ÇUKUROVA UNIVERSITY
FACULTY OF ENGINEERING
COMPUTER ENGINEERING DEPARTMENT

Lecture 3: Computability

Computability: How to formally specify an algorithm?

➔ Turing (Thue, Post, Klune, Church, …)

3.1. Computability with Turing Machine

A Turing Machine (TM), thought of by the mathematician Alan Turing in 1936, is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected. Despite its simplicity, the TM can simulate ANY computer algorithm, no matter how complicated it is!

Repetition from last week: A TM can be formally described as a 7-tuple (Σ, T, S, δ, s_0, #, F) where

- Σ is the input alphabet
- **T** is the tape alphabet (wheras $\Sigma \subset$ **T**)
- **S** is a finite set of states
- **δ** is a transition function; $\delta : S \times T \rightarrow S \times T \times$ {Left_shift, Right_shift, or no_movement}.
- $\mathbf{s_0}$ is the initial state
- # is the blank symbol
- **F** is the set of final states

TM Example: $L(TM) = \{a^n b^n \mid n \geq 1\}$ => **ab, aabb, aaabbb, etc.**

Turing machine M = (Σ, T, S, δ, S_0, #, F) with

- Σ = {a,b}
- T = {a, b, A, B, #}
- S = {s_0, s_1, s_2, s_3, s_4}
- S_0 = {s_0}
- # = blank symbol
- F = {s_4}

δ is given by

Current state	Symbol read		Symbol written	New state	Move instruction
s_0	a	→	A	s_1	Right
s_0	B	→	B	s_3	Right
s_1	a	→	a	s_1	Right
s_1	b	→	B	s_2	Left
s_1	B	→	B	s_1	Right
s_2	a	→	a	s_2	Left
s_2	A	→	A	s_0	Right
s_2	B	→	B	s_2	Left
s_3	B	→	B	s_3	Right
s_3	#	→	#	s_4	Right

Input word: a^2b^2

$$s_0aabb\# \vdash As_1abb\# \vdash Aas_1bb\# \vdash As_2aBb\# \vdash s_2AaBb\# \vdash As_0aBb\# \vdash AAs_1Bb\#$$
$$\vdash AABs_1b\# \vdash AAs_2BB\# \vdash As_2ABB\# \vdash AAs_0BB\# \vdash AABs_3B\#$$
$$\vdash AABBs_{3\#} \vdash AABB\#s_4$$

➔ **A problem is computable if there is a TM that can solve that problem.**
➔ $L(TM) = \{a^n b^n \mid n \geq 1\}$ is computable!

Homework: Design a TM for the following language: $L(TM) = \{a^n b^n c^n \mid n \geq 1\}$

3.2. Computability with Queue automaton and Pushdown automata with 2 stacks

Queue automaton (QA)

QA is similar to Pushdown Automaton (PDA) but has a queue instead of a stack which helps Queue automaton to recognize languages beyond Context Free Languages. QA works according to First-in First-out (FIFO) principle.

The working tape is a queue, i.e. a data structure that only allows two types of access−

- **enqueue** − a new symbol is added at the rear of the queue.
- **dequeue** − the symbol at the front of the queue is removed.

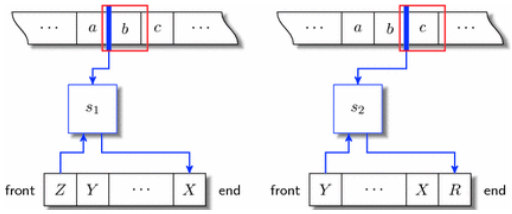

Figure 2: Design of QA

A QA is a **6-tuple (Σ, Γ, S, δ, s_0, #)**, where

1. **Σ** is a set of finite **input alphabet**,
2. **Γ** is a set of finite **queue alphabet**,
3. **S** is a set of **states**,
4. $\delta \subseteq S \times \Gamma \times S \times \Gamma^*$

   Configuration $c = (s, \gamma) \in S \times \Gamma^*$

   Transition: $(S \times \Gamma^*) \times (S \times \Gamma^*)$ defined through

   $(s, A\alpha) \vdash (s', \alpha\beta)$ iff. $(s, A, s', \beta) \in \delta$, $\quad s, s' \in S, A \in \Gamma, \alpha, \beta \in \Gamma^*$
5. s_0 is the **initial state** $\in S$
6. # is the blank symbol $\in \Gamma - \Sigma$

Example: $L(QA) = \{a^n b^n c^n \mid n \geq 1\}$ => abc, aabbcc, aaabbbccc, etc.

Idea: $a^n b^n c^n$ is processed in rounds: in each round an *a*, a *b* and a *c* are deleted, and all other letters are added to the end of the word. If the word belongs to the language, the queue is processed in n rounds. If the input word does not belong to the language, the final configuration is not reached.

$$Q = (\{a, b, c\}, \{a, b, c, \#\}, \{s_0, s_a, s_b, s_c\}, \delta, s_0, \#)$$

$$\delta = \begin{Bmatrix} (s_0, \#, s_0, \varepsilon), \\ (s_0, a, s_a, \varepsilon), \\ (s_a, a, s_a, \mathrm{a}), \\ (s_a, b, s_b, \varepsilon), \\ (s_b, b, s_b, \mathrm{b}), \\ (s_b, c, s_c, \varepsilon), \\ (s_c, c, s_c, \mathrm{c}), \\ (s_c, \#, s_0, \#) \end{Bmatrix}$$

Input word: $a^2b^2c^2$

$$(s_0, a^2b^2c^2\#) \vdash (s_a, ab^2c^2\#) \vdash (s_a, b^2c^2\#a) \vdash (s_b, bc^2\#a) \vdash (s_b, c^2\#ab) \vdash (s_c, c\#ab)$$
$$\vdash (s_c, \#abc) \vdash (s_0, abc\#) \vdash (s_a, bc\#) \vdash (s_b, c\#) \vdash (s_c, \#) \vdash (s_0, \#) \vdash (s_0, \varepsilon)$$

Note: It has been proven that a queue machine is equivalent to a Turing machine by showing that a queue machine can simulate a Turing machine and vice versa.

Furthermore, it has been proved that a PDA with 2 stacks (2PDA) is also equivalent to a Turing machine.

Consequently (!):

In addition to Turing computability, two further computability terms are the **Queue computability** and the **2PDA computability**.

QA_Σ: Queue Automat (First in First out – FIFO)
$QA_\Sigma = TA_\Sigma$

Pushdown automata (PDA) with 2 stacks (2PDA)
$2PDA_\Sigma = TA_\Sigma$

3.3. Computability with programming language approaches: loop, while, goto

In this section we introduce further computability concepts. Unlike Turing machines, they are not based on the formalization of an intuitive concept of computability, but on concepts of procedural programming languages. We will show that goto and while computability are equivalent to each other and to Turing computability, and that loop computability is weaker than these.

The programming language LOOP

We first define the syntax of the LOOP programming language, then we define the semantics of LOOP programs.

The LOOP alphabet includes:

- variables **x_0, x_1, x_2,** … which serve as identifiers for memory locations, each of which can contain a natural number,
- Constants **0, 1, 2, . . .**, which serve as identifiers for natural numbers,
- the separator **;**,
- the assignment symbol **:=**,
- the operator symbols + and **-**,
- the keywords **read, write, loop, do, end**.

A LOOP statement is defined as: **loop** x **do** A **end**;

If A is a LOOP statement, then

read$(x_1, x_2, \ldots x_k)$; A; write (x_0)

is a LOOP program.

Example LOOP program:

read (x_1, x_2);
$x_0 := x_1 + 0$;
loop x_2 **do**
$x_0 := x_0 + 1$
end;
write (x_0)

The programming language LOOP has no explicit selection statement, i.e., no statement of the kind

if B **then** A_1 **else** A_2 **endif**

where B is a condition and A_1 and A_2 are two statements. However, if statements can be simulated by LOOP statements. For the instruction

if $x_p = 0$ **then** A_1 **else** A_2 **endif**

the following LOOP statement sequence represents a possible simulation:

$x_q := 1$
$x_r := 1$
loop x_p **do** $x_q := 0$ **end**;
loop x_q **do** A_1 ; $x_r := 0$ **end**;
loop x_r **do** A_2 **end** ;

The programming language WHILE

We add another loop statement to the LOOP programming language, the **while** statement, and call the extended programming language WHILE. In contrast to the loop statement, the number of executions of the loop body in the while statement is not fixed before the beginning of the evaluation, but the number of executions can depend on the results of the loop body executions. Accordingly, the loop variable in the loop body of the while statement may not only be used, but also manipulated, i.e., it may also be "write" accessed.

while x ≠0 **do** A **end**;

The WHILE programming language is defined as follows:

i. Each LOOP statement is a WHILE statement.

ii. Let A be a WHILE statement, then

while x_i ≠0 **do** A **endwhile**

is also a WHILE statement.

iii. If A is a WHILE statement, then

read(x_i, , x_k); A; **write**(x_0)

a WHILE program.

The programming language GOTO

As a further programming language, we want to consider the language GOTO. In addition to elementary statements, it does not contain any loop statements, but two jump statements.

label x; A **goto** x;

In addition to symbols for variables x_0, x_1, … and constants c_0, c_1, …, the alphabet of GOTO programming language includes symbols for labels, which we denote by l_1, l_2, ... (l stands for label), as well as the keywords **read**, **write**, **goto**, **if**, **then**, and **stop**, the assignment symbol **:=**,

the operation symbols =, +, -, and the separators **:** and **;**. The following statements are unmarked GOTO statements:

i. the assignments $x_i := x_j + c_k$ and $x_i := x_i - c_k$,
ii. (unconditional) jumps **goto** l_i,
iii. conditional jumps **if** $x_i = c_i$ **then goto** l_k,
iv. the stop statement **stop**.

If A is an unmarked **goto** statement and l is a mark, then

l : A

is called a marked goto statement.

A syntactically correct GOTO program consists of a sequence of marked GOTO statements surrounded by a read and write statement:

read $(x_1, \dots, x_k)$;
$l_1 := A_1$;
.
.
.
$l_n := A_n$;
write (x_0)

Relationship:

⇨ loop ⊂ while = goto = Turing computability

It is known that

⇨ f: loop-computable functions → total computable functions

Question: How about the other direction?

⇨ loop-computable functions ← total computable functions **?**

∉

Ackermann function

3.4. Primitive Recursive Functions, Composition, μ-Recursion

Primitive Functions:

- **Null function:** $0: \mathbb{N}_0 \rightarrow \mathbb{N}_0 \;\; 0(x) = 0$
- **Successor function:** $S: \mathbb{N}_0 \rightarrow \mathbb{N}_0 \; f(x) = x + 1$
- **Projection:** $\pi_i^k: \mathbb{N}_0^k \rightarrow \mathbb{N}_0 \;\; \pi_i^k(x_1, \dots, x_k) = x_i$

Defining Primitive Recursive Functions (PRFs):

Let g: $\mathbb{N}_0^n \rightarrow \mathbb{N}_0$, h: $\mathbb{N}_0^{n+2} \rightarrow \mathbb{N}_0$ be primitive recursive. Then f: $\mathbb{N}_0^{n+1} \rightarrow \mathbb{N}_0$ with

$$f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$$

$$f\big(x_1, \dots, x_n, S(y)\big) = h\big(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)\big)$$

is also primitive recursive.

PRF #1: Addition

Definition:

$$add(x, 0) = x$$

$$add\big(x, S(y)\big) = S\left(\pi_3^3\big(x, y, add(x, y)\big)\right)$$

Example:

$$add(2,3) = S\left(\pi_3^3\big(2,3, add(2,2)\big)\right)$$

$$= S\left(\pi_3^3\left(2,3, S\left(\pi_3^3\big(2,3, add(2,1)\big)\right)\right)\right)$$

$$= S\left(\pi_3^3\left(2,3, S\left(\pi_3^3\left(2,3, S\left(\pi_3^3\big(2,3, add(2,0)\big)\right)\right)\right)\right)\right)$$

$$= S\left(\pi_3^3\left(2,3, S\left(\pi_3^3\left(2,3, S\big(\pi_3^3(2,3,2)\big)\right)\right)\right)\right)$$

$$= S\left(\pi_3^3\left(2,3, S\left(\pi_3^3\big(2,3, S(2)\big)\right)\right)\right)$$

$$= S\left(\pi_3^3\left(2,3, S\big(\pi_3^3(2,3,3)\big)\right)\right)$$

$$= S\left(\pi_3^3(2,3,S(3))\right)$$
$$= S(\pi_3^3(2,3,4))$$
$$= S(4)$$
$$= 5$$

Composition of Primitive Recursive Functions:

Let $h_1, \ldots, h_n: \mathbb{N}_0 \to \mathbb{N}_0$, $g: \mathbb{N}_0^n \to \mathbb{N}_0$ be primitive recursive, then $f: \mathbb{N}_0^k \to \mathbb{N}_0$ with

$$f(x_1, \ldots, x_k) = g\big(h_1(x_1, \ldots, x_k), h_2(x_1, \ldots, x_k), \ldots, h_n(x_1, \ldots, x_k)\big)$$

is also primitive recursive.

or

$h_1, \ldots, h_n: \mathbb{N}_0 \to \mathbb{N}_0$, $g: \mathbb{N}_0^n \to \mathbb{N}_0$ is a primitive rekursive function, then $f: \mathbb{N}_0^k \to \mathbb{N}_0$ with

$$f(x_1, \ldots, x_k) = g \circ (h_1, h_2, \ldots, h_n)\,(x_1, \ldots, x_n)$$

is also primitive recursive.

PRF #2: Multiplication

Definition:

$$mult(x, 0) = 0$$

$$mult\big(x, S(y)\big) = add \circ (\pi_1^3, \pi_3^3)\big(x, y, mult(x, y)\big)$$

Example:

$$mult(3,2) = add \circ (\pi_1^3, \pi_3^3)\big(3,2, mult(3,1)\big)$$
$$= add \circ (\pi_1^3, \pi_3^3)\left(3,2, add \circ (\pi_1^3, \pi_3^3)\big(3,2, mult(3,0)\big)\right)$$
$$= add \circ (\pi_1^3, \pi_3^3)\big(3,2, add \circ (\pi_1^3, \pi_3^3)(3,2,0)\big)$$
$$= add \circ (\pi_1^3, \pi_3^3)\big(3,2, add(3,0)\big)$$
$$= add \circ (\pi_1^3, \pi_3^3)(3,2,3)$$
$$= add(3,3)$$
$$= 6$$

PRF #3: Subtraction

$$sub: N_0\ x\ N_0 \rightarrow N_0 \quad sub(x,y) = \begin{cases} x-y, & x \geq y \\ 0, & x < y \end{cases}$$

To show that the sub function is primitive recursive, we need a predecessor function P. The predecessor function P acts as the opposite of the successor function S and is recursively defined by the rules:

$$P: N_0 \rightarrow N_0 \quad P(x) = \begin{cases} x-1, & x \geq y \\ 0, & x < y \end{cases}$$

Definition:

$$P(0) = 0$$
$$P\big(S(x)\big) = \pi_1^2\big(x, P(x)\big)$$

Example:

Predecessor of 4?

$$\begin{aligned} P\big(S(3)\big) &= \pi_1^2\big(3, P(2)\big) \\ &= \pi_1^2\left(3, \pi_1^2\big(3, P(1)\big)\right) \\ &= \pi_1^2\left(3, \pi_1^2\left(3, \pi_1^2\big(3, P(0)\big)\right)\right) \\ &= \pi_1^2\left(3, \pi_1^2\big(3, \pi_1^2(3,0)\big)\right) \\ &= \pi_1^2\big(3, \pi_1^2(3,3)\big) \\ &= \pi_1^2(3,3) \\ &= 3 \end{aligned}$$

With the help of the predecessor function P, now we can show that the sub function is primitive recursive:

Definition:

$$sub(x,0) = x$$
$$sub\big(x, S(y)\big) = P\left(\pi_3^3\big(x, y, sub(x,y)\big)\right)$$

Example:

$$\begin{aligned} sub(4,2) &= P\left(\pi_3^3\big(4,2, sub(4,1)\big)\right) \\ &= P\left(\pi_3^3\left(4,2, P\left(\pi_3^3\big(4,2, sub(4,0)\big)\right)\right)\right) \\ &= P\left(\pi_3^3\left(4,2, P\big(\pi_3^3(4,2,4)\big)\right)\right) \end{aligned}$$

$$= P\left(\pi_3^3(4,2,P(4))\right)$$
$$= P\left(\pi_3^3(4,2,3)\right)$$
$$= P(3)$$
$$= 2$$

PRF #4: Exponent

$$exp: N_0\ x\ N_0 \rightarrow N_0 \quad exp(b,y) = b^y$$

Definition:

$$exp(x,0) = 1$$
$$exp\left(x,S(y)\right) = mult \circ (\pi_1^3, \pi_3^3)\left(x,y,exp(x,y)\right)$$

Example:

$$exp(2,3) = mult \circ (\pi_1^3, \pi_3^3)\left(2,3,exp(2,2)\right)$$
$$= mult \circ (\pi_1^3, \pi_3^3)\left(2,3,mult \circ (\pi_1^3, \pi_3^3)\left(2,3,exp(2,1)\right)\right)$$
$$= mult \circ (\pi_1^3, \pi_3^3)\left(2,3,mult \circ (\pi_1^3, \pi_3^3)\left(2,3,mult \circ (\pi_1^3, \pi_3^3)\left(2,3,exp(2,0)\right)\right)\right)$$
$$= mult \circ (\pi_1^3, \pi_3^3)\left(2,3,mult \circ (\pi_1^3, \pi_3^3)\left(2,3,mult \circ (\pi_1^3, \pi_3^3)(2,3,1)\right)\right)$$
$$= mult \circ (\pi_1^3, \pi_3^3)\left(2,3,mult \circ (\pi_1^3, \pi_3^3)\left(2,3,mult(2,1)\right)\right)$$
$$= mult \circ (\pi_1^3, \pi_3^3)\left(2,3,mult \circ (\pi_1^3, \pi_3^3)(2,3,2)\right)$$
$$= mult \circ (\pi_1^3, \pi_3^3)\left(2,3,mult(2,2)\right)$$
$$= mult \circ (\pi_1^3, \pi_3^3)(2,3,4)$$
$$= mult(2,4)$$
$$= 8$$

Homeworks:

$$sign: N_0 \rightarrow N_0 \quad sign(x) = \begin{cases} 1, & x \geq 1 \\ 0 & x = 0 \end{cases}$$

$$equal: N_0\ x\ N_0 \rightarrow N_0 \quad equal(x,y) = \begin{cases} 1, & x = y \\ 0 & x \neq y \end{cases}$$

➔ Primitive Recursive Functions (PRF) = Loop-Computable Functions

µ-Operator (Mikro-Operator):

f: $\mathbb{N}_0^{n+1} \rightarrow \mathbb{N}_0$ µ[f]: $\mathbb{N}_0^n \rightarrow \mathbb{N}_0$

$\mu[f]\ (x_1, \dots, x_k) = min\{z \mid (x_1, \dots, x_n, y) \in Def(f)\ for\ y \leq z\ and\ f(x_1, \dots, x_n, z) = 0\}$

Example:

log: $\mathbb{N}_0^2 \rightarrow \mathbb{N}_0$ $log_b(x, y) = x - exp(b, y)$

Assumption: b is constant. The µ-recursive computation of log: $\mathbb{N}_0^2 \rightarrow \mathbb{N}_0$ with the following definition:

$\mu[\log]\ (x) = min\{z \mid log(x, y) \in Def(f)\ for\ y \leq z\ and\ log(x, z) = 0\}$

The procedure for b = 2, x = 4

$\mu[\log]\ (4, 0) = 3$
$\mu[\log]\ (4, 1) = 2$
$\mu[\log]\ (4, 2) = 0$ ← The correct result (Note: The log function is not total)

- ⇨ The class that comes together by combining/nesting **primitive recursion**, **composition**, and/or the **µ-operator**, is called as the **class of µ-recursive functions** $\mathbb{P}$ (also referred to as the **class of partial recursive functions).**
- ⇨ A distinction is made between the set of µ-recursive functions that are **total**, i.e. $\mathbb{R}$.

Ackermann Function

We know that

- ⇨ f: loop-computable functions → total computable functions

Question: How about the other direction?

- ⇨ loop-computable functions ← total computable functions **?**

∉

Ackermann function

Ackermann function, referred to as $ack(n,m)$ or $ack_n(m)$, is defined as follows:

$$ack(0,m) = m+1$$
$$ack(n,0) = ack(n-1,1)$$
$$ack(n,m) = ack\big(n-1, ack(n,m-1)\big)$$

Example #1:

$$\begin{aligned} ack(1,2) &= ack\big(0, ack(1,1)\big) \\ &= ack\Big(0, ack\big(0, ack(1,0)\big)\Big) \\ &= ack\Big(0, ack\big(0, ack(0,1)\big)\Big) \\ &= ack\big(0, ack(0,2)\big) \\ &= ack(0,3) \\ &= 4 \end{aligned}$$

Example #2:

$$\begin{aligned} ack(2,2) &= ack\big(1, ack(2,1)\big) \\ &= ack\big(1, ack(1, ack(2,0)\big) \\ &= ack\big(1, ack(1, ack(1,1)\big) \\ &= ack\Big(1, ack\big(1, ack(0, ack(1,0)\big)\Big) \\ &= ack\Big(1, ack\big(1, ack(0, ack(0,1)\big)\Big) \\ &= ack\big(1, ack(1, ack(0,2)\big) \\ &= ack\big(1, ack(1,3)\big) \\ &= ack\Big(1, ack\big(0, ack(1,2)\big)\Big) \\ &= ack\Big(1, ack\Big(0, ack\big(0, ack(1,1)\big)\Big)\Big) \\ &= ack\Big(1, ack\Big(0, ack\Big(0, ack\big(0, ack(1,0)\big)\Big)\Big)\Big) \\ &= ack\Big(1, ack\Big(0, ack\Big(0, ack\big(0, ack(0,1)\big)\Big)\Big)\Big) \\ &= ack\Big(1, ack\Big(0, ack\big(0, ack(0,2)\big)\Big)\Big) \\ &= ack\Big(1, ack\big(0, ack(0,3)\big)\Big) \\ &= ack\Big(1, ack\big(0, ack(0,3)\big)\Big) \\ &= ack\big(1, ack(0,4)\big) \\ &= ack(1,5) \\ &= ack\big(0, ack(1,4)\big) \\ &= ack\Big(0, ack\big(0, ack(1,3)\big)\Big) \end{aligned}$$

$$= ack\left(0, ack\left(0, ack\left(0, ack(1,2)\right)\right)\right)$$
$$= ack\left(0, ack\left(0, ack\left(0, ack\left(0, ack(1,1)\right)\right)\right)\right)$$
$$= ack\left(0, ack\left(0, ack\left(0, ack\left(0, ack\left(0, ack(1,0)\right)\right)\right)\right)\right)$$
$$= ack\left(0, ack\left(0, ack\left(0, ack\left(0, ack\left(0, ack(0,1)\right)\right)\right)\right)\right)$$
$$= ack\left(0, ack\left(0, ack\left(0, ack\left(0, ack(0,2)\right)\right)\right)\right)$$
$$= ack\left(0, ack\left(0, ack\left(0, ack(0,3)\right)\right)\right)$$
$$= ack\left(0, ack\left(0, ack(0,4)\right)\right)$$
$$= ack\left(0, ack(0,5)\right)$$
$$= ack(0,6)$$
$$= 7$$

Growth of the Ackerman function:

The Ackermann function is a tremendously increasing function. For example, for all y:

$ack(1, y) = y + 2,$

$ack(2, y) = 2y + 3,$

$ack(3, y) = 2^{y+3} - 3$

$ack(4, y) = 2^{2^{\cdots^{2}}} - 3$

The number of twos in the last equation is $y + 3$. It applies e.g.

$ack(4,0) = 16 - 3 = 13,$

$ack(4,1) = 2^{16} - 3 = 65.533,$

$ack(4,2) = 2^{2^{16}} - 3 = 2^{65.536} - 3 > 10^{19.660} =?$

$A(m,n)$	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
$m=0$	1	2	3	4	5	6
$m=1$	2	3	4	5	6	7
$m=2$	3	5	7	9	11	13
$m=3$	5	13	29	61	125	253
$m=4$	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	$A(3,2^{2^{65536}}-3)$	$A(3,A(4,4))$
$m=5$	65533	$A(4,65533)$	$A(4,A(4,65533))$	$A(4,A(5,2))$	$A(4,A(5,3))$	$A(4,A(5,4))$
$m=6$	$A(4,65533)$	$A(5,A(4,65533))$	$A(5,A(6,1))$	$A(5,A(6,2))$	$A(5,A(6,3))$	$A(5,A(6,4))$

Ackermann's function, while Turing computable, grows faster than any primitive recursive function. The reason can be seen from the table: the index for the next value is also growing.

Properties of Ackermann function:

$ack(m,n)$

⇕

$ack_m(n)$ → Family of Ackermann functions with m=0, m=1, m=2, etc.

⇨ It applies to all primitive recursive functions: In this sequence of Ackermann functions, there is a particular function that grows faster than the primitive recursive function being considered.

1. $ack_i(x) \leq ack_{i+j}(x+y), \quad j,y \geq 0$
2. $ack_i(x+1) \leq ack_{i+1}(x), \quad x \geq x_0$

   $ack_i(x+j) \leq ack_{i+j}(x), \quad x \geq x_0$ ← Generalization of the 2. expression above

3. $\forall f \in \text{loop}\ \exists i, x_0 : f(x) \leq ack_i(x), \quad x \geq x_0$

Use the Ackerman function as benchmark

The Ackermann function, due to its definition in terms of extremely deep recursion, can be used as a benchmark of a compiler's ability to optimize recursion. The first use of Ackermann's function in this way was by Yngve Sundblad, The Ackermann function. A Theoretical, computational and formula manipulative study.

- For example, a compiler which, in analyzing the computation of A(3, 30), is able to save intermediate values like A(3, n) and A(2, n) in that calculation rather than recomputing them, can speed up computation of A(3, 30) by a factor of hundreds of thousands.

- Also, if A(2, n) is computed directly rather than as a recursive expansion of the form A(1, A(1, A(1,...A(1, 0)...))), this will save significant amounts of time.

- Instead, shortcut formulas such as $A(3, n) = 8 \times 2n - 3$ are used as an optimization to complete some of the recursive calls.

Conclusion:

So far, we have seen five formal concepts for the notion of computability: Turing, loop, while and goto computability as well as μ-recursive functions. Turing computability starts from an intuitive understanding of computability, 'calculating with pencil and paper'. Loop, while, and goto computability are programming language-like concepts, and the μ-recursive functions describe computability using only mathematical functions.

Note that they are many other computability concepts available, including Universal Register Machines (URM), λ calculus, and Markov.

Church's thesis: The class of functions that can be calculated in the intuitive sense is precisely the class of functions that can be calculated by Turing-computable (and thus while-, goto, Queue-, 2PDA-computable, …) functions.