UNIT-II

Finding The Square Root Of A Number

Given a number 'm' devise an algorithm to compute its square root.

square root of a number 4 is 2

9 is 3

from the above examples we can say that square root n, of another number m must satisfy the equation.

$$n * n = m \qquad (1)$$

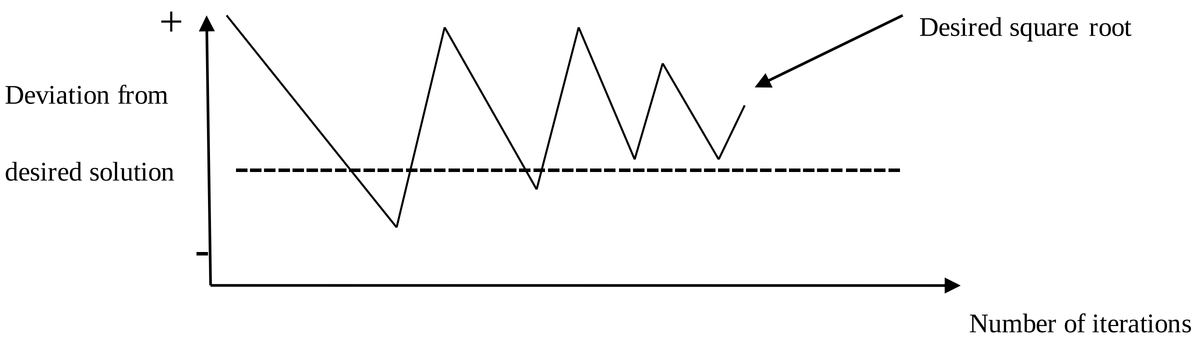
to find the square root of a given number the following systematic approach could be adopted.

1. Choose a number **n** less than the number **m** want the square root of.

2. Square **n** and if it is greater than **m** decrease **n** by 1 and repeat step **2**, else go to step **3**.

3. when the square of our guess at the square root is less than m we can start increasing n by 0.1 until we again compute a guess greater than m.

At this point, we start decreasing our guess by 0.01 and so on until we have computed the square root we require the desired accuracy.



Through this algorithm , the number of iterations it requires depends critically on how good our initial guess is.

At this stage we should do some mathematical analysis and do the algorithm with a good strategy.

Algorithm Description:

1. Establish 'm' the number whose square root is required and the termination condition error 'e'.

2. Set the initial guess g2 to m/2.

3. Repeatedly

(a) let g1 assume the role of g2,

(b) generate a better estimate g2 of the square root using the averaging formula,

until the absolute difference between g1 and g2 is less than error 'e'.

4. Return the estimated square root g2.

<u>**Pascal Implementation**</u>:

```
function sqroot(m, error:real):real;

var g1{previous estimate of square root},

    g2{current estimate of square root}: real;

begin{estimates square root of number m}

g2: m/2;

repeat

g1=g2;

g2=(g1+m/g1)/2;

until abs(g1-g2)<error;

sqroot:= g2;

end
```

<u>Notes:</u>

This design is implemented with feedback principle. that is we keep making corrections to our estimate in way dependent on how much the previous solution deviated from the desired result.

The Smallest Divisor Of An Integer

Given an integer n devise an algorithm that will find its smallest exact divisor other than one.

All even numbers are divisible by 2. it follows that if the number is not divisible by 2 it will not divisible by 4,6,8,10,..... And So if the number we are testing is odd, we should consider odd numbers as potential smallest divisor candidates.

The overall algorithm cam be written as:

```
1. If the number 'n' is even, then the smallest divisor is 2
else
 (a) Compute the square root r of n,
(b) while no exact divisor less than square root of n do
   (b.1) test next divisor in sequence 3, 5, 7, ...
```

Algorithm Description

```
1. Establish 'n' the integer whose smallest divisor is required.
2. if 'n' is not odd then return 2 as the smallest divisor
else
  (a) Compute 'r' the square root of 'n',
  (b) initialize divisor 'd' to 3,
  (c) while not an exact divisor and square root limit not reached do
     (c.1)  generate next number in odd sequence d,
(d) if current odd value d is an exact divisor, then divide it as the
   exact divisor of 'n'
```

```
else
    return 1 as the smallest divisor of 'n'.
```

Pascal Implementation

```
function sdivisor(n:integer): integer;
var d {current divisor and member of odd sequence},
      r { integer less than or equal to square root of n}: integer;
begin{finds the smallest exact divisor of an integer n, returns 1 if n prime}
      if not odd(n) then
            sdivisor:=2
else
   begin{terminates search for smallest divisor at sqrt (n)}
         r:=trunc(sqrt(n));
         d:=3;
     while(n mod d< >0) and (d < r) do
    d:=d+2;
     if n mod d =0 then
      sdivisor:=d
else
      sdivisor:=1       end     end
```

The Greatest Common Divisor of Two Integers

Given two positive non-zero integers n and m design an algorithm for finding their greatest common divisor.

The gcd of two integers is the largest integer that will divide exactly into the two integers with no remainder. We can build up to the common divisor of the two integers by considering an exact divisor of a single integer. An exact divisor of a number is another smaller number that divides the original number up into set of equal parts.

The gcd of two numbers cannot be bigger than the smaller of the two numbers.

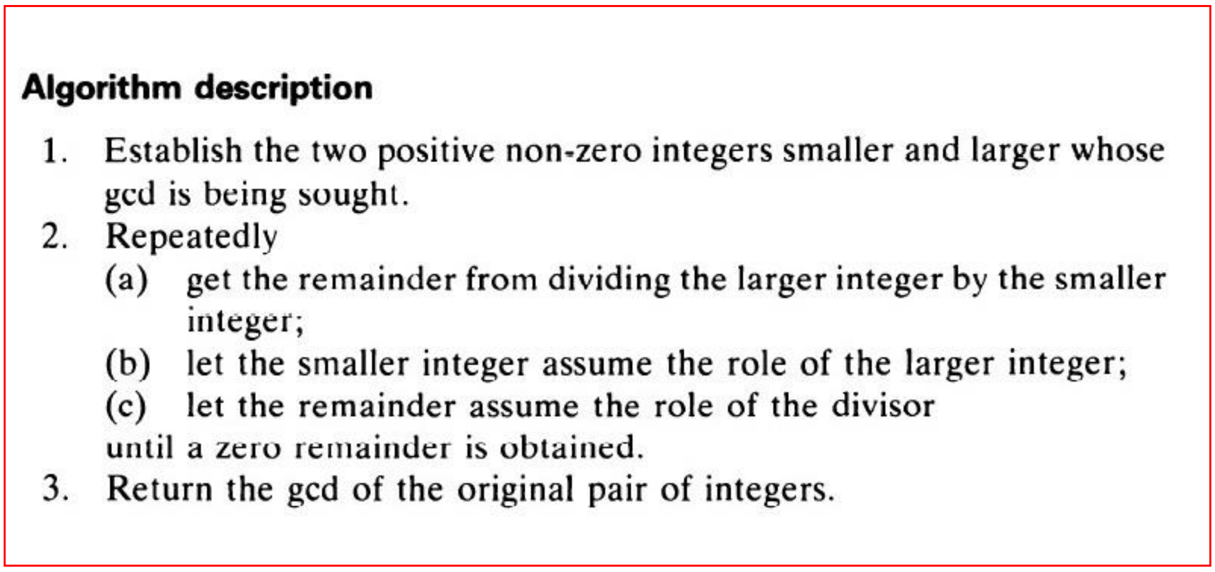
Next point is how to continue when the smaller of the two numbers n and m is not their gcd

The basic strategy for computing the gcd of two numbers:

```
1. Divide the larger of two numbers by the smaller number.
2. if the smaller number exactly divides into larger number
      then the smaller number is the gvd
   else
      remove from the larger number the part common to the smaller number and repeat the whole procedure with the new pair of numbers.
```

Terminating the gcd mechanism can be detected by there being no remainder after mod function

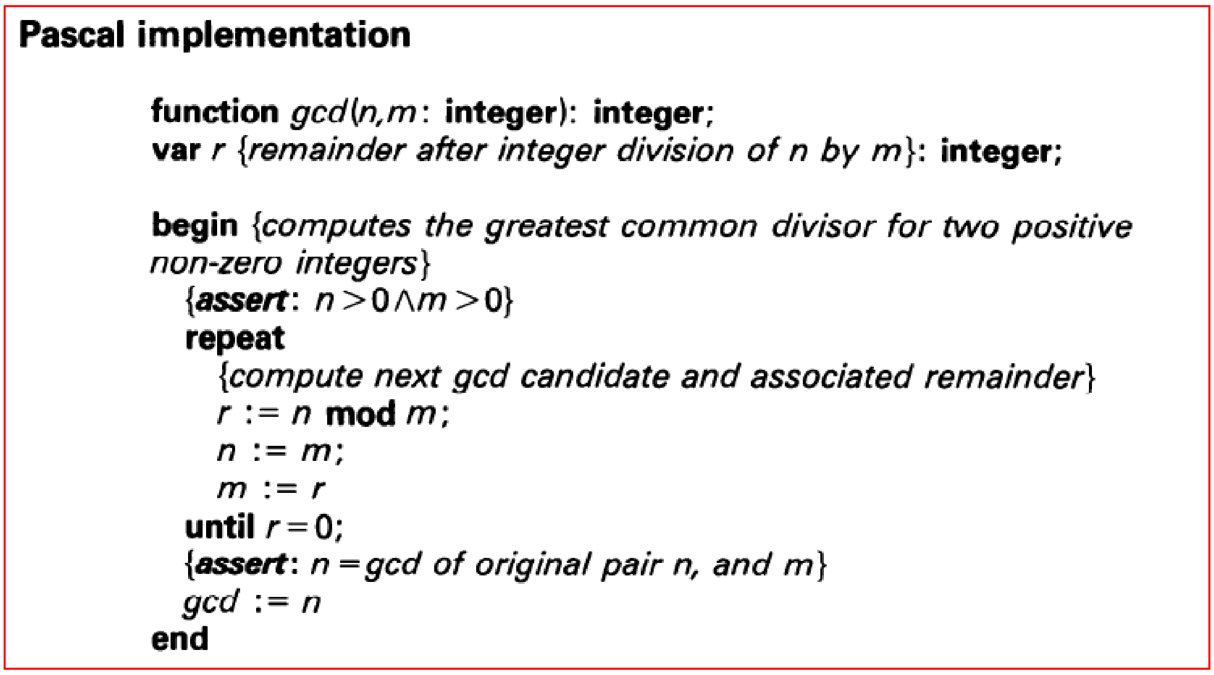
```
                r:= n mod m
if r is zero then m is the gcd, else
while gcd not found do
    a. get remainder by dividing larger integer by the smaller integer;
    b. let the smaller integer assume the role of the larger integer;
    c. let the remainder assume the role of the smaller integer.
    repeat the same process until  zero remainder.
```



Algorithm description

1. Establish the two positive non-zero integers smaller and larger whose gcd is being sought.
2. Repeatedly
   (a) get the remainder from dividing the larger integer by the smaller integer;
   (b) let the smaller integer assume the role of the larger integer;
   (c) let the remainder assume the role of the divisor

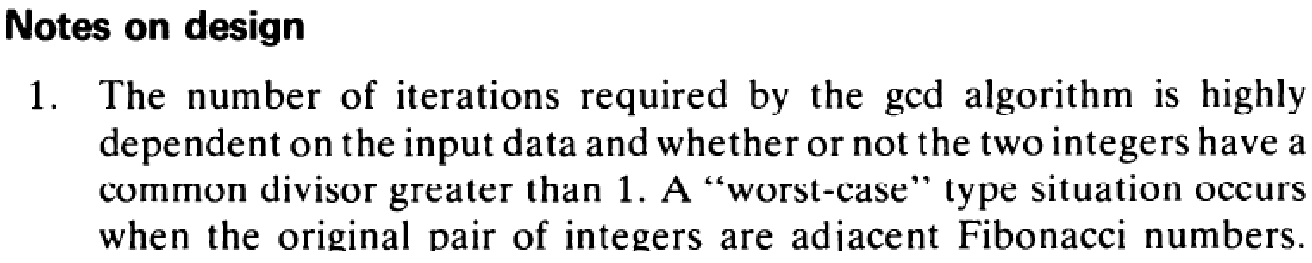
   until a zero remainder is obtained.
3. Return the gcd of the original pair of integers.



Pascal implementation

```
function gcd(n,m: integer): integer;
var r {remainder after integer division of n by m}: integer;

begin {computes the greatest common divisor for two positive
non-zero integers}
  {assert: n>0∧m>0}
  repeat
    {compute next gcd candidate and associated remainder}
    r := n mod m;
    n := m;
    m := r
  until r=0;
  {assert: n=gcd of original pair n, and m}
  gcd := n
end
```



Notes on design

1. The number of iterations required by the gcd algorithm is highly dependent on the input data and whether or not the two integers have a common divisor greater than 1. A "worst-case" type situation occurs when the original pair of integers are adjacent Fibonacci numbers.

Applications:

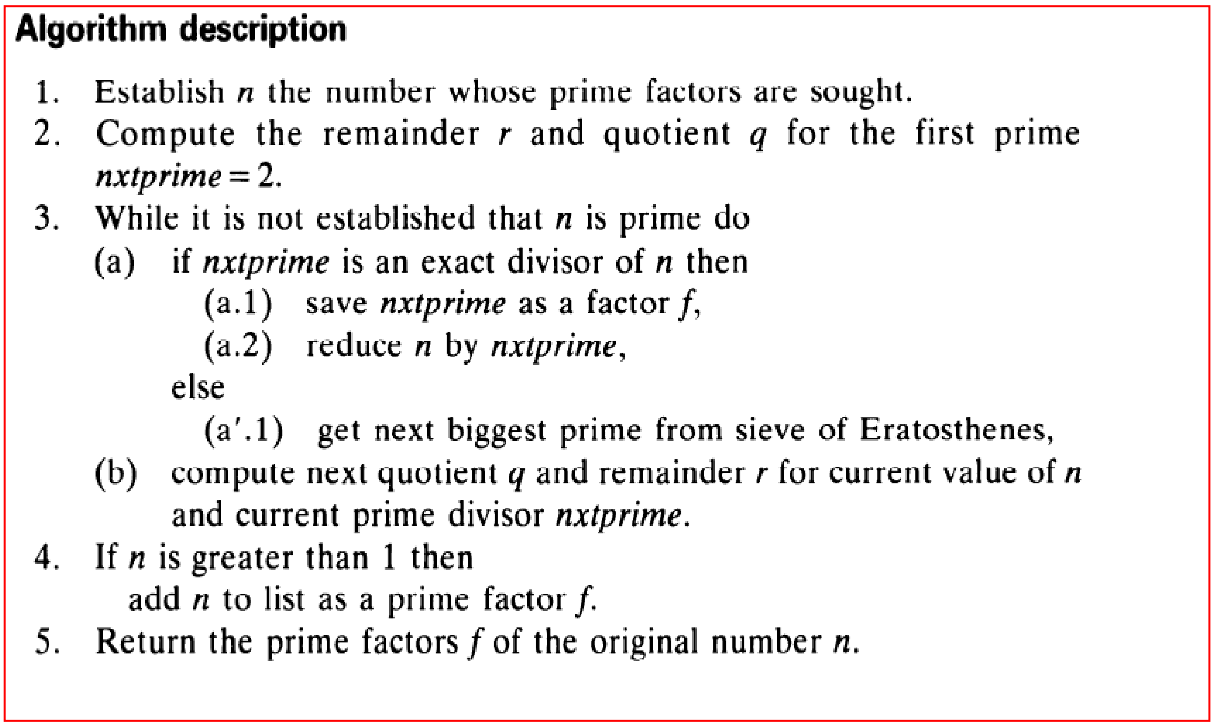
Reducing a fraction to its lowest terms.

Computing the Prime Factors of an Integer

Every integer can be expressed as a product of prime numbers. Design an algorithm to compute all the prime factors of an integer n.

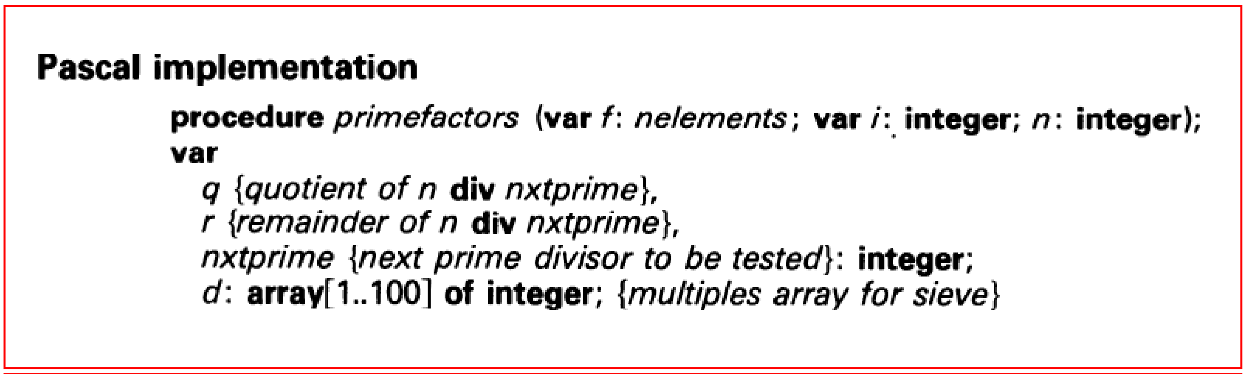
All the prime factors of 'n' must be less than or equal to √n. This suggests that to produce a list of primes up to √n before going through the process of trying to establish the prime factors of n. But this may end up in computing a lot more primes that are needed as divisors.

A better and more economical strategy is therefore to only compute prime divisors as needed.



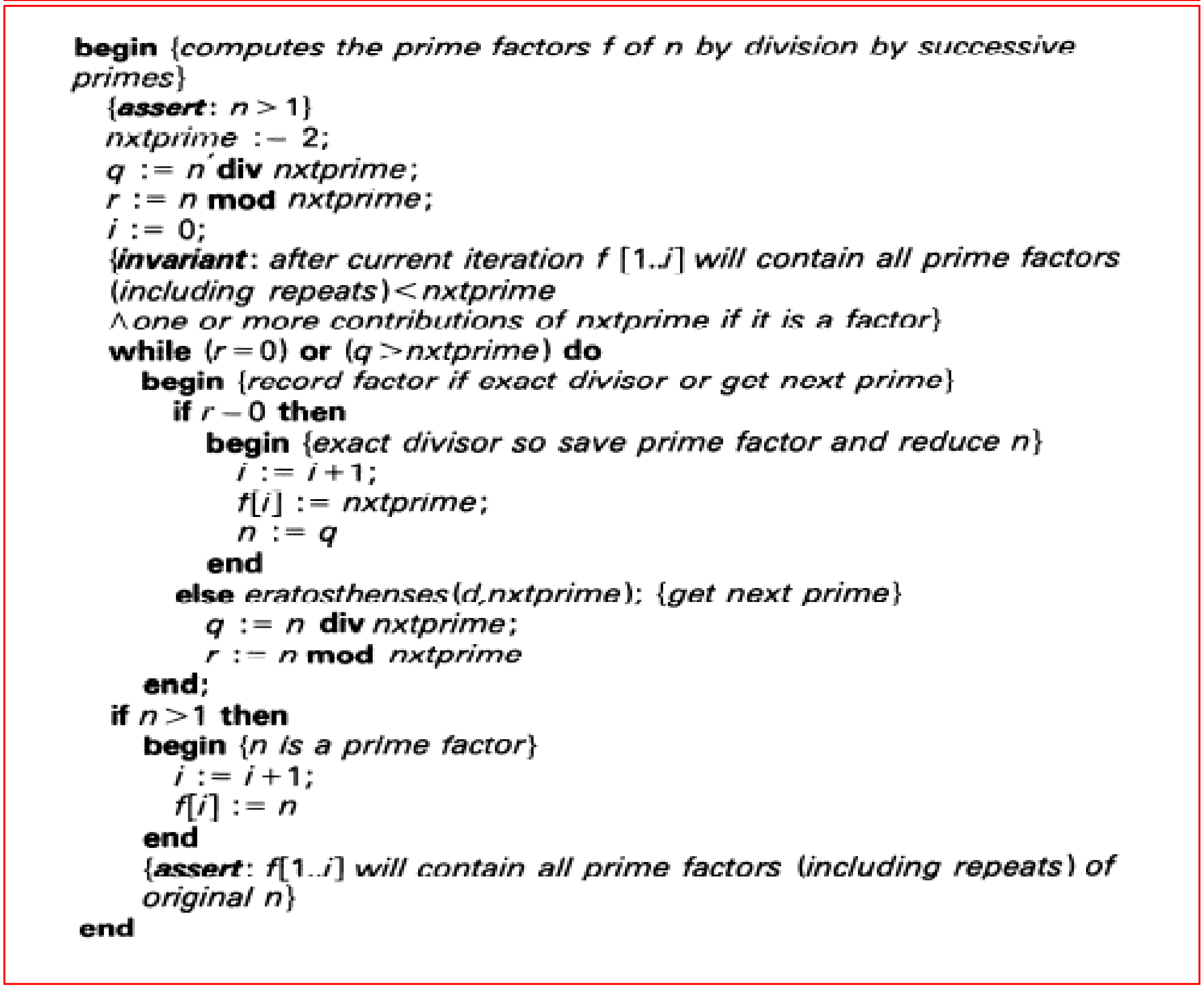
Algorithm description

```
1. Establish $n$ the number whose prime factors are sought.
2. Compute the remainder $r$ and quotient $q$ for the first prime
   $nxtprime = 2$.
3. While it is not established that $n$ is prime do
   (a) if $nxtprime$ is an exact divisor of $n$ then
         (a.1) save $nxtprime$ as a factor $f$,
         (a.2) reduce $n$ by $nxtprime$,
       else
         (a'.1) get next biggest prime from sieve of Eratosthenes,
   (b) compute next quotient $q$ and remainder $r$ for current value of $n$
       and current prime divisor $nxtprime$.
4. If $n$ is greater than 1 then
     add $n$ to list as a prime factor $f$.
5. Return the prime factors $f$ of the original number $n$.
```



Pascal implementation

```
procedure primefactors (var f: nelements; var i: integer; n: integer);
var
  q {quotient of n div nxtprime},
  r {remainder of n div nxtprime},
  nxtprime {next prime divisor to be tested}: integer;
  d: array[1..100] of integer; {multiples array for sieve}
```



```
begin {computes the prime factors f of n by division by successive
primes}
  {assert: n > 1}
  nxtprime := 2;
  q := n div nxtprime;
  r := n mod nxtprime;
  i := 0;
  {invariant: after current iteration f [1..i] will contain all prime factors
  (including repeats) < nxtprime
  ∧ one or more contributions of nxtprime if it is a factor}
  while (r = 0) or (q > nxtprime) do
    begin {record factor if exact divisor or get next prime}
      if r = 0 then
        begin {exact divisor so save prime factor and reduce n}
          i := i + 1;
          f[i] := nxtprime;
          n := q
        end
      else eratosthenses(d,nxtprime); {get next prime}
        q := n div nxtprime;
        r := n mod nxtprime
    end;
  if n > 1 then
    begin {n is a prime factor}
      i := i + 1;
      f[i] := n
    end
    {assert: f[1..i] will contain all prime factors (including repeats) of
    original n}
 end
```

Applications:

Factoring numbers with up to six digits.

Generation of Pseudo Random numbers

Use the linear congruential method to generate a uniform set of pseudo random numbers.

Algorithm Development

Random numbers generators are frequently used in computing science for among other things, testing and analysing the behaviour of algorithms.

A sequence of random numbers should exhibit the following behaviour

i) The sequence should appear as though each member had occurred by chance.

ii) each number should have a specified probability of falling within a given range.

The implementation of the linear congruential method is very straight forward. Successive members of the linear congruential sequence {x} are generated using the expression

$$x_{n+1} = (ax_n+b) \bmod m \qquad \text{for } n>=0$$

The parameters a, b and m are referred as the multiplier, increment and modulus respectively.

All parameters should be integers greater than or equal to zero and m should be greater than x0, a and b.

Parameter X_0

The parameter X_0 can chosen arbitrarily within the range $0<= X_0 <m$.

Parameter m

The value of m should be greater than or equal to the length of the random sequence required.

It must be possible to do the computation (a * x+ b) mod m without roundoff.

Parameter a

The choice of a depends on the choice of m. If m is a power of 2 then a should satisfy the condition

$$a \bmod 8 = 5$$

a should be larger than $\sqrt{m}$ and less than $m-\sqrt{m}$

Parameter b

The constant b should be odd and not a multiple of 5.

Notes on design.

The linear congruential method is a simple, efficient and practical method for generating pseudo random numbers.

The theoretical basis for the choice of parameters involves a highly sophisticated analysis.

Raising A Number To A Larger Power

Given some integer x, compute the value of X^n where n is a positive integer considerably greater than 1.

Evaluating the expression

$p=X^n$

In the power generation process, one of two conditions apply

(a) where we have an odd power it must have been generated from the power that is one less (e.g $X^{23} = X^{22} * X$)

(b) where we have an even power, it can be computed from a power that is half its size (e.g $X^{22} = X^{112} * X^{11}$)

These last two statements capture the essence of the algorithm. This means that our algorithm will need to be in two parts.

1. a part that determines the multiplication strategy and

2. a second part that actually does the power evaluation.

To map out the multiplication procedure, we can start with the power required and determine whether it is even or odd.

The next step is to integer-divide the current power by 2 and repeat the even/odd determination procedure.

if the current d array element is zero then

(a) we simply square the current power product p

else

(a$'$) we square the current product p and multiply by X to generate an odd power.

The algorithm will need to terminate when n is reduced to zero

Algorithm description

1. Establish *n*, the integer power, and *x* the integer to be raised to the power *n*.

2. Initialize the power sequence and product variable for the zero power case.
3. While the power *n* is greater than zero do
   (a) if next most significant binary digit of power *n* is one then
      (a.1) multiply accumulated product by current power sequence value;
   (b) reduce power *n* by a factor of two using integer division;
   (c) get next power sequence member by multiplying current value by itself.
4. Return *x* raised to the power *n*.

Pascal implementation

```
function power(x, n: integer): integer;
var product {current accumulated product, eventually contains result},
  psequence {current power sequence value}: integer;

begin {computes x raised to the power n using doubling strategy}
  {assert: x>0∧n>=0∧n0=n}
  product := 1;
  psequence := x;
  {invariant: product * (psequence)↑n=x↑n0∧n>=0}
  while n>0 do
    begin {incorporate power for next most significant binary digit if
    not zero}
      if (n mod 2)=1 then
        product := product * psequence;
        n := n div 2;
        psequence := psequence * psequence
    end;
    {assert: product=x↑n0}
  power := product
end
```

Applications.

Encryption and testing for non-priority of numbers.

<u>Computing the nth Fibonacci number</u>

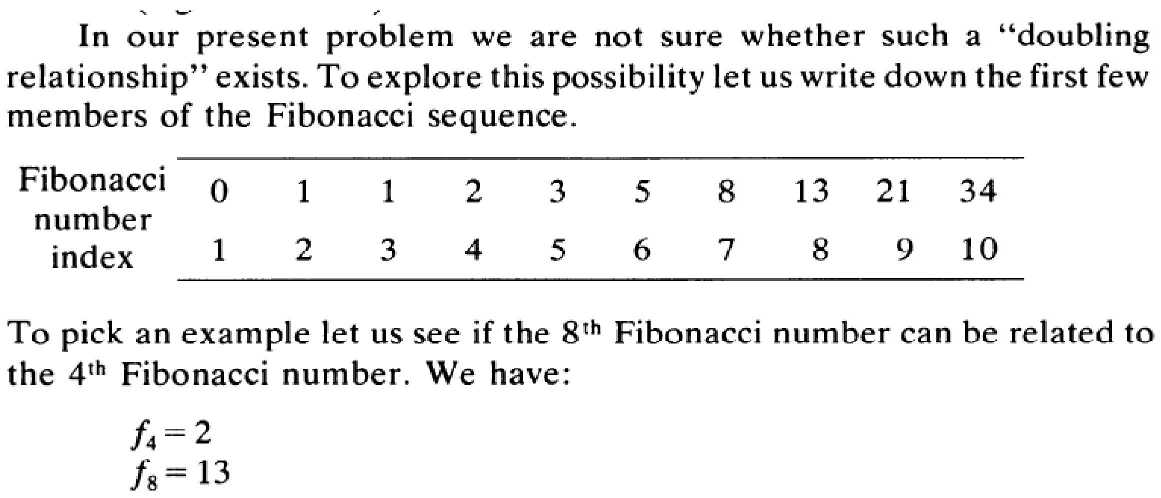
Given a number n generate the nth number of the Fibonacci sequence

The nth member of the fibonacci sequence f_n is defined recursively as follows

$f_1 = 0$

$f_2 = 1$

$f_n = f_{n-1} + f_{n-2}$ for $n>2$



In our present problem we are not sure whether such a "doubling relationship" exists. To explore this possibility let us write down the first few members of the Fibonacci sequence.

Fibonacci number	0	1	1	2	3	5	8	13	21	34
index	1	2	3	4	5	6	7	8	9	10

To pick an example let us see if the 8th Fibonacci number can be related to the 4th Fibonacci number. We have:

$f_4 = 2$
$f_8 = 13$

The relationship is

$$f_8 = 6 \times f_4 + 1$$

we couldn't find any doubling sequence in the above expression.

The next thing is try some combination of two fibonacci sequence to generate the doubled fibonacci number.

Let us try to write f_8 in terms of f_4 and f_5

$$f_4 = 2$$
$$f_5 = 3$$
$$f_6 = f_5 + f_4$$
$$f_7 = f_6 + f_5 = (f_5 + f_4) + f_5 \quad \text{(substituting for } f_6\text{)}$$
$$f_8 = f_7 + f_6 = [(f_5 + f_4) + f_5] + f_5 + f_4$$

collecting terms we get:

$$f_8 = 3f_5 + 2f_4 = 3\times 3 + 2\times 2 = 13$$

Since $f_5 = 3$ and $f_4 = 2$ the last expression suggests:

$$f_8 = f_5^2 + f_4^2$$

Checking with f_{10}, f_5 and f_6 to see if the formula generalizes, we get

$$f_{10} = f_6^2 + f_5^2 = 5^2 + 3^2 = 34$$

Without going into a detailed proof, we might be satisfied that in general:

$$f_{2n} = f_{n+1}^2 + f_n^2$$

To generate the f_{2n+1} fibonacci number we need f_{2n} and f_{2n-1}

$$f_{2n+1} = f_{2n} + f_{2n-1}$$

we finally get, after some substitution and checking:

$$f_{2n+1} = 2f_n f_{n+1} + f_{n+1}^2 \qquad (\text{for } n \geqslant 1)$$

In this case our algoirthm will need to be in two parts:

1. a part that determines the doubling strategy by generating the binary representation of *n*, and
2. a second part that actually computes the *n*th Fibonacci number according to the doubling strategy.

Notice that "doubling" (f_2, f_3) only gives us (f_4, f_5). However, to generate the next "doubled" pair (f_{10}, f_{11}) we need the pair (f_5, f_6). Fortunately we can simply use the standard formula to generate f_6 (i.e. $f_6 = f_5 + f_4$).

Consider another example where $n = 13$ (Table 3.3).

Table 3.3

Pair of Fibonacci numbers needed	*Fibonacci number*	*Binary representation*
(f_{13}, f_{14})	$\rightarrow f_{13}$	$d[1] = 1$
(f_6, f_7)	$\rightarrow f_6$	$d[2] = 0$
(f_3, f_4)	$\rightarrow f_3$	$d[3] = 1$
(f_1, f_2)	f_1	$d[4] = 1$

We always start out with the pair (f_1, f_2) and to generate the required Fibonacci number we will always need to make one less doubling step $(i-1)$

than there are binary digits i in the representation of n (e.g. the binary representation of $n=13$ is 1011 and hence $4-1=3$ doubling steps are required). The binary digit that is always accounted for by the initial conditions is always the most significant digit (e.g. for $n=13$ it is $d[4]$) when n has been reduced to 1. We can eliminate this digit by stopping generating binary digits one step earlier.

If we use *fn* and *fnp*1 for the Fibonacci numbers f_n and f_{n+1} respectively and *f*2*n* and *f*2*np*1 for f_{2n} and f_{2n+1} we will have the initial doubling steps:

```
f2n := fn*fn+fnp1*fnp1;
f2np1 := 2*fn*fnp1+fnp1*fnp1
```

and then the following reassignments in preparation for the next doubling step:

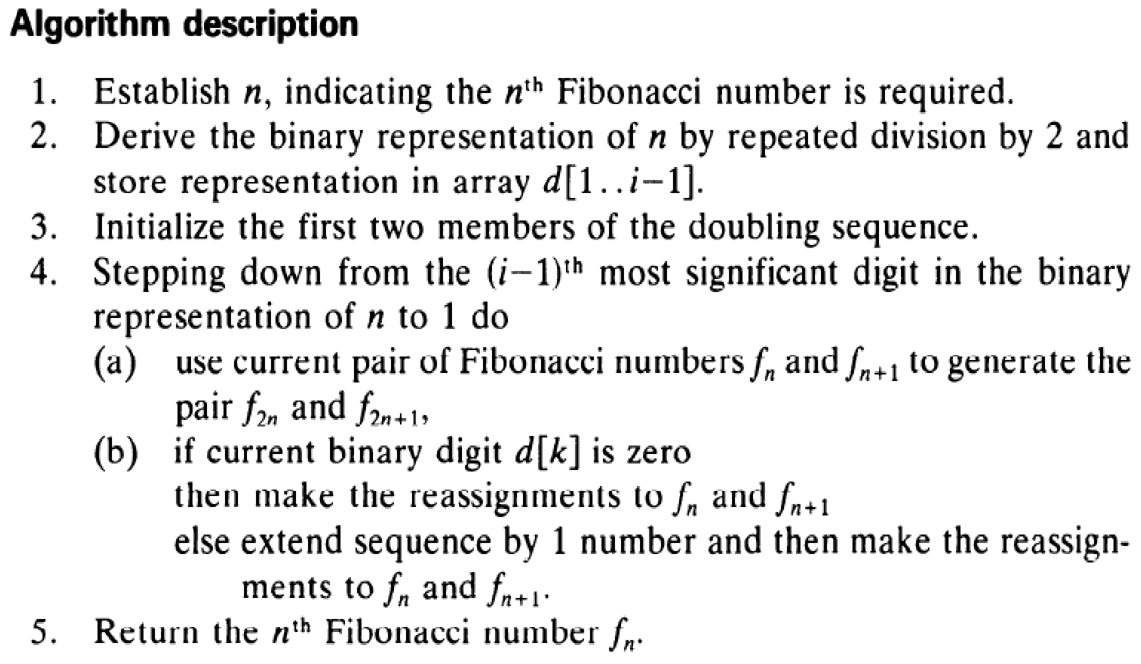
```
fn := f2np1;
fnp1 := f2np1+f2n       (extension by 1)
```

Using the variables *fn*, *fnp*1, *f*2*n*, *f*2*np*1 again, we have in this case first the initially doubling steps as before: that is,

```
f2n := fn*fn+fnp1*fnp1;
f2np1 := 2*fn*fnp1+fnp1*fnp1
```

and then the following reassignments in preparation for the next doubling step:

```
fn := f2n;
fnp1 := f2np1
```



Algorithm description

1. Establish n, indicating the n^{th} Fibonacci number is required.
2. Derive the binary representation of n by repeated division by 2 and store representation in array $d[1..i-1]$.
3. Initialize the first two members of the doubling sequence.
4. Stepping down from the $(i-1)^{th}$ most significant digit in the binary representation of n to 1 do
   (a) use current pair of Fibonacci numbers f_n and f_{n+1} to generate the pair f_{2n} and f_{2n+1},
   (b) if current binary digit $d[k]$ is zero
   then make the reassignments to f_n and f_{n+1}
   else extend sequence by 1 number and then make the reassignments to f_n and f_{n+1}.
5. Return the n^{th} Fibonacci number f_n.

<u>Generating Prime Numbers</u>

Design an algorithm to establish all the primes in first n positive numbers

A prime number is a positive integer that is exactly divisible only by 1 and itself. The first few prime are

2,3,5,7,11,13,17,19,.....

All primes apart from 2 are odd.

Beyond 5 the alternating sequence of differences 2,4

It is easy to construct below with dx initially 4

dx=abs(dx-6)

At this stage we can propose a basic structure for our algorithm:

```
while $x<n$ do
  begin
    (a) generate next $x$ using the construct $dx := abs(dx-6)$,
    (b) test whether $x$ is prime using all primes $\leqslant\sqrt{x}$,
    (c) if a prime is found that is less than $\sqrt{n}$ then store it for
        later testing against larger $x$ values.
  end
```

To test all integers up to n for primality we will need to retain all primes up to $\sqrt{n}$.

Every time a new x is brought up for testing we will need to ensure that we have the appropriate set of primes to divide into x.

Working through some examples we find:

x range	*prime divisors required*
$2\leqslant x<9$	2
$9\leqslant x<25$	2, 3
$25\leqslant x<49$	2, 3, 5
$49\leqslant x<121$	2, 3, 5, 7

Starting out with

```
    $p[1] := 2; p[2] := 3; p[3] := 5;$
and $plimsq := 25; limit := 3$
```

we can include the following conditional statement *before* testing each x value for primality

```
if $x>=plimsq$ then
  begin
    $limit := limit+1;$
    $plimsq := sqr(p[limit])$
  end
```

It is only necessary to increase *limit* by 1 with this test because the difference between the squares of adjacent primes is always greater than 4, the largest increment that is made in x.

[Type text]

Once we have established the proper set of prime divisors we need to test a given x, the next step is to actually test x for primality. For this purpose we can use a loop that successively tests all the prime divisors with indices less than *limit* against x.

Some thought reveals there are two conditions under which this loop should terminate:

1. an exact divisor of x has been found—so it cannot be prime;
2. we have reached the divisor with index one less than limit.

Using the **mod** function to test for exact division and using the remainder *rem* to set the Boolean condition *prime* we get:

```
j := 3; prime := true;
while prime and (j<limit) do
  begin
    rem := x mod p[j];
    prime := rem <> 0;
    j := j+1
  end
```

Algorithm description

1. Initialize and write out the first 3 primes. Also initialize the square of the 3rd prime.
2. Initialize x to 5.
3. While x less than n do
   (a) get next x value excluding multiples of 2 and 3;
   (b) if not past end of multiples list then
      (b.1) if $x \geqslant$ square of largest prime then
         (1.a) include next prime multiple as its square,
         (1.b) update square by squaring next prime $>\sqrt{x}$;
   (c) while have not established x is non-prime with valid prime multiples do
      (c.1) while current prime multiple is less than x, increment by current prime value doubled,
      (c.2) do prime test by comparing x with current multiple;
   (d) if current x prime then
      (d.1) write out x and if it is less than $\sqrt{n}$ store it.