have to add to A in order to satisfy P. We need 9 people and when we've got them we can stop. But notice that there is not a unique way to satisfy P (assuming that we know more than 9 people). Any way of adding 3 people to the set will work.
- Let A be the Address relation (which we defined earlier as "lives at same address as"). Since relations are sets, we should be able to treat Address just as we've treated the sets of people in our last two examples. We know that Address is an equivalence relation. So we'll let P be the property of being an equivalence relation (i.e., reflexive, symmetric, and transitive). But suppose we are only able to collect facts about living arrangements in a piecemeal fashion. For example, we may learn that Address contains {(Dave, Mary), (Sue, Pete), (John, Bill)}. Immediately we know, because Address must be reflexive, that it must also contain {(Dave, Dave), (Mary, Mary), (Sue, Sue), (Pete, Pete), (John, John), (Bill, Bill)}. And, since Address must also be symmetric it must contain {(Mary, Dave), (Pete, Sue), (Bill, John)}. Now suppose that we discover that Mary lives with Sue. We add {(Mary, Sue)}. To make Address symmetric again, we must add {(Sue, Mary)}. But now we also have to make it transitive by adding {(Dave, Sue), (Sue, Dave)}.
- Let A be the set of natural numbers. Let P be "the sum of any two elements of A is also in A." Now we've got a property that is already satisfied. The sum of any two natural numbers is a natural number. This time, we don't have to add anything to A to establish P.
- Let A be the set of natural numbers. Let P be "the quotient of any two elements of A is also in A." This time we have a problem. 3/5 is not a natural number. We can add elements to A to satisfy P. If we do, we end up with exactly the rational numbers.

In all of these cases, we're going to want to say that A is ***closed*** with respect to P if it possesses P. And, if we have to add elements to A in order to satisfy P, we'll call a smallest such expanded A that does satisfy P a ***closure*** of A with respect to P. What we need to do next is to define both of these terms more precisely .

8.1 Defining Closure

The first set of definitions of closure that we'll present is very general, although it does require that we can describe property P as an n-ary relation (for some value of n). We can use it to describe what we did in all but one of the examples above, although in a few cases it will be quite cumbersome to do so.

Let n be an integer greater than or equal to 1. Let R be an n-ary relation on a set D. Thus elements of R are of the form $(d_1, d_2, \ldots, d_n)$. We say that a subset S of D is ***closed under*** R if, whenever:

1. $d_1, d_2, \ldots d_{n-1} \in S$, (all of the first n-1 elements are already in the set S) and
2. $(d_1, d_2, \ldots d_{n-1}, d_n) \in R$ (the last element is related to the n-1 other elements via R)

it is also true that $d_n \in S$.

A set S' is a ***closure*** of S with respect to R (defined on D) iff:
1. $S \subseteq S'$,
2. S' is closed under R, and
3. $\forall T$ ($T \subseteq D$ and T is closed under R) $\Rightarrow |S'| \leq |T|$.

In other words, S' is a closure of S with respect to R if it is an extension of S that is closed under R and if there is no smaller set that also meets both of those requirements. Note that we can't say that S' must be the smallest set that will do the job, since we do not yet have any guarantee that there is a unique such smaller set (recall the softball example above).

These definitions of closure are a very natural way to describe our first example above. Drawing from a set A of people, you start with S equal to your friends. Then, to compute your invitee list, you simply take the closure of S with respect to the relation FriendOf, which will force you to add to A your friends' friends, their friends, and so forth.

Now consider our second example, the case of the baseball team. Here there is no relation R that specifies, if one or more people are already on the team, then some specific other person must also be on. The property we care about is a property of the team (set) as a whole and not a property of patterns of individuals (elements). Thus this example, although similar, is not formally an instance of closure as we have just defined it. This turns out to be significant and leads us to the following definition:

Any property that asserts that a set S is closed under some relation R is a ***closure property*** of S. It is possible to prove that if P is a closure property, as just defined, on a set A and S is a subset of A, then the closure of S with respect to R exists and is unique. In other words, there exists a unique minimal set S' that contains S and is closed under R. Of all of our examples above, the baseball example is the only one that cannot be described in the terms of our definition of a closure property. The theorem that we have just stated (without proof) guarantees, therefore, that it will be the only one that does not have a unique minimal solution.

The definitions that we have just provided also work to describe our third example, in which we want to compute the closure of a relation (since, after all, a relation is a set). All we have to do is to come up with relations that describe the properties of being reflexive, symmetric, and transitive. To help us see what those relations need to be, let's recall our definitions of symmetry, reflexivity, and transitivity:

- A relation $R \subseteq A \times A$ is ***reflexive*** if, for each $a \in A$, $(a, a) \in R$.
- A relation $R \subseteq A \times A$ is ***symmetric*** if, whenever $(a, b) \in R$, so is (b, a).
- A relation $R \subseteq A \times A$ is ***transitive*** if, whenever $(a, b) \in R$ and $(b, c) \in R$, $(a, c) \in R$.

Looking at these definitions, we can come up with three relations, Reflexivity, Symmetry, and Transitivity. All three are relations on relations, and they will enable us to define these three properties using the closure definitions we've given so far. All three definitions assume a base set A on which the relation we are interested is defined:

- $\forall a \in A$, $((a, a)) \in$ Reflexivity. Notice the double parentheses here. Reflexivity is a unary relation, where each element is itself an ordered pair. It doesn't really "relate" two elements. It is simply a list of ordered pairs. To see how it works to define reflexive closure, imagine a set A = {x, y}. Now suppose we start with a relation R on A = {(x, y)}. Clearly R isn't reflexive. And the Reflexivity relation tells us that it isn't because the reflexivity relation on A contains {((x, x)), ((y, y))}. This is a unary relation. So n, in the definition of closure, is 1. Consider the first element ((x, x)). We consider all the components before the nth (i.e., first) and see if they're in A. This means we consider the first zero components. Trivially, all zero of them are in A. So the nth (the first) must also be. This means that (x, x) must be in R. But it isn't. So to compute the closure of R under Reflexivity, we add it. Similarly for (y, y).
- $\forall a,b \in A$, $a \neq b \Rightarrow [((a, b), (b, a)) \in$ Symmetry]. This one is a lot easier. Again, suppose we start with a set A = {x, y} and a relation R on A = {(x, y)}. Clearly R isn't symmetric. And Symmetry tells us that. Symmetry on A = {((x, y), (y, x)), ((y, x), (x, y))}. But look at the first element of Symmetry. It tells us that for R to be closed, whenever (x, y) is in R, (y, x) must also be. But it isn't. To compute the closure of R under Symmetry, we must add it.

- $\forall a,b,c \in A, [a \neq b \wedge b \neq c] \Rightarrow [((a, b), (b, c), (a, c)) \in \text{Transitivity}]$. Now we will exploit a ternary relation. Whenever the first two elements of it are present in some relation R, then the third must also be if R is transitive. This time, let's start with a set A = {x, y, z} and a relation R on A = {(x, y), (y, z)}. Clearly R is not transitive. The Transitivity relation on A is {((x, y), (y, z), (x, z)), ((x, z), (z, y), (x, y)), ((y, x), (x, z), (y, z)), ((y, z), (z, x), (y, x)), ((z, x), (x, y), (z, y)), ((z, y), (y, x), (z, x))}. Look at the first element of it. Both of the first two components of it are in R. But the third isn't. To make R transitive, we must add it.

These definitions also work to enable us to describe the closure of the integers under division as the rationals. Following the definition, A is the set of rationals. S (a subset of A) is the integers and R is QuotientClosure, defined as:

- $\forall a,b,c \in A, [a/b = c] \Rightarrow [(a, b, c) \in \text{QuotientClosure}]$.

So we've got a quite general definition of closure. And it makes it possible to prove the existence of a unique closure for any set and any relation R. The only constraint is that this definition works only if we can define the property we care about as an n-ary relation for some finite n. There are cases of closure where this isn't possible, as we saw above, but we won't need to worry about them in this class.

So we don't really need any new definitions. We've offered a general definition of closure of a set (any set) under some relation (which is the way we use to define a property). But most of the cases of closure that we'll care about involve the special case of the closure of a binary relation given some property that may or may not be naturally describable as a relation. For example, one could argue that the relations we just described to define the properties of being reflexive, symmetric, and transitive are far from natural. Thus it will be useful to offer the following alternative definitions. Don't get confused though by the presence of two definitions. Except when we cannot specify our property P as a relation (and we won't need to deal with any such cases), these new definitions are simply special cases of the one we already have.

We say that a binary relation B on a set T is ***closed under*** property P if B possesses P. For example, LessThanOrEqual is closed under transitivity since it is transitive. Simple enough. Next:

Let B be a binary relation on a set T. A relation B' is a ***closure*** of B with respect to some property P iff:

1. $B \subseteq B'$,
2. B' is closed under P, and
3. There is no smaller relation B" that contains B and is closed under P.

So, for example, the transitive closure of B = {(1, 2), (2, 3)} is the smallest new relation B' that contains B but is transitive. So B' = {(1, 2), (2, 3), (1, 3)}.

You'll generally find it easier to use these definitions than our earlier ones. But keep in mind that, with the earlier definitions it is possible to prove the existence of a unique closure. Since we went through the process of defining reflexivity, symmetry, and transitivity using those definitions, we know that there always exists a unique reflexive, symmetric, and transitive closure for any binary relation. We can exploit that fact that the same time that we use the simpler definitions to help us find algorithms for computing those closures.

8.2 Computing Closures

Suppose we are given a set and a property and we want to compute the closure of the set with respect to the property. Let's consider two examples:

- Compute the symmetric closure of a binary relation B on a set A. This is trivial. Simply make sure that, for all elements x of A, $(x, x) \in B$.
- Compute the transitive closure of a binary relation B on a set A. This is harder. We can't just add a fixed number of elements to B and then quit. Every time we add a new element, such as (x, y), we have to look to see whether there's some element (y, z) so now we also have to add (x, z). And, similarly, we must check for any element (w, x) that would force us to add (w, y). If A is infinite, there is no guarantee that this process will ever terminate. Our theorem that guaranteed that a unique closure exists did not guarantee that it would contain a finite number of elements and thus be computable in a finite amount of time.

We can, however, guarantee that the transitive closure of any binary relation on a *finite* set is computable. How? A very simple approach is the following algorithm for computing the transitive closure of a binary relation B with N elements on a set A:

```
Set Trans = B;                                  /* Initially Trans is just the original relation.
/* We need to find all cases where (x, y) and (y, z) are in Trans.  Then we must insert (x, z) into Trans if
/*     it isn't already there.
Boolean AddedSomething = True;                  /* We'll keep going until we make one whole pass through
                                                   without adding any new elements to Trans.
while AddedSomething = True do
        AddedSomething = False;
        Xcounter := 0;
        Foreach element of Trans do
                xcounter := xcounter + 1;
                x = Trans[xcounter][1]          /* Pull out the first element of the current element of Trans
                y = Trans[xcounter][2]          /* Pull out the second element of the i'th element of Trans
                                                /* So if the first element of Trans is (p, q), then
                                                /* x = p and y = q the first time through.
                zcounter := 0;
                foreach element of Trans do
                        zcounter := zcounter + 1;
                        if Trans[zcounter][1] = y then do   /* We've found another element (y, ?) and we may need to
                                z = Trans[zcounter][2];                     /* add (x, ?) to Trans.
                                if (x, z) ∉ Trans then do   /* we have to add it
                                        Insert(Trans, (x, z))
                                        AddedSomething = True;
        end;    end;    end;    end;    end;
```

This algorithm works. Try it on some simple examples. But it's very inefficient. There are much more efficient algorithms. In particular, if we represent a relation as an incidence matrix, we can do a lot better. Using Warshall's algorithm, for example, we can find the transitive closure of a relation of n elements using $2n^3$ bit operations. For a description of that algorithm, see, for example, Kenneth Rosen, *Discrete Mathematics and its Applications*, McGraw-Hill.

9 Proof by Mathematical Induction

In the last section but one, as a sideline to our main discussion of cardinality, we presented diagonalization as a proof technique. In this section, we'll present one other very useful proof technique.

Mathematical induction is a technique for proving assertions about the set of positive integers. A proof by induction of assertion A about some set of positive integers greater than or equal to some specific value has two steps:

1. Prove that A holds for the smallest value we're concerned with. We'll call this value v. Generally v = 0 or 1, but sometimes A may hold only once we get past some initial unusual cases.
2. Prove that $\forall n \geq v$, $A(n) \Rightarrow A(n+1)$

We'll call A(n) the ***induction hypothesis***. Since we're trying to prove that $A(n) \Rightarrow A(n+1)$, we can assume the induction hypothesis as an axiom in our proof.

Let's do a simple example and use induction to prove that the sum of the first n odd positive integers is n^2. Notice that this appears to be true:

(n = 1) 1 = 1 = 1^2
(n = 2) 1 + 3 = 4 = 2^2
(n = 3) 1 + 3 + 5 = 9 = 3^2
(n = 4) 1 + 3 + 5 + 7 = 16 = 4^2, and so forth.

To prove it, we need to follow the two steps that we just described:

1. Let v = 1: $1 = 1^2$
2. Prove that, $\forall n \geq 0$,

$$(\sum_{i=1}^{n} \text{Odd}_i = n^2) \Rightarrow (\sum_{i=1}^{n+1} \text{Odd}_i = (n+1)^2)$$

To do this, we observe that the sum of the first n+1 odd integers is the sum of the first n of them plus the n+1'st, i.e.,

$$\begin{aligned} \sum_{i=1}^{n+1} \text{Odd}_i &= \sum_{i=1}^{n} \text{Odd}_i + \text{Odd}_{n+1} \\ &= n^2 + \text{Odd}_{n+1} \quad \text{(Using the induction hypothesis)} \\ &= n^2 + 2n + 1 \quad (\text{Odd}_{n+1} \text{ is } 2n + 1) \\ &= (n + 1)^2 \end{aligned}$$

Thus we have shown that the sum of the first n+1 odd integers must be equivalent to $(n+1)^2$ if it is known that the sum of the first n of them is equivalent to n^2.

Mathematical induction lets us prove properties of positive integers. But it also lets us prove properties of other things if the properties are described in terms of integers. For example, we could talk about the size of finite sets, or the length of finite strings. Let's do one with sets: For any finite set A, $|2^A| = 2^{|A|}$. In other words, the cardinality of the power set of A is 2 raised to the power of the cardinality of A. We'll prove this by induction on the number of elements of A (|A|). We follow the same two steps:

1. Let v = 0. So A is ∅, |A| = 0, and A's power set is {∅}, whose cardinality is $1 = 2^0 = 2^{|A|}$.
2. Prove that, $\forall n \geq 0$, if $|2^A| = 2^{|A|}$ is true for all sets A of cardinality n, then it is also true for all sets S of cardinality n+1. We do this as follows. Since $n \geq 0$, and any such S has n + 1 elements, S must have at least one element. Pick one and call it a. Now consider the set T that we get by removing a from S. |T| must be n. So, by the induction hypothesis (namely that $|2^A| = 2^{|A|}$ if |A| = n), our claim is true for T and we have $|2^T| = 2^{|T|}$. Now let's return to the power set of the original set S. It has two parts: those subsets that include a and those that don't. The second part is exactly 2^T, so we know that it has $2^{|T|} = 2^n$ elements. The first part (all the subsets that include a) is exactly all the subsets that don't include a with a added in). Since there are 2^n subsets that don't include a and there are the same number of them once we add a to each, we have that the total number of subsets of our original set S is 2^n (for the ones that don't include a) plus 2^n (for the ones that do include a), for a total of $2(2n) = 2^{n+1}$, which is exactly $2^{|S|}$.

Why does mathematical induction work? It relies on the ***well-ordering property*** of the integers, which states that every nonempty set of nonnegative integers has a least element. Let's see how that property assures us that mathematical induction is valid as a proof technique. We'll use the technique of proof by contradiction to show that, given the well-ordering property, mathematical induction must be valid. Once we have done an induction proof, we know that A(v) (where v is 0 or 1 or some other starting value) is true and we know that $\forall n \geq 0$, A(n) ⇒ A(n+1). What we're using the technique to enable us to claim is that, therefore, $\forall n \geq v$, A(n). Suppose the technique were not valid and there was a set S of nonnegative integers ≥ v for which A(n) is False. Then, by the well-ordering property, there is a smallest element in this set. Call it x. By definition, x must be equal to or greater than v. But it cannot actually be v because we proved A(v). So it must be greater than v. But now consider x - 1. Since x - 1 is less than x, it cannot be in S (since we chose x to be the smallest value in S). If it's not in S, then we know A(x - 1). But we proved that $\forall n \geq 0$, A(n) ⇒ A(n+1), so A(x - 1) ⇒ A(x). But we assumed ¬ A(x). So that assumption led us to a contradiction; thus it must be false.

Sometimes the principle of mathematical induction is stated in a slightly different but formally equivalent way:

1. Prove that A holds for the smallest value v with which we're concerned.
2. State the ***induction hypothesis*** H, which must be of the form, "There is some integer $n \geq v$ such that A is true for all integers k where $v \leq k \leq n$."
3. Prove that $(\forall n \geq v,\ A(n)) \Rightarrow A(n + 1)$. In other words prove that whenever A holds for all nonnegative integers starting with v, up to an including n, it must also hold for n + 1.

You can use whichever form of the technique is easiest for a particular problem.