Arbitrary-precision integers (included)

Arbitrary-precision integers (included)
You are encouraged to solve this task according to the task description, using any language you may know.

Using the in-built capabilities of your language, calculate the integer value of:

$$5^{4^{3^2}}$$

- Confirm that the first and last twenty digits of the answer are:

```
62060698786608744707...92256259918212890625
```

- Find and show the number of decimal digits in the answer.

Note:

- Do not submit an *implementation* of arbitrary precision arithmetic. The intention is to show the capabilities of the language as supplied. If a language has a single, overwhelming, library of varied modules that is endorsed by its home site – such as CPAN for Perl or Boost for C++ – then that *may* be used instead.
- Strictly speaking, this should not be solved by fixed-precision numeric libraries where the precision has to be manually set to a large value; although if this is the **only** recourse then it may be used with a note explaining that the precision must be set manually to a large enough value.

Related tasks

- Long multiplication
- Exponentiation order
- exponentiation operator
- Exponentiation with infix operators in (or operating on) the base

11l

Translation of: Python

```
V y = String(BigInt(5) ^ 4 ^ 3 ^ 2)
print(‘5^4^3^2 = #....#. and has #. digits’.format(y[0.<20], y[(len)-20..], y.len))
```

Output:

```
5^4^3^2 = 62060698786608744707...92256259918212890625 and has 183231 digits
```

8th

```
200000 n#
5 4 3 2 bfloat ^ ^ ^
"%.0f" s:strfmt
dup s:len . " digits" . cr
dup 20 s:lsub . "..." .  20 s:rsub . cr
```

Output:

```
183231 digits
62060698786608744707...92256259918212890625
```

ACL2

```
(in-package "ACL2")

(include-book "arithmetic-3/floor-mod/floor-mod" :dir :system)

(set-print-length 0 state)

(defun arbitrary-precision ()
   (declare (xargs :mode :program))
   (let* ((x (expt 5 (expt 4 (expt 3 2))))
          (s (mv-let (col str)
                     (fmt1-to-string "~xx"
                                     (list (cons #\x x))
                                     0)
                (declare (ignore col))
                str)))
         (cw "~s0 ... ~x1 (~x2 digits)~%"
             (subseq s 0 20)
             (mod x (expt 10 20))
             (1- (length s)))))
```

Output:

```
62060698786608744707 ... 92256259918212890625 (183231 digits)
```

Ada

Library: GMP

Using GMP, Ada bindings provided in GNATColl

```
with Ada.Text_IO; use Ada.Text_IO;
with GNATCOLL.GMP; use GNATCOLL.GMP;
with GNATCOLL.GMP.Integers; use GNATCOLL.GMP.Integers;
procedure ArbitraryInt is
   type stracc is access String;
   BigInt : Big_Integer;
   len : Natural;
   str : stracc;
begin
   Set (BigInt, 5);
   Raise_To_N (BigInt, Unsigned_Long (4**(3**2)));
   str := new String'(Image (BigInt));
   len := str'Length;
   Put_Line ("Size is:"& Natural'Image (len));
   Put_Line (str (1 .. 20) & "....." & str (len - 19 .. len));
end ArbitraryInt;
```

Output:

```
Size is: 183231
62060698786608744707.....92256259918212890625
```

ALGOL 68

```
BEGIN
COMMENT
   The task specifies

   "Strictly speaking, this should not be solved by fixed-precision
   numeric libraries where the precision has to be manually set to a
   large value; although if this is the only recourse then it may be
   used with a note explaining that the precision must be set manually
   to a large enough value."

   Now one should always speak strictly, especially to animals and
   small children and, strictly speaking, Algol 68 Genie requires that
   a non-default numeric precision for a LONG LONG INT be specified by
   "precision=<integral denotation>" either in a source code PRAGMAT
   or as a command line argument.  However, that specification need
   not be made manually.  This snippet of code outputs an appropriate
   PRAGMAT

   printf (($gg(0)xgl$, "PR precision=",
        ENTIER (1.0 + log (5) * 4^(3^(2))), "PR"));

   and the technique shown in the "Call a foreign-language function"
   task used to write, compile and run an Algol 68 program in which
   the precision is programmatically determined.

   The default stack size on this machine is also inadequate but twice
   the default is sufficient.  The PRAGMAT below can be machine
   generated with

   printf (($gg(0)xgl$, "PR stack=", 2 * system stack size, "PR"));

COMMENT
   PR precision=183231 PR
   PR stack=16777216 PR
   INT digits = ENTIER (1.0 + log (5) * 4^(3^(2))), exponent = 4^(3^2);
   LONG LONG INT big = LONG LONG 5^exponent;
   printf (($gxg(0)l$, " First 20 digits:", big % LONG LONG 10 ^ (digits - 20)));
   printf (($gxg(0)l$, "  Last 20 digits:", big MOD LONG LONG 10 ^ 20));
   printf (($gxg(0)l$, "Number of digits:", digits))
```

```
END
```

Output:

```
 First 20 digits: 62060698786608744707
  Last 20 digits: 92256259918212890625
Number of digits: 183231
```

Alore

```
def Main()
  var len as Int
  var result as Str
  result = Str(5**4**3**2)
  len = result.length()
  Print(len)
  Print(result[:20])
  Print(result[len-20:])
end
```

Arturo

```
num: to :string 5^4^3^2

print [first.n: 20 num ".." last.n: 20 num "=>" size num "digits"]
```

Output:

```
62060698786608744707 .. 92256259918212890625 => 183231 digits
```

bc

```
/* 5432.bc */

y = 5 ^ 4 ^ 3 ^ 2
c = length(y)
" First 20 digits: "; y / (10 ^ (c - 20))
"  Last 20 digits: "; y % (10 ^ 20)
"Number of digits: "; c
quit
```

Output:

```
$ time bc 5432.bc
 First 20 digits: 62060698786608744707
  Last 20 digits: 92256259918212890625
Number of digits: 183231
    0m24.81s real     0m24.81s user     0m0.00s system
```

Bracmat

At the prompt type the following one-liner:

```
{?} @(5^4^3^2:?first [20 ? [-21 ?last [?length)&str$(!first "..." !last "\nlength " !length)
{!} 62060698786608744707...92256259918212890625
length 183231
    S   2,46 sec
```

C

Library: GMP

```
#include <gmp.h>
#include <stdio.h>
#include <string.h>

int main()
{
    mpz_t a;
    mpz_init_set_ui(a, 5);
    mpz_pow_ui(a, a, 1 << 18); /* 2**18 == 4**9 */

    int len = mpz_sizeinbase(a, 10);
    printf("GMP says size is: %d\n", len);

    /* because GMP may report size 1 too big; see doc */
    char *s = mpz_get_str(0, 10, a);
    printf("size really is %d\n", len = strlen(s));
    printf("Digits: %.20s...%s\n", s, s + len - 20);

    // free(s); /* we could, but we won't. we are exiting anyway */
    return 0;
}
```

Output:

```
GMP says size is: 183231
size really is 183231
Digits: 62060698786608744707...92256259918212890625
```

Library: OpenSSL

OpenSSL is about 17 times slower than GMP (on one computer), but still fast enough for this small task.

```
/* 5432.c */

#include <openssl/bn.h>     /* BN_*() */
#include <openssl/err.h>    /* ERR_*() */
#include <stdlib.h>     /* exit() */
#include <stdio.h>      /* fprintf() */
#include <string.h>     /* strlen() */
```

```
void
fail(const char *message)
{
    fprintf(stderr, "%s: error 0x%08lx\n", ERR_get_error());
    exit(1);
}

int
main()
{
    BIGNUM two, three, four, five;
    BIGNUM answer;
    BN_CTX *context;
    size_t length;
    char *string;

    context = BN_CTX_new();
    if (context == NULL)
        fail("BN_CTX_new");

    /* answer = 5 ** 4 ** 3 ** 2 */
    BN_init(&two);
    BN_init(&three);
    BN_init(&four);
    BN_init(&five);
    if (BN_set_word(&two, 2) == 0 ||
        BN_set_word(&three, 3) == 0 ||
        BN_set_word(&four, 4) == 0 ||
        BN_set_word(&five, 5) == 0)
        fail("BN_set_word");
    BN_init(&answer);
    if (BN_exp(&answer, &three, &two, context) == 0 ||
        BN_exp(&answer, &four, &answer, context) == 0 ||
        BN_exp(&answer, &five, &answer, context) == 0)
        fail("BN_exp");

    /* string = decimal answer */
    string = BN_bn2dec(&answer);
    if (string == NULL)
        fail("BN_bn2dec");

    length = strlen(string);
    printf(" First 20 digits: %.20s\n", string);
    if (length >= 20)
        printf("  Last 20 digits: %.20s\n", string + length - 20);
    printf("Number of digits: %zd\n", length);

    OPENSSL_free(string);
    BN_free(&answer);
    BN_free(&five);
    BN_free(&four);
    BN_free(&three);
    BN_free(&two);
    BN_CTX_free(context);

    return 0;
}
```

Output:

```
$ make LDLIBS=-lcrypto 5432
cc -O2 -pipe    -o 5432 5432.c -lcrypto
$ time ./5432
 First 20 digits: 62060698786608744707
  Last 20 digits: 92256259918212890625
Number of digits: 183231
    0m1.30s real     0m1.30s user     0m0.00s system
```

C#

`System.Numerics.BigInteger` was added in C# 4. The exponent of `BigInteger.Pow()` is limited to a 32-bit signed integer, which is not a problem in this specific task.

Works with: C# version 4+

```
using System;
using System.Diagnostics;
using System.Linq;
using System.Numerics;

static class Program {
    static void Main() {
        BigInteger n = BigInteger.Pow(5, (int)BigInteger.Pow(4, (int)BigInteger.Pow(3, 2)));
        string result = n.ToString();

        Debug.Assert(result.Length == 183231);
        Debug.Assert(result.StartsWith("62060698786608744707"));
        Debug.Assert(result.EndsWith("92256259918212890625"));

        Console.WriteLine("n = 5^4^3^2");
        Console.WriteLine("n = {0}...{1}",
            result.Substring(0, 20),
            result.Substring(result.Length - 20, 20)
            );

        Console.WriteLine("n digits = {0}", result.Length);
    }
}
```

Output:

```
n = 5^4^3^2
n = 62060698786608744707...92256259918212890625
n digits = 183231
```

C++

Library: Boost-Multiprecision (GMP Backend)

To compile link with GMP `-lgmp`

```
#include <iostream>
#include <boost/multiprecision/gmp.hpp>
#include <string>

namespace mp = boost::multiprecision;

int main(int argc, char const *argv[])
{
    // We could just use (1 << 18) instead of tmpres, but let's point out one
    // pecularity with gmp and hence boost::multiprecision: they won't accept
    // a second mpz_int with pow(). Therefore, if we stick to multiprecision
    // pow we need to convert_to<uint64_t>().
    uint64_t tmpres = mp::pow(mp::mpz_int(4)
                              , mp::pow(mp::mpz_int(3)
                                      , 2).convert_to<uint64_t>()
                                        ).convert_to<uint64_t>();
    mp::mpz_int res = mp::pow(mp::mpz_int(5), tmpres);
    std::string s = res.str();
    std::cout << s.substr(0, 20)
              << "..."
              << s.substr(s.length() - 20, 20) << std::endl;
    return 0;
}
```

Output:

```
62060698786608744707...92256259918212890625
```

Ceylon

Be sure to import ceylon.whole in your module.ceylon file.

```
import ceylon.whole {
    wholeNumber,
    two
}

shared void run() {

    value five = wholeNumber(5);
    value four = wholeNumber(4);
    value three = wholeNumber(3);

    value bigNumber = five ^ four ^ three ^ two;

    value firstTwenty = "62060698786608744707";
    value lastTwenty =  "92256259918212890625";
    value bigString = bigNumber.string;

    "The number must start with ``firstTwenty`` and end with ``lastTwenty``"
    assert(bigString.startsWith(firstTwenty), bigString.endsWith(lastTwenty));

    value bigSize = bigString.size;
    print("The first twenty digits are ``bigString[...19]``");
    print("The last twenty digits are ``bigString[(bigSize - 20)...]``");
    print("The number of digits in 5^4^3^2 is ``bigSize``");
}
```

Output:

```
The first twenty digits are 62060698786608744707
The last twenty digits are 92256259918212890625
The number of digits in 5^4^3^2 is 183231
```

Clojure

```
(defn exp [n k] (reduce * (repeat k n)))
```

```
(def big (->> 2 (exp 3) (exp 4) (exp 5)))
(def sbig (str big))

(assert (= "62060698786608744707" (.substring sbig 0 20)))
(assert (= "92256259918212890625" (.substring sbig (- (count sbig) 20))))
(println (count sbig) "digits")

(println (str (.substring sbig 0 20) ".."
          (.substring sbig (- (count sbig) 20)))
     (str "(" (count sbig) " digits)"))
```

Output:

```
output> 62060698786608744707..92256259918212890625 (183231 digits)
```

Redefining *exp* as follows speeds up the calculation of *big* about a hundred times:

```
(defn exp [n k]
  (cond
    (zero? (mod k 2)) (recur (* n n) (/ k 2))
    (zero? (mod k 3)) (recur (* n n n) (/ k 3))
    :else (reduce * (repeat k n))))
```

CLU

This program uses the `bigint` type that is supplied with Portable CLU, in `misc.lib`. The program must be merged with that library in order to work.

The type is not included in the CLU specification, however it is included as a library with the reference implementation.

```
start_up = proc ()
    % Get bigint versions of 5, 4, 3 and 2
    five: bigint := bigint$i2bi(5)
    four: bigint := bigint$i2bi(4)
    three: bigint := bigint$i2bi(3)
    two: bigint := bigint$i2bi(2)

    % Calculate 5**4**3**2
    huge_no: bigint := five ** four ** three ** two

    % Turn answer into string
    huge_str: string := bigint$unparse(huge_no)

    % Scan for first digit (the string will have some leading whitespace)
    i: int := 1
    while huge_str[i] = ' ' do i := i + 1 end

    po: stream := stream$primary_output()
    stream$putl(po, "First 20 digits: "
            || string$substr(huge_str, i, 20))
    stream$putl(po, "Last  20 digits: "
            || string$substr(huge_str, string$size(huge_str)-19, 20))
    stream$putl(po, "Amount of digits: "
            || int$unparse(string$size(huge_str) - i + 1))
end start_up
```

Output:

```
First 20 digits: 62060698786608744707
Last  20 digits: 92256259918212890625
Amount of digits: 183231
```

COBOL

This entry might be pushing the limits of the spirit of the task. COBOL does not have arbitrary-precision integers in the spec, but it does mandate a precision of some 1000 digits with intermediate results, from 10^-999 through 10^1000, for purposes of rounding financially sound decimal arithmetic. GnuCOBOL uses libgmp or equivalent to meet and surpass this requirement, but this precision is not exposed to general programming in the language. The capabilities are included in the GnuCOBOL implementation run-time support, but require access to some of the opaque features of libgmp for use in this task.

This listing includes a few calculations, 12345**9 is an example that demonstrates the difference between the library's view of certain string lengths and a native C view of the data.

Works with: GnuCOBOL
Library: GMP

```
       identification division.
       program-id. arbitrary-precision-integers.
       remarks. Uses opaque libgmp internals that are built into libcob.

       data division.
       working-storage section.
       01 gmp-number.
          05 mp-alloc          usage binary-long.
          05 mp-size           usage binary-long.
          05 mp-limb           usage pointer.
       01 gmp-build.
          05 mp-alloc          usage binary-long.
          05 mp-size           usage binary-long.
          05 mp-limb           usage pointer.

       01 the-int              usage binary-c-long unsigned.
       01 the-exponent         usage binary-c-long unsigned.
       01 valid-exponent       usage binary-long value 1.
          88 cant-use          value 0 when set to false 1.

       01 number-string        usage pointer.
       01 number-length        usage binary-long.

       01 window-width         constant as 20.
       01 limit-width          usage binary-long.
       01 number-buffer        pic x(window-width) based.

       procedure division.
       arbitrary-main.

      *> calculate 10 ** 19
       perform initialize-integers.
       display "10 ** 19        : " with no advancing
       move 10 to the-int
       move 19 to the-exponent
       perform raise-pow-accrete-exponent
```

```
           perform show-all-or-portion
           perform clean-up

          *> calculate 12345 ** 9
           perform initialize-integers.
           display "12345 ** 9      : " with no advancing
           move 12345 to the-int
           move 9 to the-exponent
           perform raise-pow-accrete-exponent
           perform show-all-or-portion
           perform clean-up

          *> calculate 5 ** 4 ** 3 ** 2
           perform initialize-integers.
           display "5 ** 4 ** 3 ** 2: " with no advancing
           move 3 to the-int
           move 2 to the-exponent
           perform raise-pow-accrete-exponent
           move 4 to the-int
           perform raise-pow-accrete-exponent
           move 5 to the-int
           perform raise-pow-accrete-exponent
           perform show-all-or-portion
           perform clean-up
           goback.
          *> ***************************************************************

           initialize-integers.
           call "__gmpz_init" using gmp-number returning omitted
           call "__gmpz_init" using gmp-build returning omitted
           .

           raise-pow-accrete-exponent.
          *> check before using previously overflowed exponent intermediate
           if cant-use then
               display "Error: intermediate overflow occured at "
                       the-exponent upon syserr
               goback
           end-if
           call "__gmpz_set_ui" using gmp-number by value 0
               returning omitted
           call "__gmpz_set_ui" using gmp-build by value the-int
               returning omitted
           call "__gmpz_pow_ui" using gmp-number gmp-build
               by value the-exponent
               returning omitted
           call "__gmpz_set_ui" using gmp-build by value 0
               returning omitted
           call "__gmpz_get_ui" using gmp-number returning the-exponent
           call "__gmpz_fits_ulong_p" using gmp-number
               returning valid-exponent
           .

          *> get string representation, base 10
           show-all-or-portion.
           call "__gmpz_sizeinbase" using gmp-number
               by value 10
               returning number-length
           display "GMP length: " number-length ", " with no advancing

           call "__gmpz_get_str" using null by value 10
               by reference gmp-number
               returning number-string
           call "strlen" using by value number-string
               returning number-length
           display "strlen: " number-length

          *> slide based string across first and last of buffer
           move window-width to limit-width
           set address of number-buffer to number-string
```

```
        if number-length <= window-width then
            move number-length to limit-width
            display number-buffer(1:limit-width)
        else
            display number-buffer with no advancing
            subtract window-width from number-length
            move function max(0, number-length) to number-length
            if number-length <= window-width then
                move number-length to limit-width
            else
                display "..." with no advancing
            end-if
            set address of number-buffer up by
                function max(window-width, number-length)
            display number-buffer(1:limit-width)
        end-if
        .

        clean-up.
        call "free" using by value number-string returning omitted
        call "__gmpz_clear" using gmp-number returning omitted
        call "__gmpz_clear" using gmp-build returning omitted
        set address of number-buffer to null
        set cant-use to false
        .

        end program arbitrary-precision-integers.
```

Output:

```
prompt$ cobc -xj arbitrary-integer.cob
10 ** 19        : GMP length: +0000000020, strlen: +0000000020
10000000000000000000
12345 ** 9      : GMP length: +0000000038, strlen: +0000000037
6659166111488656281486807152009765625
5 ** 4 ** 3 ** 2: GMP length: +0000183231, strlen: +0000183231
62060698786608744707...92256259918212890625
```

Common Lisp

Common Lisp has arbitrary precision integers, inherited from MacLisp: "[B]ignums—arbitrary precision integer arithmetic—were added [to MacLisp] in 1970 or 1971 to meet the needs of Macsyma users." [*Evolution of Lisp* [1] (http://dreamsongs.com/Files/Hopl2.pdf), 2.2.2]

```
(let ((s (format () "~s" (expt 5 (expt 4 (expt 3 2))))))
  (format t "~a...~a, length ~a" (subseq s 0 20)
          (subseq s (- (length s) 20)) (length s)))
```

Output:

```
62060698786608744707...92256259918212890625, length 183231
```

Crystal

```
require "big"

z = (BigInt.new(5) ** 4 ** 3 ** 2).to_s
zSize = z.size
puts "5**4**3**2 = #{z[0, 20]} .. #{z[zSize-20, 20]} and has #{zSize} digits"
```

Output:

```
5**4**3**2 = 62060698786608744707 .. 92256259918212890625 and it has 183231 digits
```

D

```
void main() {
  import std.stdio, std.bigint, std.conv;

  auto s = text(5.BigInt ^^ 4 ^^ 3 ^^ 2);
  writefln("5^4^3^2 = %s..%s (%d digits)", s[0..20], s[$-20..$], s.length);
}
```

Output:

```
5^4^3^2 = 62060698786608744707..92256259918212890625 (183231 digits)
```

With dmd about 0.55 seconds compilation time (-release -noboundscheck) and about 3.3 seconds run time.

Dart

Originally Dart's integral type **int** supported arbitrary length integers, but this is no longer the case; Dart now supports integral type **BigInt**.

```
import 'dart:math' show pow;

int fallingPowers(int base) =>
    base == 1 ? 1 : pow(base, fallingPowers(base - 1));

void main() {
  final exponent = fallingPowers(4),
      s = BigInt.from(5).pow(exponent).toString();
  print('First twenty:     ${s.substring(0, 20)}');
  print('Last twenty:      ${s.substring(s.length - 20)}');
  print('Number of digits: ${s.length}');
```

Output:

```
First twenty:     62060698786608744707
Last twenty:      92256259918212890625
Number of digits: 183231
```

dc

Translation of: bc

```
[5432.dc]sz

5 4 3 2 ^ ^ ^ sy                [y = 5 ^ 4 ^ 3 ^ 2]sz
ly Z sc                     [c = length of y]sz
[ First 20 digits: ]P ly 10 lc 20 - ^ / p sz    [y / (10 ^ (c - 20))]sz
[  Last 20 digits: ]P ly 10 20 ^ % p sz     [y % (10 ^ 20)]sz
[Number of digits: ]P lc p sz
```

Output:

```
$ time dc 5432.dc
 First 20 digits: 62060698786608744707
  Last 20 digits: 92256259918212890625
Number of digits: 183231
    0m24.80s real     0m24.81s user     0m0.00s system
```

Delphi

Library: System.SysUtils
Library: Velthuis.BigIntegers

Thanks for Rudy Velthuis, BigIntegers Library [2] (https://github.com/rvelthuis/DelphiBigNumbers).

```
program Arbitrary_precision_integers;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  Velthuis.BigIntegers;

var
  value: BigInteger;
  result: string;

begin
  value := BigInteger.pow(3, 2);
  value := BigInteger.pow(4, value.AsInteger);
  value := BigInteger.pow(5, value.AsInteger);
  result := value.tostring;
  Write('5^4^3^2 = ');
  Write(result.substring(0, 20), '...');
  Write(result.substring(result.length - 20, 20));
  Writeln(' (', result.Length,' digits)');
  readln;
end.
```

Output:

```
5^4^3^2 = 62060698786608744707...92256259918212890625 (183231 digits)
```

E

E implementations are required to support arbitrary-size integers transparently.

```
? def value := 5**(4**(3**2)); null
? def decimal := value.toString(10); null
? decimal(0, 20)
# value: "62060698786608744707"

? decimal(decimal.size() - 20)
# value: "92256259918212890625"

? decimal.size()
# value: 183231
```

EchoLisp

```
;; to save space and time, we do'nt stringify Ω = 5^4^3^2 ,
;; but directly extract tail and head and number of decimal digits

(lib 'bigint) ;; arbitrary size integers

(define e10000 (expt 10 10000)) ;; 10^10000

(define (last-n big (n 20))
(string-append "..." (number->string (modulo big (expt 10 n)))))

(define (first-n big (n 20))
    (while (> big e10000)
        (set! big (/ big e10000))) ;; cut 10000 digits at a time
    (string-append (take (number->string big) n) "..."))

;; faster than directly using (number-length big)
(define (digits big (digits 0))
    (while (> big e10000)
        (set! big (/ big e10000))
        (set! digits (1+ digits)))
    (+ (* digits 10000) (number-length big)))

(define Ω (expt 5 (expt 4 (expt 3 2))))

(last-n Ω )
    → "...92256259918212890625"
(first-n Ω )
    → "62060698786608744707..."
(digits Ω )
    → 183231
```

Elixir

Translation of: Erlang

```
defmodule Arbitrary do
  def pow(_,0), do: 1
  def pow(b,e) when e > 0, do: pow(b,e,1)

  defp pow(b,1,acc), do: acc * b
  defp pow(b,p,acc) when rem(p,2)==0, do: pow(b*b,div(p,2),acc)
  defp pow(b,p,acc), do: pow(b,p-1,acc*b)
  def test do
    s = pow(5,pow(4,pow(3,2))) |> to_string
    l = String.length(s)
    prefix = String.slice(s,0,20)
    suffix = String.slice(s,-20,20)
    IO.puts "Length: #{l}\nPrefix:#{prefix}\nSuffix:#{suffix}"
  end
end
Arbitrary.test
```

Output:

```
Length: 183231
Prefix:62060698786608744707
Suffix:92256259918212890625
```

Emacs Lisp

As of Emacs 27.1, bignums are supported via GMP. However, there is a configurable limit on the maximum bignum size. If the limit is exceeded, an overflow error is raised.

```
(let* ((integer-width (* 65536 16)) ; raise bignum limit from 65536 bits to avoid overflow error
       (answer (number-to-string (expt 5 (expt 4 (expt 3 2)))))
       (length (length answer)))
  (message "%s has %d digits"
       (if (> length 40)
           (format "%s...%s"
               (substring answer 0 20)
               (substring answer (- length 20) length))
         answer)
       length))
```

Emacs versions older than 27.1 do not support bignums, but include Calc, a library that implements big integers. The `calc-eval` function takes an algebraic formula in a string, and returns the result in a string.

Library: Calc

```
(let* ((answer (calc-eval "5**4**3**2"))
       (length (length answer)))
  (message "%s has %d digits"
       (if (> length 40)
           (format "%s...%s"
               (substring answer 0 20)
               (substring answer (- length 20) length))
         answer)
       length))
```

This implementation is *very slow*; one computer, running GNU Emacs 23.4.1, needed about seven minutes to find the answer.

Output:

```
62060698786608744707...92256259918212890625 has 183231 digits
```

Erlang

Erlang supports arbitrary precision integers. However, the math:pow function returns a float. This implementation includes an implementation of pow for integers with exponent greater than 0.

```
-module(arbitrary).
-compile([export_all]).

pow(B,E) when E > 0 ->
    pow(B,E,1).

pow(_,0,_) -> 0;
pow(B,1,Acc) -> Acc * B;
pow(B,P,Acc) when P rem 2 == 0 ->
    pow(B*B,P div 2, Acc);
pow(B,P,Acc) ->
    pow(B,P-1,Acc*B).

test() ->
    I = pow(5,pow(4,pow(3,2))),
    S = integer_to_list(I),
    L = length(S),
    Prefix = lists:sublist(S,20),
    Suffix = lists:sublist(S,L-19,20),
    io:format("Length: ~b~nPrefix:~s~nSuffix:~s~n",[L,Prefix,Suffix]).
```

Output:

23> arbitrary:test().

```
Length: 183231
Prefix:62060698786608744707
Suffix:92256259918212890625
ok
```

F#

You can specifiy arbitrary-precision integers (bigint or System.Numeric.BigInteger) in F# by postfixing the number with the letter 'I'. While '**' is the power function, two things should be noted:

- bigint does not support raising to a power of a bigint
- The int type does not support the power method

```
let () =
    let answer = 5I **(int (4I ** (int (3I ** 2))))
    let sans = answer.ToString()
    printfn "Length = %d, digits %s ... %s" sans.Length (sans.Substring(0,20))
(sans.Substring(sans.Length-20))
;;
Length = 183231, digits 62060698786608744707 ... 92256259918212890625
```

Factor

Factor has built-in bignum support. Operations on integers overflow to bignums.

```
USING: formatting kernel math.functions math.parser sequences ;
IN: rosettacode.bignums

: test-bignums ( -- )
    5 4 3 2 ^ ^ ^ number>string
    [ 20 head ] [ 20 tail* ] [ length ] tri
    "5^4^3^2 is %s...%s and has %d digits\n" printf ;
```

It prints: `5^4^3^2 is 62060698786608744707...92256259918212890625 and has 183231 digits`

Forth

ANS Forth has no in-built facility for arbitrarily-sized numbers, but libraries are available.

Works with: ANS Forth
Library: The Forth Scientific Library

Here is a solution using The Forth Scientific Library, available here (https://www.taygeta.com/fsl/scilib.html).

The big.fth (https://www.taygeta.com/fsl/library/big.fth) needs to be patched to allow printing of numbers with more than 256 digits. Note that this restriction is only on the printing routines, as the size of numbers handled by the library is only limited by the available memory.

The patch is:

```
388c388
< CREATE big_string 256 CHARS ALLOT
---
> CREATE big_string 500000 CHARS ALLOT
394c394
<   big_string 256 CHARS + bighld ! ;  \ Haydon p 67
---
>   big_string 500000 CHARS + bighld ! ;  \ Haydon p 67
403c403
<   big_string 256 CHARS +     \ One past end of string
---
>   big_string 500000 CHARS +  \ One past end of string
```

Here is the solution:

```
INCLUDE big.fth

big_digit_pointer *exp                  \ iteration counter
big_digit_pointer *base                 \ point to base of number being exponentiated

big_digit_pointer *0                    \ store 0
big_digit_pointer *1                    \ store 1
big_digit_pointer *result               \ point to result of exponentiation
big_digit_pointer *temp                 \ point to temporary result

: big--                                 ( addr-n -- )
    DUP 0 *1 big-                       ( addr-n addr-n-1 ) \ decrement value
    SWAP DUP @ ABS 1+ CELLS MOVE    ( ) \ overwrite addr-n with addr-n-1
;

: big>                              ( addr1 addr2 -- flag )
    2DUP big< >R                    ( addr1 addr2 ) ( R: flag1 )
    big=                            ( flag2 ) ( R: flag1 )
    R> OR 0=                        ( ! <= )
;

: big^                              ( addr-base addr-exp - addr )
    to_pointer *exp
    to_pointer *base

    HERE 1 , 0 , to_pointer *0      \ create limit value for counter
    HERE 1 , 1 , to_pointer *1      \ create subtraend for counter
    HERE 1 , 1 , to_pointer *result \ create result = 1
    BEGIN
        0 *exp 0 *0 big>            ( flag ) \ loop while exp > 0
        0 *exp big--                \ prepare for next iteration
    WHILE
        0 *result 0 *base big*
        to_pointer *temp            \ temp = result * base

        0 *temp 0 *result
        reposition                  \ move [temp,HERE[ to [result,result+size[
    REPEAT
    0 *result 0 *0
    reposition                      \ overwrite *0 with result
    0 *0                            \ and return its address
;

: big-show                          ( addr -- )
    \ show first 20 and last 20 digits of the big number
    <big# big#s #big>               ( addr n )  \ returns string
    DUP . ."  digits" CR            \ show number of digits
    DUP 50 > IF
        OVER 20 TYPE ." ..."        \ show first 20 digits
        + 20 - 20 TYPE CR           \ show last 20 digits
    ELSE
        TYPE CR
    THEN
;

\ compute 5^(4^(3^2))
big 5 big 4 big 3 big 2 big^ big^ big^ big-show CR
```

and the output:

```
183231  digits
62060698786608744707...92256259918212890625
```

Fortran

Modern Fortran has no in-built facility for arbitrarily-sized numbers, but libraries are available.

FM library

Here is a solution using David M. Smith's FM library, available here (http://myweb.lmu.edu/dmsmith/fmlib.html).

```
program bignum
    use fmzm
    implicit none
    type(im) :: a
    integer :: n

    call fm_set(50)
    a = to_im(5)**(to_im(4)**(to_im(3)**to_im(2)))
    n = to_int(floor(log10(to_fm(a))))
    call im_print(a / to_im(10)**(n - 19))
    call im_print(mod(a, to_im(10)**20))
end program
```

```
62060698786608744707
92256259918212890625
```

FreeBASIC

freebasic has it's own gmp static library. Here, a power function operates via a string and uinteger.

```
#Include once "gmp.bi"
Dim Shared As Zstring * 100000000 outtext

Function  Power(number As String,n As Uinteger) As String'automate precision
    #define dp 3321921
    Dim As __mpf_struct _number,FloatAnswer
    Dim As Ulongint ln=Len(number)*(n)*4
    If ln>dp Then ln=dp
    mpf_init2(@FloatAnswer,ln)
    mpf_init2(@_number,ln)
    mpf_set_str(@_number,number,10)
    mpf_pow_ui(@Floatanswer,@_number,n)
    gmp_sprintf( @outtext,"%." & Str(n) & "Ff",@FloatAnswer )
    Var outtxt=Trim(outtext)
    If Instr(outtxt,".") Then outtxt= Rtrim(outtxt,"0"):outtxt=Rtrim(outtxt,".")
    Return Trim(outtxt)
End Function

Extern gmp_version Alias "__gmp_version" As Zstring Ptr
Print "GMP version ";*gmp_version
Print

var ans=power("5",(4^(3^2)))
Print Left(ans,20) + " ... "+Right(ans,20)
Print "Number of digits ";Len(ans)
Sleep
```

Output:

```
GMP version 5.1.1

62060698786608744707 ... 92256259918212890625
Number of digits  183231
```

Frink

Frink has built-in arbitrary-precision integers and all operations automatically promote to arbitrary precision when needed.

Fun Fact: The drastically faster arbitrary-precision integer operations that landed in Java 8 (for much faster multiplication, exponentiation, and toString) were taken from Frink's implementation and contributed to Java. Another fun fact is that it took employees from Java 11 years to integrate the improvements.

```
a = 5^4^3^2
as = "$a"     // Coerce to string
println["Length=" + length[as] + ", " + left[as,20] + "..." + right[as,20]]
```

This prints `Length=183231, 62060698786608744707...92256259918212890625`

FutureBasic

Translated from c

Thanks, Ken

```
/*

Uses GMP for Multiple Precision Arithmetic
Install GMP using terminal and Homebrew command, "brew install gmp"
before running this app in FutureBasic.

Homebrew available here, https://brew.sh

*/


include "NSLog.incl"

void local fn GMPoutput
  CFStringRef     sourcePath = fn StringByAppendingPathComponent( @"/tmp/", @"temp.m" )
  CFStringRef executablePath = fn StringByAppendingPathComponent( @"/tmp/", @"temp" )
  CFStringRef         gmpStr = @"#import <Foundation/Foundation.h>\n#import <gmp.h>\n¬
  int main(int argc, const char * argv[]) {\n¬
  @autoreleasepool {\n¬
  mpz_t a;\n¬
  mpz_init_set_ui(a, 5);\n¬
  mpz_pow_ui(a, a, 1 << 18);\n¬
  size_t len = mpz_sizeinbase(a, 10);\n¬
  printf(\"GMP says size is: %zu\\n\", len);\n¬
  char *s = mpz_get_str(0, 10, a);\n¬
  size_t trueLen = strlen(s);\n¬
  printf(\"  Actual size is: %zu\\n\", trueLen);\n¬
  printf(\"First & Last 20 digits: %.20s…%s\\n\", s, s + trueLen - 20);\n¬
  }\n¬
  return 0;\n¬
  }"
  fn StringWriteToURL( gmpStr, fn URLFileURLWithPath( sourcePath ), YES, NSUTF8StringEncoding, NULL )

  TaskRef task = fn TaskInit
  TaskSetExecutableURL( task, fn URLFileURLWithPath( @"usr/bin/clang" ) )
  CFArrayRef arguments = @[@"-o", executablePath, sourcePath, @"-lgmp", @"-fobjc-arc"]
  TaskSetArguments( task, arguments )
  PipeRef pipe = fn PipeInit
  TaskSetStandardInput( task, pipe )
  fn TaskLaunch( task, NULL )
  TaskWaitUntilExit( task )

  if ( fn TaskTerminationStatus( task ) == 0 )
    TaskRef executionTask = fn TaskInit
    TaskSetExecutableURL( executionTask, fn URLFileURLWithPath( executablePath ) )
    PipeRef executionPipe = fn PipeInit
    TaskSetStandardOutput( executionTask, executionPipe )
    FileHandleRef executionFileHandle = fn PipeFileHandleForReading( executionPipe )
    fn TaskLaunch( executionTask, NULL )
    TaskWaitUntilExit( executionTask )
    CFDataRef outputData = fn FileHandleReadDataToEndOfFile( executionFileHandle, NULL )
    CFStringRef outputStr  = fn StringWithData( outputData, NSUTF8StringEncoding )
    NSLog( @"%@", outputStr )
  else
    alert 1,, @"GMP required but not installed"
  end if

  fn FileManagerRemoveItemAtURL( fn URLFileURLWithPath( sourcePath     ) )
  fn FileManagerRemoveItemAtURL( fn URLFileURLWithPath( executablePath ) )
end fn

fn GMPoutput

HandleEvents
```

Output:

```
GMP says size is: 183231
  Actual size is: 183231
First & Last 20 digits: 62060698786608744707…92256259918212890625
```

Fōrmulæ

Fōrmulæ programs are not textual, visualization/edition of programs is done showing/manipulating structures but not text. Moreover, there can be multiple visual representations of the same program. Even though it is possible to have textual representation —i.e. XML, JSON— they are intended for storage and transfer purposes more than visualization and edition.

Programs in Fōrmulæ are created/edited online in its website (https://formulae.org).

In **this page (https://formulae.org/?script=examples/Arbitrary-precision_integers_%28included%29)** you can see and run the program(s) related to this task and their results. You can also change either the programs or the parameters they are called with, for experimentation, but remember that these programs were created with the main purpose of showing a clear solution of the task, and they generally lack any kind of validation.

Solution

In the following script, the result is converted to a string, in order to calculate its size, and its first/last digits.

local s ← ToString$\left(5^{\left(4^{\left(3^2\right)}\right)}\right)$

Length	Length(s)
First 20 digits	SubstringPosN(s, 1, 20)
Last 20 digits	SubstringPosToEnd(s, -20)

Length	183,231
First 20 digits	“62060698786608744707”
Last 20 digits	“92256259918212890625”

GAP

```
n:=5^(4^(3^2));;
s := String(n);;
m := Length(s);
# 183231
s{[1..20]};
# "62060698786608744707"
s{[m-19..m]};
# "92256259918212890625"
```

Go

Using math/big's `Int.Exp (https://golang.org/pkg/math/big/#Int.Exp)`.

```
package main

import (
    "fmt"
    "math/big"
)

func main() {
    x := big.NewInt(2)
    x = x.Exp(big.NewInt(3), x, nil)
    x = x.Exp(big.NewInt(4), x, nil)
    x = x.Exp(big.NewInt(5), x, nil)
    str := x.String()
    fmt.Printf("5^(4^(3^2)) has %d digits: %s ... %s\n",
        len(str),
        str[:20],
        str[len(str)-20:],
    )
}
```

Output:

```
5^(4^(3^2)) has 183231 digits: 62060698786608744707 ... 92256259918212890625
```

Golfscript

```
5 4 3 2???  # Calculate 5^(4^(3^2))
`..         # Convert to string and make two copies
20<p        # Print the first 20 digits
-20>p       # Print the last 20 digits
,p          # Print the length
```

The *p* command prints the top element from the stack, so the output of this program is just three lines:

```
"62060698786608744707"
"92256259918212890625"
183231
```

Groovy

Solution:

```
def bigNumber = 5G ** (4 ** (3 ** 2))
```

Test:

```
def bigString = bigNumber.toString()

assert bigString[0..<20] == "62060698786608744707"
assert bigString[-20..-1] == "92256259918212890625"

println bigString.size()
```

Output:

```
183231
```

Haskell

Haskell comes with built-in support for arbitrary precision integers. The type of arbitrary precision integers is `Integer`.

```
main :: IO ()
main = do
  let y = show (5 ^ 4 ^ 3 ^ 2)
  let l = length y
  putStrLn
    ("5**4**3**2 = " ++
     take 20 y ++ "..." ++ drop (l - 20) y ++ " and has " ++ show l ++ " digits")
```

Output:

```
5**4**3**2 = 62060698786608744707...92256259918212890625 and has 183231 digits
```

Hoon

```
=+  big=(pow 5 (pow 4 (pow 3 2)))
=+  digits=(lent (skip <big> |=(a/* ?:(=(a '.') & |))))
[digits (div big (pow 10 (sub digits 20))) (mod big (pow 10 20))]
```

Output:

```
[183.231 62.060.698.786.608.744.707 92.256.259.918.212.890.625]
```

As of 23 July 2016, the standard library lacks a base-10 logarithm, so the length is computed by pretty-printing the number and counting the length of the resulting string without grouping dots.

Icon and Unicon

Both Icon and Unicon have built-in support for bignums.

Note: It takes far longer to convert the result to a string than it does to do the computation itself.

```
procedure main()
    x := 5^4^3^2
    write("done with computation")
    x := string(x)
    write("5 ^ 4 ^ 3 ^ 2 has ",*x," digits")
    write("The first twenty digits are ",x[1+:20])
    write("The last twenty digits are  ",x[0-:20])
end
```

Sample run:

```
->ap
done with computation
5 ^ 4 ^ 3 ^ 2 has 183231 digits
The first twenty digits are 62060698786608744707
The last twenty digits are  92256259918212890625
->
```

J

J has built-in support for extended precision integers. See also J:Essays/Extended Precision Functions.

```
   Pow5432=: 5^4^3^2x
   Pow5432=: ^/ 5 4 3 2x                  NB. alternate J solution
   # ": Pow5432                          NB. number of digits
183231
   20 ({. , '...' , -@[ {. ]) ": Pow5432    NB. 20 first & 20 last digits
```

```
62060698786608744707...92256259918212890625
```

Java

Java library's `BigInteger` class provides support for arbitrary precision integers.

```
import java.math.BigInteger;

class IntegerPower {
    public static void main(String[] args) {
        BigInteger power =
BigInteger.valueOf(5).pow(BigInteger.valueOf(4).pow(BigInteger.valueOf(3).pow(2).intValueExact()).intValu
eExact());
        String str = power.toString();
        int len = str.length();
        System.out.printf("5**4**3**2 = %s...%s and has %d digits%n",
                str.substring(0, 20), str.substring(len - 20), len);
    }
}
```

Output:

```
5**4**3**2 = 62060698786608744707...92256259918212890625 and has 183231 digits
```

JavaScript

The ECMA-262 JavaScript standard now defines a BigInt [3] (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt) type for for abitrary precision integers.

BigInt is already implemented by Chrome and Firefox, but not yet by Explorer or Safari.

```
>>> const y = (5n**4n**3n**2n).toString();
>>> console.log(`5**4**3**2 = ${y.slice(0,20)}...${y.slice(-20)} and has ${y.length} digits`);
5**4**3**2 = 62060698786608744707...92256259918212890625 and has 183231 digits
```

jq

Works with gojq, the Go implementation of jq

There is a BigInt.jq library for jq, but gojq, the Go implementation of jq, supports unbounded-precision integer arithmetic, so the output shown below is that produced by gojq.

```
def power($b): . as $in | reduce range(0;$b) as $i (1; . * $in);

5|power(4|power(3|power(2))) | tostring
| .[:20], .[-20:], length
```

Output:

```
62060698786608744707
92256259918212890625
183231
```

Julia

Julia includes built-in support for arbitrary-precision arithmetic using the GMP (http://gmplib.org/) (integer) and GNU MPFR (http://www.mpfr.org/) (floating-point) libraries, wrapped by the built-in `BigInt` and `BigFloat` types, respectively.

```
julia> @elapsed bigstr = string(BigInt(5)^4^3^2)
0.017507363

julia> length(bigstr)
183231

julia> bigstr[1:20]
"62060698786608744707"

julia> bigstr[end-19:end]
"92256259918212890625"
```

Klong

```
   n::$5^4^3^2
   .p("5^4^3^2 = ",(20#n),"...",((-20)#n)," and has ",($#n)," digits")
```

Output:

5^4^3^2 = 62060698786608744707...92256259918212890625 and has 183231 digits

Kotlin

Translation of: Java

```
import java.math.BigInteger

fun main(args: Array<String>) {
    val x = BigInteger.valueOf(5).pow(Math.pow(4.0, 3.0 * 3.0).toInt())
    val y = x.toString()
    val len = y.length
    println("5^4^3^2 = ${y.substring(0, 20)}...${y.substring(len - 20)} and has $len digits")
}
```

Output:

```
5^4^3^2 = 62060698786608744707...92256259918212890625 and has 183231 digits
```

Lambdatalk

Just using Javascript's BigInt

```
{def N {BI.** 5 {BI.** 4 {BI.** 3 2}}}} -> N
length: {def L {W.length {N}}}         -> L = 183231
20 first digits: {W.slice 0 20 {N}}    -> 62060698786608744707
20 last digits:  {W.slice -20 {L} {N}} -> 92256259918212890625
```

langur

Arbitrary precision is native in langur.

```
val xs = string(5 ^ 4 ^ 3 ^ 2)

writeln len(xs), " digits"

if len(xs) > 39 and s2s(xs, 1..20) == "62060698786608744707" and
    s2s(xs, -20 .. -1) == "92256259918212890625" {

    writeln "SUCCESS"
}
```

Output:

```
183231 digits
SUCCESS
```

Lasso

Interestingly, we have to define our own method for integer powers.

```
define integer->pow(factor::integer) => {
    #factor <= 0
        ? return 0

    local(retVal) = 1

    loop(#factor) => { #retVal *= self }

    return #retVal
}

local(bigint) = string(5->pow(4->pow(3->pow(2))))
#bigint->sub(1,20) + ` ... ` + #bigint->sub(#bigint->size - 19)
"\n"
`Number of digits: ` + #bigint->size
```

Output:

```
62060698786608744707 ... 92256259918212890625
Number of digits: 183231
```

Liberty BASIC

Interestingly this takes a LONG time in LB.

It takes however only seconds in RunBASIC, which is written by the same author, shares most of LB's syntax, and is based on later Smalltalk implementation.

Note the brackets are needed to enforce the desired order of exponentiating.

```
a$ = str$( 5^(4^(3^2)))
print len( a$)
print left$( a$, 20); "......"; right$( a$, 20)
```

Output:

```
183231
62060698786608744707......92256259918212890625
```

Lua

Pure/native off-the-shelf Lua does not include support for arbitrary-precision arithmetic. However, there are a number of optional libraries that do, including several from an author of the language itself - one of which this example will use. (citing the "..*may* be used instead" allowance)

```
bc = require("bc")
-- since 5$=5^4$, and IEEE754 can handle 4$, this would be sufficient:
-- n = bc.pow(bc.new(5), bc.new(4^3^2))
-- but for this task:
n = bc.pow(bc.new(5), bc.pow(bc.new(4), bc.pow(bc.new(3), bc.new(2))))
s = n:tostring()
print(string.format("%s...%s (%d digits)", s:sub(1,20), s:sub(-20,-1), #s))
```

Output:

```
62060698786608744707...92256259918212890625 (183231 digits)
```

Maple

Maple supports large integer arithmetic natively.

```
> n := 5^(4^(3^2)):
> length( n ); # number of digits
                                 183231

> s := convert( n, 'string' ):
> s[ 1 .. 20 ], s[ -20 .. -1 ]; # extract first and last twenty digits
             "62060698786608744707", "92256259918212890625"
```

In the Maple graphical user interface it is also possible to set things up so that only (say) the first and last 20 digits of a large integer are displayed explicitly. This is done as follows.

```
> interface( elisiondigitsbefore = 20, elisiondigitsafter = 20 ):
> 5^(4^(3^2)):
           62060698786608744707[...183191 digits...]92256259918212890625
```

Mathematica / Wolfram Language

Mathematica can handle arbitrary precision integers on almost any size without further declarations. To view only the first and last twenty digits:

```
s:=ToString[5^4^3^2];
Print[StringTake[s,20]<>"..."<>StringTake[s,-20]<>" ("<>ToString@StringLength@s<>" digits)"];
```

Output:

```
62060698786608744707...92256259918212890625 (183231 digits)
```

MATLAB

Using the Variable Precision Integer (http://www.mathworks.com/matlabcentral/fileexchange/22725-variable-precision-integer-arithmetic) library this task is accomplished thusly:

```
>> answer = vpi(5)^(vpi(4)^(vpi(3)^vpi(2)));
>> numDigits = order(answer) + 1

numDigits =

      183231

>> [sprintf('%d',leadingdigit(answer,20)) '...' sprintf('%d',trailingdigit(answer,20))]
%First and Last 20 Digits

ans =

62060698786608744707...92256259918212890625
```

Maxima

```
block([s, n], s: string(5^4^3^2), n: slength(s), print(substring(s, 1, 21), "...", substring(s, n - 19)),
n);
/* 62060698786608744707...92256259918212890625
183231 */
```

Nanoquery

Integer values are arbitrary-precision by default in Nanoquery.

```
value = str(5^(4^(3^2)))

first20 = value.substring(0,20)
```

```
last20 = value.substring(len(value) - 20)

println "The first twenty digits are " + first20
println "The last twenty digits are " + last20

if (first20 = "62060698786608744707") && (last20 = "92256259918212890625")
    println "\nThese digits are correct.\n"
end

println "The result is " + len(str(value)) + " digits long"
```

Output:

```
The first twenty digits are 62060698786608744707
The last twenty digits are 92256259918212890625

These digits are correct.

The result is 183231 digits long
```

Nemerle

Translation of: C#

```
using System.Console;
using System.Numerics;
using System.Numerics.BigInteger;

module BigInt
{
    Main() : void
    {
        def n = Pow(5, Pow(4, Pow(3, 2) :> int) :> int).ToString();
        def len = n.Length;
        def first20 = n.Substring(0, 20);
        def last20 = n.Substring(len - 20, 20);

        assert (first20 == "62060698786608744707", "High order digits are incorrect");
        assert (last20 == "92256259918212890625", "Low order digits are incorrect");
        assert (len == 183231, "Result contains wrong number of digits");

        WriteLine("Result: {0} ... {1}", first20, last20);
        WriteLine($"Length of result: $len digits");
    }
}
```

Output:

```
Result: 62060698786608744707 ... 92256259918212890625
Length of result: 183231 digits
```

=

NewLisp

```
;;; No built-in big integer exponentiation
(define (exp-big x n)
    (setq x (bigint x))
    (let (y 1L)
        (if (= n 0)
            1L
            (while (> n 1)
                (if (odd? n)
                    (setq y (* x y)))
                (setq x (* x x) n (/ n 2)))
        (* x y))))
;
;;; task
(define (test)
    (local (res)
        ;   drop the "L" at the end
        (setq res (0 (- (length res) 1) (string (exp-big 5 (exp-big 4 (exp-big 3 2))))))
        (println "The result has:  " (length res) " digits")
        (println "First 20 digits: " (0 20 res))
        (println "Last 20 digits:  " (-20 20 res))))
```

Output:

```
The result has:  183231 digits
First 20 digits: 62060698786608744707
Last 20 digits:  92256259918212890625
```

NetRexx

Using Java's BigInteger Class

```
/* NetRexx */

options replace format comments java crossref savelog symbols

import java.math.BigInteger

numeric digits 30 -- needed to report the run-time

nanoFactor = 10 ** 9

t1 = System.nanoTime
x = BigInteger.valueOf(5)
x = x.pow(BigInteger.valueOf(4).pow(BigInteger.valueOf(3).pow(2).intValue()).intValue())
n = Rexx(x.toString)
t2 = System.nanoTime
td = t2 - t1
say "Run time in seconds:" td / nanoFactor
say

check = "62060698786608744707...92256259918212890625"
sample = n.left(20)"..."n.right(20)

say "Expected result:" check
say "  Actual result:" sample
say "         digits:" n.length
say

if check = sample
then
  say "Result confirmed"
else
  say "Result does not satisfy test"
```

```
return
```

Output:

```
Run time in seconds: 6.696671

Expected result: 62060698786608744707...92256259918212890625
  Actual result: 62060698786608744707...92256259918212890625
         digits: 183231

Result confirmed
```

Using Java's BigDecimal Class

```
/* NetRexx */

options replace format comments java crossref savelog symbols

import java.math.BigDecimal

numeric digits 30 -- needed to report the run-time

nanoFactor = 10 ** 9

t1 = System.nanoTime
x = BigDecimal.valueOf(5)
x = x.pow(BigDecimal.valueOf(4).pow(BigDecimal.valueOf(3).pow(2).intValue()).intValue())
n = Rexx(x.toString)
t2 = System.nanoTime
td = t2 - t1
say "Run time in seconds:" td / nanoFactor
say

check = "62060698786608744707...92256259918212890625"
sample = n.left(20)"..."n.right(20)

say "Expected result:" check
say "  Actual result:" sample
say "         digits:" n.length
say

if check = sample
then
  say "Result confirmed"
else
  say "Result does not satisfy test"

return
```

Output:

```
Run time in seconds: 7.103424

Expected result: 62060698786608744707...92256259918212890625
  Actual result: 62060698786608744707...92256259918212890625
         digits: 183231

Result confirmed
```

Using NetRexx Built-In Math

Like Rexx, NetRexx comes with built-in support for numbers that can be manually set to very large values of precision. Compared to the two methods shown above however, the performance is extremely poor.

Note

Translation of: REXX

```
/* NetRexx */

options replace format comments java crossref savelog symbols

/* precision must be set manually */

numeric digits 190000

nanoFactor = 10 ** 9

t1 = System.nanoTime
n = 5 ** (4  ** (3 ** 2))
t2 = System.nanoTime
td = t2 - t1
say "Run time in seconds:" td / nanoFactor
say

check = "62060698786608744707...92256259918212890625"
sample = n.left(20)"..."n.right(20)

say "Expected result:" check
say "  Actual result:" sample
say "         digits:" n.length
say

if check = sample
then
  say "Result confirmed"
else
  say "Result does not satisfy test"
```

Output:

```
Run time in seconds: 719.660995

Expected result: 62060698786608744707...92256259918212890625
  Actual result: 62060698786608744707...92256259918212890625
         digits: 183231

Result confirmed
```

Nim

Library: bigints

```
import bigints

var x = 5.pow 4.pow 3.pow 2
var s = $x
```

```
echo s[0..19]
echo s[s.high - 19 .. s.high]
echo s.len
```

Output:

```
62060698786608744707
92256259918212890625
183231
```

OCaml

```
open Num
open Str
open String

let () =
  let answer = (Int 5) **/ (Int 4) **/ (Int 3) **/ (Int 2) in
  let answer_string = string_of_num answer in
  Printf.printf "has %d digits: %s ... %s\n"
                (length answer_string)
                (first_chars answer_string 20)
                (last_chars answer_string 20)
```

A more readable program can be obtained using Delimited Overloading (http://forge.ocamlcore.org/projects/pa-do/):

```
let () =
  let answer = Num.(5**4**3**2) in
  let s = Num.(to_string answer) in
  Printf.printf "has %d digits: %s ... %s\n"
    (String.length s) (Str.first_chars s 20) (Str.last_chars s 20)
```

Output:

```
has 183231 digits: 62060698786608744707 ... 92256259918212890625
```

Oforth

Oforth handles arbitrary precision integers :

```
import: mapping

5 4 3 2 pow pow pow >string dup left( 20 ) . dup right( 20 ) . size .
```

Output:

```
62060698786608744707 92256259918212890625 183231
```

Ol

```
(define x (expt 5 (expt 4 (expt 3 2))))
(print
   (div x (expt 10 (- (log 10 x) 20)))
   "..."
   (mod x (expt 10 20)))
(print "totally digits: " (log 10 x))
```

Output:

```
62060698786608744707...92256259918212890625
totally digits: 183231
```

ooRexx

Translation of: REXX

```
--REXX program to show arbitrary precision integers.
numeric digits 200000
check = '62060698786608744707...92256259918212890625'

start = .datetime~new
n = 5 ** (4 ** (3**2))
time = .datetime~new - start
say 'elapsed time for the calculation:' time
say
sampl = left(n, 20)"..."right(n, 20)

say ' check:' check
say 'Sample:' sampl
say 'digits:' length(n)
say

if check=sampl then say 'passed!'
               else say 'failed!'
```

Output:

```
prompt$ rexx rexx-arbitrary.rexx
elapsed time for the calculation: 00:00:45.373140

 check: 62060698786608744707...92256259918212890625
Sample: 62060698786608744707...92256259918212890625
digits: 183231

passed!
```

Oz

```
declare
  Pow5432 = {Pow 5 {Pow 4 {Pow 3 2}}}
  S = {Int.toString Pow5432}
  Len = {Length S}
in
```

```
  {System.showInfo
   {List.take S 20}#"..."#
   {List.drop S Len-20}#" ("#Len#" Digits)"}
```

Output:

```
62060698786608744707...92256259918212890625 (183231 Digits)
```

PARI/GP

PARI/GP natively supports integers of arbitrary size, so one could just use `N=5^4^3^2`. But this would be foolish (using a lot of unneeded memory) if the task is to get just the number of, and the first and last twenty digits. The number of and the leading digits are given as 1 + the integer part, resp. 10^(fractional part + offset), of the logarithm to base 10 (not as log(N) with N=A^B, but as B*log(A); one needs at least 20 correct decimals of the log, on 64 bit machines the default precision is 39 digits, but on 32 bit architecture one should set `default(realprecision,30)` to be on the safe side). To get the trailing digits, one would use modular exponentiation which is also built-in and very efficient even for extremely huge exponents:

```
num_first_last_digits(a=5,b=4^3^2,n=20)={ my(L = b*log(a)/log(10), m=Mod(a,10^n)^b);
    [L\1+1, 10^frac(L)\10^(1-n), lift(m)] \\ where x\y = floor(x/y) but more efficient
}
print("Length, first and last 20 digits of 5^4^3^2: ", num_first_last_digits()) \\ uses default values
a=5, b=4^3^2, n=20
```

Output:

```
Length, first and last 20 digits of 5^4^3^2: [183231, 62060698786608744707, 92256259918212890625]
```

If an integer is already given, then `logint(N,10)+1` is the most efficient way to get its number of digits.

An alternate but much slower method for counting decimal digits is `#Str(n)`. Note that `sizedigit` is not exact—in particular, it may be off by one (thus the function below).

```
digits(x)={
    my(s=sizedigit(x)-1);
    if(x<10^s,s,s+1)
};

N=5^(4^(3^2));
[precision(N*1.,20), Mod(N,10^20), digits(N)]
```

Output:

```
[6.20606987866087447074832055728 E183230, Mod(92256259918212890625, 100000000000000000000), 183231]
```

Pascal

Works with: Free_Pascal
Library: math
Library: GMP

FreePascal comes with a header unit for gmp. Starting from the C program, this is a Pascal version:

```
program GMP_Demo;

uses
  math, gmp;

var
  a:   mpz_t;
  out: pchar;
  len: longint;
  i:   longint;

begin
  mpz_init_set_ui(a, 5);
  mpz_pow_ui(a, a, 4 ** (3 ** 2));
  len := mpz_sizeinbase(a, 10);
  writeln('GMP says size is: ', len);
  out := mpz_get_str(NIL, 10, a);
  writeln('Actual size is:   ', length(out));
  write('Digits: ');
  for i := 0 to 19 do
    write(out[i]);
  write ('...');
  for i := len - 20 to len do
    write(out[i]);
  writeln;
end.
```

Output:

```
GMP says size is: 183231
Actual size is:   183231
Digits: 62060698786608744707...92256259918212890625
```

Perl

Perl's `Math::BigInt` core module handles big integers:

```
use Math::BigInt;
my $x = Math::BigInt->new('5') ** Math::BigInt->new('4') ** Math::BigInt->new('3') ** Math::BigInt-
>new('2');
my $y = "$x";
printf("5**4**3**2 = %s...%s and has %i digits\n", substr($y,0,20), substr($y,-20), length($y));
```

You can enable "transparent" big integer support by enabling the `bigint` pragma:

```
use bigint;
my $x = 5**4**3**2;
my $y = "$x";
printf("5**4**3**2 = %s...%s and has %i digits\n", substr($y,0,20), substr($y,-20), length($y));
```

Math::BigInt is very slow. Perl 5.10 was about 120 times slower than Ruby 1.9.2 (on one computer); Perl used more than one minute, but Ruby used less than one second.

Output:

```
$ time perl transparent-bigint.pl
5**4**3**2 = 62060698786608744707...92256259918212890625 and has 183231 digits
    1m4.28s real     1m4.30s user     0m0.00s system
```

Phix

Library: Phix/basics
Library: Phix/mpfr

```
with javascript_semantics
include mpfr.e
atom t0 = time()
mpz res = mpz_init()
mpz_ui_pow_ui(res,5,power(4,power(3,2)))
string s = mpz_get_short_str(res),
       e = elapsed(time()-t0)
printf(1,"5^4^3^2 = %s (%s)\n", {s,e})
```

Output:

```
5^4^3^2 = 62060698786608744707...92256259918212890625 (183,231 digits) (0.1s)
```

PHP

PHP has two separate arbitrary-precision integer services.

The first is the BC library.[4] (http://us3.php.net/manual/en/book.bc.php) It represents the integers as strings, so may not be very efficient. The advantage is that it is more likely to be included with PHP.

```
<?php
$y = bcpow('5', bcpow('4', bcpow('3', '2')));
printf("5**4**3**2 = %s...%s and has %d digits\n", substr($y,0,20), substr($y,-20), strlen($y));
?>
```

Output:

```
5**4**3**2 = 62060698786608744707...92256259918212890625 and has 183231 digits
```

The second is the GMP library.[5] (http://us3.php.net/manual/en/book.gmp.php) It represents the integers as an opaque type, so may be faster. However, it is less likely to be compiled into your version of PHP (it isn't compiled into mine).

Picat

```
main =>
    X = to_string(5**4**3**2),
    Y = len(X),
    println("Result: "),
    print("Number of digits: "), println(Y),
    println("First 20 digits: " ++ X[1..20]),
    println("Last 20 digits: " ++ X[Y-19..Y]).
```

PicoLisp

```
(let L (chop (** 5 (** 4 (** 3 2))))
   (prinl (head 20 L) "..." (tail 20 L))
   (length L) )
```

Output:

```
62060698786608744707...92256259918212890625
-> 183231
```

Pike

```
> string res = (string)pow(5,pow(4,pow(3,2)));
> res[..19] == "62060698786608744707";
Result: 1
> res[<19..] == "92256259918212890625";
Result: 1
> sizeof(result);
Result: 183231
```

PowerShell

```
#  Perform calculation
$BigNumber = [BigInt]::Pow( 5, [BigInt]::Pow( 4, [BigInt]::Pow( 3, 2 ) ) )

#  Display first and last 20 digits
$BigNumberString = [string]$BigNumber
$BigNumberString.Substring( 0, 20 ) + "..." + $BigNumberString.Substring( $BigNumberString.Length - 20,
20 )

#  Display number of digits
$BigNumberString.Length
```

Output:

```
62060698786608744707...92256259918212890625
183231
```

Processing

```
import java.math.BigInteger;

// Variable definitions
BigInteger _5, _4, powResult;
_5 = BigInteger.valueOf(5);
_4 = BigInteger.valueOf(4);

//calculations
powResult          = _5.pow(_4.pow(9).intValueExact());
String powStr      = powResult.toString();
int    powLen      = powStr.length();
String powStrStart = powStr.substring(0, 20);
String powStrEnd   = powStr.substring(powLen - 20);

//output
System.out.printf("5**4**3**2 = %s...%s and has %d digits%n", powStrStart, powStrEnd, powLen);
```

Output:

```
5**4**3**2 = 62060698786608744707...92256259918212890625 and has 183231 digits
```

Prolog

Works with: SWI-Prolog version 6.6

```
task(Length) :-
    N is 5^4^3^2,

    number_codes(N, Codes),
    append(`62060698786608744707`, _,  Codes),
    append(_, `92256259918212890625`, Codes),

    length(Codes, Length).
```

Query like so:

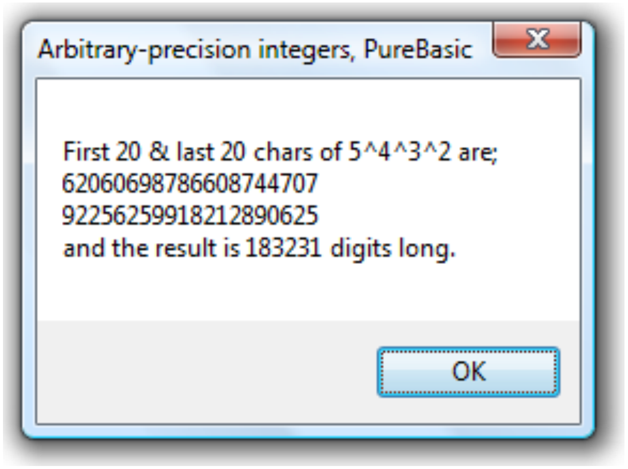
```
?- task(N).
N = 183231 ;
false.
```

PureBasic

PureBasic has in its current version (today 4.50) no internal support for large numbers, but there are several free libraries for this.

Using Decimal.pbi (http://www.purebasic.fr/english/viewtopic.php?p=309763#p309763), e.g. the same included library as in Long multiplication#PureBasic, this task is solved as below.

```
IncludeFile "Decimal.pbi"

;- Declare the variables that will be used
Define.Decimal *a
Define n, L$, R$, out$, digits.s

;- 4^3^2 is withing 32 bit range, so normal procedures can be used
n=Pow(4,Pow(3,2))

;- 5^n is larger then 31^2, so the same library call as in the "Long multiplication" task is used
*a=PowerDecimal(IntegerToDecimal(5),IntegerToDecimal(n))

;- Convert the large number into a string & present the results
out$=DecimalToString(*a)
L$ = Left(out$,20)
R$ = Right(out$,20)
digits=Str(Len(out$))
out$="First 20 & last 20 chars of 5^4^3^2 are;"+#CRLF$+L$+#CRLF$+R$+#CRLF$
out$+"and the result is "+digits+" digits long."

MessageRequester("Arbitrary-precision integers, PureBasic",out$)
```



First 20 & last 20 chars of 5^4^3^2 are;
62060698786608744707
92256259918212890625
and the result is 183231 digits long.

Python

Python comes with built-in support for arbitrary precision integers. The type of arbitrary precision integers is `long (http://docs.python.org/library/stdtypes.html#typesnumeric)` in Python 2.x (overflowing operations on `int`'s are automatically converted into `long`'s), and `int (http://docs.python.org/3.1/library/stdtypes.html#typesnumeric)` in Python 3.x.

```
>>> y = str( 5**4**3**2 )
>>> print ("5**4**3**2 = %s...%s and has %i digits" % (y[:20], y[-20:], len(y)))
5**4**3**2 = 62060698786608744707...92256259918212890625 and has 183231 digits
```

Quackery

As a dialogue in the Quackery shell. (REPL)

```
 > quackery

Welcome to Quackery.
```

```
Enter "leave" to leave the shell.

/O> 5 4 3 2 ** ** **
... number$ dup 20 split swap echo$
... say "..." -20 split echo$ drop cr
... size echo say " digits" cr
...
62060698786608744707...92256259918212890625
183231 digits

Stack empty.

/O>
```

R

R does not come with built-in support for arbitrary precision integers, but it can be implemented with the GMP library.

```
library(gmp)
large <- pow.bigz(5, pow.bigz(4, pow.bigz(3, 2)))
largestr <- as.character(large)
cat("first 20 digits:", substr(largestr, 1, 20), "\n",
    "last 20 digits:", substr(largestr, nchar(largestr) - 19, nchar(largestr)), "\n",
    "number of digits: ", nchar(largestr), "\n")
```

Output:

```
first 20 digits: 62060698786608744707
 last 20 digits: 92256259918212890625
 number of digits:  183231
```

Racket

```
#lang racket

(define answer (number->string (foldr expt 1 '(5 4 3 2))))
(define len (string-length answer))

(printf "Got ~a digits~n" len)
(printf "~a ... ~a~n"
        (substring answer 0 20)
        (substring answer (- len 20) len))
```

Output:

```
Got 183231 digits
62060698786608744707 ... 92256259918212890625
```

Raku

(formerly Perl 6)

Works with: Rakudo version 2022.07

```
given [**] 5, 4, 3, 2 {
  use Test;
  ok /^ 62060698786608744707 <digit>* 92256259918212890625 $/,
     '5**4**3**2 has expected first and last twenty digits';
  printf 'This number has %d digits', .chars;
}
```

Output:

```
ok 1 - 5**4**3**2 has expected first and last twenty digits
This number has 183231 digits
```

REXX

REXX comes with built-in support for fixed precision integers that can be manually set to a large value of precision (digits).
Most REXXes have a practical limit of around eight million bytes, but that is mostly an underlying limitation of addressing virtual storage.

manual setting of decimal digits

Note: both REXX versions (below) don't work with:

- PC/REXX
- Personal REXX

as those REXX versions have a practical maximum of around **3,700** or less for **numeric digits** (officially, it's **4K**).

The **3,700** limit is based on the setting of RXISA, program size, and the amount of storage used by REXX variables.

Both (below) REXX programs have been tested with:

- PC/REXX (can't execute correctly)
- Personal REXX (can't execute correctly)
- Regina REXX
- R4
- ROO
- ooRexx (tested by Walter Pachl)

```
/*REXX program calculates and demonstrates  arbitrary precision numbers (using powers). */
```

```
numeric digits 200000                               /*two hundred thousand decimal digits. */

    # = 5 ** (4 ** (3 ** 2) )                       /*calculate multiple exponentiations.  */

true=62060698786608744707...92256259918212890625 /*what answer is supposed to look like.*/
rexx= left(#, 20)'...'right(#, 20)                  /*the left and right 20 decimal digits.*/

say  '  true:'    true                              /*show what the  "true"  answer is.    */
say  '  REXX:'    rexx                              /*  "    "    "    REXX       "    "    */
say  'digits:'    length(#)                         /*  "    "    "   length  of answer is. */
say
if true == rexx   then say 'passed!'                /*either it passed,  ···               */
                  else say 'failed!'                /*    or it didn't.                    */
                                                    /*stick a fork in it,  we're all done. */
```

output:

```
 check: 62060698786608744707...92256259918212890625
sample: 62060698786608744707...92256259918212890625
digits: 183231

passed!
```

automatic setting of decimal digits

```
/*REXX program calculates and demonstrates  arbitrary precision numbers (using powers). */
numeric digits 5                                    /*just use enough digits for 1st time. */

                  #=5** (4** (3** 2) )              /*calculate multiple exponentiations.  */

parse var  #  'E'  pow  .                           /*POW   might be null,  so   N  is OK. */

if pow\==''  then do                                /*general case:   POW  might be < zero.*/
                  numeric digits  abs(pow) + 9      /*recalculate with more decimal digits.*/
                  #=5** (4** (3** 2) )              /*calculate multiple exponentiations.  */
                  end                               /* [↑]  calculation is the real McCoy. */

true=62060698786608744707...92256259918212890625 /*what answer is supposed to look like.*/
rexx= left(#, 20)'...'right(#, 20)                  /*the left and right 20 decimal digits.*/

say  '  true:'    true                              /*show what the  "true"  answer is.    */
say  '  REXX:'    rexx                              /*  "    "    "    REXX       "    "    */
say  'digits:'    length(#)                         /*  "    "    "   length  of answer is. */
say
if true == rexx   then say 'passed!'                /*either it passed,  ···               */
                  else say 'failed!'                /*    or it didn't.                    */
                                                    /*stick a fork in it,  we're all done. */
```

output is the same as the 1st REXX version.

Ruby

Ruby comes with built-in support for arbitrary precision integers.

```
y = ( 5**4**3**2 ).to_s
puts "5**4**3**2 = #{y[0..19]}...#{y[-20..-1]} and has #{y.length} digits"
```

Output:

```
5**4**3**2 = 62060698786608744707...92256259918212890625 and has 183231 digits
```

Run BASIC

```
x$ = str$( 5^(4^(3^2)))
print "Length:";len( x$)
print left$( x$, 20); "......"; right$( x$, 20)
```

Output:

```
Length:183231
62060698786608744707......92256259918212890625
```

Rust

This is accomplished via the `num` crate. This used to be part of the standard library, but was relegated to an external crate when Rust hit 1.0. It is still owned and maintained by members of the Rust core team and is the de-facto library for numerical generics and arbitrary precision arithmetic.

```
extern crate num;
use num::bigint::BigUint;
use num::FromPrimitive;
use num::pow::pow;

fn main() {
    let big = BigUint::from_u8(5).unwrap();
    let answer_as_string = format!("{}", pow(big,pow(4,pow(3,2))));

      // The rest is output formatting.
    let first_twenty: String = answer_as_string.chars().take(20).collect();
    let last_twenty_reversed: Vec<char> = answer_as_string.chars().rev().take(20).collect();
    let last_twenty: String = last_twenty_reversed.into_iter().rev().collect();
    println!("Number of digits: {}", answer_as_string.len());
    println!("First and last digits: {:?}..{:?}", first_twenty, last_twenty);
}
```

Output:

```
Number of digits: 183231
First and last digits: "62060698786608744707".."92256259918212890625"
```

Sather

```
class MAIN is
  main is
    r:INTI;
    p1 ::= "62060698786608744707";
    p2 ::= "92256259918212890625";

    -- computing 5^(4^(3^2)), it could be written
    -- also e.g. (5.inti)^((4.inti)^((3.inti)^(2.inti)))
    r  := (3.pow(2)).inti;
    r  := (4.inti).pow(r);
    r  := (5.inti).pow(r);

    sr ::= r.str; -- string rappr. of the number
    if sr.head(p1.size) = p1
       and sr.tail(p2.size) = p2 then
         #OUT + "result is ok..\n";
    else
         #OUT + "oops\n";
    end;
    #OUT + "# of digits: " + sr.size + "\n";
  end;
end;
```

Output:

```
result is ok..
# of digits: 183231
```

Scala

Scala does not come with support for arbitrary precision integers powered to arbitrary precision integers, except if performed on a module. It can use arbitrary precision integers in other ways, including powering them to 32-bits integers.

```
scala> BigInt(5) modPow (BigInt(4) pow (BigInt(3) pow 2).toInt, BigInt(10) pow 20)
res21: scala.math.BigInt = 92256259918212890625

scala> (BigInt(5) pow (BigInt(4) pow (BigInt(3) pow 2).toInt).toInt).toString
res22: String = 6206069878660874470748320557284679309194219265199117173177383244
78446890420544620839553285931321349485035253770303663683982841794590287939217907
89641300156281305613064874236198955114921296922487632406742326659692228562195387
46210423235340883954495598715281862895110697243759768434501295076608139350684049
01191160699929926568099301259938271975526587719565309995276438998093283175080241
55833224724855977970015112594128926594587205662421861723789001208275184293399910
13912158886504596553858675842231519094813553261073608575593794241686443569888058
92732524316323249492420512640962691673104618378381545202638771401061171968052873
21414945463925055899307933774904078819911387324217976311238875802878310483037255
33789567769926391314746986316354035923183981697660495275234703657750678459919...
scala> res22 take 20
res23: String = 62060698786608744707

scala> res22 length
res24: Int = 183231

scala>
```

Scheme

R^{4}RS (http://people.csail.mit.edu/jaffer/r4rs_8.html#SEC52) and R^{5}RS (http://schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.2.3) encourage, and R^{6}RS (http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-6.html#node_sec_3.4) requires, that exact integers be of arbitrary precision.

```
(define x (expt 5 (expt 4 (expt 3 2))))
(define y (number->string x))
(define l (string-length y))
(display (string-append "5**4**3**2 = " (substring y 0 20) "..." (substring y (- l 20) l) " and has "
(number->string l) " digits"))
(newline)
```

Output:

```
5**4**3**2 = 62060698786608744707...92256259918212890625 and has 183231 digits
```

Seed7

```
$ include "seed7_05.s7i";
  include "bigint.s7i";

const proc: main is func
  local
    var bigInteger: fiveToThePowerOf262144 is 5_ ** 4 ** 3 ** 2;
    var string: numberAsString is str(fiveToThePowerOf262144);
  begin
    writeln("5**4**3**2 = " <& numberAsString[..20] <&
            "..." <& numberAsString[length(numberAsString) - 19 ..]);
    writeln("decimal digits: " <& length(numberAsString));
  end func;
```

Output:

```
5**4**3**2 = 62060698786608744707...92256259918212890625
decimal digits: 183231
```

Sidef

```
var x = 5**(4**(3**2))
var y = x.to_s
printf("5**4**3**2 = %s...%s and has %i digits\n", y.first(20), y.last(20), y.len)
```

Output:

```
5**4**3**2 = 62060698786608744707...92256259918212890625 and has 183231 digits
```

SIMPOL

This example is **incomplete**. Number of digits in result not given. Please ensure that it meets all task requirements and remove this message.

SIMPOL supports arbitrary precision integers powered to arbitrary precision integers. This is the only integer data type in SIMPOL. SIMPOL supports conversion from its integer data type to other formats when calling external library functions.

```
constant FIRST20 "62060698786608744707"
constant LAST20  "92256259918212890625"

function main()
  integer i
  string s, s2

  i = .ipower(5, .ipower(4, .ipower(3, 2)))
  s2 = .tostr(i, 10)
  if .lstr(s2, 20) == FIRST20 and .rstr(s2, 20) == LAST20
    s = "Success! The integer matches both the first 20 and the last 20 digits. There are " +
.tostr(.len(s2), 10) + " digits in the result.{d}{a}"
  else
    s = ""
    if .lstr(s2, 20) != FIRST20
      s = "Failure! The first 20 digits are: " + .lstr(s2, 20) + " but they should be: " + FIRST20 + "{d}
{a}"
    end if
    if .rstr(s2, 20) != LAST20
      s = s + "Failure! The first 20 digits are: " + .lstr(s2, 20) + " but they should be: " + LAST20 + "
{d}{a}"
    end if
  end if
end function s
```

Smalltalk

This code in Squeak Smalltalk [1] returns a string containing the first 20 digits, last 20 digits and length of the result.

A very simple approach:

```
|num|
num := (5 raisedTo: (4 raisedTo: (3 raisedTo: 2))) asString.
Transcript
   show: (num first: 20), '...', (num last: 20); cr;
   show: 'digits: ', num size asString.
```

On a Transcript window:

```
62060698786608744707...92256259918212890625
digits: 183231
```

And a more advanced one:

```
|num numstr|
num := (2 to: 5) fold: [:exp :base| base raisedTo: exp].
numstr := num asString.
```

```
'<1s>...<2s>  digits:<3p>'
   expandMacrosWith: (numstr first: 20)
   with: (numstr last: 20)
   with: numstr size.
```

Output:

```
'62060698786608744707...92256259918212890625  digits: 183231'
```

Note 1) should work in all Smalltalk dialects; tried in Smalltalk/X and VisualWorks.

SPL

```
t = #.str(5^(4^(3^2)))
n = #.size(t)
#.output(n," digits")
#.output(#.mid(t,1,20),"...",#.mid(t,n-19,20))
```

Output:

```
183231 digits
62060698786608744707...92256259918212890625
```

Standard ML

```
let
  val answer = IntInf.pow (5, IntInf.toInt (IntInf.pow (4, IntInf.toInt (IntInf.pow (3, 2)))))
  val s = IntInf.toString answer
  val len = size s
in
  print ("has " ^ Int.toString len ^ " digits: " ^
         substring (s, 0, 20) ^ " ... " ^
         substring (s, len-20, 20) ^ "\n")
end;
```

it took too long to run

mLite

mLite does not have a logarithm function so one was constructed (see fun log10)

```
fun
    ntol (0, x) = if len x < 1 then [0] else x
       | (n, x) = ntol (n div 10, (n mod 10) :: x)
       | n      = ntol (n, [])
and
    powers_of_10 9 = 1000000000
               | 8 = 100000000
               | 7 = 10000000
               | 6 = 1000000
               | 5 = 100000
               | 4 = 10000
```

```
                | 3 = 1000
                | 2 = 100
                | 1 = 10
                | 0 = 1
and
    size (c, 0) = c
       | (c, n > 9999999999) = size (c + 10, trunc (n / 10000000000))
       | (c, n)              = size (c +  1, trunc (n / 10))
       | n                   = size (     0, trunc (n / 10))
and
    makeVisible L = map (fn x = if int x then chr (x + 48) else x) L
and
    log10 (n, 0, x) = ston ` implode ` makeVisible ` rev x
        | (n, c, x) =
            let val n' = n^10;
              val size_n' = size n'
            in
              log10 (n' / powers_of_10 size_n', c - 1, size_n' :: x)
            end
        | (n, c) =
            let
              val size_n = size n
            in
              log10 (n / 10^size_n, c, #"." :: rev (ntol size_n) @ [])
            end
;
val fourThreeTwo = 4^3^2;
val fiveFourThreeTwo = 5^fourThreeTwo;

val digitCount = trunc (log10(5,6) * fourThreeTwo + 0.5);
print "Count  = "; println digitCount;

val end20 = fiveFourThreeTwo mod (10^20);
print "End 20 = "; println end20;

val top20 = fiveFourThreeTwo div (10^(digitCount - 20));
print "Top 20 = "; println top20;
```

Output

```
Count = 183231
End 20 = 92256259918212890625
Top 20 = 62060698786608744707
```

Took 1 hour and 9 minutes to run (AMD A6, Windows 10)

Stata

Stata does not have builtin support for arbitrary-precision integers. However, since version 16 Stata has builtin support for Python (https://www.stata.com/new-in-stata/python-integration/), so arbitrary-precision integers are readily available from a Python prompt within Stata.

Tcl

Tcl supports arbitrary precision integers (and an exponentiation operator) from 8.5 onwards.

Works with: Tcl version 8.5

```
set bigValue [expr {5**4**3**2}]
```

```
puts "5**4**3**2 has [string length $bigValue] digits"
if {[string match "62060698786608744707*92256259918212890625" $bigValue]} {
    puts "Value starts with 62060698786608744707, ends with 92256259918212890625"
} else {
    puts "Value does not match 62060698786608744707...92256259918212890625"
}
```

Output:

```
5**4**3**2 has 183231 digits
Value starts with 62060698786608744707, ends with 92256259918212890625
```

Transd

```
#lang transd

MainModule: {
    _start:(λ locals: a BigLong(5) ss StringStream() s ""
        (textout to: ss (pow a (pow 4 (pow 3 2))))
        (= s (str ss))
        (with len (size s)
            (lout "The number of digits is: " len)
            (lout (sub s 0 20) " ... " (sub s (- len 20))))
    )
}
```

Output:

```
The number of digits is: 183231
62060698786608744707 ... 92256259918212890625
```

TXR

```
@(bind (f20 l20 ndig)
       @(let* ((str (tostring (expt 5 4 3 2)))
               (len (length str)))
          (list [str :..20] [str -20..:] len)))
@(bind f20 "62060698786608744707")
@(bind l20 "92256259918212890625")
@(output)
@f20...@l20
ndigits=@ndig
@(end)
```

Output:

```
62060698786608744707...92256259918212890625
ndigits=183231
```

Ursa

The Ursa standard library provides the module `unbounded_int` which contains the definition of the `unbounded_int` type. In Cygnus/X Ursa, `unbounded_int` is essentially a wrapper for `java.math.BigInteger`

Usage

```
import "unbounded_int"
decl unbounded_int x
x.set ((x.valueof 5).pow ((x.valueof 4).pow ((x.valueof 3).pow 2)))

decl string first last xstr
set xstr (string x)

# get the first twenty digits
decl int i
for (set i 0) (< i 20) (inc i)
    set first (+ first xstr<i>)
end for

# get the last twenty digits
for (set i (- (size xstr) 20)) (< i (size xstr)) (inc i)
    set last (+ last xstr<i>)
end for

out "the first and last digits of 5^(4^(3^2)) are " first "..." console
out last " (the result was " (size xstr) " digits long)" endl endl console

if (and (and (= first "62060698786608744707") (= last "92256259918212890625")) (= (size xstr) 183231))
    out "(pass)" endl console
else
    out "FAIL" endl console
end if
```

Output

Output:

```
the first and last digits of 5^(4^(3^2)) are 62060698786608744707...92256259918212890625 (the result was
183231 digits long)

(pass)
```

Ursala

There are no infix arithmetic operators in the language, but there is a `power` function in the `bcd` library, which is part of the standard distribution from the home site.

There is no distinction between ordinary and arbitrary precision integers, but the binary converted decimal representation used here is more efficient than the usual binary representation in calculations that would otherwise be dominated by the conversion to decimal output.

```
#import std
#import nat
```

```
#import bcd

#show+

main = <.@ixtPX take/$20; ^|T/~& '...'--@x,'length: '--@h+ %nP+ length@t>@h %vP power=> <5_,4_,3_,2_>
```

With this calculation taking about a day to run, correct results are attainable but not performant.

```
62060698786608744707...92256259918212890625
length: 183231
```

Visual Basic .NET

Translation of: C#
Library: System.Numerics

Addressing the issue of the **BigInteger.Pow()** function having the exponent value limited to **Int32.MaxValue** (2147483647), here are a couple of alternative implementations using a **BigInteger** for the exponent.

```
Imports System.Console
Imports BI = System.Numerics.BigInteger

Module Module1

    Dim Implems() As String = {"Built-In", "Recursive", "Iterative"},
        powers() As Integer = {5, 4, 3, 2}

    Function intPowR(val As BI, exp As BI) As BI
        If exp = 0 Then Return 1
        Dim ne As BI, vs As BI = val * val
        If exp.IsEven Then ne = exp >> 1 : Return If (ne > 1, intPowR(vs, ne), vs)
        ne = (exp - 1) >> 1 : Return If (ne > 1, intPowR(vs, ne), vs) * val
    End Function

    Function intPowI(val As BI, exp As BI) As BI
        intPowI = 1 : While (exp > 0) : If Not exp.IsEven Then intPowI *= val
            val *= val : exp >>= 1 : End While
    End Function

    Sub DoOne(title As String, p() As Integer)
        Dim st As DateTime = DateTime.Now, res As BI, resStr As String
        Select Case (Array.IndexOf(Implems, title))
            Case 0 : res = BI.Pow(p(0), CInt(BI.Pow(p(1), CInt(BI.Pow(p(2), p(3))))))
            Case 1 : res = intPowR(p(0), intPowR(p(1), intPowR(p(2), p(3))))
            Case Else : res = intPowI(p(0), intPowI(p(1), intPowI(p(2), p(3))))
        End Select : resStr = res.ToString()
        Dim et As TimeSpan = DateTime.Now - st
        Debug.Assert(resStr.Length = 183231)
        Debug.Assert(resStr.StartsWith("62060698786608744707"))
        Debug.Assert(resStr.EndsWith("92256259918212890625"))
        WriteLine("n = {0}", String.Join("^", powers))
        WriteLine("n = {0}...{1}", resStr.Substring(0, 20),  resStr.Substring(resStr.Length - 20, 20))
        WriteLine("n digits = {0}", resStr.Length)
        WriteLine("{0} elasped: {1} milliseconds." & vblf, title, et.TotalMilliseconds)
    End Sub

    Sub Main()
        For Each itm As String in Implems : DoOne(itm, powers) : Next
        If Debugger.IsAttached Then Console.ReadKey()
    End Sub
End Module
```

Output:

```
n = 5^4^3^2
n = 62060698786608744707...92256259918212890625
n digits = 183231
Built-In elasped: 2487.4002 milliseconds.

n = 5^4^3^2
n = 62060698786608744707...92256259918212890625
n digits = 183231
Recursive elasped: 2413.0434 milliseconds.

n = 5^4^3^2
n = 62060698786608744707...92256259918212890625
n digits = 183231
Iterative elasped: 2412.5477 milliseconds.
```

Remarks: Not much difference in execution times for three methods. But the exponents are relatively small. If one does need to evaluate an exponent greater than **Int32.MaxValue**, the execution time will be measured in hours.

V (Vlang)

```
import math.big
import math

fn main() {

    mut x := u32(math.pow(3,2))
    x = u32(math.pow(4,x))
    mut y := big.integer_from_int(5)
    y = y.pow(x)
    str := y.str()
    println("5^(4^(3^2)) has $str.len digits: ${str[..20]} ... ${str[str.len-20..]}")
}
```

Output:

```
5^(4^(3^2)) has 183231 digits: 62060698786608744707 ... 92256259918212890625
```

Wren

Library: Wren-fmt
Library: Wren-big

```
import "./fmt" for Fmt
import "./big" for BigInt

var p = BigInt.three.pow(BigInt.two)
p = BigInt.four.pow(p)
p = BigInt.five.pow(p)
var s = p.toString
Fmt.print("5 ^ 4 ^ 3 ^ 2 has $,d digits.\n", s.count)
```

```
System.print("The first twenty are    : %(s[0..19])")
System.print("and the last twenty are : %(s[-20..-1])")
```

Output:

```
5 ^ 4 ^ 3 ^ 2 has 183,231 digits.

The first twenty are    : 62060698786608744707
and the last twenty are : 92256259918212890625
```

Zig

```
const std = @import("std");
const bigint = std.math.big.int.Managed;

pub fn main() !void {
    var a = try bigint.initSet(std.heap.c_allocator, 5);
    try a.pow(&a, try std.math.powi(u32, 4, try std.math.powi(u32, 3, 2)));
    defer a.deinit();

    var as = try a.toString(std.heap.c_allocator, 10, .lower);
    defer std.heap.c_allocator.free(as);

    std.debug.print("{s}...{s}\n", .{ as[0..20], as[as.len - 20 ..] });
    std.debug.print("{} digits\n", .{as.len});
}
```

Output:

```
62060698786608744707...92256259918212890625
183231 digits
```

zkl

Using the GNU big num library:

```
var BN=Import("zklBigNum");
n:=BN(5).pow(BN(4).pow(BN(3).pow(2)));
s:=n.toString();
"%,d".fmt(s.len()).println();
println(s[0,20],"...",s[-20,*]);
```

Output:

```
183,231
62060698786608744707...92256259918212890625
```

Retrieved from "https://rosettacode.org/wiki/Arbitrary-precision_integers_(included)?oldid=365254"

-