File Structures

A **file** is a collection of data stored on mass storage (e.g., disk or tape)
Why on mass storage?

- too big to fit in main memory
- share data between programs
- backup (disks and tapes are less volatile than main memory)

The data is subdivided into **records** (e.g., student information).

Each record contains a number of **fields** (e.g., name, GPA).

One (or more) field is the **key** field (e.g., name).

Issue: how to organize the records on the mass storage to provide convenient access for the user?

We will discuss sequential files, indexed files, and hashed files.

Sequential Files

Records are *conceptually* organized in a sequential list and can only be accessed sequentially.

The actual storage might or might not be sequential:

- On a tape, it usually is.
- On a disk, it might be distributed across sectors and the operating system would use a linked list of sectors to provide the illusion of sequentiality.

Convenient way to batch (group together) a number of updates:

- Store the file in sorted order of key field.
- Sort the updates in increasing order of key field.
- Scan through the file once, doing each update in order as the matching record is reached.

Not a convenient organization for accessing a particular record quickly.

Indexed Files

Sequential search is even slower on disk/tape than in main memory. Try to improve performance using more sophisticated data structures.

An **index** for a file is a list of key field values occurring in the file along with the address of the corresponding record in the mass storage.

Typically the key field is much smaller than the entire record, so the index will fit in main memory.

The index can be organized as a list, a search tree, a hash table, etc. To find a particular record:

- Search the index for the desired key.
- When the search returns the index entry, extract the record's address on mass storage.
- Access the mass storage at the given address to get the desired record.

Multiple indexes, one per key field, allow searches based on different fields.

Hashed Files

An alternative to storing the index as a hash table is to not have an index at all.

Instead, hash on the key to find the address of the desired record and use open addressing to resolve collisions.

The usual hashing considerations arise.

Databases

A **database** is a collection of data in mass storage that can

- take on a variety of appearances and
- can be used by a variety of applications.

Example: Collection of student records can be viewed as a database to be used by:

- payroll
- mailing out report cards
- preparing tuition bills
- etc.

The advantages of consolidating the data:

- saves space
- saves duplication of effort to enter, update or correct information
- centralized control within the organization

Database System Organization

The “software architecture” of a database system is usually layered:

- End user calls application software to access the data. End user thinks of data in terms of the application
- Application software calls database management system (DBMS) software. The applications software has a conceptual view of the data.
- DBMS deals with the nitty gritty details of data storage (indexing, sectors, etc.).

As usual, the advantages of layering are that changes can be made to lower level implementations without affecting higher levels.

Communication with a Database

Databases usually provide a useful and powerful interface for obtaining information from them. So far, we've just seen requests of the form:

- add/delete/search for a record with a given key
- find min/max/pred/succ
- print out all the keys

But suppose you'd like to print out the names of all students that are freshman and either have a 4.0 GPA or whose names start with X.

There are ways to conceptually organize the data to allow such **queries** to be answered efficiently, using what are called **tables** or **relations**.

- The application software communicates with the DBMS in terms of this "relational model".
- The DBMS must translate from the relational model into the actual storage data structures.

Database Integrity

Data in a database is typically

- long-lived and
- of crucial importance to the organization.

Thus it must not get corrupted.

Data can be corrupted if several different programs (or **transactions**) accessing the database at the same time.

Example of corrupted data:

- T1 transfers \$100 from account x to account y.
- T2 inventories how much money the bank has.

Suppose this sequence of events occurs:

- T1 subtracts \$100 from account x.
- T2 gets the balance from account x.
- T2 gets the balance from account y.
- T1 adds \$100 to account y.

T2's total balance is \$100 too small.

DB Serializability

To prevent transactions from interfering with each other, the DBMS should *provide the illusion that each transaction runs in isolation*.

This property is called **serializability**.

The DMBS does not have to (and should not) actually make the transactions run serially, but if there is a potential conflict, the DBMS must take steps.

One solution is **two-phase locking:**

- Before accessing any data item, the transaction must obtain a **lock** for *every* data item it plans to access.
- Only one transaction at a time can have a lock on the same data item.
- If another transaction already has the lock, then the first one must wait.
- After accessing all the data items, transaction releases all its locks.

Committing and Aborting a Transaction

Two-phase locking can lead to **deadlock**, e.g.:

- T1 locks data item A
- T2 locks data item B
- T1 waits for data item B
- T2 waits for data item

The DBMS must periodically check for deadlock, and if one is discovered, it must choose a transaction to be **aborted** to break the deadlock.

If the aborted transaction has already made changes to the database, the DBMS must **roll back** those changes:

- either keep a **log** of the changes made (the before and after values) or
- don’t actually make the changes in the log until the transaction has completed.

Once the transaction has successfully completed, then it is **committed**, and the changes are installed in the database.

Artificial Intelligence

Goal: Develop machines that communicate with their environment through traditionally human sensory means, such as

- vision
- speech recognition

and proceed ”intelligently” without human intervention, e.g.,

- planning
- expert systems
- reasoning

Distinct but related goals:

1. trying to make machines actually ”intelligent” (whatever that would mean),
2. improving technology,
3. understanding how the human mind works by trying to model it

8-Puzzle Example

Given a 3-by-3 box that holds 8 tiles, numbered 1 through 8. One tile is missing. The goal is to start with the tiles scrambled and move them around so that they are in order:

1	2	3
4	5	6
7	8	

We will try to solve this problem by a machine that has

- a gripper, to hold the box
- a video camera, to see where the tiles are
- a computer, to decide how to move the tiles
- a “finger”, to move the tiles.

Ideas from mechanical engineering can be used to implement the gripper and the finger. We will talk about how to “see” where the tiles are, and how to decide how to move the tiles.

Computer Vision

It is not enough to simply store the image obtained from the camera. The program must be able to *understand* the image:

- figure out which parts of the image are the salient objects, called **feature extraction**
- and then recognize the objects by comparing them to known symbols, called **feature evaluation**.

For the 8-puzzle, this problem can be highly simplified:

- always expect the digits to be the same size (by holding the box at a constant distance from the camera)
- same perspective
- small set of different images to be handled (8 numbers and blank)
- no obstruction (one object overlapping another)

But in general this is a very difficult problem and one where there has been extensive research.

Reasoning

How can the program solve the puzzle?

One solution is to preprogram solutions, i.e., look up the solution in a table. For example, if the input is

1	2	3
4	5	6
7		8

then the solution is to move the bottom right tile to the left.

But in this case there are approximately 9! = 362,880 different inputs, some of which require a long sequence of moves to solve, and it would require a lot of space.

Plus, someone would have to figure out all the answers in advance.

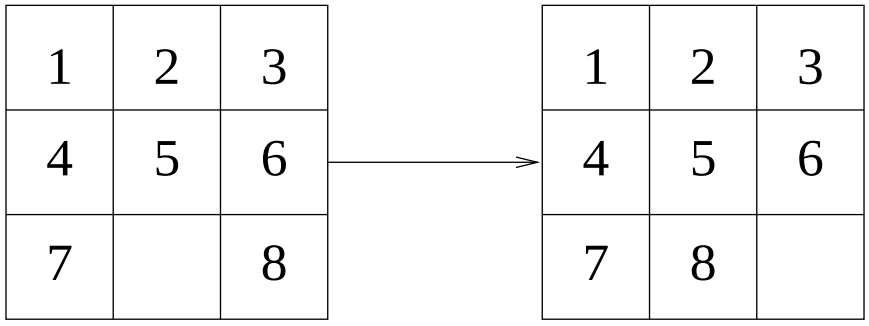
Production Systems

Instead, have the program figure out the solution. One approach is the **production system model**.

First, consider the **state graph** of the problem:

- Every possible **state** of the system is a node.
- Draw an arrow from one node to another if a single **move** (or **production**, or **rule**) takes you from one state to the other.

Here is a tiny piece of the state graph for the 8-puzzle:



Identify the **start** and **goal** states of the state graph.

The **control system** figures out how to get from the start state to the goal state, by following arrows in the state graph.

Solving a Production System

We must find a path through the state graph from the start state to the goal state.

Luckily, finding paths in graphs is a very general problem that has been much studied.

One way is to build a **search tree** (not to be confused with a binary search tree), which indicates the part of the state graph that has been explored so far.

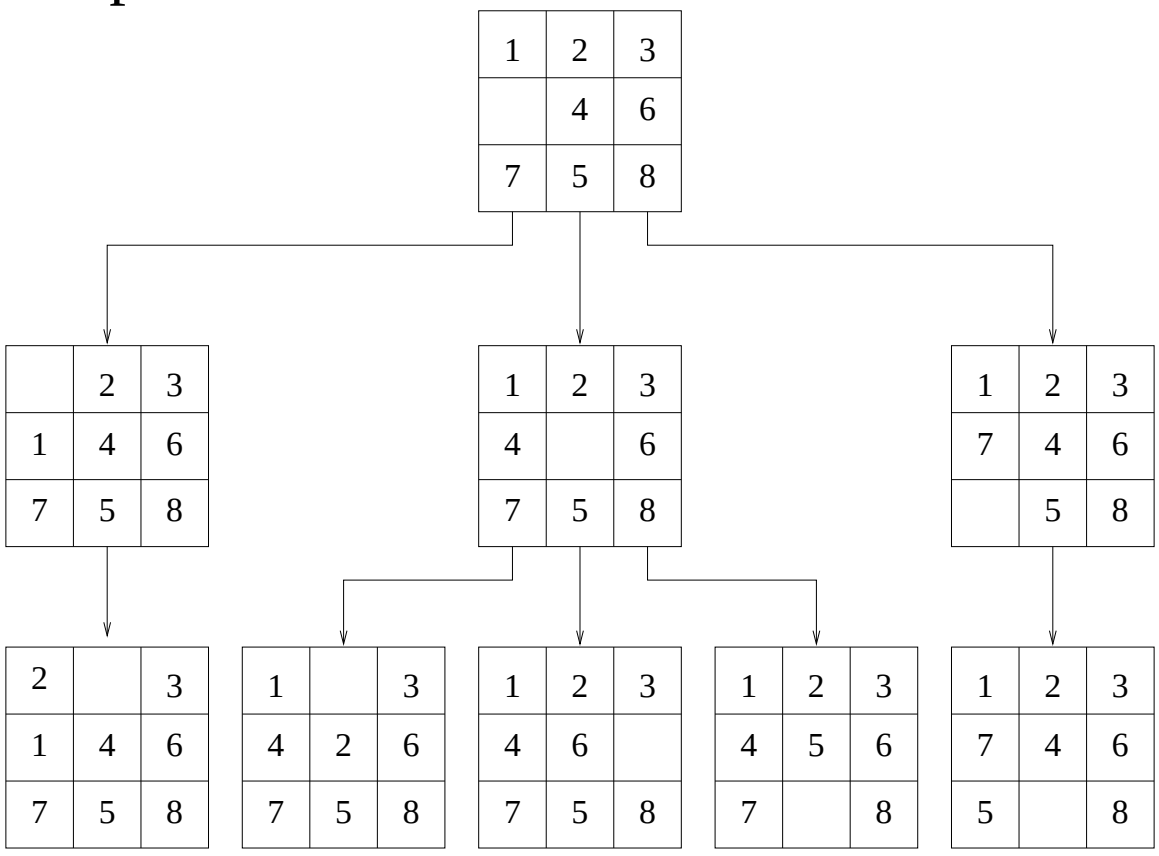
Two solutions are breadth-first search and depth-first search.

Breadth-First Search

Build the search tree in a **breadth-first** manner:

- The root is the start state.
- The next level is all states reachable from the start state with a single production.
- The next level is all states reachable from states in the first level with a single production. Etc.

For example:



But the search tree grows exponentially.

Depth-First Search

Another approach is to search the state space **depth first**, instead of breadth first.

Pursue more promising paths to greater depths and consider other options only if the original choices turn out to be false leads.

To implement this idea, we need some criterion to decide which paths are promising, or *appear* to be promising.

Such criteria are called **heuristics**. A heuristic is a rule of thumb for the program.

We need something quantitative so we can compare different choices and choose the best.

Heuristic for 8-Puzzle

For the 8-puzzle example, our intuitive rule of thumb is to try to move pieces toward their final destination.

A quantitative heuristic measure is: take the sum, over all the tiles, of the minimum number of moves needed to get that tile to its final position (ignoring the presence of other tiles).

For instance, if the input is

1	2	6
5		3
8	7	4

then the heuristic measure is
0 + 0 + 1 + 3 + 1 + 1 + 1 + 1 = 8.

This heuristic has two desirable properties:

1. it is a reasonable estimate of the remaining work
2. it is easy to calculate

Using a Heuristic in Depth-First Search

- Repeatedly check *all* leaves in the search tree,
- Choose the leaf with the smallest heuristic measure.
- Generate all children of that leaf.
- Continue until goal state is found.

In the 8-puzzle example above:

- Generate the root. Its heuristic measure is 3.
- Generate all children of the root. They have measures 4, 2, and 4.
- Choose the leaf with measure 2 and generate all its children. They have measures 3, 3, 1.
- Choose the leaf with measure 1 and generate all its children. They have measures 2 and 0. Goal state is found.

In this depth-first search, we only had to generate 9 states, instead of approximately 17 in the breadth-first case.

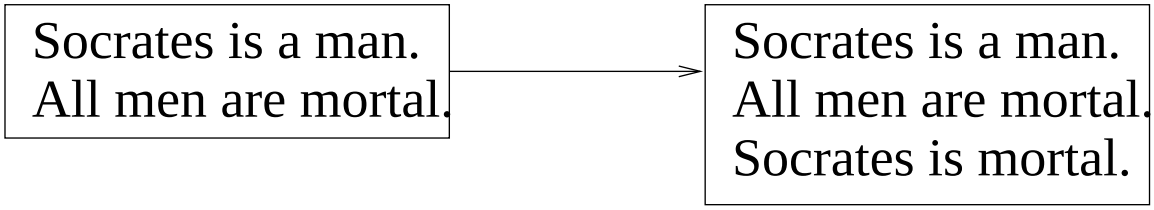
Other Applications of Production Systems

Many problems can be formulated as production systems. In addition to the 8-puzzle, chess can be also.

You can even model the process of drawing logical conclusions from a set of given facts as a production system. In this case,

- each state is a collection of facts that are known to be true.
- a production/rule/move corresponds to a rule of logic that allows an additional fact to be deduced.

For instance, part of the state graph might be:



since there is a rule of logic that says: Given the facts

1. X is a Y
2. All Y are Z

then you can deduce that “X is Z” is also a fact.

Some Other Areas of AI

Neural Networks: Try to take advantage of the power of parallelism (multiprocessor computer architectures) using a paradigm that (roughly) follows the model of neurons in biological systems.

Robotics: Hardware and software working together, e.g., automated manufacturing. Great interest in having machines explore and function in uncontrolled and unpredictable environments, such as

- outer space
- underwater
- inside a nuclear waste dump

Expert Systems: Combine domain specific knowledge from human experts with some kind of deduction system. For example:

- medical
- seismic exploration for oil and gas

Time Complexity of an Algorithm

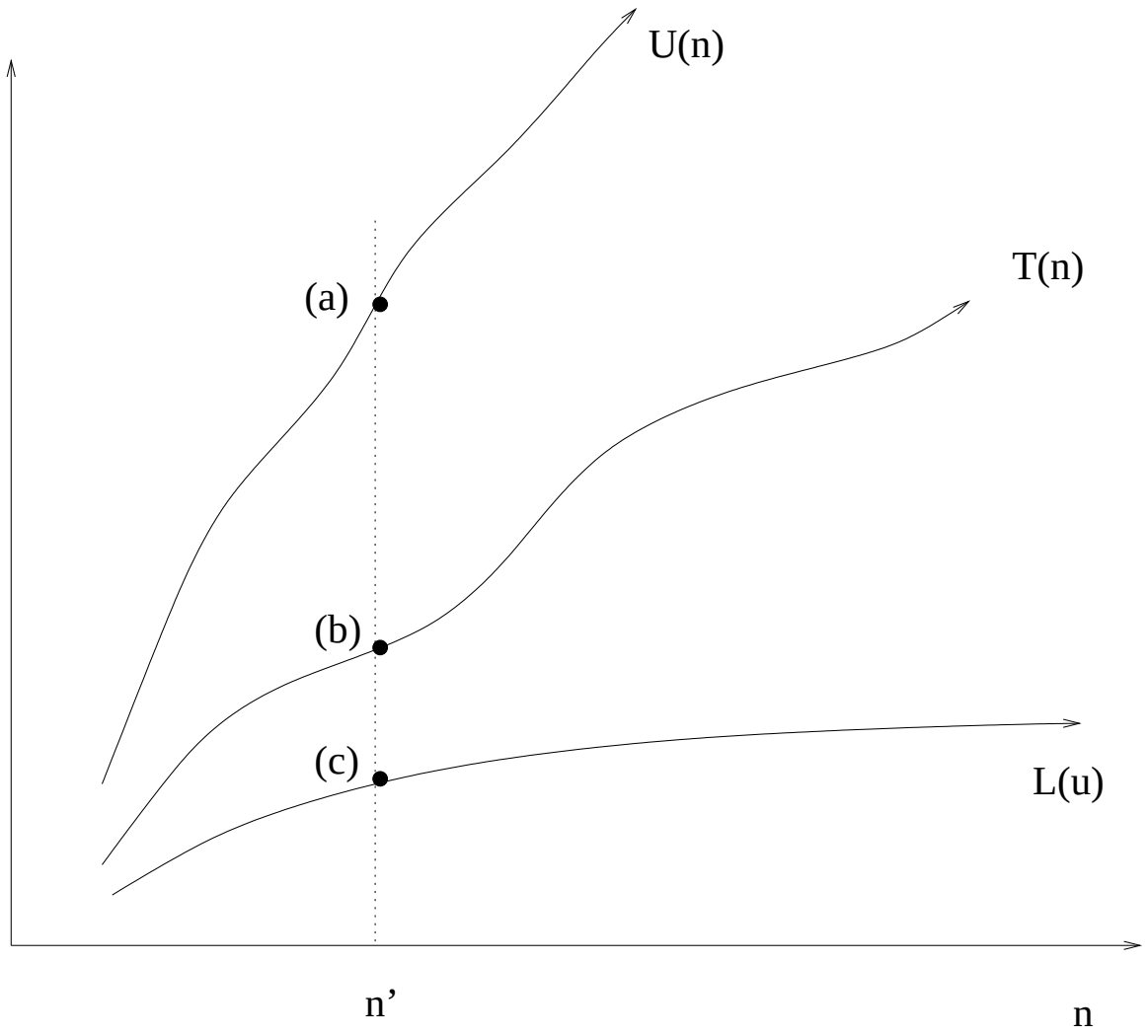
Time complexity of an algorithm: the function $T(n)$ that describes the (worst-case) running time as input size, n, increases.

Given a particular algorithm, discover this function by attacking the problem from two directions:

- find an *upper bound* $U(n)$ on the function $T(n)$, i.e., convince ourselves that the algorithm will *never take more than* $U(n)$ *time on any input of size* n.
- find a *lower bound* $L(n)$ on the function $T(n)$, i.e., convince ourselves that, for each n, there is *at least one input of size* n *on which the algorithm takes at least time* $L(n)$.

Try to find smallest U and largest L, so that T is squeezed in between and has no room to hide.

Time Complexity of an Algorithm (cont'd)



(a) No execution on an input of size n' takes more time than this.
(b) The slowest execution on all inputs of size n' takes exactly this much time.
(c) At least one execution on an input of size n' takes at least this much time.

Time Complexity of Heapsort

Let $T(n)$ be the time complexity of heapsort.

First cut at upper bound: each heap operation never takes more than $O(n)$ time. Thus $T(n)$ is at most $O(n^2)$.

First cut at lower bound: each heap operation always takes at least $O(1)$ time. Thus $T(n)$ is at least $O(n)$.

Refined argument for upper bound: each heap operation never takes more than $O(\log n)$ time. Thus $T(n)$ is at most $O(n \log n)$.

Refined argument for lower bound: Describe a particular input that causes running time of at least $O(n \log n)$.

On input $n, n-1, n-2, \ldots, 3, 2, 1$, running time is at least $O(\log 1 + \log 2 + \ldots + \log n) = O(n \log n)$.

Thus $T(n)$ now precisely identified as $O(n \log n)$ (to within constant factors).

Time Complexity of a Problem

Time complexity of a problem: the time complexity for the *fastest possible algorithm* for the problem.

To show that a problem has time complexity $T(n)$:

- Identify a specific algorithm for the problem with time complexity $T(n)$.
- Then prove that *any algorithm* for the problem has time complexity at least $T(n)$.

Example: Sorting problem has time complexity $O(n \log n)$.

- Heapsort has time complexity $O(n \log n)$.
- It can be proved that no (comparison-based) sorting algorithm can have better time complexity.

Problems can be classified by their time complexity. *Harder* problems are considered to be those with larger time complexity.

The Class P

All *problems* (not algorithms) whose time complexity is at most some polynomial are said to be **in the class P** (P for polynomial).

Example: Sorting is in P, since $O(n \log n)$ is less than $O(n^2)$.

Not all problems are in P.

Example: Consider the problem of listing all permutations of the integers 1 through n.

- Output size is $n!$.
- Thus running time is at least $O(n!)$.
- $n!$ is larger than 2^n, thus larger than any polynomial.

NP-Complete Problems

There is an important class of problems that might or might not be in P — nobody knows!

These problems are called **NP-complete**.

These problems have the following characteristic:

- *A candidate solution can be verified in polynomial time as being a real solution or not.*
- *However, there are an exponential number of candidate solutions.*

Many real-world problems in science, math, engineering, operations research, etc. are NP-complete.

Traveling Salesman Problem

An example NP-complete problem is the **traveling salesman problem:**

Given a set of cities and the distances between them, determine an order in which to visit all the cities that does not exceed the salesman's allowed mileage.

A candidate solution for TSP is a particular listing of the cities.

To check whether the allowed mileage is exceeded, add up the distances between adjacent cities in the listing, which will take time linear in the number of cities.

But the total number of different candidate solutions is $n!$, so it's not feasible to check them all.

P vs. NP

Imagine an (unrealistically) powerful model of computation in which the computer first makes a lucky **guess** (a nondeterministic choice) as to a candidate solution in constant time, and then behaves as an ordinary computer and verifies the solution.

Problems solvable on this computer in polynomial time are **in the class NP** (nondeterministic polynomial).

NP includes all the NP-complete problems.

Having polynomial running time on this funny computer would not seem to ensure polynomial running time on a real computer.

That is, it seems likely that NP is a strictly larger class of problems than P, and that the NP-complete problems cannot be solved in polynomial time.

But no one has yet been able to prove $P \neq NP$. Outstanding open question in CS since the 1970's.

Computability Theory

Complexity theory focuses on how expensive it is to solve various problems.

Computability theory focuses on which problems are *solvable at all* by a computer (i.e., with an algorithm), regardless of how expensive a solution might be.

We will focus on computing (mathematical) *functions*, with inputs and outputs.

We would like to know if there exist functions that are so complicated that no algorithm can compute them.

Church-Turing Thesis

First, we have to decide what constitutes an algorithm.

- Assembly languages have restricted sets of primitives.
- High-level languages have a wider choice of primitives.
- What's to say you couldn't have some language with very powerful primitives?

Church-Turing thesis: ("thesis" means "conjecture") Anything that can reasonably be considered an algorithm can be represented as a **Turing machine**.

A Turing machine is a very abstract, yet low-level, model of computation.

Every actual programming language is equivalent, in computational power, to the Turing machine model.

Thus, for theoretical purposes, the choice of programming language is irrelevant.

Computing Functions

Some sample functions:

- $f(n) = 3$: very easy to compute, always return 3, no matter what the input is
- $f(n) = 2n$: easy to compute, since multiplication can be done with an algorithm
- $f(n) = \sin n$: getting more complicated, especially with issues of precision

There exist **non-computable** functions, functions whose input/output relationships are so complicated that there is no well-defined, step-by-step process for determining the function's output based on its input value.

We will assume

- your favorite programming language L,
- with a very powerful implementation, in which integers can be any length, and
- only consider programs in L that take a single integer input and produce a single integer output.

Goedel Number of a Program

Here is a way to convert a program into an integer.

- Convert all the characters in the program to their ASCII codes.
- Interpret the result as a (big) integer. Call this integer the **Goedel** number of the program.

Conversely, any integer can be converted into a series of characters:

- Most of the time, the result is garbage.
- Sometimes it isn't garbage, but it isn't a legal program in language L.
- Rarely, it is a legal program in L.
- More rarely, the resulting program has a single integer input and single integer output.

Use this numbering scheme to *list all the programs in language L*. The list is infinite.

An Uncomputable Function

Define a function h called the **halting problem**:

- If the program with Goedel number n halts when its input is n, then $h(n) = 1$.
- If the program with Goedel number n does not halt when its input is n, then $h(n) = 0$.

Theorem: h is uncomputable (has no program in the Goedel listing).

Proof: Assume in contradiction that h is computable. Then some program H (in the Goedel listing) computes the function h.

Define another program I (which will be in the listing):

1. n is the input
2. run program H as a subroutine on input n
3. let x be the output returned by H
4. if $x = 0$ then return 0
5. else go into an infinite loop

An Uncomputable Function (cont'd)

Let n_I be the Goedel number of I. What does I do on input n_I?

Case 1: I halts on input n_I. Then in Line 4, $x = 0$, i.e., the subroutine *H* returned 0, meaning that I does *not* halt on n_I. Contradiction.

Case 2: I does not halt on input n_I. Then in Line 4, $x = 1$, i.e., the subroutine *H* returned 1, meaning that I *does* halt on n_I. Contradiction.

Thus the hypothetical program H cannot exist.
□

Another way to view this result is that there is only a countably infinite number of programs (algorithms), but there are uncountably many functions.