Introduction to Languages and Computation

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Reducibility

- We look at several unsolvable problems beyond A_{TM}.
- The primary method for proving that problems are computationally unsolvable is called **reducibility**.
- A **reduction** is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem.
- Reducibility always involves two problems, which we call A and B.
  - If A reduces to B, we can use a solution to B to solve A.
  - Reducibility says nothing about solving A or B alone, but only about the solvability of A, given a solution to B.
- Examples:
  - The problem of measuring the area of a rectangle reduces to the problem of measuring its length and width.
  - The problem of solving a system of linear equations reduces to the problem of inverting a matrix.

Reducibility and Decidability

- Reducibility plays an important role in classifying problems by decidability and later in complexity theory as well:
  - When A is reducible to B, solving A cannot be harder than solving B because a solution to B gives a solution to A.
  - In terms of computability theory, if A is reducible to B and B is decidable, A is also decidable.
  - Equivalently, if A is undecidable and reducible to B, B is undecidable.
- To prove that a problem is undecidable, show that some other problem already known to be undecidable reduces to it.

Subsection 1

Undecidable Problems from Language Theory

The Halting Problem

- We have already established the undecidability of A_{TM}, the problem of determining whether a Turing machine accepts a given input.
- We consider now $\mathrm{HALT}_{\mathrm{TM}}$, the problem of determining whether a Turing machine halts by accepting or rejecting, on a given input.
- We use the undecidability of A_{TM} to prove the undecidability of $\mathrm{HALT}_{\mathrm{TM}}$ by reducing A_{TM} to $\mathrm{HALT}_{\mathrm{TM}}$.
- Let $\mathrm{HALT}_{\mathrm{TM}} = \{\langle M, w\rangle : M \text{ is a TM and } M \text{ halts on input } w\}$.

Theorem

$\mathrm{HALT}_{\mathrm{TM}}$ is undecidable.

- We assume that $\mathrm{HALT}_{\mathrm{TM}}$ is decidable and use that assumption to show that A_{TM} is decidable, a contradiction. The key idea is to show that A_{TM} is reducible to $\mathrm{HALT}_{\mathrm{TM}}$. Assume that we have a TM R that decides $\mathrm{HALT}_{\mathrm{TM}}$. Then we use R to construct a TM S that decides A_{TM}.

Undecidability of the Halting Problem

- Given an input of the form $\langle M, w \rangle$, S must output accept, if M accepts w, and reject, if M loops or rejects on w. Use R to determine whether M halts on w. If R indicates that M does not halt on w, then reject. However, if R indicates that M does halt on w, you can simulate M on w without any danger of looping. Thus, if TM R exists, we can decide A_{TM}, but we know that A_{TM} is undecidable. So, R does not exist. Therefore, $\mathrm{HALT}_{\mathrm{TM}}$ is undecidable.
- Let us assume for the purposes of obtaining a contradiction that TM R decides $\mathrm{HALT}_{\mathrm{TM}}$. We construct TM S to decide A_{TM}:
  S: On input $\langle M, w \rangle$, an encoding of a TM M and a string w,
  1. Run TM R on input $\langle M, w \rangle$.
  2. If R rejects, reject.
  3. If R accepts, simulate M on w until it halts.
  4. If M accepts, accept; if M rejects, reject.

  Clearly, if R decides $\mathrm{HALT}_{\mathrm{TM}}$, then S decides A_{TM}. Because A_{TM} is undecidable, $\mathrm{HALT}_{\mathrm{TM}}$ must also be undecidable.

Emptiness Testing for Turing Machines

- Let $\mathrm{E}_{\mathrm{TM}} = \{\langle M \rangle : M \text{ is a TM and } L(M) = \emptyset\}$.

Theorem

E_{TM} is undecidable.

- We assume for the purposes of obtaining a contradiction that E_{TM} is decidable and then show that A_{TM} is decidable, a contradiction.

  Let R be a TM that decides E_{TM}. We use R to construct TM S that decides A_{TM}. When S receives input $\langle M, w \rangle$, we run R on a modification of $\langle M \rangle$ that guarantees that M rejects all strings except w, but on input w it works as usual. Then we use R to determine whether the modified machine recognizes the empty language. The only string the machine can now accept is w, so its language will be nonempty if and only if it accepts w. If R accepts when it is fed a description of the modified machine, we know that the modified machine does not accept anything and that M does not accept w.

Undecidability of Emptiness Testing

- We call the modified machine, described above, M_1:
  M_1: On input x
  1. If $x \neq w$, reject.
  2. If $x = w$, run M on input w and accept if M does.

  This machine has the string w as part of its description. It conducts the test of whether $x = w$ by scanning the input and comparing it character by character with w to determine if they are the same.
- Assume TM R decides E_{TM} and construct TM S that decides A_{TM}:
  S: On input $\langle M, w \rangle$, an encoding of a TM M and a string w,
  1. Use the description of M and w to construct the TM M_1 just described.
  2. Run R on input $\langle M_1 \rangle$.
  3. If R accepts, reject; if R rejects, accept.

  S must actually be able to compute a description of M_1 from a description of M and w. This only requires adding extra states to M that perform the $x = w$ test. If R were a decider for E_{TM}, S would be a decider for A_{TM}. Since a decider for A_{TM} cannot exist, E_{TM} must be undecidable.

Equivalence with a Finite Automaton

- Let $\text{REGULAR}_{\text{TM}}$ be the problem of determining whether a given Turing machine has an equivalent finite automaton.
- This problem is the same as determining whether the Turing machine recognizes a regular language.

  $\text{REGULAR}_{\text{TM}} = \{\langle M \rangle : M \text{ is a TM and } L(M) \text{ is regular language}\}$.

Theorem

$\text{REGULAR}_{\text{TM}}$ is undecidable.

- The proof is by reduction from A_{TM}. We assume that $\text{REGULAR}_{\text{TM}}$ is decidable by a TM R and use this assumption to construct a TM S that decides A_{TM}. To use R, S, provided with its input $\langle M, w \rangle$, modifies M so that the resulting TM recognizes a regular language if and only if M accepts w.

Undecidability of the Equivalence

- We design the modified machine M_2 to recognize the nonregular language $\{0^n1^n : n \geq 0\}$, if M does not accept w, and to recognize the regular language Σ^*, if M accepts w.

  We must specify how S can construct such an M_2 from M and w. M_2 works by automatically accepting all strings in $\{0^n1^n : n \geq 0\}$. In addition, if M accepts w, M_2 accepts all other strings.
- We let R be a TM that decides $\text{REGULAR}_{\text{TM}}$ and construct TM S to decide A_{TM}:

  S: On input $\langle M, w \rangle$, where M is a TM and w is a string,
  1. Construct the following TM M_2.
     M_2: On input x
     1. If x has the form 0^n1^n, accept.
     2. If x does not have this form, run M on input w and accept if M accepts w.
  2. Run R on input $\langle M_2 \rangle$.
  3. If R accepts, accept; if R rejects, reject.

Properties of Turing Recognizable Languages

- The problems of testing whether the language of a Turing machine is
  - a context-free language,
  - a decidable language,
  - a finite language

  can be shown to be undecidable with similar proofs.
- **Rice's Theorem** states that testing any property of the languages recognized by Turing machines is undecidable.
- So far, our strategy for proving languages undecidable involves a reduction from A_{TM}.
- Sometimes reducing from some other undecidable language is more convenient when we are showing undecidability.
- We use reduction from $\mathrm{HALT}_{\mathsf{TM}}$ and from E_{TM}, respectively, to prove Rice's Theorem and to show that testing the equivalence of two Turing machines is undecidable.

Rice's Theorem

- Let $\mathcal{S}$ be a proper nonempty subset of the set of all Turing-recognizable languages.
- Let $\mathcal{S}\text{MEMB}_{\text{TM}}$ be the problem of determining whether a given Turing machine recognizes a language in $\mathcal{S}$:

$$\mathcal{S}\text{MEMB}_{\text{TM}} = \{\langle M \rangle : M \text{ is a TM and } L(M) \in \mathcal{S}\}.$$

Theorem (Rice's Theorem)

Let $\mathcal{S}$ be a proper nonempty subset of the set of all Turing-recognizable languages. Then $\mathcal{S}\text{MEMB}_{\text{TM}}$ is undecidable.

- The proof is by reduction from HALT_{TM}. We assume that $\mathcal{S}\text{MEMB}_{\text{TM}}$ is decidable by a TM S and use this assumption to construct a TM H that decides HALT_{TM}. To use S, H, provided with its input $\langle M, w \rangle$, uses a TM M_A recognizing a fixed language $A \in \mathcal{S} \neq \emptyset$ to construct a TM M_w and uses $\langle M_w \rangle$ as input to S. The construction ensures that M halts on w iff $L(M_w) \in \mathcal{S}$.

Proof of Rice's Theorem

- We assume that $\emptyset \notin \mathcal{S}$; otherwise, we use $\overline{\mathcal{S}}$.
- Since $\mathcal{S} \neq \emptyset$, there exists $A \in \mathcal{S}$. Since $\mathcal{S}$ consists of Turing recognizable languages, there exists a TM M_A recognizing A.
- Given $\langle M, w \rangle$, a description of a TM M and a string w, we design a machine M_w, such that:
  - If M halts on w, M_w recognizes $A \in \mathcal{S}$;
  - If M does not halt on w, M_w recognizes $\emptyset \notin \mathcal{S}$.

  Thus testing M_w for membership in $\mathcal{S}$ solves the halting of M on w.
- We let S be a TM that decides $\mathcal{S}\text{MEMB}_{\text{TM}}$ and construct TM H to decide HALT_{TM}:

  H: On input $\langle M, w \rangle$, where M is a TM and w is a string,
  1. Construct the following TM M_w.

     M_w: On input y
     1. Run M on w.
     2. If M halts, simulate M_A on y.
     3. If it accepts, accept.
  2. Run S on input $\langle M_w \rangle$.
  3. If S accepts, accept; if S rejects, reject.

Correctness of the Reduction

- If S decides $\mathcal{S}\text{MEMB}_{\text{TM}}$, then H decides HALT_{TM}:
  - Suppose $\langle M, w\rangle \in \text{HALT}_{\text{TM}}$. Then M halts on w. Thus, M_w runs M_A on y and accepts iff $y \in A$. Therefore, in this case, $L(M_w) = A \in \mathcal{S}$. This means that S will accept $\langle M_w\rangle$ and, thus, H will also accept.
  - If $\langle M, w\rangle \notin \text{HALT}_{\text{TM}}$, then M does not halt on w. Thus, on all inputs y, M_w does not halt. In his case, $L(M_w) = \emptyset \notin \mathcal{S}$. Therefore, S will not accept $\langle M_w\rangle$ and, thus, H will reject.

  Thus H is a decider for HALT_{TM}, which is a contradiction. Therefore, $\mathcal{S}\text{MEMB}_{\text{TM}}$ is also undecidable.

Equivalence of Turing Machines

- Let $\mathrm{EQ}_{\mathrm{TM}} = \{\langle M_1, M_2\rangle : M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$.

Theorem

$\mathrm{EQ}_{\mathrm{TM}}$ is undecidable.

- We show that, if $\mathrm{EQ}_{\mathrm{TM}}$ were decidable, E_{TM} would be decidable, by giving a reduction from E_{TM} to $\mathrm{EQ}_{\mathrm{TM}}$. If one of the languages tested for equivalence happens to be $\emptyset$, the problem reduces to determining whether the language of the other machine is empty, i.e., the E_{TM} problem. Thus, in a sense, E_{TM} is a special case of $\mathrm{EQ}_{\mathrm{TM}}$ wherein one of the machines is fixed to recognize the empty language.
- We let TM R decide $\mathrm{EQ}_{\mathrm{TM}}$ and construct TM S to decide E_{TM}:
  S: On input $\langle M\rangle$, where M is a TM,
  1. Run R on input $\langle M, M_1\rangle$, where M_1 is a TM that rejects all inputs.
  2. If R accepts, accept; if R rejects, reject.

  If R decides $\mathrm{EQ}_{\mathrm{TM}}$, S decides E_{TM}. But E_{TM} is undecidable, whence $\mathrm{EQ}_{\mathrm{TM}}$ must also be undecidable.

Computation Histories

- The computation history method is an important technique for proving that A_{TM} is reducible to certain languages.
- This method is often useful when the problem to be shown undecidable involves testing for the existence of something.
- An example is the undecidability of Hilbert's tenth problem.
- The computation history for a Turing machine on an input is simply the sequence of configurations that the machine goes through as it processes the input, i.e., a complete record of its computation:

Definition (Computation History)

Let M be a Turing machine and w an input string. An **accepting computation history** for M on w is a sequence of configurations, $C_1, C_2, \ldots, C_\ell$, where C_1 is the start configuration of M on w, C_ℓ is an accepting configuration of M, and each C_i legally follows from C_{i-1} according to the rules of M. A **rejecting computation history** for M on w is defined similarly, except that C_ℓ is a rejecting configuration.

Linear Bounded Automata

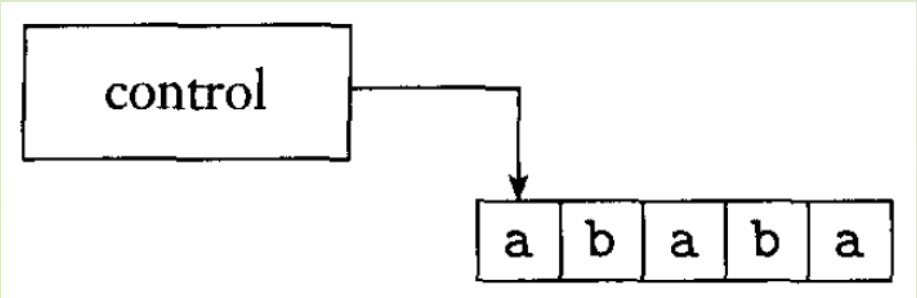
- Computation histories are finite sequences.
- If M does not halt on w, no accepting or rejecting computation history exists for M on w.
- Deterministic machines have at most one computation history on any given input.
- Nondeterministic machines may have many computation histories on a single input, corresponding to the various computation branches.
- For now, we continue to focus on deterministic machines.

Definition (Linear Bounded Automaton)

A **linear bounded automaton** is a restricted type of Turing machine wherein the tape head is not permitted to move off the portion of the tape containing the input.
If the machine tries to move its head off either end of the input, the head stays where it is, in the same way that the head will not move off the left-hand end of an ordinary Turing machine's tape.

Memory of Linear Bounded Automaton

- A linear bounded automaton is a Turing machine with a limited amount of memory:



- It can only solve problems requiring memory that can fit within the tape used for the input.
- Using a tape alphabet larger than the input alphabet allows the available memory to be increased up to a constant factor.
- We express this by saying that, for an input of length n, the amount of memory available is linear in n, whence the name of this model.

Acceptance Testing for LBAs

- Despite their memory constraint, linear bounded automata (LBAs) are quite powerful:
  - The deciders for $\mathrm{A}_{\mathrm{DFA}}$, $\mathrm{A}_{\mathrm{CFG}}$, $\mathrm{E}_{\mathrm{DFA}}$, and $\mathrm{E}_{\mathrm{CFG}}$ all are LBAs.
  - Every CFL can be decided by an LBA.
  - In fact, coming up with a decidable language that cannot be decided by an LBA takes some work.
- $\mathrm{A}_{\mathrm{LBA}}$ is the problem of determining whether an LBA accepts its input.
- Even though $\mathrm{A}_{\mathrm{LBA}}$ is the same as the undecidable problem A_{TM} where the Turing machine is restricted to be an LBA, we can show that $\mathrm{A}_{\mathrm{LBA}}$ is decidable.
- Let $\mathrm{A}_{\mathrm{LBA}} = \{\langle M, w\rangle : M \text{ is an LBA that accepts string } w\}$.
- An LBA can have only a limited number of configurations when a string of length n is the input.

Number of Configurations of an LBA

Lemma

Let M be an LBA with q states and g symbols in the tape alphabet. There are exactly qng^n distinct configurations of M for a tape of length n.

- Recall that a configuration of M is like a snapshot in the middle of its computation. A configuration consists of the state of the control, position of the head, and contents of the tape. Here, M has q states. The length of its tape is n, so the head can be in one of n positions, and g^n possible strings of tape symbols appear on the tape. The product of these three quantities is the total number of different configurations of M with a tape of length n.

Decidability of Acceptance Testing for LBAs

Theorem

A_{LBA} is decidable.

- In order to decide whether LBA M accepts input w, we simulate M on w. During the course of the simulation, if M halts and accepts or rejects, we accept or reject accordingly. The difficulty occurs if M loops on w. We need to be able to detect looping so that we can halt and reject.
  The idea for detecting when M is looping is that, as M computes on w, it goes from configuration to configuration. If M ever repeats a configuration it would go on to repeat this configuration over and over again and thus be in a loop. Because M is an LBA, the amount of tape available to it is limited. By the lemma, M can be in only a limited number of configurations on this amount of tape. Therefore only a limited amount of time is available to M before it will enter some configuration that it has previously entered.

Decidability of Acceptance Testing for LBAs: The Proof

- Detecting that M is looping is possible by simulating M for the number of steps given by the lemma. If M has not halted by then, it must be looping.
- The algorithm that decides $\mathrm{A}_{\mathrm{LBA}}$:

  L: On input $\langle M, w \rangle$, where M is an LBA and w is a string,

  1. Simulate M on w for qng^n steps or until it halts.
  2. If M halts, accept if it has accepted and reject if it has rejected. If it has not halted, reject.

  If M on w has not halted within qng^n steps, it must be repeating a configuration according to the preceding lemma and, therefore, looping. So the algorithm must reject in this instance.

Emptiness Testing for LBAs: Proof Outline

- For LBAs the acceptance problem is decidable, but for TMs it is not.
- Certain other problems involving LBAs remain undecidable.
- Let $\mathrm{E}_{\mathrm{LBA}} = \{\langle M \rangle : M \text{ is an LBA where } L(M) = \emptyset\}$.
- We give a reduction that uses the computation history method.

Theorem

$\mathrm{E}_{\mathrm{LBA}}$ is undecidable.

- This proof is by reduction from A_{TM}. We show that, if $\mathrm{E}_{\mathrm{LBA}}$ were decidable, A_{TM} would also be. Suppose that $\mathrm{E}_{\mathrm{LBA}}$ is decidable. For a TM M and an input w we can determine whether M accepts w by constructing a certain LBA B and then testing whether $L(B)$ is empty. The language that B recognizes comprises all accepting computation histories for M on w. If M accepts w, this language contains one string and so is nonempty. If M does not accept w, this language is empty. If we can determine whether B's language is empty, clearly we can determine whether M accepts w.

Emptiness Testing for LBAs: Computation History

- We need to show more than the mere existence of B: We have to show how a Turing machine can obtain a description of B, given descriptions of M and w. We construct B to accept its input x, if x is an accepting computation history for M on w. Recall that an accepting computation history is the sequence of configurations, $C_1, C_2, \ldots, C_\ell$ that M goes through as it accepts some string w. For the purposes of this proof we assume that the accepting computation history is presented as a single string, with the configurations separated from each other by the # symbol:

$$\#\underbrace{\qquad}_{C_1}\#\underbrace{\qquad}_{C_2}\#\underbrace{\qquad}_{C_3}\#\cdots\#\underbrace{\qquad}_{C_\ell}\#$$

Operation of the LBA B

- The LBA B works as follows: When it receives an input x, B is supposed to accept if x is an accepting computation for M on w.
  - First, B breaks up x into the strings $C_1, C_2, \ldots, C_\ell$.
  - Then B determines whether the C_i satisfy the three conditions of an accepting computation history:
    1. C_1 is the start configuration for M on w.
    2. Each C_{i+1} legally follows from C_i.
    3. C_ℓ is an accepting configuration for M.
  - The start configuration C_1 for M on w is the string $q_0 w_1 w_2 \cdots w_n$, where q_0 is the start state for M on w. Here, B has this string directly built in, so it is able to check the first condition.
  - An accepting configuration is one that contains the q_{accept} state, so B can check the third condition by scanning C_ℓ for q_{accept}.
  - The second condition is the hardest to check. For each pair of adjacent configurations, B checks whether C_{i+1} legally follows from C_i. This step involves verifying that C_i and C_{i+1} are identical except for the positions under and adjacent to the head in C_i. These positions must be updated according to the transition function of M.

Finishing the Preparation for the Proof

- B verifies that the updating was done properly by zig-zagging between corresponding positions of C_i and C_{i+1}. To keep track of the current positions while zig-zagging, B marks the current position with dots on the tape.
- If conditions 1, 2, and 3 are satisfied, B accepts its input.

- Note that the LBA B is not constructed for the purposes of actually running it on some input.
  - We construct B only for the purpose of feeding a description of B into the decider for $\mathrm{E}_{\mathrm{LBA}}$ that we have assumed to exist.
  - Once this decider returns its answer we can invert it to obtain the answer to whether M accepts w.

  This decides A_{TM}, a contradiction.

The Proof

- Now we are ready to state the reduction of A_{TM} to $\mathrm{E}_{\mathrm{LBA}}$. Suppose that TM R decides $\mathrm{E}_{\mathrm{LBA}}$. We construct TM S that decides A_{TM} as follows.

  S: On input $\langle M, w \rangle$, where M is a TM and w is a string,

  1. Construct LBA B from M and w as described in the proof idea.
  2. Run R on input $\langle B \rangle$.
  3. If R rejects, accept; if R accepts, reject.

  If R accepts $\langle B \rangle$, then $L(B) = \emptyset$. Thus M has no accepting computation history on w and M does not accept w. Consequently S rejects $\langle M, w \rangle$. Similarly, if R rejects $\langle B \rangle$, the language of B is nonempty. The only string that B can accept is an accepting computation history for M on w. Thus M must accept w. Consequently S accepts $\langle M, w \rangle$.

Universal Inclusion Testing for CFGs

- We can also use the technique of reduction via computation histories to establish the undecidability of certain problems related to context-free grammars and pushdown automata.
- We have presented an algorithm to decide whether a context-free grammar generates any strings, i.e., whether $L(G) = \emptyset$.
- Now we show that the problem of determining whether a context-free grammar generates all possible strings is undecidable.
- This is also the main step in showing that the equivalence problem for context-free grammars is undecidable.
- Let $\text{ALL}_{\text{CFG}} = \{\langle G \rangle : G \text{ is a CFG and } L(G) = \Sigma^*\}$.

Theorem

ALL_{CFG} is undecidable.

- This proof is by contradiction. We assume that ALL_{CFG} is decidable and use this assumption to show that A_{TM} is decidable. The proof consists of a reduction from A_{TM} via computation histories.

Description of the Reduction

- We develop a decision procedure for A_{TM} using one for $\mathrm{ALL}_{\mathrm{CFG}}$.
- For a TM M and an input w we construct a CFG G that generates all strings if and only if M does not accept w.
- So, if M does accept w, G does not generate some particular string. This string is the accepting computation history for M on w.
- Thus, G is designed to generate all strings that are not accepting computation histories for M on w: An accepting computation history for M on w appears as $\#C_1\#C_2\#\cdots\#C_\ell\#$, where C_i is the configuration of M on the ith step of the computation on w. G generates all strings that:
  1. do not start with C_1, or
  2. do not end with an accepting configuration, or
  3. where some C_i does not properly yield C_{i+1} under the rules of M.

  If M does not accept w, no accepting computation history exists, so all strings fail in one way or another. Therefore G would generate all strings.

Construction of Grammar G

- Instead of constructing the grammar G, we construct a PDA D. D can then be converted to a CFG. Designing a PDA is easier than designing a CFG.
- D will start by nondeterministically branching to guess which of the preceding three conditions to check.
  - One branch checks on whether the beginning of the input string is C_1 and accepts if it is not.
  - Another branch checks on whether the input string ends with a configuration containing the accept state and accepts if it does not.
  - The third branch is supposed to accept if some C_i does not properly yield C_{i+1}. It works by scanning the input until it nondeterministically decides that it has come to C_i. Next, it pushes C_i onto the stack until it comes to the end as marked by the # symbol. Then D pops the stack to compare with C_{i+1}. They are supposed to match except around the head position where the difference is dictated by the transition function of M. Finally, D accepts if it is a mismatch or an improper update.

A Twist in the Construction of the PDA

- There is a problem with this construction: When D pops C_i off the stack, it is in reverse order and not suitable for comparison with C_{i+1}.
- To deal with this, we write the accepting computation history differently: Every other configuration appears in reverse order.
  - The odd positions remain written in the forward order;
  - the even positions are written backward.

$$\#\underbrace{\rightarrow}_{C_1}\#\underbrace{\leftarrow}_{C_2^{\mathcal{R}}}\#\underbrace{\rightarrow}_{C_3}\#\underbrace{\leftarrow}_{C_4^{\mathcal{R}}}\#\cdots\#\underbrace{\quad}_{C_\ell}\#$$

- In this form, the PDA is able to push a configuration so that when it is popped, the order is suitable for comparison with the next one.
- We design D to accept any string that is not an accepting computation history in the modified form.

Subsection 2

Mapping Reducibility

Formalizing Reducibility

- We have shown how to use the reducibility technique to prove that various problems are undecidable.
- We now formalize the notion of reducibility in order to be able to use it in more refined ways, e.g., for proving that certain languages are not Turing-recognizable.
- We choose to define a simple type of reducibility, called **mapping reducibility** or **many-one reducibility**.
  - Being able to reduce problem A to problem B by using a mapping reducibility means that a computable function exists that converts instances of problem A to instances of problem B.
  - If we have such a conversion function, called a **reduction**, we can solve A with a solver for B: Given any instance of A,
    - First, use the reduction to convert it to an instance of B;
    - Then apply the solver for B.

Computable Functions

- A Turing machine computes a function by starting with the input to the function on the tape and halting with the output of the function on the tape.

Definition (Computable Function)

A function $f : \Sigma^* \to \Sigma^*$ is a **computable function** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

- Example: All usual arithmetic operations on integers are computable functions. For example, we can make a machine that takes input $\langle m, n \rangle$ and returns $m + n$, the sum of m and n.

Transformations of Machine Descriptions

- Computable functions may be transformations of machine descriptions.
- Example: For example, one computable function f takes input w.
  - If $w = \langle M \rangle$ is an encoding of a Turing machine M, then f returns the description of a Turing machine $\langle M' \rangle$:

    > The machine M' is a machine that recognizes the same language as M, but never attempts to move its head off the left-hand end of its tape.

    The function f accomplishes this task by adding several states to the description of M.
  - If w is not a legal encoding of a Turing machine, then f returns ε

Mapping Reducibility

- Recall the representation of computational problems by languages.

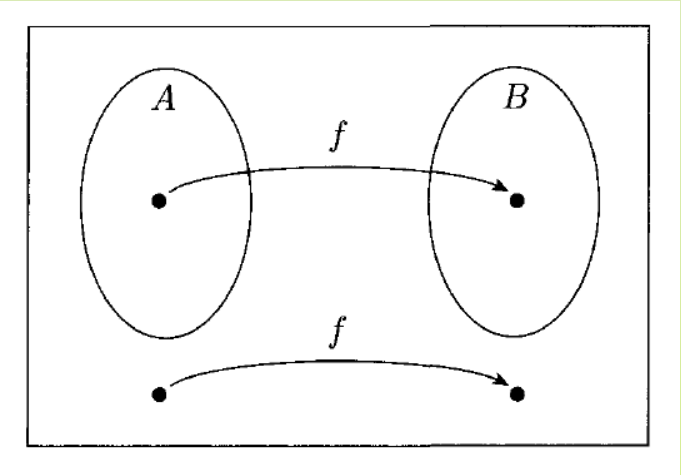
Definition (Mapping Reducibility)

Language A is **mapping reducible** to language B, written $A \leq_m B$, if there is a computable function $f : \Sigma^* \to \Sigma^*$, such that, for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the **reduction** of A to B.

- The following figure illustrates mapping reducibility:



Reducibility and Decidability

- A mapping reduction of A to B provides a way to convert questions about membership testing in A to membership testing in B.

  To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$.

Theorem

If $A \leq_m B$ and B is decidable, then A is decidable.

- We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A:

  N: On input w

  1. Compute $f(w)$.
  2. Run M on input $f(w)$ and output whatever M outputs.

  Clearly, $w \in A$ iff $f(w) \in B$, because f is a reduction from A to B. Thus, M accepts $f(w)$ iff $w \in A$. So N works as desired.

Corollary

If $A \leq_m B$ and A is undecidable, then B is undecidable.

Reducing Acceptability Testing to Halting

- Some earlier proofs that used the reducibility method provide examples of mapping reducibilities.
- Example: We have used a reduction from A_{TM} to prove that $\mathrm{HALT}_{\mathsf{TM}}$ is undecidable. This reduction showed how a decider for $\mathrm{HALT}_{\mathsf{TM}}$ could be used to give a decider for A_{TM}.
- To demonstrate a mapping reducibility from A_{TM} to $\mathrm{HALT}_{\mathsf{TM}}$, we must present a computable function f that takes input of the form $\langle M, w\rangle$ and returns output of the form $\langle M', w'\rangle$, where

$$\langle M, w\rangle \in \mathrm{A}_{\mathsf{TM}} \quad \text{if and only if} \quad \langle M', w'\rangle \in \mathrm{HALT}_{\mathsf{TM}}.$$

The following machine F computes a reduction f.

Description of the Machine F

F: On input $\langle M, w \rangle$

1. Construct the following machine M':
   M': On input x
   1. Run M on x.
   2. If M accepts, accept.
   3. If M rejects, enter a loop.
2. Output $\langle M', w \rangle$.

A minor issue arises concerning improperly formed input strings.

> If TM F determines that its input is not of the correct form as specified in the input line and, hence, that the input is not in A_{TM}, the TM outputs a string not in $\mathrm{HALT}_{\mathrm{TM}}$. Any string not in $\mathrm{HALT}_{\mathrm{TM}}$ will do.

- In general, when we describe a Turing machine that computes a reduction from A to B, improperly formed inputs are assumed to map to strings outside of B.

Two More Examples

- Example: A mapping reduction f from E_{TM} to $\mathrm{EQ}_{\mathrm{TM}}$ maps the input $\langle M \rangle$ to the output $\langle M, M_1 \rangle$, where M_1 is the machine that rejects all inputs.
- Example: The proof showing that E_{TM} is undecidable illustrates the difference between the formal notion of mapping reducibility and the informal notion of reducibility.
  - The proof shows that E_{TM} is undecidable by reducing A_{TM} to it. From the original reduction a function f can be constructed that takes input $\langle M, w \rangle$ and produces output $\langle M_1 \rangle$, where M_1 is the Turing machine described in the proof.
  - M accepts w iff $L(M_1)$ is not empty. So f is a mapping reduction from A_{TM} to $\overline{\mathrm{E}_{\mathrm{TM}}}$.
    - The reduction still shows that E_{TM} is undecidable because decidability is not affected by complementation. However, It does not give a mapping reduction from A_{TM} to E_{TM}.
  - In fact, no reduction from A_{TM} to E_{TM} exists.

Reducibility and Turing Recognizability

- The sensitivity of mapping reducibility to complementation is important in the use of reducibility to prove non-recognizability.

Theorem

If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable.

- The proof is the same as that of the theorem concerning decision, except that M and N are recognizers instead of deciders.

Corollary

If $A \leq_m B$ and A is not Turing-recognizable, then B is not either.

- In a typical application of this corollary, we let A be $\overline{\mathrm{A_{TM}}}$, the complement of $\mathrm{A_{TM}}$. We know that $\overline{\mathrm{A_{TM}}}$ is not Turing-recognizable. The definition of mapping reducibility implies that $A \leq_m B$ means the same as $\overline{A} \leq_m \overline{B}$. To prove that B is not recognizable we may show that $\mathrm{A_{TM}} \leq_m \overline{B}$.

$\mathrm{EQ}_{\mathrm{TM}}$ is not Turing-Recognizable

- We can also use mapping reducibility to show that certain problems are neither Turing-recognizable nor co-Turing-recognizable.

Theorem

$\mathrm{EQ}_{\mathrm{TM}}$ is neither Turing-recognizable nor co-Turing-recognizable.

- First we show that $\mathrm{EQ}_{\mathrm{TM}}$ is not Turing-recognizable. We reduce A_{TM} to $\mathrm{EQ}_{\mathrm{TM}}$. The following TM F computes the reducing function f:
  F: On input $\langle M, w \rangle$, where M is a TM and w a string,
  1. Construct the following two machines M_1 and M_2.
     M_1: On any input
     1. Reject.
     M_2: On any input
     1. Run M on w. If it accepts, accept.
  2. Output $\langle M_1, M_2 \rangle$.

  M_1 accepts nothing. If M accepts w, M_2 accepts everything, and the two machines are not equivalent. If M does not accept w, M_2 accepts nothing, and they are equivalent. Thus, f reduces A_{TM} to $\mathrm{EQ}_{\mathrm{TM}}$.

$\mathrm{EQ}_{\mathrm{TM}}$ is not co-Turing-Recognizable

- To show that $\overline{\mathrm{EQ}_{\mathrm{TM}}}$ is not Turing-recognizable we give a reduction from A_{TM} to $\mathrm{EQ}_{\mathrm{TM}}$. The following TM G computes the reducing function g:

  G: On input $\langle M, w \rangle$, where M is a TM and w a string,

  1. Construct the following two machines M_1 and M_2.
     M_1: On any input
     1. Accept.

     M_2: On any input
     1. Run M on w.
     2. If it accepts, accept.
  2. Output $\langle M_1, M_2 \rangle$.

  The only difference between f and g is in machine M_1. In f, machine M_1 always rejects, whereas in g it always accepts. In both f and g, M accepts w iff M_2 always accepts. In g, M accepts w iff M_1 and M_2 are equivalent. So g is a reduction from A_{TM} to $\mathrm{EQ}_{\mathrm{TM}}$.

Subsection 3

Turing Reducibility

Turing versus Mapping Reducibility

- We introduced reducibility as a way of using a solution to one problem to solve other problems.

  If A is reducible to B, and we find a solution to B, we can obtain a solution to A.

- We then described mapping reducibility, a specific form of reducibility.
- It turns out that mapping reducibility does not capture our intuitive concept of reducibility in the most general way.
- Example: Consider the two languages A_{TM} and $\overline{\mathrm{A}_{\mathrm{TM}}}$.
  - Intuitively, they are reducible to one another because a solution to either could be used to solve the other by simply reversing the answer.
  - However, we know that $\overline{\mathrm{A}_{\mathrm{TM}}}$ is not mapping reducible to A_{TM} because A_{TM} is Turing-recognizable but $\overline{\mathrm{A}_{\mathrm{TM}}}$ is not.
- A very general form of reducibility which captures our intuitive concept of reducibility more closely is **Turing reducibility**.

Oracle Turing Machines

Definition (Oracle Turing Machine)

- An **oracle** for a language B is an external device that is capable of reporting whether any string w is a member of B.
- An **oracle Turing machine** is a modified Turing machine that has the additional capability of querying an oracle.
- We write M^B to describe an oracle Turing machine that has an oracle for language B.

- The way an oracle determines its responses is of no concern.
- Example: Consider an oracle for A_{TM}. An oracle Turing machine with an oracle for A_{TM} can decide more languages than an ordinary Turing machine can:

  Such a machine can (obviously) decide A_{TM} itself, by querying the oracle about the input.

Another Example of an Oracle Turing Machine

- An oracle Turing machine with an oracle for $\mathrm{A_{TM}}$ can also decide $\mathrm{E_{TM}}$, the emptiness testing problem for TMs, with the following procedure called $T^{\mathrm{A_{TM}}}$.

  $T^{A_{\mathrm{TM}}}$: On input $\langle M \rangle$, where M is a TM,

  1. Construct the following TM N.
     N: On any input
     1. Run M in parallel on all strings in Σ^*.
     2. If M accepts any of these strings, accept.
  2. Query the oracle to determine whether $\langle N, 0 \rangle \in A_{\mathrm{TM}}$
  3. If the oracle answers NO, accept; if YES, reject.

- If M's language is not empty, N will accept every input and, in particular, input 0. Hence the oracle will answer YES, and $T^{\mathrm{A_{TM}}}$ will reject. Conversely, if M's language is empty, $T^{\mathrm{A_{TM}}}$ will accept. Thus $T^{\mathrm{A_{TM}}}$ decides $\mathrm{E_{TM}}$.
- We say that $\mathrm{E_{TM}}$ is **decidable relative to** $\mathrm{A_{TM}}$.

Turing Reducibility

Definition (Turing Reducibility)

Language A is **Turing reducible** to language B, written $A \leq_T B$, if A is decidable relative to B.

- Example: We saw that E_{TM} is Turing reducible to A_{TM}.
- Turing reducibility satisfies our intuitive concept of reducibility:

Theorem

If $A \leq_T B$ and B is decidable, then A is decidable.

- If B is decidable, then we may replace the oracle for B by an actual procedure that decides B. Thus we may replace the oracle Turing machine that decides A by an ordinary Turing machine that decides A.
- Turing reducibility is a generalization of mapping reducibility:
  If $A \leq_m B$, then $A \leq_T B$, because the mapping reduction may be used to give an oracle Turing machine that decides A relative to B.
- Despite deciding more languages than ordinary TMs, oracle Turing machines with an oracle for A_{TM} cannot decide all languages.

Subsection 4

The Definition of Information

Algorithm and Information in Computer Science

- The concepts of algorithm and information are fundamental in computer science.
- The Church-Turing Thesis gives a universally applicable, model-independent, definition of algorithm.
- No equally comprehensive definition of information is known.
- One way of defining information is by using computability theory.
- Example: Consider the information content of the following two binary sequences:

$$A = 0101010101010101010101010101010101010101$$
$$B = 1110010110100011101010000111010011010111$$

  - Intuitively, sequence A contains little information because it is merely a repetition of the pattern 01 twenty times.
  - In contrast, sequence B appears to contain more information.

Information Content of a Sequence

- We define the quantity of information contained in an object to be the size of that object's smallest representation or description.

  By a **description** of an object we mean a precise and unambiguous characterization of the object so that we may recreate it from the description alone.

- Consider again

$$A = 01010101010101010101010101010101010101$$
$$B = 11100101101000111010100001110100110101111$$

  - Sequence A contains little information because it has a small description;
  - Sequence B contains more information because it seems to have no concise description.

Shortest Descriptions and Restriction to Binary Strings

- Clearly, we may always describe an object, such as a string, by placing a copy of the object directly into the description.
- This type of description is never shorter than the object itself and does not tell us anything about its information quantity.
- A description that is significantly shorter than the object implies that the information contained within it can be compressed into a small volume implying that the amount of information cannot be very large.
- That is the reason why the size of the shortest description determines the amount of information.
- To formalize this intuitive idea:
  - First, we restrict our attention to objects that are binary strings. Other objects can be represented as binary strings, so this restriction does not limit the scope of the theory.
  - Second, we consider only descriptions that are themselves binary strings. By imposing this requirement, we may easily compare the length of the object with the length of its description.

Discussion of Possible Encodings

- One way to use algorithms to describe strings is to:
  - Construct a TM that outputs the string when it starts on a blank tape;
  - Then represent the Turing machine itself as a string.

  The string representing the TM is a description of the original string.
- However, a Turing machine cannot represent a table of information concisely with its transition function:
  - Representing a string of n bits might use n states and n rows in the transition function table.
  - That would result in a description that is too long for our purpose.
- Instead, we describe a binary string x with a Turing machine M and a binary input w to M. The length of the description is the combined length of representing M and w.
- We write this description with our usual notation for encoding several objects into a single binary string $\langle M, w \rangle$. But, here, we must pay additional attention to the encoding operation $\langle \bullet, \bullet \rangle$ because we need to produce a concise result.

Adoption of a Suitable Encoding

- We define the string $\langle M, w \rangle$ to be $\langle M \rangle w$, where we concatenate the binary string w onto the end of the binary encoding of M.
- The encoding $\langle M \rangle$ of M may be done in any standard way, except for the subtlety that we describe next:
  - Concatenating w onto the end of $\langle M \rangle$ to yield a description of x might create trouble if the point at which $\langle M \rangle$ ends and w begins is not discernible from the description itself.
  - Then, several ways of partitioning the description $\langle M \rangle w$ into a syntactically correct TM and an input may occur, in which case the description would be ambiguous and hence invalid.
- To avoid this problem, we ensure that we can locate the separation between $\langle M \rangle$ and w in $\langle M \rangle w$.

  One way to do so is to write each bit of $\langle M \rangle$ twice, writing 0 as 00 and 1 as 11, and then follow it with 01 to mark the separation point:

$$\langle M, w \rangle = \underbrace{11001111001100 \cdots 11000 \overbrace{01}^{\text{delimiter}}}_{\langle M \rangle} \underbrace{01101011 \cdots 010}_{w}$$

Descriptive or Kolmogorov Complexity

Definition (Descriptive or Kolmogorov Complexity)

Let x be a binary string. The **minimal description** of x, written $d(x)$, is the shortest string $\langle M, w \rangle$, where TM M on input w halts with x on its tape. If several such strings exist, select the lexicographically first among them. The **descriptive complexity** or **Kolmogorov complexity** or **Kolmogorov-Chaitin complexity** of x, written $\mathrm{K}(x)$, is $\mathrm{K}(x) = |d(x)|$.

- In other words, $\mathrm{K}(x)$ is the length of the minimal description of x.
- The definition of $\mathrm{K}(x)$ is intended to capture our intuition for the amount of information in the string x.

An Upper Bound on the Kolmogorov Complexity

Theorem

There exists a constant c, such that, for all x, $\mathrm{K}(x) \leq |x| + c$.

- The theorem says that the descriptive complexity of a string is at most a fixed constant more than its length.
- The constant is a universal one, not dependent on the string.
- To prove an upper bound on $\mathrm{K}(x)$, we need only demonstrate some description of x that is no longer than the stated bound. Then the minimal description of x may be shorter than the demonstrated description, but not longer. We describe the string x by considering the Turing machine M that halts as soon as it is started. This machine computes the identity function, i.e., its output is the same as its input. A description of x is simply $\langle M \rangle x$. Letting c be the length of $\langle M \rangle$ completes the proof.

Descriptive Complexity of xx

- The preceding result verifies our intuition that the amount of information contained in a string cannot be much more than its length.
- Similarly, intuition says that the information contained by the string xx is not significantly more than the information contained by x:

Theorem

There exists a constant c, such that, for all x, $\mathrm{K}(xx) \leq \mathrm{K}(x) + c$.

- Consider the TM M, which expects an input of the form $\langle N, w \rangle$, where N is a Turing machine and w is an input for it:
  M: On input $\langle N, w \rangle$, where N is a TM and w is a string,
  1. Run N on w until it halts and produces an output string s.
  2. Output the string ss.

  A description of xx is $\langle M \rangle d(x)$. Recall that $d(x)$ is a minimal description of x. The length of this description is $|\langle M \rangle| + |d(x)|$, which is $c + \mathrm{K}(x)$, where c is the length of $\langle M \rangle$.

Descriptive Complexity of xy

- We might be led to believe that the complexity of the concatenation is at most the sum of the individual complexities (plus a fixed constant), but the cost of combining two descriptions leads to a greater bound:

Theorem

There exists a constant c, such that, for all strings x and y,

$$\mathrm{K}(xy) \leq 2\mathrm{K}(x) + \mathrm{K}(y) + c.$$

- We construct a TM M that breaks its input w into two separate descriptions. The bits of the first description $d(x)$ are all doubled and terminated with string 01 before the second description $d(y)$ appears. Once both descriptions have been obtained, they are run to obtain the strings x and y and the output xy is produced. The length of this description of xy is twice the complexity of x plus the complexity of y plus a fixed constant for describing M. This sum is $2\mathrm{K}(x) + \mathrm{K}(y) + c$.

Improvements on the Upper Bound of $\mathrm{K}(xy)$

- We may improve this theorem somewhat by using a more efficient method of indicating the separation between the two descriptions to avoid doubling the bits of $d(x)$.
- Instead we prepend the length of $d(x)$ as a binary integer that has been doubled to differentiate it from $d(x)$.
- The description still contains enough information to decode it into the two descriptions of x and y, and it now has length at most

$$2\log_2 \mathrm{K}(x) + \mathrm{K}(x) + \mathrm{K}(y) + c.$$

- Further small improvements are possible, but it can be shown that we cannot reach the bound

$$\mathrm{K}(x) + \mathrm{K}(y) + c.$$

Alternative Algorithmic Descriptions

- The adopted definition of $\mathrm{K}(x)$ has an optimality property among all possible ways of defining descriptive complexity with algorithms.
- Suppose that we consider a general **description language** to be any computable function $p : \Sigma^* \to \Sigma^*$. Define the **minimal description** of x with respect to p, written $d_p(x)$, to be the lexicographically shortest string s where $p(s) = x$. Define $\mathrm{K}_p(x) = |d_p(x)|$.
- Example: Consider a programming language, such as LISP (encoded into binary), as the description language.
  - $d_{\mathrm{LISP}}(x)$ would be the minimal LISP program that outputs x.
  - $\mathrm{K}_{\mathrm{LISP}}(x)$ would be the length of the minimal program.
- The following theorem shows that any description language of this type is not significantly more concise than the language of Turing machines and inputs that determines the Kolmogorov complexity.

Comparison With Descriptive Complexity

Theorem

For any description language p, a fixed constant c exists that depends only on p, where, for all x, $\mathrm{K}(x) \leq \mathrm{K}_p(x) + c$.

- We illustrate the idea of this proof by using the LISP example. Suppose that x has a short description w in LISP. Let M be a TM that can interpret LISP and use the LISP program for x as M's input w. Then $\langle M, w \rangle$ is a description of x that is only a fixed amount larger than the LISP description of x. The extra length is for the LISP interpreter M.
- Take any description language p and consider the following Turing machine M:
  M: On input w
  1. Output $p(w)$.

  Then $\langle M \rangle d_p(x)$ is a description of x whose length is at most a fixed constant greater than $\mathrm{K}_p(x)$. The constant is the length of $\langle M \rangle$.

Incompressibility

- We proved that a string's minimal description is never much longer than the string itself.
- For some strings, the minimal description may be much shorter if the information in the string appears sparsely or redundantly.
- Do some strings lack short descriptions? I.e., is the minimal description of some strings actually as long as the string itself?

Definition (Incompressible Strings)

Let x be a string.

- Say that x is c-**compressible** if $\mathrm{K}(x) \leq |x| - c$.
- If x is not c-compressible, we say that x is **incompressible by** c.
- If x is incompressible by 1, we say that x is **incompressible**.

- In other words, if x has a description that is c bits shorter than its length, x is c-compressible. If not, x is incompressible by c.
- If x does not have a description shorter than itself, then x is incompressible.

Existence of Incompressible Strings

Theorem

Incompressible strings of every length exist.

- The number of binary strings of length n is 2^n. Each description is a binary string. Thus, the number of descriptions of length less than n is at most
$$\sum_{i=0}^{n-1} 2^i = 1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1.$$
The number of short descriptions is less than the number of strings of length n. Therefore at least one string of length n is incompressible.

Corollary

At least $2^n - 2^{n-c+1} + 1$ strings of length n are incompressible by c.

- At most $2^{n-c+1} - 1$ strings of length n are c-compressible, because at most that many descriptions of length at most $n - c$ exist. The remaining $2^n - (2^{n-c+1} - 1)$ are incompressible by c.

Properties "Holding For Almost All Strings"

- Incompressible strings have many properties that we would expect to find in randomly chosen strings.
  - Any incompressible string of length n has roughly an equal number of 0s and 1s.
  - The length of its longest run of 0s is approximately $\log_2 n$, as we would expect to find in a random string of that length.
- We only prove a theorem that forms the basis for these statements:

  It shows that any computable property that holds for "almost all" strings also holds for all sufficiently long incompressible strings.
- A **property of strings** is simply a function f that maps strings to $\{\text{TRUE}, \text{FALSE}\}$. We say that a property **holds for almost all strings** if the fraction of strings of length n on which it is FALSE approaches 0 as n grows large.
- A randomly chosen long string is likely to satisfy a computable property that holds for almost all strings, whence random strings and incompressible strings share such properties.

Compressibility and Properties of Almost All Strings

Theorem

Let f be a computable property that holds for almost all strings. Then, for any $b > 0$, the property f is FALSE on only finitely many strings that are incompressible by b.

- Let M be the following algorithm:
  M: On input i, a binary integer,
  1. Find the i-th string s where $f(s) =$ FALSE, considering the strings ordered lexicographically.
  2. Output string s.

  We can use M to obtain short descriptions of strings that fail to have property f as follows: For any such string x, let i_x be the position or **index** of x on a list of all strings that fail to have property f, ordered by length and lexicographically within each length. Then $\langle M, i_x \rangle$ is a description of x. The length of this description is $|i_x| + c$, where c is the length of $\langle M \rangle$. Because few strings fail to have property f, the index of x is small and its description is correspondingly short.

Proving Compressibility by any $b > 0$

- Fix any number $b > 0$. Select n such that at most a $\frac{1}{2^{b+c+1}}$ fraction of strings of length n or less fail to have property f. All sufficiently large n satisfy this condition because f holds for almost all strings. Let x be a string of length n that fails to have property f. We have a total of $2^{n+1} - 1$ strings of length n or less, so

$$i_x \leq \frac{2^{n+1} - 1}{2^{b+c+1}} \leq 2^{n-b-c}.$$

Therefore, $|i_x| \leq n - b - c$, so the length of $\langle M, i_x \rangle$ is at most $(n - b - c) + c = n - b$, which implies that $\mathrm{K}(x) \leq n - b$. Thus, every sufficiently long x that fails to have property f is compressible by b. Hence, only finitely many strings that fail to have property f are incompressible by b, which proves the theorem.

Uncomputability of Descriptive Complexity

- At this point exhibiting some examples of incompressible strings would be appropriate.
- However:
  - The K measure of complexity is not computable.
  - No algorithm can decide in general whether strings are incompressible.
  - No infinite subset of them is Turing-recognizable.
- So there is no way to obtain long incompressible strings.
- There is no way either to determine whether a string is incompressible even if one was produced.
- We finish by describing certain strings that are nearly incompressible, without providing a way to exhibit them explicitly.

Near Incompressibility of Minimal Description

Theorem

There exists a constant b, such that, for every string x, the minimal description $d(x)$ of x is incompressible by b.

- Consider the following TM M:
  M: On input $\langle R, y\rangle$, where R is a TM and y is a string,
  1. Run R on y and reject if its output is not of the form $\langle S, z\rangle$.
  2. Run S on z and halt with its output on the tape.

  Let b be $|\langle M\rangle| + 1$.

  Claim: b satisfies the theorem.

  Suppose, to the contrary, $d(x)$ is b-compressible for some string x. Then $|d(d(x))| \leq |d(x)| - b$. But then $\langle M\rangle d(d(x))$ is a description of x whose length is at most

$$|\langle M\rangle| + |d(d(x))| \leq (b-1) + (|d(x)| - b) = |d(x)| - 1.$$

  This description is shorter than $d(x)$, contradicting minimality.