ArsDigita University
Month 2: Discrete Mathematics - Professor Shai Simonson

Lecture Notes

What is Discrete Math?

Example of continuous math – Given a fixed surface area, what are the dimensions of a cylinder that maximizes volume?

Example of Discrete Math – Given a fixed set of characters, and a length, how many different passwords can you construct? How many edges in graph with n vertices? How many ways to choose a team of two people from a group of n? How many different binary trees (is it worth checking them all to find a minimum spanning tree of a graph – a tree that includes all the vertices of a weighted edge graph, with minimum sum of weights)? How many ways to arrange n arrays for multiplication? How many ways to draw n pairs of balanced parens?

Note that the last 3 examples have the same answers (not obvious).

Note the second and third examples have the same answer (obvious).

Counting is an important tool in discrete math as we will see later.

What are proofs?

Formal definitions and logic versus…

A proof is a clear explanation, accepted by the mathematical community, of why something is true.

Examples….

Ancient Babylonian and Egyptian mathematics had no proofs, just examples and methods.
Proofs in the way we use them today began with the Greeks and Euclid.

1. The square root of two is irrational - A proof by contradiction from Aristotle.

Assume that a/b = $\sqrt{2}$, where a and b are relatively prime. Squaring both sides of the equation gives a^2/b^2 = 2. Then a^2 = 2b^2, and since an even number is any number that can be written as 2k, a^2 must be even. By a separate lemma, we know that if a^2 is even, then a must also be even. So write a = 2m. Then a^2 = (2m)^2 and a^2 = 4m^2, and 2b^2 = 4m^2, so b^2 is even, and b is even.. But we assumed without any loss of generality that a and b were relatively prime, and now we have deduced that both are even! This is a contradiction, hence our assumption that a/b = $\sqrt{2}$ cannot be right.

2. There are an infinite number of prime numbers – A proof by contradiction by Euclid.

Assume that there is a finite number of prime numbers. Construct their product and add one. None of the prime numbers divide this new number evenly, because they will all leave a remainder of one. Hence, the number is either prime itself, or it is divisible by another prime not on the original list. Either way we get a prime number not in the original list. This is a contradiction to the assumption that there is a finite number of prime numbers. Hence our assumption cannot be correct.

Discovering theorems is as important as proving them.

Examples:

1. How many pairs of people are possible given a group of n people?

   Constructive counting method: The first person can pair up with n-1 people. The next person can pair up with n-2 people etc, giving (n-1) + (n-2) + … + 2 + 1

   Counting argument: Each person of n people can pair up with n-1 other people, but counting pairs this way, counts each pair twice, once from each end. Hence we get a total of n(n-1)/2.

2. Define the triangle numbers. How big is the nth triangle number?

   Geometric argument – If n is even, (n+1)(n/2). If n is odd, (n)((n+1)/2). These cases seem unnecessary to our algebraic eyes, but in the middle ages, before algebra, each of these was listed as a separate theorem described in words.

   A pairing idea - Pair the numbers up one from each end, working inwards. The Gauss legend tells a story of the 8-year old wunderkind being told by a teacher to add up the numbers from to 100. The teacher had hoped this would keep Gauss busy for a few minutes. Gauss presumably derived this formula on the spot and blurted back 5050. Note that later in his life it is well documented that Gauss was quite proud of his proof that any integer can be written as a sum of at most three triangle numbers.

3. How many pieces do you get from cutting a circle with n *distinct* cuts? (make sure we define distinct carefully).

   The first few numbers of cuts and pieces can be listed below as we experiment:

   Cuts   Pieces

1	2
2	4
3	7
4	11

We can argue that the P_(n+1) = P_n + n+1. Every new cut intersects each of the old cuts in one unique place. Hence each new cut creates 1 more region than the number of cuts already made, because it creates a region as it exits the circle. This is called a recurrence equation and we can solve it directly (see week 3 in syllabus).

Note that T_(n+1) = T_n + n+1. This is the same equation, but P_n does not equal T_n! What gives? The difference is that P_1 = 2 and T_1 = 1.

We know that T_n = (n)(n+1)/2 and it seems that P_n = T_n + 1.

Can we prove this last fact, namely P_n = T_n + 1? If so, it would immediately imply that P_n = (n^2 + n + 2)/2. There are many ways to prove this formula including a neat technique called finite differences, but we will use a technique called mathematical induction

Proofs by induction – The most common method of proof in computer science.

Strategy – To prove something for an infinite number of cases. Start by identifying a variable which will be used to index the infinite number of cases. In our case, this will be n. The proof proceeds “by induction on n”. Note that sometimes the choice of variable is not immediately obvious and a good choice can make the proof simpler.

Show that the theorem is true for a start value of n. In our case we can use n =1 . Since P_1 = 2, we can check that (1^2 + 1 + 2)/2 = 2, and it does.

Then try to show that IF the theorem is true for the nth case, then it must also be true for the n+1st case. The idea is to focus on the transition from smaller cases to larger cases.

In our case, let’s assume that P_n = T_n + 1, and try to show that P_(n+1) = T_(n+1) + 1.
We know from our own analysis that P_(n+1) = P_n + n + 1, and from our assumption, we can derive that P_(n+1) = (T_n + 1) + n + 1. Also, we know that T_(n+1) = T_n + n + 1, so we conclude that P_(n+1) = T_(n+1) + 1, QED.

It takes a lot of experience before proofs by mathematical induction start to lose their magic, and yield up their real ideas. The interactive lectures supporting these notes is a crucial guide to the ideas here.

Recitation – Proof by induction of Euler’s Thm on planar Graphs. A Combinatorial card trick.

Formal Proof, Logic and Boolean Algebra

We can represent facts by Boolean variables, variables whose values are true or false (1 or 0). We can combine these variables using various operators, AND, OR and NOT. We can specify all sorts of logical statements using other operators, but they can always be transformed back to a formula containing just AND, OR and NOT.

Example:
Let W = wet outside. Let R = raining.

It is raining and it's wet outside.	W AND R	WR	$W \wedge R$
It is raining or it's wet outside.	W OR R	W+R	$W \vee R$
It is not raining	NOT R	$\neg R$	
If it's raining then its wet outside.	$R \Rightarrow W$		
Either it's raining or it's wet outside but not both.	$(R+W)\ \neg(RW)$	$(\neg RW)+(\neg WR)$	

Let's look at the fourth example. The logic of this is equivalent to: if R is true then W is true; but if R is false then W can be anything. Let's make a truth table of this below:

R	W	$R \Rightarrow W$
0	0	1
0	1	1
1	0	0
1	1	1

This idea of a truth table is a sure fire to show the equivalence of Boolean expressions.

It can be seen that the above formula $R \Rightarrow W$ is equivalent to: $(\neg R \neg W)+(\neg RW)+(RW)$. It is constructed by looking in each row that has a 1 appearing at the right end. These are the rows for which the formula is true. We simply write down the possible values for each combination of variables that can make these 1's occur, and OR them altogether. For each combination of variables we AND the conditions on each variable. The method used here to compute this formula implies a proof that any Boolean expression can be represented by a combination of ANDs ORs and NOTs. It is also equivalent to $\neg R + W$.

Truth tables can be made for AND, OR, NOT, Exclusive OR (the fifth example), implies (the 4^{th} example). Note there may be many different Boolean expressions that are equivalent to each other logically. Note that with n variables, a truth table will have 2^n rows.

Last Example. Make a truth table for $(R \Rightarrow W)$ AND $(W \Rightarrow R)$. This is sometimes called $\Leftrightarrow$ or simply =.

R	W	$R \Leftrightarrow W$

0	0	1
0	1	0
1	0	0
1	1	1

The Algebra of Bits – Boolean Algebra

Here we treat the manipulation of Boolean expressions syntactically and note the analogy to addition and multiplication, where true is the value 1 and false is the value 0. AND, OR are commutative, and they are mutually distributive. There are two rules called DeMorgan's Laws that relate NOT to AND and OR.

Here is a summary of the rules of Boolean Algebra. They all can be verified by truth tables and the definitions of the operators.

$P + \text{true} = \text{true}$	$\neg PP = \text{false}$
$P(\text{true}) = P$	$\neg P + P = \text{true}$
$P + \text{false} = P$	$P(Q+R) = PQ + PR$
$P(\text{false}) = \text{false}$	$PQ + R = (P+R)(Q+R)$

(note that this last beautiful dual rule is not true for regular addition and multiplication.).

DeMorgan's Laws: $\neg(P + Q) = \neg P\neg Q$ $\qquad \neg(PQ\ ) = \neg P + \neg Q$

Boolean algebra is useful not only in logic but more importantly in the design of digital circuits at the heart of making a computer work. It allows the manipulation of Boolean expressions from one form to another without the need for truth table verification.

Example: Show that $\neg X(X+Y) \Rightarrow Y$ is equal to true.

$\neg X(X+Y) \Rightarrow Y$	
$\neg(\neg X(X+Y)) + Y$	$P\Rightarrow Q$ equals $\neg P + Q$
$X + \neg(X+Y) + Y$	De Morgan's Laws
$(X+Y) + \neg(X+Y)$	Commutativity and Associativity of +
true	$\neg P + P = \text{true}$

In this example, you should identify which rule is applicable at each step.

Example: $(R+W)\ \neg(RW) \quad = \quad (\neg RW)+(\neg WR)$

$R\neg(RW) + W\neg(RW)$

$R(\neg R+\neg W) + W(\neg R + \neg W)$

$\neg RR + \neg WR + \neg RW + \neg WW$

$(\neg RW)+(\neg WR)$

Theorem: Any Boolean function can be described using just AND, OR and NOT operators.

Proof by example above.

The resulting expression is an OR of a collection of variables or their negations that are ANDed together. This is called Disjunctive Normal Form. The Conjunctive Normal form of a Boolean expression can also always be constructed and it is an AND of a collection of variables or their negations that are ORed together. Note again the intense symmetry in Boolean Algebra.

Complete Operators

A set of operators that can describe an arbitrary Boolean function is called complete. The set {AND, OR, NOT} is complete. There are certain operators that alone can describe any Boolean function. One example is the NOR operator $\downarrow$. $P \downarrow Q$ is defined to be $\neg(P+Q)$. You can verify that

$\neg P = (P \downarrow P)$

$PQ = (P \downarrow Q) \downarrow (P \downarrow Q)$

$P+Q = (P \downarrow P) \downarrow (Q \downarrow Q)$

These three equations imply that NOR is complete.

Recitation – Predicates and higher order Logic. Quantifiers and rules for substitution and pushing through of negations.

Applications in Computer Science:

Example: The Satisfiability problem and NP-Completeness.

Reductions

Informally, a reduction is a transformation of one problem into another. It is a fundamental notion in algorithms, theory of computation, and good software design.

The idea behind Reductions:

“Q: What do you feed a blue elephant for breakfast?”
“A: Blue elephant toasties”.
“Q: What do you feed a pink elephant for breakfast?”
“A: You tell the pink elephant not to breathe until he turns blue, then you feed him blue elephant toasties”.

This comes from The Funnybone Book of Jokes and Riddles, ISBN 0-448-1908-x.

Reductions are crucial to showing that a problem is hard. We cannot in general prove that a problem is hard. We would have to show that no algorithm is efficient, and there are a lot of algorithms! On the other hand, we can show that a problem is easy but exhibiting just one good algorithm. What computer scientists can do, is to prove that a problem is NP-Complete. This does NOT mean it is definitely hard, but it means it is at least as hard as a whole host of other well known difficult problems.

NP is the set of all problems solvable in polynomial time by a non-deterministic program. Yikes, what does that mean? Wait until the algorithms course. But basically, it means that you can verify a guess of the solution in polynomial time. Non-determinism gives you lots of power. No one knows how to simulate non-deterministic programs efficiently with deterministic (normal) programs. Any general simulation known requires an exponential growth in time requirements.

An NP-Complete problem is a problem in NP to which all the problems in NP can be reduced in polynomial time. This means that if you could solve the NP-Complete problem in polynomial time, then you could solve all the problems in NP in polynomial time. So if your boss gives you a hard problem, you can't say "Sorry boss, it can't be done efficiently", but at least you can say "I can't do it boss, but neither can all these other smart people".

P is the set of problems that can be solved by normal deterministic programs in polynomial time

The greatest open question in computer science is whether P = NP. If a problem is NP-Complete, and someone comes up with a polynomial time algorithm for it, then P = NP. No one really believes that P = NP, but showing otherwise has eluded the best minds in the world.

Satisfiability was the first problem proved to be NP-Complete. The problem gives you a Boolean formula in conjunctive normal form, and asks whether or not there is an assignment of True/False to the variables, which makes the formula true. Note that a brute force algorithm for this problem runs in 2^n * m time where n is the number of variables and m is the number of clauses. A non-deterministic polynomial time algorithm verifies a guess of the solution in m time.

Satisfiability reduces to 3SAT.

An input to SAT is a formula F in conjunctive normal form (AND of ORs). Convert the clauses in F according to the following rules:

We show how to convert formulas with an arbitrary number of variables per clause, into an equivalent set with exactly 3 per clause.

One variable in the clause: (x) = (x+a+b)(x+a+-b)(x+-a+b)(x+-a+-b)
Two variables in the clause: (x+y) = (x+y+c)(x+y+-c)
Three variables in the clause: (x+y+z) = (x+y+z)
Four or more variables in the clause: (u+v+w+x+y+z) = (u+v+d)(-d+w+e)(-e+x+f)(-f+y+z)

You can prove that the new set of clauses is satisfiable iff F is satisfiable. Also the new set has exactly 3 variables per clause. Finally note that this reduction can be done in time proportional to the m*n, where m is the number clauses and n is the number of variables. An example will be done in class.

This implies that 3SAT is at least as hard as Satisfiability.

2SAT reduces to Cycles in Graph.

Given a Boolean expression in conjunctive normal form with two variables per clause, create a graph G = (V, E) where V = {x, -x} for all variables x, and E = {(-x, y), (-y, x) for each (x+y) clause. The formula is not satisfiable if and only if there is a cycle in G including x and –x for some vertex x. This is equivalent to a strongly connected component containing both x and –x. This can be done in O(edges) time.

Note that a directed edge in the graph from x to y means that if x is true in the formula then y must be true. This idea is the key to the reduction. For example (x + -y) (y + -w) (-x + -y) (z + y) (-z + w) is not satisfiable and will result in a graph with a cycle including y and –y. Note how much information the graph shows at a glance. It shows that if y is true then x and –x must be true, hence y must be false. But if y is false then –y is true and that implies via a chain through z and w, that y is true. Hence there is no satisfiable assignment that works. Graphs are a superb tool for visualization of subtle dependencies.

This implies that 2SAT is no harder than the Cycles in Graph problem.

Note how reductions can be used to show that a problem is easy or hard, depending on the problems to and from which we are reducing. To show a problem is hard, reduce a hard problem to it. To show a problem is easy, reduce it to an easy problem. This is why we choose the <= symbol to indicate A <= B when a problem A reduces to a problem B.

Example: Theorem Proving by Resolution:

Mechanical Theorem proving is a wide area of research whose techniques are applicable to the wider field of database query processing, and the logic paradigm of programming languages.

To prove a theorem we can represent the hypotheses H_i by logic expressions, and the theorem T by another expression. H_1 and H_2 and … H_n implies T, can be checked mechanically, by checking whether H_1 and H_2 and … H_n and NOT(T) is false. If it is false, then the theorem is true. Theorem Proving is a large area of research, but one basic idea uses resolution. Resolution is a way to reduce two expressions into an implied simpler expression. In particular, (A or Q) and (B or –Q) is equivalent to the simpler expression (A or B).

Let R mean it's raining, W mean it's wet outside, C mean my driveway is clean. Say we know that R implies W, W implies C, and that now it is either raining or wet outside. Prove that my driveway is clean.

1. -R + W	given
2. -W + C	given
3. W + R	given
4. –C	theorem negated
5. W	resolve 1, 3
6. C	resolve 2, 5
7. false	resolve 4, 6 QED.

Theorem proving usually works in higher order logic, where the idea is identical, except for the presence of quantifiers and functions. Your Scheme text talks about unification, to handle matching up clauses. But this is out of our territory. All you really need to know is that a universal quantifier can be replaced with any value you like, and an existential quantifier can be replaced with a specific that must not be dependent on other variables.

Recitation – Resolution with quantifiers and unification.

Logic Based Programming Languages –

Another place where theorem proving shows up in disguise is in the implementation of a Logic based programming languages, namely Prolog. The execution of a Prolog program, is theorem proving in disguise. The program is described by a list of FACTS and RULES, and we provide the system with a QUERY, which it tries to prove from the FACTS and RULES. The same idea comes up in the query processing for database languages.

Recitation - Some examples of Prolog programs and how they are executed.

Example: Digital Circuits, Binary Addition – Half Adders, Threshhold Circuits 2,3

A half adder takes two binary inputs and outputs their sum. The truth table is shown below:

Bit1	Bit2	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

We can calculate by our algorithm a disjunctive normal form:
Carry = Bit1 and Bit2 Sum = (-Bit1 and Bit2) or (Bit1 and –Bit2)

In class we will make the pictures for these circuits as explained in section 9.3 of the text.

A threshold circuit is a type of circuit used to simulate neurons. An a,b threshhold circuit has b inputs and one output. The output is 1 iff a input bits or more are 1. For example:

In1	In2	In3	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1

1	1	0	1
1	0	1	1
1	1	1	1

Out = (-In1 and In2 and In3) or (In1 and -In2 and In3) or (In1 and In2 and -In3) or (In1 and -In2 and In3) or (In1 and In2 and In3).

Note this is equivalent to (In1 and In2) or (In1 and In3) or (In1 and In3). DNF is not always the simplest formula.

Sets

What are sets? Unordered collections of things.

In computer science we see them in theory, software engineering, data structures and algorithms, (for example, did someone choose one of the legal set of choices in a program?) In algorithms there is an efficient algorithm called Union-Find which allows us to combine smaller objects into larger ones, and identify an object by name. It is used in many applications including minimum spanning tree, where the sets contain edges.

We specify sets using curly brackets with a list of elements, or we can describe the elements. For example V = {a, e, i, o, u} is the set of vowels in English. B = {0, 1} the set of symbols in the binary number system. O = {2, 4, 6, …} = {x : where x is an even positive integer}. Sets can be infinite of course. Whenever we speak use sets there is an implicit Universal set of which all the sets in question are subsets. There is also an empty set {}= $\varnothing$.

The notion of a subset, a proper subset, union, intersection and complement must be defined through logic. There are many theorems regarding the relationship between these operators on sets. For the most part they have counterparts to similar theorems in Boolean algebra.

Universal and Complement Laws

$A \cup \varnothing = A$ $A \cap \varnothing = \varnothing$ $A \cup U = U$ $A \cap U = A$

$A \cup A^c = U$ $U^c = \varnothing$ $A \cap A^c = \varnothing$

Commutative Laws

$A \cup B = B \cup A$ $A \cap B = B \cap A$

Associative Laws

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

Distributive Laws

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

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

De Morgan's Laws

$(A \cap B)^c = A^c \cup B^c$
$(A \cup B)^c = A^c \cap B^c$

We will prove the distributive laws by unraveling the expressions about sets into Boolean expressions. The laws involving union, intersection and complement come from their counterparts of OR, AND and Complement.

Example:

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

The set of all elements of the left side equals $\{x \mid (x \in A) \text{ or } ((x\in B) \text{ and } (x\in C))\} = \{x \mid ((x \in A) \text{ or } (x\in B)) \text{ and } ((x\in A) \text{ or } (x\in C))\} = \{x \mid x\in (A\cup B) \text{ and } x\in (A \cup C)\} = \{x \mid x \in (A \cup B) \cap (A \cup C)\}$ = the set of all elements on the right side.

Once we have proved this theorem about sets by unraveling the associated Boolean logic, we can prove more theorems about sets by induction:

For example: Let's prove a generalization of the distributive theorem we just proved before. Namely: $A \cup (B1 \cap B2 \cap \ldots \cap Bn) = (A\cup B1) \cap (A\cup B2) \cap \ldots \cap (A \cup Bn)$

The proof is by induction on n.

The base case is when n=2. This is the theorem we previously proved.

Now let's prove that:
$A \cup (B1 \cap B2 \cap \ldots \cap Bn \cap Bn+1) = (A\cup B1) \cap (A\cup B2) \cap \ldots \cap (A \cup Bn) \cap (A \cup Bn+1)$

By associativity of intersection,
$A \cup (B1 \cap B2 \cap \ldots \cap Bn \cap Bn+1) = A \cup ((B1 \cap B2 \cap \ldots \cap Bn) \cap Bn+1)$.
By the distributive theorem (base case again) we know that:
$A \cup ((B1 \cap B2 \cap \ldots \cap Bn) \cap Bn+1) = (A \cup (B1 \cap B2 \cap \ldots \cap Bn)) \cap (A \cup Bn+1)$
By the induction hypothesis,
$A \cup (B1 \cap B2 \cap \ldots \cap Bn) = (A\cup B1) \cap (A\cup B2) \cap \ldots \cap (A \cup Bn)$. Hence,
$A \cup (B1 \cap B2 \cap \ldots \cap Bn \cap Bn+1) = (A\cup B1) \cap (A\cup B2) \cap \ldots \cap (A \cup Bn) \cap (A \cup Bn+1)$ QED.

This theorem screams for an inductive proof. Some theorems are more naturally conducive to inductive proofs than others. The key feature to look for is the ease with which larger cases can be made to depend specifically on the smaller cases.

There are two major tricks for counting:

A. If you can't count what you want – count the complement instead.
B. Count double in a controlled fashion.

A nice example of the former trick, is when we want to count the number of ways n people can have at least one common birthday. Instead we count the number of ways for n people to have all different birthdays. This value is then subtracted from the total number of ways for n people to have birthdays. It is generally easier to count things when the conditions are ANDed together, as in "person 1 has a different birthday AND person 2 has a different birthday etc", as opposed to when the conditions are ORed together, as in "person 1 has the same birthday as someone else OR person 2 has the same birthday etc.".

A nice example of the latter trick is the triangle numbers (again). To count the maximum number of edges in a graph with n vertices, (or equivalently, the number of pairs of people we can choose from a set of n people), we can say that each one of the n vertices can connect to n-1 other vertices. This gives n(n-1) edges. But we have counted each edge exactly twice, once from each end of the edge. This means the total number of edges is n(n-1)/2.

Venn Diagrams can be used to illustrate relationships between sets and to motivate an important counting theorem regarding sets – the inclusion/exclusion theorem.

The incl/excl theorem for sets makes use of both of these kinds of tricks.

We will discuss this theorem in class and prove it for n=2 and 3. A more general proof by induction can be constructed in a style similar to the proof of the general distributive law, as we did before.

Let |X| be the number of elements in a set X. This is often called the cardinality of X.

Then the incl/excl theorem for n=2 states.

$$|A \cup B| = |A| + |B| - |A \cap B|$$

A Venn diagram makes it clear why this theorem is true. A and B count the elements in $A \cup B$, but they count the $A \cap B$ section twice. The case for 3 can be analyzed with a little difficulty from a 3-circle diagram and results in:

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

The theorem generalizes to any number of sets, by adding up the cardinalities of the single sets, subtracting the cardinalities of the intersections of each pair, adding the cardinalities of the intersections of each triple etc.

The theorem can be used to solve a variety of straightforward and subtle counting problems. An example of a famous but subtle use is to calculate the number of derangements of a particular set. For example, if all of you bring lunch, and I collect them and redistribute them randomly, how many of the n! random permutations result in none of you getting your own lunches back? We will solve this problem in the unit on counting in two weeks.

An easier kind of problem that can be solved with incl/excl is the following type:

How many numbers between 1 and 100 are divisible by 3 or 7?

This is hard to count, but the number divisible by 3 and 7 is easy to count. It is just the number of numbers divisible by 21. Let A = the number of numbers between 1 and 100 that are divisible by 7, and B = the number of numbers between 1 and 100 that are divisible by 3. The theorem states that $|A \cup B| = |A| + |B| - |A \cap B|$. Hence the number of numbers between 1 and 100 that are divisible by 7 or 3 equals = 100/7 + 100/3 – 100/21 = 14 + 33 – 4 = 43.

Assume there are 12 people all of whom are either computer scientists or smart or both. Ten of them are smart and 5 are computer scientists. How many people are both smart and computer scientists? 12 = 10 + 5 – x. So x = 3.

There are more complicated versions of these kind of problems (see psets).

Sets as data structures

In most programming languages, sets are represented by bit strings. The number of bits is equal to the number of elements in the universal set. Ones appear in the slots of the elements contained in that set. Note that this implies an ordering to the elements of the set which does not strictly exist in the mathematical definition of a set.

It is convenient to store sets this way because:

1. It uses very little space, and
2. Set operations can be done using and/or/not bit-wise operators.

For example, assume you have 16 elements in the universal set and you want to know whether your set A contains element 3, then you can compute: A and '0010000000000000'. If this equals 0 then the answer is false, else true. Note that this is sometimes called masking, where the 0's mask out the bits in A that we do not care to look at. This also motivates the reason why in many languages, all 0's is considered false and anything else is true.

Any kinds of operations you want to do with sets can be simulated this way with bit operations.

The idea is often used in a different context when we want to look at particular bits in an arithmetic algorithm for overflow or carry information.

Functions and Countability of Sets

It is easy to compare the cardinality of finite sets by just seeing which set has greater or fewer elements. Comparing infinite sets is a more difficult issue. For example, which set has more elements, the set of all integers or the set of all even integers? Cantor, in the late 1800s, gave us a way to compare infinite sets. He suggested that two sets would have the same "size" if and only if there is a 1-1 correspondence between the elements in the two sets. In the previous example, there is such a 1-1 correspondence. An element x in the set of all even integers corresponds to the element x/2 in the set of all integers. This means that we must change our intuition to think of such sets as the same size even though there seems to be twice as many in one as the other.

We say that a set is countable iff it is the same size as the set of natural numbers.

Example: The set of all integers is countable.

Let x in the set of integers correspond to the natural number 2x if x >=0 and -(2x+1) if x<0.

Recitation: Pairs of integers are countable. Real numbers are not countable. Diagonalization. In the pset you will show that triples and n-tuples of integers are countable. Rational numbers are like pairs so they are countable.

The power set of a set A is the set of all subsets of A. The cardinality of the power set of A is 2^|A|. This can be proved by induction – see pset 2 – or by creating a 1-1 correspondence with the number of rows in a truth table. Given an assignment of T/F to a set of variables, associate the set of all variables that are marked true. Since we know there are 2^n assignments of T/F to n variables, then there are 2^|A| subsets of a set with |A| elements.

Cantor proved that the power set of A has cardinality greater than A. This gives a hierarchy of infinities. This hierarchy, as you will learn, implies the existence of functions that have no programs to compute them. That is, there are more functions than there are programs. In class we will discuss the relationship of this idea and diagonalization. In particular there is no program that computes whether an arbitrary program accepts itself or not.

Functions and Order of Growth

A function is a rule that maps each value in a domain to a particular value in some range. A function is onto, when every value in the range has at least one value in the domain that maps to it. A function is 1-1 when every value in the range has at most one value in the domain that maps to it. (These definitions are not always standard formal way to define these ideas, but they are equivalent). A function is a 1-1 correspondence when it is both onto and 1-1. When this is the case, then the inverse of the function is also a function. This is the kind of function that Cantor insisted on.

For future reference, R is the set of real numbers, N is the set of natural numbers, Z is the set of integers, Q is the set of rationals.

Examples: f(x) = x^2 maps from R to R. It is onto but not 1-1. It's inverse (square root) is not a function. f(x) = x+1 from R to R is onto and 1-1. It's inverse is f(x) = x-1. f(x) = x^2 maps from N to N is 1-1 but not onto. It does not have an inverse because not every integer has an integer square root. f(x) = $\lceil x \rceil$ maps from R to Z, is onto but not 1-1. Its inverse is not a function because one value gets mapped to many.

Functions, especially those with finite domain and range, are sometimes represented by a picture with arrows showing the mapping.

In CS, it is fundamental to be able to measure one function's rate of growth relative to another. Functions often represent time complexity of an algorithm where the input to the function is the size of the input to the algorithm. In order to compare which algorithm is theoretically faster or slower, we need to know what happens to the function as the size of the input grows. It is not enough to do some engineering and measure particular sample inputs on particular machines. Experimental measurements are worth doing but they can be misleading. We would prefer a metric that is independent of implementation and hardware. Note, this preference is an ideal, and the theory does not always win out over engineering.

We say that f(x) is O(g(x)) iff there exists constants c > 0 and x_0 > 0, such that f(x) <= cg(x) for all x > x_0. It means that f(x) is bounded above by g(x) once we get passed x_0, as long as we don't quibble about constant factors. This means intuitively that 3n^3 is O(n^3). Bounded below is defined similarly using Omega and >=. Bounded strictly above is defined using < and small-o.

We now work through a few examples showing how to find appropriate c and x_0 to prove that certain functions are Big-O of other functions.

2n^3 – n^2 + 8n + 6 is O(n^3). Let c = 17. 2n^3 – n^2 + 8n + 6 <= 2n^3 + n^3 + 8n^3 + 6n^3 = 17n^3, for all n >0.

Bubble sort gives a time complexity of n(n-1)/2. This is Omega(n^2) because n(n-1)/2 >= (1/3) n^2 for all n>3.

The minimum of steps to sort n items is lg (n!). We prove that this is BigTheta(nlogn).
lg n! is O(nlogn) by setting c=1, and noting that lg n! <= lg n^n = nlgn.
lg n! is Omega(n lgn) can be seen by writing n! >= n (n-1) … (n/2) >= (n/2)^(n/2). Hence lg n! >= (n/2)(lg n/2) = n/2(lgn) – n/2 <= n/4 (log n) for all n >4.

In recitation you can see an easier proof of this using Stirling's approximation for n!.

One can show that 2^(n+1) is O(2^n) but that 2^(2n) is not O(2^n). In the first case, set c=2. In the second case, note that the limit as n approaches infinite of (2^(2n))/ 2^n is infinite. Hence no c will ever work. This limit technique is especially useful. For example, we can prove that 2^n is not O(n^2), since the lim 2^n/n^2 = lim (2^n)''/(n^2)'' = lim ((ln 2)(ln2) 2^n/2) = infinite. (This uses L'Hospital's rule.

Sometimes we must make a change of variables to be able to more easily compare functions. Which is larger x ^lgx or (lg x)^x? Let x=2^n. Then x^(lgx) = 2^(n^2) and (lg x)^x = n^(2^n) = 2^((lg n)2^n). Hence (lg x)^x is larger because log n 2^n is bigger than n^2, as we showed just earlier.

An easier problem this time. Prove that both x lg (x^2) and (lg x^x) are big theta (x log x). Details left to you.

There are other techniques for estimating growth including integration, and example of which will be discussed in recitation, where we show that the sum of 1/I for I = 1 to n is Big theta log n.

Working with sums.

It is worth getting good at manipulating sums in discrete math because they come up so often. Today we look at the sum of the first n squares and derive a formula. This formula can be estimated by integration ($n^2/3$), and it can be proved by induction, but the proof by induction is not so helpful in discovering the formula. Contrast this with the proof for the sum of the first n cubes on your pset, where the induction implies the formula. In 1321, Levi ben gershon proved formulas for the sum of the first n integers, squares and cubes. He used induction only for the cubes.

Let's start with the sum 1 + 3 + 5 + 7 + .. + 2n-1. It doesn't take too long to realize that this equals n^2. A picture proves it.

```
*      *|*     *|*|*    *|*|*|*
       * *     * *|*    * *|*|*
               * * *    * * *|*
                        * * * *
```

This can of course also be proved by induction and the proof is natural.

Now note that $1^2 + 2^2 + 3^2 + \ldots = 1 + (1+3) + (1+3+5) + \ldots$

= Sum from i = 1 to n of (2i-1)(n-i+1). This is more clear if you write the sum above like this:

1+
1+3+
1+3+5+
1+3+5+7+…

The key point here is that notation is not only useful as a shorthand – but it affects the way we think and what we are able to think about.

This example will help us learn how to manipulate Sum notation, and appreciate the need for it.

Sum from i = 1 to n of (2i-1)(n-i+1) = Sum 1 to n of ($2in - 2i^2 + 2i - n + i - 1$)

This implies that 3 * Sum of squares = (2n+3)Sum(i) $-n^2 - n$ = (2n+3)(n)(n+1)/2 – n(n+1)

Hence Sum of squares = (2n+1)(n)(n+1)/6

Recurrence Relations and Geometric Sums

Compound Interest….

Start with X dollars at 10% year.
The number of dollars after the nth year equals 1.1 times the number of dollars after the previous year. That is, $D(n) = 1.1 * D(n-1)$, and $D(0) = X$.

This is called a recurrence equation. Recursion, mathematical induction, and recurrence equations are three legs of a three-legged stool. The algorithm uses recursion, the proof it word uses mathematical induction, and the analysis of the time requirements results in a recurrence equation.

The easiest and most common sense way to solve a recurrence equation is to use what I call repeated substitution. It is a primitive brute force method that relies, in difficult cases, on the ability to manipulate sums.

$D(n) = 1.1 * D(n-1)$. So we substitute for $D(n-1)$, using $D(n-1) = 1.1 * D(n-2)$, and we get:
$D(n) = 1.1^2 * D(n-2)$. Continuing this r times, gives:

$D(n) = 1.1^r * D(n-r)$.

Now $D(0) = X$, so if we let $r = n$, then we get:

$D(n) = 1.1^n * X$

The number of dollars after n years is shown below:

Years	Dollars
0	X
1	$1.1 * X$
2	$1.1^2 * X$
…	
n	$1.1^n * X$

This recurrence is the simplest possible example. The sum often develops into a geometric sum or something more complicated. Sometimes the method gives a sum that's too ugly to work with, and we need to use a different method.

Binary Search –

Ask if your guess is higher or lower to guess my secret number between 1 and n. Each guess that you make halves the possible remaining secret numbers. The time for this algorithm is:
$T(n) = T(n/2) + 1$ and $T(1) = 0$.

Using our substitution method we get:

T(n) = T(n/2^r) + r, after r iterations.

Let r = lg n and this becomes T(n) = lgn

Towers of Hanoi -- the legend is only 100 years old or so ☺

Define ToH (n, From, To, Using)

```
If n >0 {
        ToH (n-1, From, Using, To);
        Display ('move disk from' From 'to' To);
        ToH (n-1, Using, To, From);
        }
```

Let's analyze the time complexity, solve the resulting recurrence equation, and look at some special cases. Then we look at a graph that will give us a bird's eye view of the Hanoi recursive jungle. The power of graphs will be seen in this example, and throughout the pset.

T(n) = 2T(n-1) + 1   T(0) = 0

After 1 iteration T(n) = 2^2 T(n-2) + 2 + 1

After r iterations T(n) = 2^r T(n-r) + 2^(r-1) + 2^(r-2) + … + 4 + 2 + 1

Letting r = n, we get Sum i = 1 to n-1 of 2^i.

This is called a geometric series. In a geometric series, each subsequent term increases by a fixed multiplicative factor. Euclid (300 B.C.E.) knew all about geometric series and how to sum them. An arithmetic series is one where each subsequent term increases by a fixed sum. The triangle numbers represent an arithmetic series.

The trick to sum a geometric series is to multiply the series by the fixed multiplicative factor, and note that the result is the same series just shifted over one term. For example.

Let x = 1 + 2 + 2^2 + … + 2^(n-1)

Then 2x = 2 + 2^2 + … + 2^(n-1) + 2^n

At this point we subtract the two equations to give:

2x – x = x = 2^n – 1

Hence for ToH, T(n) = 2^n –1

Now that we have opened the box of geometric series, let's review 1/2^i.

Let's also consider the sum of i(2^i), or i^2(2^i), or i^k(2^i). None of these are geometric series but they can all be handled by the same trick in an especially inductive way, where the next case reduces to the simpler case, and finally to the original geometric series.

Example:

Let x = 1*2 + 2*2^2 + 3*2^3 + … + n*2^n

Here 2x –x = x = n*2^(n+1) – 2 – (2^2 + 2^3 + … + 2^n)

We get a formula with a geometric series in it. This formula equals n*2^(n+1) – 2 – (2^(n+1) – 4) = (n-1) 2^(n+1) +2

Many other basic sums can be managed with repeated use of this one trick.

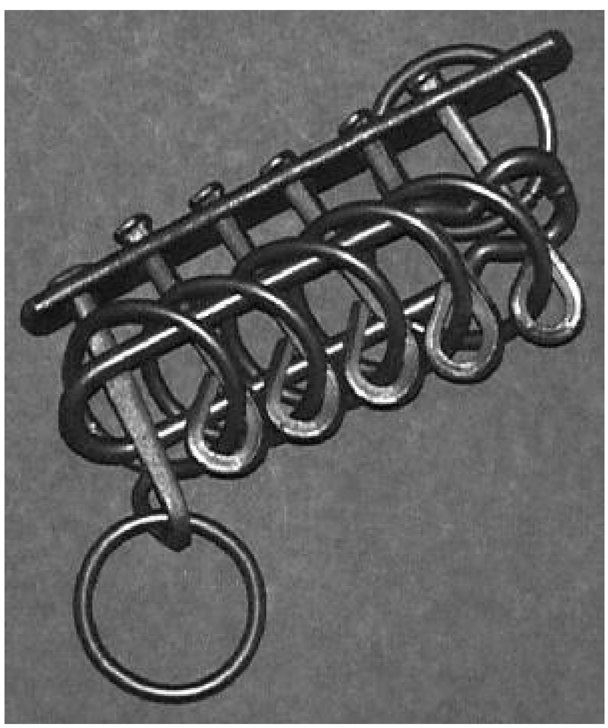
The Hanoi Graph

The Hanoi graph will be shown and discussed in class. You can look for a picture on the web on Eric Weisstein's math site mathworld.wolfram.com. It is constructed recursively, defined inductively and analyzed. It will give us a blueprint of the computation for ToH. Note that a solution to ToH is a path through this graph.

The next example of recursion is an excellent one for motivating induction. We will discover the truth about the seven rings puzzle, and discover its connection to Hamiltonian circuits in hypercubes, and to Gray Codes.

An Example of Motivating Mathematical Induction for Computer Science

The Chinese Rings or Patience Puzzle



A Recursive Method to Remove Rings and Unlock the Puzzle

To Remove the n rings:

Reduce the puzzle to an n-1 ring puzzle.

Remove the leftmost n-1 rings.

End

To Reduce the puzzle to an n-1 ring puzzle:

Remove the leftmost n-2 rings.

Remove the nth ring.

Replace the leftmost n-2 rings.

End

Resulting Recurrence Equation

$$T(n) = 1 + T(n-1) + 2T(n-2)$$

$$T(1) = 1 \qquad T(2) = 2$$

Analysis and Solution

For Towers of Hanoi $T(n) = 2T(n-1) + 1$, $T(1)= 1$, we solved the recurrence by repeated substitution.

Substituting $T(n-1) = 2T(n-2) + 1$ back into $T(n) = 2T(n-1) + 1$ implies $T(n) = 4T(n-2) + 1 + 2$

After r substitutions we get:

$T(n) = 2^r\, T(n-r) + (1 + 2 + 4 + \ldots + 2^{r-1})$, and
$T(n) = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1$

But here…

$$T(n) = 1 + T(n-1) + 2T(n-2)$$

$$T(1) = 1 \qquad T(2) = 2$$

The same technique after one iteration would imply:

$$T(n) = 1 + 1 + 2 + T(n-2) + 4T(n-3) + 4T(n-4)$$

Should we continue? Ugh!!!

Let's Experiment

(Recursion versus Dynamic Programming)

$$T(n) = 1 + T(n-1) + 2T(n-2)$$

$$T(1) = 1 \qquad T(2) = 2$$

n	T(n)
1	1
2	2
3	5
4	10
5	21
6	42
7	85

Let's Guess…

When n is even: $T(n) = 2T(n-1)$

When n is odd: $T(n) = 2T(n-1) + 1$

Proving this directly is not obvious. However, a proof by induction is natural and easy.

Logo Program to Experiment

```
to chinese :n              ; an inefficient recursive program
       if (= :n 1) op 1
       if (= :n 2) op 2
       op (+ 1 (chinese (- :n 1)) (* 2 chinese (- :n 2))
end

to chinese2 :n             ; a fast computation of the closed form
       if (= :n 1) op 1
       if (= :n 2) op 2
       if (even? :n) op (/(* 2 (-(exp 2 :n) 1)) 3)
       op (/ (-(exp 2 (+ :n 1)) 1) 3)
end

to exp :a :b
       make "x 1
       repeat :b [make "x (* :a :x)]
       op :x
end

to even? :any
       op (= (remainder :any 2) 0)
end

to odd? :any
       op (not (even? :any)
end
```

Exercises:

Write an Iterative Version.
Write a Tail Recursive Version.

Solution and Closed Form

$$T(n) = 1 + T(n-1) + 2T(n-2)$$
$$T(1) = 1 \qquad T(2) = 2$$

When n is even: $T(n) = 2T(n-1)$
When n is odd: $T(n) = 2T(n-1) + 1$

Now we can use repeated substitution to get:

$T(n) = 4T(n-2) + 2$, when n is even.
$T(n) = 4T(n-2) + 1$, when n is odd.

Continuing our substitutions gives:

$T(n) = 2/3\ (2^n - 1)$, when n is even.
$T(n) = 1/3\ (2^{n+1} - 1)$, when n is odd.

The Chinese Ring Puzzle motivates:

1. An Understanding of Recursion.
2. *Natural* proofs by induction.
3. Construction, analysis and solution of recurrence equations.
4. Complexity analysis of recursive programming versus dynamic programming.
5. Binary Grey Codes.
6. Graph Representations and data structures.
7. Experimenting and Guessing.