1 Introduction: motivating examples

To motivate our study of discrete mathematics, let's see a couple of examples of mathematical ideas appearing in various areas of computer science.

Facebook Graph API

The rise of social networking in the 21st century has made it significantly easier for people to stay at home and pretend to make meaningful connections. But how do we model such connections—not just between people, but between those people and things they like, or companies they work for, or places they visit? We could use something like a "friend adjacency matrix", where each person on Earth is represented as a row/column and the state of "x being friends with y" is denoted by a 1 entry in row x and column y. However, it isn't exactly nice to deal with a $7500000000 \times 7500000000$ matrix. We could instead work under the assumption that not every person on Earth can be friends with every other person on Earth, thus saving us from having to maintain entries between one person and everyone else as we did in the adjacency matrix. Instead of representing a person's friend list as a 7500000000-entry vector, we could list every person in one set and the connections between every person in another (much smaller) set. This model lends itself well to the vertex/edge structure of a graph. We could even extend this model to include organizations, companies, pages, or places a person "likes" by adding that entity to our vertex set; as an added benefit, there is no need to modify the way we handle our edge set. Indeed, graphs constitute the underlying structure of Facebook, the world's largest social networking website. According to their documentation, Facebook's Graph API allows developers to "programmatically query data, post new stories, manage ads, upload photos, and perform a wide variety of other tasks". The Graph API uses vertices (called "nodes" in the documentation) to represent users, photos, pages, and so on, and edges denote relations between nodes; for instance, photos on a user's profile.

Bitcoin

If you've turned on a television or used the Internet in the past decade, you've likely heard about Bitcoin. Bitcoin is a digital currency that is distributed via a peer-to-peer network, rather than via a central authority like a bank. It is possibly the most well-known example of a cryptocurrency: a currency that uses cryptography to secure funds and to verify transactions. The security of Bitcoin is implemented using two technologies: public key cryptography and the blockchain. In a public key cryptography system, two users share information by encrypting and decrypting it with "keys"; essentially, keys are pairs of very large numbers. The public can encrypt messages intended for a single user with that user's public key, but only the user can decrypt the message using their private key. Therefore, once a message is encrypted, only the sender and the intended recipient will know what it says. The Bitcoin protocol uses public key cryptography in a slightly different way: keys are used not only to transmit information, but also to verify information. The person sending Bitcoin attaches the recipient's public key to the transaction and "signs" the transaction with their private key. The recipient can then prove both that they are the new owner of the Bitcoin (due to the recipient's public key being attached to the transaction) and that the sender was truly responsible for the transaction (by using the sender's public key against the private key used to "sign" the transaction). Since Bitcoin is a distributed system, how can we be sure that everyone has a consistent historical record of transactions? This is the job of the second technology: the blockchain. A blockchain is like a "transaction tree"; the root of the tree, called the genesis block, stores timestamp data and instructions on how to recreate the block, and other transaction blocks are added as children of a previous transaction. As a security measure, each block also contains a hashed version of the block preceding it to verify that it is located in the correct position within the blockchain.

Running time of algorithms

When a program is run on a computer, two of the most important considerations are how long it will take and how much memory it will require. Algorithm analysis deals with estimating

upper and lower bounds for their time complexity, or space complexity. The time complexity is measured as a function on input length. Because the exact running time of an algorithm often is a complex expression, the running time is estimated using *asymptotic analysis* (or, so called big-Oh notation). The time complexity is represented as a function on positive integers and asymptotic analysis is concerned with determining the behavior of the function with large argument values (i.e., with large size inputs). The crucial question is normally to determine whether the growth rate of the time used by an algorithm is exponential or polynomial.

Probabilistic algorithms

A *probabilistic algorithm* is designed to use the outcome of a random process. We can think that the algorithm has instructions to "flip a coin" and the result of the coin flip influences the subsequent execution and output of the algorithm. Certain types of problems seem more easiliy (i.e., more efficiently) solvable by probabilistic algorithms than by deterministic algorithms. For example, statisticians use random sampling to determine information about large populations where querying all individuals would be too time consuming. Randomness is useful in algorithms also in less obvious ways. For example, the well-known Rabin's primality testing algorithm uses probabilistic choices. Generally probabilistc algorithms are classified as Monte Carlo or Las Vegas algorithms. A Monte Carlo algorithm may produce an incorrect answer with small probability. The answer produced by a Las Vegas algorithm is always correct, but with small probability the algorithm may return an inconclusive answer (i.e., reply "don't know").

2 Sets: basic concepts

This section recalls material from CISC-102.

Sets are fundamental discrete structures on which all other discrete structures are built. Also the notions of *function* and *relation* are defined in terms of sets.

We begin by recalling basic notation and definitions on sets.

A *set* is an unordered collection of objects represented as a unit. A set may contain any types of objects: numbers, symbols and even other sets. The objects belonging to a set are called *elements* or *members* of the set.

One way of defining a set is to list it's elements inside braces

$$A = \{5, 8, 13, 21\}.$$

The order of describing the elements does not matter, and $\{5, 21, 8, 13\}$ is the same as the set A (this is what the term "unordered collection" refers to.) An element cannot belong to a set more than once, and $\{5, 5, 5, 8, 8, 13, 21, 21\}$ is again equal to the set A. Normally elements of a finite set are listed only once.

An *infinite set* contains infinitely many elements. We cannot write a list of all the elements of an infinite set so we sometimes use the "three dots" notation to mean "continue the sequence forever". The set of natural numbers is $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ and the set of integers is

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}.$$

It would be inconvenient to describe the set of rational numbers just using the "three dots" notation. When we want to describe a set containing elements satisfying some condition we write $\{x \mid \text{condition(s) on } x\}$. The set of rational numbers can be written as

$$\mathbb{Q} = \{\ \frac{m}{n} \mid m, n \in \mathbb{Z}, n \neq 0\ \}.$$

A set with zero elements is called the *empty set* and denoted $\emptyset$. A set with one element is called a *singleton*.

The *cardinality* (or size) of a finite set A, $|A|$, is the number of elements of A. For example,

$$|\{a_1, \ldots, a_n\}| = n, \quad n \in \mathbb{N}.$$

If A is not finite (e.g. $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$) we say that the cardinality of A is infinite. More generally it is possible to compare the sizes of infinite sets, but this is a topic not covered by our textbook.

2.1 Operations on sets and some notation

In the following let A and B be sets. Examples of the following operations and notation will be done in class.

The *intersection* of sets A and B, denoted $A \cap B$, is the set consisting of elements that are both in A and B. The *union* of sets A and B, denoted $A \cup B$, is the set consisting of all elements of A and of all elements of B. The elements of $A \cap B$ are listed only once in the union $A \cup B$ (since an element is not normally repeated in the set listing.)

Sets are sometimes depicted using pictures called *Venn diagrams.* A Venn diagram represents a set as a region enclosed by circular lines. Examples of elements in the set may be represented as points inside the region.

In Figure 1 (i) the shaded area represents the intersection of A and B. The union of A and B contains all elements of the sets A and B (Figure 1 (ii)), the elements in $A \cap B$ are normally listed only once.

The *set difference* of A and B, $A - B$, consists of all elements of A that are not elements of B. The set difference operation is illustrated in Figure 2.

The *complement of a set* A (with respect to a universe set U), $\overline{A}$, consists of all elements of U that are not in A, that is, the set $U - A$. Note that when using the complement operation we need to have an implicitly known universe set, that is, a set of all objects under consideration. For example, when talking about sets of integers, the universe set if $\mathbb{Z}$.

We will use the following notations for sets and elements of set:

- $x \in A$ (x is an element of the set A)

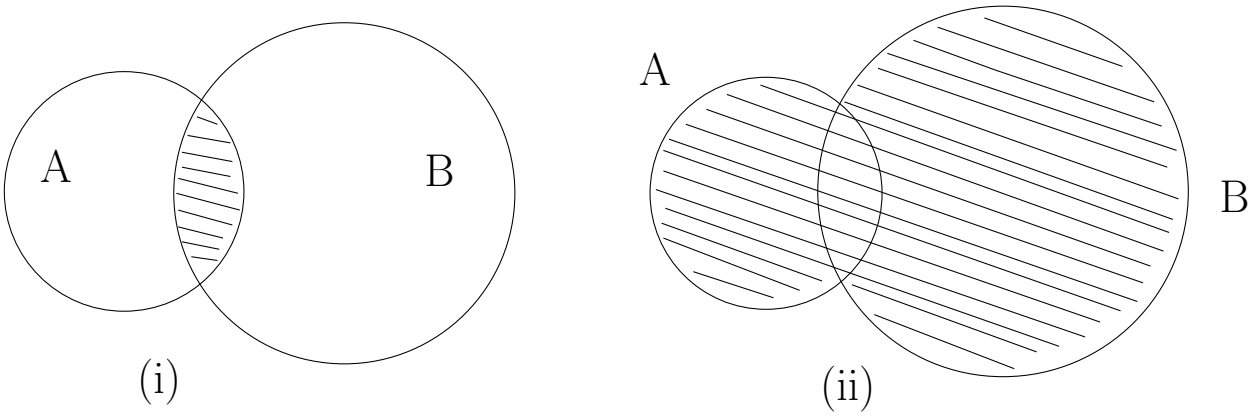

Figure 1: Venn diagrams for intersection and union.

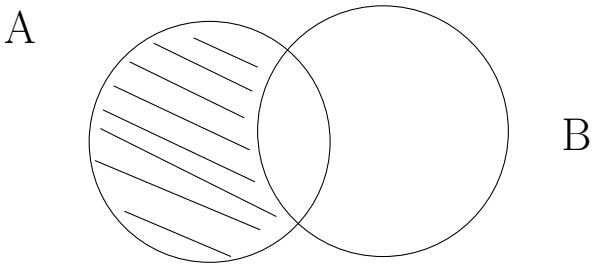

Figure 2: Venn diagram illustrating the set difference of A and B

- $x \notin A$ (x is not an element of A)
- $A \subseteq B$ (A is a subset of B, that is, every element of A is an element of B)
- $A = B$ (A and B are the same set)
- $A \neq B$ (A and B are not the same set)
- $A \subset B$ (A is a strict subset of B, that is, $A \subseteq B$ and $A \neq B$)
- $A \not\subseteq B$ (A is not a subset of B)
- $A \supseteq B$ (the same as $B \subseteq A$)
- $A \supset B$ (the same as $B \subset A$)

The *symmetric difference* of A and B is defined as

$$A \triangle B = (A - B) \cup (B - A).$$

The following equality holds in general (Proposition 12.11 in the text)

$$A \triangle B = (A \cup B) - (A \cap B)$$

To prove the equality we need to show that $A \triangle B \subseteq (A \cup B) - (A \cap B)$ and $(A \cup B) - (A \cap B) \subseteq A \triangle B$.

The following proposition lists some basic identities of set algebra. Below "iff" stands for "if and only if".

Proposition 2.1 *Let U be the set of all objects under consideration and $A, B, C \subseteq U$. Then*

1. $A \cup A = A \cap A = A$ *(idempotency laws)*
2. $A \cup B = B \cup A$ *and* $A \cap B = B \cap A$ *(commutativity)*
3. $A \cup (B \cup C) = (A \cup B) \cup C$ *and* $A \cap (B \cap C) = (A \cap B) \cap C)$ *(associativity)*
4. $A \cap (A \cup B) = A = A \cup (A \cap B)$ *(absorption laws)*
5. $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ *and* $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ *(distributivity)*
6. $A \cap \overline{A} = \emptyset$ *and* $A \cup \overline{A} = U$
7. $\overline{A \cap B} = \overline{A} \cup \overline{B}$ *and* $\overline{A \cup B} = \overline{A} \cap \overline{B}$ *(De Morgan's laws)*
8. $A \cap B = A$ *iff* $A \subseteq B$ *iff* $A \cup B = B$
9. $A \subseteq B$ *iff* $\overline{B} \subseteq \overline{A}$

Some identities from Proposition 2.1 will be proved in class or in the assignments.

In particular, due to associativity, when using multiple unions (or intersections) we can omit parentheses: $A \cup B \cup C$.

The *Cartesian product* of sets A and B, $A \times B$, is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$, that is,

$$A \times B = \{ \ (a, b) \ \mid a \in A, b \in B \}.$$

Because the product consists of ordered pairs, the operation is not commutative.

More generally the notation can be extended to a product of $n \geq 1$ sets:

$$A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n) \mid a_1 \in A_1, \ldots a_n \in A_n\}.$$

Finally, the *power set* of a set A, 2^A, is the set consisting of all subsets of A:

$$2^A = \{B \mid B \subseteq A\}.$$

The power set is a set of sets. In particular, $\emptyset$ and A are always elements of 2^A. If A is a finite set with n elements, $|2^A| = 2^n$.

3 Functions and relations

A *function* or a *mapping* is an object that sets up an input-output relationship: it takes an input and produces an output. If f is a function whose output value is y when the input is x, we write $f(x) = y$. We say that f maps x to y.

Functions are defined more precisely in terms of relations (to be discussed in subsection 3.2).[1]

For example, the absolute value function *abs* takes an integer x as input and returns x if x is positive and returns $-x$ if x is negative.

Addition is another example of a function. The input to the addition function is an ordered pair of numbers and the output is their sum.

Generally, a *function* $f : A \longrightarrow B$ from set A to set B is a rule that assigns to each element $a \in A$ a unique element $f(a) \in B$. The set A is the *domain* of the function f, the set B is the *range* of F and the set of all values of f is the *image* of f,

$$\text{im } f = \{b \in B \mid (\exists a \in A) f(a) = b\}.$$

In the absolute value function mentioned above, the argument of the function is an integer and the value is a non-negative integer so we may write abs $:\mathbb{Z} \longrightarrow \mathbb{N} \cup \{0\}$.

Consider a function $f : A \longrightarrow B$.

- The function f is *one-to-one* (or *injective*) if

$$(\forall a_1, a_2 \in A) f(a_1) = f(a_2) \text{ implies } a_1 = a_2.$$

- The function f is *onto* (or *surjective*) if

$$(\forall b \in B)(\exists a \in A) f(a) = b.$$

- The function f is a *bijection* if it is both one-to-one and onto.

[1] The textbook defines relations before functions.

The *identity function* on a set A, $1_A : A \to A$, is defined by setting, for any $a \in A$, $1_A(a) = a$. The identity function is a bijection.

The *composition* of functions $f : A \longrightarrow B$ and $g : B \longrightarrow C$ is the function $g \circ f : A \to C$ defined by

$$(\forall a \in A)\ (g \circ f)(a) = g(f(a)).$$

Note that the composition is defined only if the range of f equals the domain of g. When dealing with a function composition $g \circ f$ it is important to remember that the function f is appliead first, and then apply function g to the resulting value. The composition of functions is associative. For $f : A \longrightarrow B$, $g : B \longrightarrow C$ and $h : C \longrightarrow D$ we have

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

How would you prove this?

The inverse of a function $f : A \longrightarrow B$ associates to an element $b \in B$, elements $a \in A$ such that $f(a) = b$. The so called inverse image of an element $b \in B$ need not be unique and consequently the inverse of f (denoted f^{-1}) is a *relation* that need not be a function. The notion of relations is discussed next and we return to inverse functions in subsection 3.2.1.

3.1 Growth rate of functions and complexity of algorithms

Before continuing with the more general notion of relations, we consider how the growth rate of functions on natural numbers is used to measure the time and space complexity of algorithms.

For measuring the efficiency of computer algorithms, two most important considerations are how much *time* running the algorithm will take and how much memory it will take. The amount of memory used is commonly called *space* complexity. Different algorithms for a problem may have very different time/space complexity. In particular, we can have trade-offs where increasing the amount of memory (space) required the algorithm can run faster, or vice versa.

To illustrate time and space complexity, consider the following simple algorithmic prob-

lem.[2] Suppose n positive integers are stored in an array A and we want to determine whether they are all distinct or whether at least two of them are the same.

A straightforward algorithm `SCAN` simply scans the array A one integer at a time and looks for duplicates:

```
for i=1 to n-1 do
   for j= i+1 to n do
      if A[i] == A(j)
         then output i and j;
    exit
      else continue
```

The algorithm `SCAN` needs to store in memory the n array locations and the two variables i and j. Assuming one storage word can contain any integer used in the array, the algorithm `SCAN` requires $n + 2$ words of storage.

To determine how much time the algorithm `SCAN` uses we assume that each line of the algorithm costs one unit of time to execute. The the number of time units required naturally depends very much on the sequence of integers given as input: if the first two integers are duplicates, `SCAN` produces the answer immediately (even for very large values of n). Typically, we are interested in the question: given n integers what is the largest number of time units the algorithm requires to determine whether the sequence has duplicates.

The algorithm `SCAN` takes the longest time to execute when the first $n - 1$ integers in the list are all distinct and the two last integers coincide. Each execution of the inner loop when the if-test fails requires 3 steps. When `SCAN` begins with $i = 1$, the variable j runs from 2 to n and the inner loop is executed $n - 1$ times (total $3(n - 1)$ steps). This yields total $1 + 3(n - 1)$ steps executed with value $i = 1$. Similarly when $i = 2$, we see that `SCAN` will execute $1 + 3(n - 2)$ steps. The total time taken by the execution of `SCAN` is

$$n + 1 + 3 \cdot \sum_{k=1}^{n-1} k = \frac{3n^2 - n - 2}{2}.$$

[2]This part is modified from the text by A.K. Dewdney.

The *worst-case time complexity* of an algorithm operating on an input of size n is the maximum time the algorithm will spend on any input instance of size n. Of course the above formula estimating the time complexity of `SCAN` is based on the simplifying assumption that all instructions in the program use the same amount of time. If the execution times of different instruction vary, one might obtain a formula with different coefficients. However, the implementation of `SCAN` under any reasonable assumptions would have a worst-case running time that is quadratic in n.

For this reason, to determine the time and space complexity of an algorithm we are usually interested only in the order of magnitude that is expressed using so called "big Oh" notation.

Consider functions f and g on natural numbers. The function f is said to be *big Oh of the function* g if there exists a constant c and $n_0 \in \mathbb{N}$ such that

$$f(n) \leq c \cdot g(n) \quad \text{of all } n \geq n_0.$$

In this case we write $f(n) = O(g(n))$.

More information on related asymptotic notations like "small oh" and "Omega" can be found in section 29 in the texbook (and we may consider these notions in class).

Using this notation we can say that the time complexity of the algorithm `SCAN` is $O(n^2)$ and the space complexity is $n + 2 = O(n)$.

It is possible to use a different algorithmic approach to the problem of finding duplicates in a list of integers. The algorithm `STOR` uses the idea of storing each integer in the array A in another array B at an index equal to the integer itself:

```
for i=1 to n do
    if B[A[i]] != 0
       then output A[i];
          exit
       elseo B[A[i]]=1
```

It is assumed that the array B is initialized to contain all 0's. Each time an integer x is encountered, the entry $B[x]$ is set to 1. In this way, previously encountered integers are

detected by the if-statement. The technique used by the algorithm is a primitive form of hashing and the technique requires that the array B is large enough to contain any of the integers stored in A.

Under the simplifying assumption that all instructions require the same amount of time, when the array A contains n entries the worst-case time complexity of the algorithm `STOR` is $3n + 1 = O(n)$. On the other hand, `STOR` requires far more memory than our earlier algorithm `SCAN`. If the numbers involved use up to m bits, the operation of `STOR` requires up to 2^m memory locations.

The comparison of algorithms `SCAN` and `STOR` is an example of time vs. space trade-off: by increasing the amount of memory used the running time of the algorithm can be improved.

It is important to distinguish between, on the one hand, the time (or space) complexity of an algorithm and the complexity of the corresponding algorithmic problem, on the other. To illustrate the difference we use the following notation to indicate that a function is asymptotically smaller than another function.

We say that a function $g(n)$ is "small oh" of $f(n)$, $g(n) = o(f(n))$ if for every constant $c > 0$ there exists $n_c \in \mathbb{N}$ such that $g(n) < c \cdot f(n)$ for all $n \geq n_c$. Roughly speaking, this means that the values $g(n)$ become arbitrarily muuch smaller than $f(n)$ for larege enough arguments n.

Suppose that the fastest known algorithm A for a problem P runs in time $O(f(n))$. Now if it can be shown that no algorithm for P can have time complexity $g(n)$ where $g(n) = o(f(n))$ then the algorithm A has optimal time complexity (within a constant multiplicative factor). In this case we say that the time complexity of the problem P is $O(f(n))$.

For some problems, like integer sorting, the exact time complexity of the problem is known. Typically proving lower bounds for the time complexity of algorithmic problems is extremely hard. To solve the well known P $=^?$ NP question it would be sufficient to prove a super-polynomial lower bound for the time complexity of any of a large class of NP-complete problems. This is probably the most important open problem in theoretical computer science.

3.2 Relations

Informally speaking a "relation" means any kind of relationship that can exist between elements of certain sets. Examples geometric relations could be

- $R =$ "point p is on line s"; or,
- $S =$ "point p is in between points q and r".

Above R is a binary relaation from the set of points to the set of lines. On the other hand, S is a ternary relation in the set of points.

Below we concentrate on binary relations. Generally the notions can be extended to n-ary relations for any $n \in \mathbb{N}$.

Definition 3.1 *A (binary)* relation *from set A to set B is a subset of the Cartesian product $A \times B$.*

Suppose $R \subseteq A \times B$ is a relation. If $(a, b) \in R$ we say that element $a \in A$ is in the relation R with element $b \in B$. Sometimes this is denoted also as $a\ R\ b$.

If $R \subseteq A \times A$ we say that R is a binary relation in the set A (or a binary relation of the set A). Special cases of relations in A include the following

- the relation $\emptyset$,
- the *identity relation* (or diagonal relation) $I_A = \{(a, a) \mid a \in A\}$,
- the *universal relation* $U_A = A \times A$.

Example 3.1 *A* directed graph *is an ordered pair (V, E) where V is the set of vertices and $E \subseteq V \times V$ is the set of egges. For an edge (u, v) we say that u is the starting point and v the ending point of the edge. The graph is typically depicted by a drawing where the vertices are points (or small circles) and for each edge (u, v) there is an arrow from point u to point v. The set of edges E is, in fact, a binary relation on the set of vertices V.*

More generally, relations $R \subseteq A \times B$ can be represented by a graph that has an edge from $a \in A$ to $b \in B$ exactly then when $(a, b) \in R$.

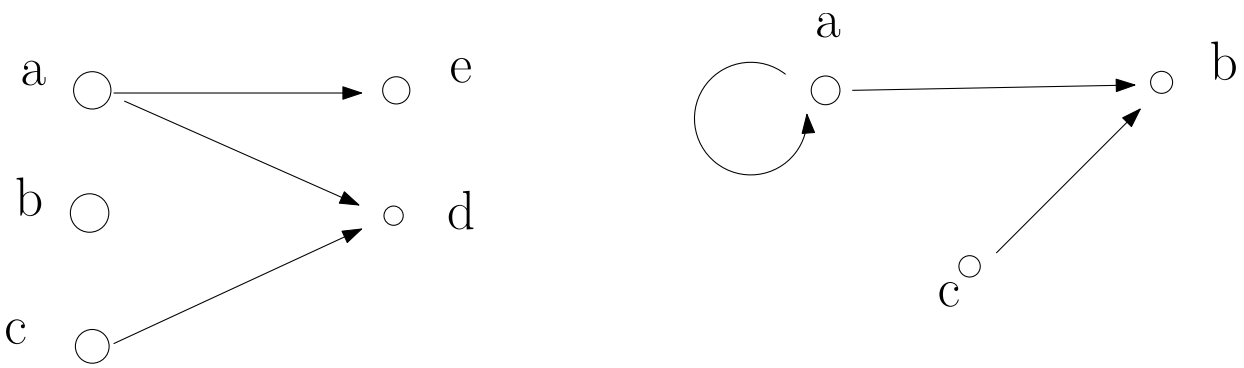

Figure 3: Relations from Example 3.2. Relation R on the left and relation S on the right.

Example 3.2 *Let* $A = \{a, b, c\}$ *and* $B = \{d, e\}$ *and consider relations*

$$R = \{(a, e), (a, d), (c, d)\}(\subseteq A \times B), \quad S = \{(a, a), (a, b), (c, b)\}(\subseteq A \times A).$$

The relations A *and* B *are depicted by the following graphs in Figure 3.1.*

Typical properties of a relation $R \subseteq A \times A$ include:

- R is *reflexive* if $I_A \subseteq R$ (that is, each element of A is in the relation with itself;
- R is *symmetric* if $(a, b) \in R$ implies $(b, a) \in R$,
- R is *transitive* if

$$(a, b) \in R \text{ and } (b, c) \in R \text{ implies } (a, c) \in R,$$

- R is *irreflexive* if $(a, a) \notin R$ for all $a \in A$,
- R is *antisymmetric* if

$$(a, b) \in R \text{ and } (b, a) \in R \text{ implies } a = b.$$

The "less than or equal to" relation $\leq \subseteq \mathbb{Z} \times \mathbb{Z}$ is reflexive, transitive and antisymmetric. The "less than relation" $< \subseteq \mathbb{Z} \times \mathbb{Z}$ is irreflexive and transitive.

Since relations are sets (subsets of a Cartesian product of sets), all set theoretic operations like union, intersection and complementation can be directly applied to relations.

For binary relations we typically use the composition and inverse operations.

Definition 3.2 *Let A, B, C be sets and R is a relation from A to B and S is a relation from B to C.*

1. *The* composition *of relations R and S is*

$$R \odot S = \{(a, c) \mid (\exists b \in B)(a, b) \in R \text{ and } (b, c) \in S\} \ \ (\subseteq A \times C).$$

2. *The* inverse *of the relation R is*

$$R^{-1} = \{(b, a) \mid (a, b) \in R\}.$$

Note: When talking about relations and functions, respectively, the composition operations $\odot$ and $\circ$, respectively, have somewhat different definition:

- the composition of relations R and S, $R \odot S$, first "applies" relation R and then "applies" relation S,
- the compositiion of functions f and g, $g \circ f$, first "applies" function f and then "applies" function g.

This is the reason why we use a different symbol for composition of relations and functions, respectively. The function composition notation $\circ$ is the same as used in our textbook.

The graph representation of an inverse relation R^{-1} is obtained from the graph representation of R simply by reversing the arrows.

Example 3.3 *Let R and S be the relations from Example 3.2. Then*

- $R^{-1} = \{(e, a), (d, a), (d, c)\}$ *and* $S^{-1} = \{(a, a), (b, a), (b, c)\}$.
- *The relation $R \odot S$ is undefined because $R \subseteq A \times B$ and $S \subseteq A \times A$ (and B and A are distinct sets). On the other hand, $S \odot R = \emptyset$.*

  Note: *The composition being undefined is <u>not</u> the same as the composition being empty!*

3.2.1 Functions as relations and inverse functions

A function is a special type of a relation. A function $f : A \longrightarrow B$ maps each element of A into exactly one element of B.

Definition 3.3 *A function from set A to set B is a relation $f \subseteq A \times B$ that satisfies the following properties:*

1. $(\forall a \in A)(\exists b \in B) : (a, b) \in f$ *(each element of A is mapped to some element of B)*
2. $(\forall a \in A)(\forall b_1, b_2 \in B) : (a, b_1) \in f$ *and* $(a, b_2) \in f$ *implies* $b_1 = b_2$ *(each element of A is mapped to only one element of B).*

Any relation $R \subseteq A \times B$ has a unique inverse relation $R^{-1} \subseteq B \times A$. However, the inverse relation of a function f need not be a function. What would be a counter-example? (See Example 24.12 in the text.)

If the inverse of f is not a function, the function f does not have an inverse. It is also possible that a function has one-sided (left or right) inverses.

Definition 3.4 *Consider a function $f : A \longrightarrow B$. A function $g : B \longrightarrow A$ is*

1. left inverse *of f if $g \circ f = 1_A$,*
2. right inverse *of f if $f \circ g = 1_B$,*
3. *a (two-sided)* inverse *of f if $g \circ f = 1_A$ and $f \circ g = 1_B$.*

The following proposition may be proved in class or in the assignments. Recall that "iff" stands for "if and only if".

Proposition 3.1 *A function $f : A \longrightarrow B$ has*

1. *a right inverse iff f is onto,*
2. *a left invervse iff f is one-to-one,*
3. *a (two-sided) inverse iff f is bijective.*

A function may have more than one left inverse or more than one right inverse. What would be an example of such a function? However, if a function has both a left inverse and a right inverse, it is the unique two-sided inverse.

Lemma 3.1 *If a function $f : A \longrightarrow B$ has a left inverse g and a right inverse h, then $g = h$.*

Lemma 3.1 implies that a two-sided inverse is unique. The two sided inverse of a function $f : A \longrightarrow B$ is denoted as $f^{-1} : B \longrightarrow A$. The function f^{-1} (if it exists) is the same as the inverse relation of f.

3.2.2 Further properties of relations

Next we list some useful observations on general binary relations. Some of the below results may be proved in class or as part of assignments.

The first lemma deals with the inverse of relations. Recall that since a binary relation is a subset of the Cartesian product $A \times B$, we can use the notation "$\subseteq$" with the standard meaning. Recall also the $U_A = A \times A$ denotes the universal relation on A.

Lemma 3.2 *For relations $R, S \subseteq A \times B$ the following hold:*

1. $R^{-1} \subseteq S^{-1}$ *iff* $R \subseteq S$,
2. $(R \cup S)^{-1} = R^{-1} \cup S^{-1}$ *and* $(R \cap S)^{-1} = R^{-1} \cap S^{-1}$,
3. $(R^{-1})^{-1} = R$, $\emptyset^{-1} = \emptyset$, $I_A^{-1} = I_A$ *and* $U_A^{-1} = U_A$.

The composition of relations is associative. That is, if $R \subseteq A \times B$, $S \subseteq B \times C$ and $T \subseteq C \times D$, we have $(R \odot S) \odot T = R \odot (S \odot T)$. Due to associatetivity, the composition of multiple relations can be written without parentheses: $R_1 \odot R_2 \odot \ldots \odot R_n$. On the other hand, it should be noted that composition of relations on a set A is not commutative.

The next lemma give the relationship between composition and inverse.

Lemma 3.3 *If $R \subseteq A \times B$ and $S \subseteq B \times C$, then*

$$(R \odot S)^{-1} = S^{-1} \odot R^{-1}.$$

The following lemma "mirrors" some properties of sets from Proposition 2.1.

Lemma 3.4 *For relations $R, R_1, R_2 \subseteq A \times B$ and $S, S_1, S_2 \subseteq B \times C$, we have*

1. *$R_1 \subseteq R_2$, $S_1 \subseteq S_2$ implies $R_1 \odot S_1 \subseteq R_2 \odot S_2$.*
2. *$R \odot (S_1 \cup S_2) = (R \odot S_1) \cup (R \odot S_2)$.*
3. *$(R_1 \cup R_2) \odot S = (R_1 \odot S) \cup (R_2 \odot S)$.*
4. *$R \odot (S_1 \cap S_2) \subseteq (R \odot S_1) \cap (R \odot S_2)$.*
5. *$(R_1 \cap R_2) \odot S \subseteq (R_1 \odot S) \cap (R_2 \odot S)$.*
6. *$\emptyset \odot S = R \odot \emptyset = \emptyset$.*
7. *$I_A \odot R = R \odot I_B = R$.*

Note that while composition "distributes" over union (items 2. and 3.), the corresponding laws for intersection (items 4. and 5.) have only inclusion in one direction.

Finally, Lemma 3.5 characterizes special properties of binary relations on a set A in a compact form.

Lemma 3.5 *Consider a binary relation $R \subseteq A \times A$. The relation R is*

1. *reflexive iff $I_A \subseteq R$,*
2. *symmetric iff $R^{-1} = R$ iff $R^{-1} \subseteq R$,*
3. *transitive iff $R \odot R \subseteq R$,*
4. *irreflexive iff $R \cap I_A = \emptyset$,*
5. *antisymmetric iff $R \cap R^{-1} \subseteq I_A$.*

3.2.3 Equivalence relations

An *equivalence relation* on a set A is a relation that is reflexive, symmetric and transitive.

By Lemma 3.5 we know that $R \subseteq A \times A$ is an equivalence relation if and only if

1. $I_A \subseteq R$,
2. $R^{-1} \subseteq R$, and,
3. $R \odot R \subseteq R$.

It is easy to verify that I_A and U_A are always equivalence relations.

Consider an equivalence relation R on a set A and $a \in A$. The equivalence class of the element a with respect to R is

$$[a]_R = \{b \in A \mid (a, b) \in R\}.$$

When the equivalence relation R is known from the context, the equivalence class $[a]_R$ is denoted simply by $[a]$.

Lemma 3.6 *Let R be an equivalence relation on a set A and $a, b \in A$.*

1. $a \in [a]$,
2. $[a] = [b]$ *if and only if* $(a, b) \in R$,
3. $[a] \cap [b] \neq \emptyset$ *if and only if* $(a, b) \in R$.

As indicated by item 3. of Lemma 3.6, equivalence relations have a close correspondence with partitiions of a set.

Definition 3.5 (Textbook Def. 16.1) *A* partition of a set A *is a set of nonempty, pairwise disjoint subsets of A whose union is A.*

Recall that 2^A denotes the power set of a set A. The following result is proved in the textbook.

Theorem 3.1 *Let A be a set.*

1. *If E is an equivalence relation of A, then*

$$\{[a]_E \mid a \in A\}$$

   is a partition of A.

2. *Suppose $\mathcal{P} \subseteq 2^A$ is a partition of A and define, for $a, b \in A$,*

$$a \equiv^{\mathcal{P}} b \text{ iff } (\exists P \in \mathcal{P}) : a, b \in P.$$

   Then $\equiv^{\mathcal{P}}$ is an equivalence relation of A.

The correspondence between equivalence relations and partitions is illustrated by the example:

Let $A = \{1, 2, 3, 4, 5, 6, 7\}$ and E is the following equivalence relation of A:

$$I_A \cup \{(1,2), (1,3), (2,1), (3,1), (2,3), (3,1), (3,2), (4,6), (6,4), (5,7), (7,5)\}.$$

The partition corresponding to the equivalence relation E is

$$\{\ \{1,2,3\}, \{4,6\}, \{5,7\}\ \}.$$