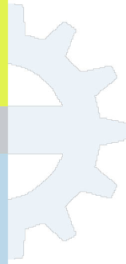
Deterministic Primality Test

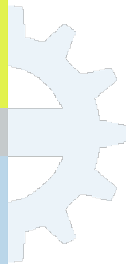
- In 2002 Agarwal, Kayal & Saxena from India managed to develop the first ”efficient” ($(\log n)^{12}$) deterministic algorithm for recognizing prime numbers
- This problem has been studied for centuries (Sieve of Eratosthenes, from approximately 240 before common era)
- The algorithm of AKS borrows techniques from randomized primality test algorithms, which are still today the most useful (efficient) solution techniques
- Because recognizing composite numbers is a complementary problem to recognizing prime numbers, also it can be solved in polynomial time deterministically
- However, one does not know how to factorize a composite number (it is, though, believed to be possible)

Theorem 3.16 *Every nondeterministic Turing machine has an equivalent deterministic Turing machine.*

- A nondeterministic TM can be simulated with a 3-tape Turing machine, which goes systematically through the possible computations of the nondeterministic TM
- Tape 1 maintains the input
- Tape 2 simulates the tape of the nondeterministic TM
- Tape 3 keeps track of the situation with the possible computations



- The tree formed of the possible computation paths of the nondeterministic TM has to be examined using breadth-first algorithm, not in depth-first order
- Let b be the largest number of possible successor states for any state in the Turing machine
- Each node in the tree can be indexed with a string over the alphabet $\{ 1, 2, ..., b \}$
- For example, 231 is the node that is the first child of the third child of the second child of the root of the tree
- All strings do not correspond to legal computations (nodes of the tree) — those get rejected
- Going through the strings in the lexicographic order corresponds to examining all computation paths, and the search order in the tree is breadth-first search



Working idea:

- Copy the input from tape 1 to tape 2
- Tape 3 tells us which is (in the lexicographic order) the next computation alternative for the nondeterministic TM. We simulate that computation targeting the updates on the tape of the original TM into tape 2 of the simulating TM
- Observe that there is a finite number of transition possibilities (successor states)
- Systematic examination of the possible computations of the nondeterministic TM ends into the accepting final state only if the original TM has an accepting computation path
- If no accepting computation exists, the simulating TM never halts

UNRESTRICTED GRAMMARS

- Context-free grammar allows to substitute only variables with strings
- In an unrestricted grammar (or a rewriting system) one may substitute any non-empty string (containing variables and terminals) with another one (also with the empty string ε)

An unrestricted grammar is a 4-tuple $G = (V, \Sigma, R, S)$, where

- V is the set of variables,
- Σ is the set of terminals,
- $\Gamma = V \cup \Sigma$ is the alphabet of G,
- $R \subseteq \Gamma^+ \times \Gamma^*$ is the set of rules, and
- $S \in V$ is the start variable

$(w, w') \in R$ is denoted as $w \rightarrow w'$

- Let
  - $G = (V, \Sigma, R, S)$,
  - strings $v \in \Gamma^+$ and $u, w, x \in \Gamma^*$ as well as
  - $v \rightarrow x$ a rule in R
- uvw *yields* string uxw in grammar G,

$$uvw \Rightarrow_G uxw$$

- String v *derives* string w in grammar G,

$$v \Rrightarrow_G w,$$

  if there exists a sequence $v_1, v_2, \ldots, v_k \in \Gamma^*$ ($k \geq 0$) s.t.

$$v \Rightarrow_G v_1 \Rightarrow_G v_2 \Rightarrow_G \ldots \Rightarrow_G v_k \Rightarrow_G w$$

- $k = 0$: $v \Rrightarrow_G v$ for any $v \in \Gamma^*$

- $u \in \Gamma^*$ is a *sentential form* of G if $S \Rightarrow_G u$
- A sentential form consisting only of terminal symbols $w \in \Sigma^*$ is a sentence of G
- The *language of the grammar* G consists of sentences

$$L(G) = \{ w \in \Sigma^* \mid S \Rightarrow_G w \}$$

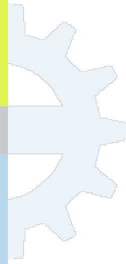
The language $\{ a^k b^k c^k \mid k \geq 0 \}$ is not a context-free one; it can be generated with an unrestricted grammar, which

1. Generates the variable sequence $L(ABC)^k$ (or ε)
2. Orders the variables lexicographically ⇨ $LA^k B^k C^k$
3. Replaces the variables with terminals

$S \rightarrow LT \mid \varepsilon$
$T \rightarrow ABCT \mid ABC$

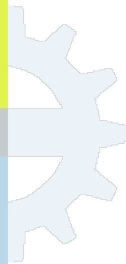
$BA \rightarrow AB$
$CB \rightarrow BC$
$CA \rightarrow AC$

$LA \rightarrow a$
$aA \rightarrow aa$
$aB \rightarrow ab$
$bB \rightarrow bb$
$bC \rightarrow bc$
$cC \rightarrow cc$



For example, we can generate the sentence $aabbcc$ as follows

$S \Rightarrow LT \Rightarrow LABCT \Rightarrow LABCABC$
$\Rightarrow LABACBC \Rightarrow LAABCBC$
$\Rightarrow LAABBCC \Rightarrow aABBCC$
$\Rightarrow aaBBCC \Rightarrow aabBCC$
$\Rightarrow aabbCC \Rightarrow aabbcC$
$\Rightarrow aabbcc$



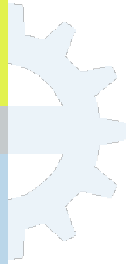
Theorem A *formal language* L *generated by an unrestricted grammar can be recognized with a Turing machine.*

Proof. Let $G = (V, \Sigma, R, S)$ be the unrestricted grammar generating language L. We devise a two-tape nondeterministic Turing machine M_G for recognizing L.

M_G maintains the input string on tape 1. On tape 2 there is some sentential form of G which we try to rewrite as the input string.

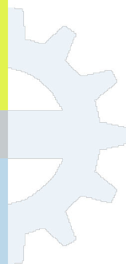
At the beginning tape 2 contains the start variable S.

The computation of M_G repeats the following stages



1. The tape head of tape 2 is (non-deterministically) moved to some location on the tape;
2. we choose (non-deterministically) some rule of G and try to apply it to the chosen location of the tape;
3. if the symbols on the tape match the symbols on the left-hand side of the rule, M_G replaces them on tape 2 with the symbols on the right-hand side of the rule;
4. we compare the strings in tapes 1 and 2 with each other;
   a) if they are equal, the Turing machine enters the accepting final state and halts,
   b) otherwise, we go back to step 1

□



Theorem *If a formal language L can be recognized with a Turing machine, then it can be generated with an unrestricted grammar.*

Proof. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \texttt{accept}, \texttt{reject})$ be a standard Turing machine recognizing language L.

Let us compose an unrestricted grammar G_M that generates L.

As variables of the grammar we take symbols representing all states $q \in Q$ of M. The configuration of the TM $u\,q\,av$ is represented as string $[uqav]$.

By the transition function of M *we* give G_M rules so that

$$[uqav] \Rightarrow_{G_M} [u'q'a'v'] \Leftrightarrow u\,q\,av \Rightarrow_M u'\,q'\,a'v'$$

Then

$$x \in L(M) \Leftrightarrow [q_0 x] \Rightarrow_{G_M} [u\mathtt{accept}v],\ \ u, v \in \Sigma^*$$

There are three types of rules in G_M:

1. Those that generate any string $x[q_0x]$, $x \in \Sigma^*$ and $[,], q_0 \in V$ from the start variable:

$$S \rightarrow T[q_0]$$
$$T \rightarrow \varepsilon$$
$$T \rightarrow aTA_a$$
$$A_a[q_0 \rightarrow [q_0A_a$$
$$A_ab \rightarrow bA_a$$
$$A_a] \rightarrow a]$$

2. Those that simulate the transition function of the Turing machine:

$\delta(q, a) = (q', b, R)$	$qa \rightarrow bq'$
$\delta(q, a) = (q', b, L)$	$cqa \rightarrow q'cb$
$\delta(q, \Box) = (q', b, R)$	$q] \rightarrow bq']$
$\delta(q, \Box) = (q', b, L)$	$cq] \rightarrow q'cb]$
$\delta(q, \Box) = (q', \Box, L)$	$cq] \rightarrow q'c]$

3. Those that replace a string of the form $[u\mathtt{accept}v]$ to an empty string

$$\mathtt{accept} \rightarrow E_L E_R$$
$$aE_L \rightarrow E_L$$
$$[E_L \rightarrow \varepsilon$$
$$E_R a \rightarrow E_R$$
$$E_R] \rightarrow \varepsilon$$

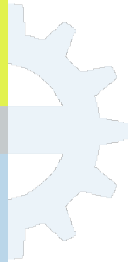
Now a string x in $L(M)$ can be generated as follows

$$S \Rightarrow_1 x[q_0 x] \Rightarrow_2 x[u\mathtt{accept}v] \Rightarrow_3 x$$

□

3.3 The Definition of Algorithm

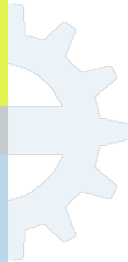
- The formulations of computation by Alonzo Church and Alan Turing were given in response to Hilbert's tenth problem which he posed in 1900 in his list of 23 challenges for the new century
- What Hilbert essentially asked for was an algorithm for determining whether a polynomial has an integral root
- Today we know that this problem is algorithmically unsolvable
- It is possible to give algorithms without them being exactly defined, but it is not possible to show that such cannot exist without a proper definition
- In was not until 1970 that Matijasevič showed that no algorithm exists for testing whether a polynomial has integral roots



- Expressed as a formal language Hilbert's tenth problem is

$$D = \{ p \mid p \text{ is a polynomial with an integral root} \}$$

- Concentrating on single variable polynomials we can see how the language D could be recognized
- In order to find the correct value of the only variable, we go through its possible integral values $0, 1, -1, 2, -2, 3, -3, \dots$
- If the polynomial attains value 0 with any examined value of the variable, then we accept the input
- A similar approach is possible when there are multiple variables in the polynomial



- For a single variable polynomial the roots must lie within

$$\pm k(c_{max} \,/\, c_1),$$

  - where k is the number of terms in the polynomial,
  - c_{max} is the coefficient with largest absolute value, and
  - c_1 is the coefficient of the highest order term

- If a root is not found within these bounds, the machine rejects
- Matijasevič's theorem shows that calculating such bounds for multivariable polynomials is impossible
- The language D can, thus, be recognized with a Turing machine, but cannot be decided with a Turing machine (may never halt)

Computability Theory

- We will examine the *algorithmic solvability* of problems
  - I.e. solvability using Turing machines
- We make a distinction between cases in which formal languages can be recognized with a Turing machine and those in which the Turing machine is required to halt with each input
- It turns out that there any many natural and interesting problems that cannot be solved using a Turing machine
- Hence, by Church-Turingin thesis these problems are unsolvable by a computer!

Definition 3.5 Call a language Turing-recognizable (or recursively enumerable, RE-language) if some Turingin machine recognizes it.

Definition 3.6 Call a language Turing-decidable (or decidable, or recursive) if some TM decides it (halts on every input, is total).

- The decision problem corresponding to language A is decidable if A is Turing-decidable.
- A problem that is not decidable is undecidable
- The decision problem corresponding to language A is semidecidable if A is Turing-recognizable
- Observe: an undecidable problem can be semidecidable.

Basic Properties of Turing-recognizable Languages

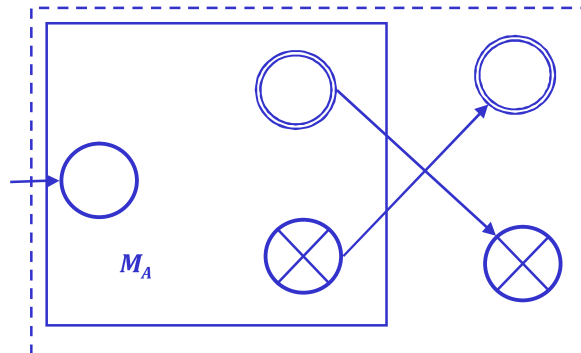
Theorem A *Let $A, B \subseteq \Sigma^*$ be Turing-decidable languages. Then also languages*

1. $\bar{A} = \Sigma^* \setminus A$,
2. $A \cup B$, *and*
3. $A \cap B$

are Turing-decidable.

Proof.

1. Let M_A be the total TM recognizing language A. By exchanging the accepting and rejecting final state of M_A with each other, we get a total Turing machine deciding the language $\bar{A}$.



The deciding machine for the complement of A

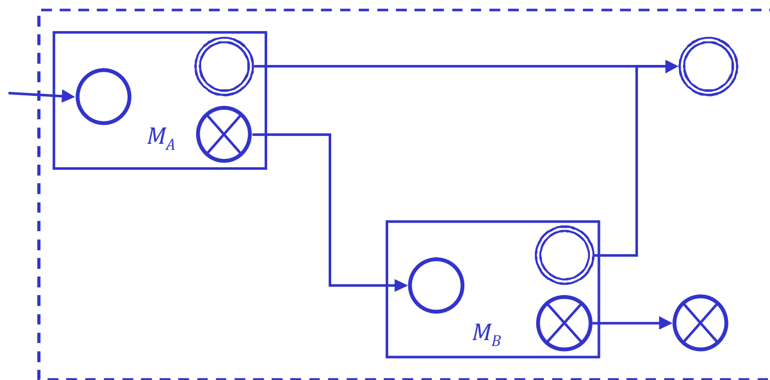
2. Let M_A and M_B, respectively, be the total Turing machines deciding A and B.
   - Let us combine them so that we first check whether M_A accepts the input.
   - If it does, so does the combined TM.
   - On the other hand, if M_A rejects the input, we pass it on to TM M_B check.
   - In this case M_B decides whether the input will be accepted. M_A must pass the original input to M_B.
   - It is clear that the combined TM is total and accepts the language

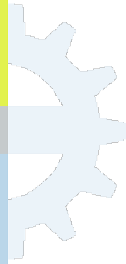
$$A \cup B = \{ x \in \Sigma^* \mid x \in A \lor x \in B \}$$

3. The Turing-decidability of $(A \cap B)$ follows from the previous results because

$$A \cap B = \overline{\overline{A} \cup \overline{B}}$$

The deciding machine for the union of languages A and B:



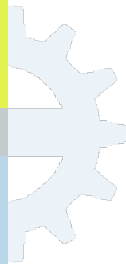


Theorem B *Let $A, B \subseteq \Sigma^*$ be Turing-recognizable languages. Then also languages $A \cup B$ and $A \cap B$ are Turing-recognizable.*

Proof. Exercises.

- As a consequence of Theorem C we get a hold of languages that are not Turing-recognizable ($\bar{A}$ is the complement of A):

Theorem D *Let $A \subseteq \Sigma^*$ be a Turing-recognizable language that is not Turing-decidable. Then $\bar{A}$ is not Turing-recognizable.*



Theorem C *Language $A \subseteq \Sigma^*$ is Turing-decidable ⇔ A and $\bar{A}$ are Turing-recognizable.*

Proof.

⇨ If A is Turing-decidable, then it is also Turing-recognizable. By Theorem A(1) the same holds also for $\bar{A}$.

⇦ Let M_A and $M_{\bar{A}}$ be the TMs recognizing A and $\bar{A}$.

For all $x \in \Sigma^*$ it holds that either $x \in A$ or $x \in \bar{A}$. In other words, either M_A or $M_{\bar{A}}$ halts on input x.

Combining machines M_A and $M_{\bar{A}}$ to run parallel gives a total Turing machine for recognizing A. □

Total Turing machine for recognizing language A: