Definition. A [partial] function f is primitive recursive ($f \in \mathbf{PRIM}$) if it can be built up in finitely many steps from the basic functions by use of the operations of composition and primitive recursion.

In other words, the set $\mathbf{PRIM}$ of primitive recursive functions is the smallest set (with respect to subset inclusion) of partial functions containing the basic functions and closed under the operations of composition and primitive recursion.

FACT: every $f \in$ PRIM is a total function

Definition. A partial function f is partial recursive ($f \in \mathbf{PR}$) if it can be built up in finitely many steps from the basic functions by use of the operations of composition, primitive recursion and minimization.

The members of $\mathbf{PR}$ that are total are called recursive functions.

Fact: there are recursive functions that are not primitive recursive. For example...

$$\mathrm{add} = \rho^1(\mathrm{proj}_1^1, \mathrm{succ} \circ \mathrm{proj}_3^3)$$

$$\mathrm{pred} = \rho^0(\mathrm{zero}^0, \mathrm{proj}_1^2)$$

$$\begin{aligned} e ::= {} & \mathrm{proj}_i^n \mid \mathrm{zero}^n \mid \mathrm{succ} \\ \mid {} & e \circ [e, \ldots, e] \\ \mid {} & \rho^n(e, e) \end{aligned}$$

Definition. A partial function f is partial recursive ($f \in \mathbf{PR}$) if it can be built up in finitely many steps from the basic functions by use of the operations of composition, primitive recursion and minimization.

The members of $\mathbf{PR}$ that are total are called recursive functions.

Fact: there are recursive functions that are not primitive recursive. For example...

it's possible to construct a computable function $e : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ satisfying

$$e(n, x) = \text{value of } n^{\text{th}} \text{ PRIM fn. at } x$$

A diagonalization argument shows $e \notin \mathrm{PRIM}$ (see CST 2017, p 6, q 4)

Examples of recursive definitions

.

$$\begin{cases} f_2(0) & \equiv 0 \\ f_2(1) & \equiv 1 \\ f_2(x+2) & \equiv f_2(x) + f_2(x+1) \end{cases}$$

$f_2(x) = x$th Fibonacci number

$f_2 \in$ PRIM even though this is not a primitive recursive definition

(see CST 2014, paper 6, question 4)

Ackermann's function

There is a (unique) function $ack \in \mathbb{N}^2 \to \mathbb{N}$ satisfying

$$\begin{array}{rcl} ack(0, x_2) & = & x_2 + 1 \\ ack(x_1 + 1, 0) & = & ack(x_1, 1) \\ ack(x_1 + 1, x_2 + 1) & = & ack(x_1, ack(x_1 + 1, x_2)) \end{array}$$

Ackermann's function

There is a (unique) function $ack \in \mathbb{N}^2 \to \mathbb{N}$ satisfying

$$\begin{array}{rcl} ack(0, x_2) & = & x_2 + 1 \\ ack(x_1 + 1, 0) & = & ack(x_1, 1) \\ ack(x_1 + 1, x_2 + 1) & = & ack(x_1, ack(x_1 + 1, x_2)) \end{array}$$

- ack is computable, hence recursive [proof: exercise].

```
        OCaml version 4.00.1

# let rec ack (x : int)(y : int) : int =
  match x ,y with
      0 , y -> y+1
    | x , 0 -> ack (x-1) 1
    | x ,y -> ack (x-1) (ack x (y-1));;
val ack : int -> int -> int = <fun>
# ack 0 0;;
- : int = 1
# ack 1 1;;
- : int = 3
# ack 2 2;;
- : int = 7
# ack 3 3;;
- : int = 61
# ack 4 4;;
Stack overflow during evaluation (looping recursion?).
#
```

Chat GPT says...

SM

what is the next number in the sequence 1, 3, 7, 61 ?

It is difficult to determine the next number in this sequence as it does not follow any recognizable pattern. The first three numbers, 1, 3, and 7, are consecutive odd numbers and may lead one to believe that the sequence involves odd numbers. However, the fourth number, 61, is not an odd number, and it is not immediately clear how it is related to the previous three numbers.

Without any additional information or context, it is impossible to determine the next number in the sequence. It is possible that there is a typo or an error in the sequence, or that it is part of a larger pattern or context that is not immediately apparent.

THE ON-LINE ENCYCLOPEDIA OF INTEGER SEQUENCES ®

founded in 1964 by N. J. A. Sloane

1,3,7,61 Search Hints

(Greetings from The On-Line Encyclopedia of Integer Sequences!)

Search: **seq:1,3,7,61**

Displaying 1-4 of 4 results found. page 1

Sort: relevance | references | number | modified | created Format: long | short | data

A183174 Numbers n such that (10^(2n+1)-6*10^n-1)/3 is prime.

+30
45

1, 3, 7, 61, 90, 92, 269, 298, 321, 371, 776, 1567, 2384, 2566, 3088, 5866, 8051, 9498, 12635, 24512, 32521, 43982 (list; graph; refs; listen; history; text; internal format)

OFFSET	1,2
COMMENTS	a(23) > 10^5. - Robert Price, Jan 29 2016
REFERENCES	C. Caldwell and H. Dubner, "Journal of Recreational Mathematics", Volume 28, No. 1, 1996-97, pp. 1-9.
LINKS	Table of n, a(n) for n=1..22. Patrick De Geest, World!Of Numbers, Palindromic Wing Primes (PWP's) Makoto Kamada, Prime numbers of the form 33...33133...33 Index entries for primes involving repunits.
FORMULA	a(n) = (A077775(n)-1)/2.
MATHEMATICA	Do[If[PrimeQ[(10^(2n + 1) - 6*10^n - 1)/3], Print[n]], {n, 3000}]
PROG	(PARI) for(n=1, 1e3, if(ispseudoprime((10^(2*n+1)-6*10^n-1)/3), print1(n", "))) \\ Charles R Greathouse IV, Jul 15 2011
CROSSREFS	Cf. A004023, A077775-A077798, A107123-A107127, A107648, A107649, A115073, A183174-A183187.
KEYWORD	nonn,base
AUTHOR	Ray Chandler, Dec 28 2010
EXTENSIONS	a(21)-a(22) from Robert Price, Jan 29 2016
STATUS	approved

A046859 Simplified Ackermann function (main diagonal of Ackermann-Péter function).

+30
8

1, 3, 7, 61 (list; graph; refs; listen; history; text; internal format)

OFFSET	0,2
COMMENTS	The next term is 2^(2^(2^(2^16))) - 3, which is too large to display in the DATA lines.

Ackermann's function

There is a (unique) function $ack \in \mathbb{N}^2 \to \mathbb{N}$ satisfying

$$\begin{array}{rcl} ack(0, x_2) & = & x_2 + 1 \\ ack(x_1 + 1, 0) & = & ack(x_1, 1) \\ ack(x_1 + 1, x_2 + 1) & = & ack(x_1, ack(x_1 + 1, x_2)) \end{array}$$

- **Fact:** ack grows faster than any primitive recursive function $f \in \mathbb{N}^2 \to \mathbb{N}$:
  $\exists N_f \ \forall x_1, x_2 > N_f \ (f(x_1, x_2) < ack(x_1, x_2))$.
  Hence ack is not primitive recursive.

Ackermann's function

There is a (unique) function $ack \in \mathbb{N}^2 \to \mathbb{N}$ satisfying

$$\begin{aligned} ack(0, x_2) &= x_2 + 1 \\ ack(x_1 + 1, 0) &= ack(x_1, 1) \\ ack(x_1 + 1, x_2 + 1) &= ack(x_1, ack(x_1 + 1, x_2)) \end{aligned}$$

In fact, writing a_x for $ack(x, -) \in \mathbb{N} \to \mathbb{N}$, one has

$$a_{x+1}(y) = (\underbrace{a_x \circ \cdots \circ a_x}_{\text{compose } y \text{ times}})(1) \leftarrow \text{this is an e.g. of a prim. rec. definition "of higher type"}$$

Lambda calculus

Notions of computability

- Church (1936): λ-calculus
- Turing (1936): Turing machines.

Turing showed that the two very different approaches determine the same class of computable functions. Hence:

Church-Turing Thesis. Every algorithm [in intuitive sense of Lect. 1] can be realized as a Turing machine.

Notation for <u>function definitions</u> in mathematical discourse:

NAMED

"let f be the function $f(x) = x^2 + x + 1$ .... $[f]$..."

ANONYMOUS

"the function $x \mapsto x^2 + x + 1$ ..."

"the function $\underline{\lambda x .\ x^2 + x + 1}$ ..."

LAMBDA NOTATION

λ-Terms, M

are built up from a given, countable collection of

- variables $x, y, z, \ldots$

by two operations for forming λ-terms:

- λ-abstraction: $(\lambda x.M)$
  (where x is a variable and M is a λ-term)
- application: $(M\, M')$
  (where M and M' are λ-terms).

Some random examples of λ-terms:

$$x \quad (\lambda x.x) \quad ((\lambda y.(x\,y))x) \quad (\lambda y.((\lambda y.(x\,y))x))$$

λ-Terms, M

Notational conventions:

- $(\lambda x_1\, x_2 \ldots x_n.M)$ means $(\lambda x_1.(\lambda x_2 \ldots (\lambda x_n.M) \ldots))$
- $(M_1\, M_2 \ldots M_n)$ means $(\ldots (M_1\, M_2) \ldots M_n)$ (i.e. application is left-associative)
- drop outermost parentheses and those enclosing the body of a λ-abstraction. E.g. write $(\lambda x.(x(\lambda y.(y\, x))))$ as $\lambda x.x(\lambda y.y\, x)$.
- $x \# M$ means that the variable x does not occur anywhere in the λ-term M.

Free and bound variables

In $\lambda x.M$, we call x the bound variable and M the body of the λ-abstraction.

An occurrence of x in a λ-term M is called

- binding if in between λ and $.$
  (e.g. $(\lambda x.y\,x)\,x$)
- bound if in the body of a binding occurrence of x
  (e.g. $(\lambda x.y\,x)\,x$)
- free if neither binding nor bound
  (e.g. $(\lambda x.y\,x)x$).

Free and bound variables

Sets of free and bound variables:

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.M) &= FV(M) - \{x\} \\
FV(M\,N) &= FV(M) \cup FV(N)
\end{aligned}
$$

$$
\begin{aligned}
BV(x) &= \emptyset \\
BV(\lambda x.M) &= BV(M) \cup \{x\} \\
BV(M\,N) &= BV(M) \cup BV(N)
\end{aligned}
$$

E.g. $FV((\lambda x.\, y\, x)\, x) = \{x, y\}$

$BV((\lambda x.\, y\, x)\, x) = \{x\}$

Free and bound variables

Sets of free and bound variables:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.M) &= FV(M) - \{x\} \\ FV(M\,N) &= FV(M) \cup FV(N) \end{aligned}$$

$$\begin{aligned} BV(x) &= \emptyset \\ BV(\lambda x.M) &= BV(M) \cup \{x\} \\ BV(M\,N) &= BV(M) \cup BV(N) \end{aligned}$$

If $FV(M) = \emptyset$, M is called a closed term, or combinator.

E.g. $FV(\lambda y.\lambda x.(\lambda x.y\,x)\,x) = \emptyset$

α-Equivalence $M =_\alpha M'$

$\lambda x.M$ is intended to represent the function f such that

$$f(x) = M \text{ for all } x.$$

So the name of the bound variable is immaterial: if $M' = M\{x'/x\}$ is the result of taking M and changing all occurrences of x to some variable $x' \# M$, then $\lambda x.M$ and $\lambda x'.M'$ both represent the same function.

For example, $\lambda x.x$ and $\lambda y.y$ represent the same function (the identity function).

α-Equivalence $M =_\alpha M'$

is the binary relation inductively generated by the rules:

$$\frac{}{x =_\alpha x} \qquad \frac{z \# (M\, N) \qquad M\{z/x\} =_\alpha N\{z/y\}}{\lambda x.M =_\alpha \lambda y.N}$$

$$\frac{M =_\alpha M' \qquad N =_\alpha N'}{M\, N =_\alpha M'\, N'}$$

where $M\{z/x\}$ is M with all occurrences of x replaced by z.

α-Equivalence $M =_\alpha M'$

For example:

$$\lambda \underline{x}.(\lambda \underline{x} x'.\underline{x})\, x' =_\alpha \lambda \underline{y}.(\lambda x\, x'.x) x'$$

because

α-Equivalence $M =_\alpha M'$

For example:

$$\lambda x.(\lambda x x'.x)\, x' =_\alpha \lambda y.(\lambda x\, x'.x) x'$$

because $\quad (\lambda z\, x'.z) x' =_\alpha (\lambda x\, x'.x) x'$

because

α-Equivalence $M =_\alpha M'$

For example:

$$\lambda x.(\lambda x x'.x)\, x' =_\alpha \lambda y.(\lambda x\, x'.x) x'$$

because $(\lambda z\, x'.z) x' =_\alpha (\lambda x\, x'.x) x'$

because $\lambda z\, x'.z =_\alpha \lambda x\, x'.x$ and $x' =_\alpha x'$

because

α-Equivalence $M =_\alpha M'$

For example:

$$\lambda x.(\lambda x x'.x)\, x' =_\alpha \lambda y.(\lambda x\, x'.x) x'$$

because $(\lambda z\, x'.z) x' =_\alpha (\lambda x\, x'.x) x'$

because $\lambda z\, x'.z =_\alpha \lambda x\, x'.x$ and $x' =_\alpha x'$

because $\lambda x'.u =_\alpha \lambda x'.u$ and $x' =_\alpha x'$

because

α-Equivalence $M =_\alpha M'$

For example:

$$\lambda x.(\lambda x x'.x)\, x' =_\alpha \lambda y.(\lambda x\, x'.x) x'$$

because $(\lambda z\, x'.z) x' =_\alpha (\lambda x\, x'.x) x'$

because $\lambda z\, x'.z =_\alpha \lambda x\, x'.x$ and $x' =_\alpha x'$

because $\lambda x'.u =_\alpha \lambda x'.u$ and $x' =_\alpha x'$

because $u =_\alpha u$ and $x' =_\alpha x'$.

α-Equivalence $M =_\alpha M'$

Fact: $=_\alpha$ is an equivalence relation (reflexive, symmetric and transitive).

We do not care about the particular names of bound variables, just about the distinctions between them. So α-equivalence classes of λ-terms are more important than λ-terms themselves.

- Textbooks (and these lectures) suppress any notation for α-equivalence classes and refer to an equivalence class via a representative λ-term (look for phrases like “we identify terms up to α-equivalence” or “we work up to α-equivalence”).
- For implementations and computer-assisted reasoning, there are various devices for picking canonical representatives of α-equivalence classes (e.g. de Bruijn indexes, graphical representations, ...).