Some problems for CS students to solve

Reg Dodds unassisted by robots
Department of Computer Science
University of the Western Cape
rdodds@uwc.ac.za

2012-12-27—2026-02-13

This is a set of simple problems that students should be able to solve by programming them proficiently in some language. Most of the solutions presented here are presented in a Python-like style or in executable Python rather than in C-like, or which is similar, Java-like pseudo code.

We have chosen Python because it is so easy to learn. A pleasant side effect is that Python forces users into the tidy habit of indenting their code with a rigor not laid down by other languages thus ensuring clear comprehension of the block structure of their code, uncluttered with braces, semicolons, or distracting block delimiters.

Many more solutions than problems are presented because although we attempt to evolve an efficient solution to each problem, there is generally more than one way of programming it. Some of the less efficient solutions present alternate viewpoints and add to our stock of programming tools. So we often tackle the same problem with solutions that differ in outlook and character. This is not limited to presenting many solutions both recursively and iteratively but also to presenting dynamic programming or divide-and-conquer solutions. Sometimes an alternate algorithm for the same problem illustrates a fundamentally different approach such as deriving the algorithm directly from an induction hypothesis, and improving the solution by refining the hypothesis.

These problems have been lying around in our cupboards in a less accessible form.

James Connan used some of them with a Pascal-like pseudo code that does not go down well with Python programmers who are used to a much more elegant type of pseudo code that almost looks like the pseudo code one finds in modern books on algorithms such as the book, now known as CLRS, by Cormen et al. [2022] that are not based on some specific language and that Cormen et al. [1990] have been using since their first edition. This style of layout, that indents the entire block including the final **END**, was already suggested by Marvin Zelkowitz in his version of PL/I called PLUM in Zelkowitz [1976]. CLRS, and Python, improved on this by altogether omitting the end statement.

1 Digits and numbers

The problems in this chapter are based mainly on the properties of numbers. We consider the digits of a number, the factors of a number, be they prime or not, the primacy of a number, and other related problems.

Problem 1.1 Count the number of digits in integer n
Calculate the number of decimal digits of the integer n.

Solution— Counting the digits of the integer n
The digits of n are manipulated by integer division and multiplication of n by 10. We indicate integer division by '//'.[1] So, **n = n // 10** *removes* the last digit of n.[2] To retrieve this removed digit necessitates some manipulation. First, consider the number **(n // 10) * 10**. It differs from n in having its final digit replaced by 0; the other digits are the same. So, to get the final digit, all we need to do is to deduct it from n as follows

```
digit = n -(n // 10) * 10
```

Another obvious way of calculating the last digit is to use the *modulo* operator, which is written as `%` in Python as follows:

```
digit = n % 10
```

But we were not asked to get the last digit, we were asked to count all the digits of n. This is done by retrieving the last digit repeatedly and counting these iterations as follows. We initialize the number of digits m to zero and at each step replace n by its value with its last digit removed.

```
m = 0
while n > 0:
  n = n // 10
  m = m + 1
```

However, this will usually be programmed by using the 'operate-and-becomes' notation and embedding the code in a function.

```
def noDigits(n):
  m = 0
  while n > 0:
    n //= 10
    m += 1
  return m

m = noDigits(n)
```

[1] Python 3 uses the binary operator '//', while most other languages, even earlier versions of Python, use the binary operator '/' embedded between two integers to give an integer result. When one of the operands of '/' is a floating point number then the result also becomes floating point and requires a function such as `int()` to convert it to an integer.

[2] Experienced programmers conventionally write **n = n // 10** as **n //= 10**.

The variable n is unaffected by the invocation **m = noDigits(n)**. This is not the last word. Python has some useful built-in operators that should be used when programming this problem. The **str()** function can be used to turn the integer into a string, and the **len()** function counts the number of characters in the string. The call below gives the same result.

```
m = len(str(n))
```

Exercise 1.2 *Display the digits of the integer in reverse.*

——— □

Problem 1.3 Given integers n and m calculate n^m
When m is positive n^m or n to the power m is defined as

$$n^m = n \times n \times n \times \ldots \ m \text{ times}.$$

More generally, the definition of n to any integer power m is

$$n^m = \begin{cases} n \times n \times n \times \ldots \ m \text{ times} & \text{if } m > 0, \\ 1 & \text{if } m = 0, \\ 1/(n \times n \times n \times \ldots \ |m| \text{ times}) & \text{if } m < 0. \end{cases}$$

Solution— Number raised to integer power
For positive values the power is quite easy to calculate. Simply start with a `product` that is 1 and replace it `m` times by `product` × `n`, i.e.,

```
product = 1
for count in range(m):
  product = product * n
```

This seems to be the last word, but it is rather inefficient. We will discuss more efficient ways of doing later in Problem 4.9. Now we will use the function below to do integer exponentiation. It takes m multiplications.

```
def power(n, m):
  product = 1
  for count in range(m):
    product = product * n
  return product
```

When the power is negative we proceed in the same way for $|m|$ multiplications but invert the product. We use a boolean variable called `negativePower` that becomes `True` when the power is negative to keep track of what action to take.

```
def power(n, m):
  product = 1
  negativePower = False
  if m < 0:
    negativePower = True
    m = -m
  for count in range(m):
      product = product * n
  if negativePower:
    product = 1/product
  return product
```

——— □

Solution— Number raised to integer power
The power n^m can be defined recursively as

$$n^m = \begin{cases} n \times n^{m-1} & m > 1 \\ 1 & m = 0 \\ 1/n^{m+1} & m < 1 \end{cases}$$

——— □

Problem 1.4 Check if n is an Armstrong number
Design an algorithm to check if a given number n is an Armstrong number.

Suppose the number n is written in decimal form as

$$n = d_1^{m-1}10^{m-1} + d_2^{m-2}10^{m-2} + \ldots + d_{m-2}^2 10^2 + d_{m-1}10 + d_m$$

where each $d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ for $i \in [1..m]$ and m is the number of decimal digits of n.

The integer n is an Armstrong number if the sum of each digit of n raised to the power of the number of digits in the number, is equal to the original number, then that number is said to be an Armstrong number, that is, n is an Armstrong number if the following equality holds

$$n = d_1^m + d_2^m + \ldots + d_{m-2}^m + d_{m-1}^m + d_m^m$$

Solution— Algorithm for Armstrong number
Given the integer n, the program must split it into digits, count them, and then sum their powers and check if the sum equals n.

The digits of n are peeled of it from the rear. The least significant digit of n is found by integer division and multiplication. Integer division gives an integer result by dropping the fractional part of the division. Multiplying this result by the divisor produces the original number with a zero in the last digit. Deducting this zero-ended number from n yields the least significant digit—the normally most right-hand digit of n.

```
digit = n -(n // 10) * 10
```

In order to proceed to the next-to-last digit the last digit needs to be removed. Integer division by 10 gets rid of the last digit. We adapt our code slightly so that the final digit can be repeatedly removed until nothing—zero—is left. The following code must be iterated to get each digit.

```
digit = n -(n // 10) * 10
n = n // 10
```

By introducing a new variable $ndiv10$ we can avoid doing the division twice. The **while** statement runs until n becomes zero.

```
while n > 0:
  ndiv = n // 10
  digit = n -ndiv10 * 10
  n = ndiv10
```

After running the while only the leftmost digit will be available, so we need to do the summation for the Armstrong number inside the loop.

```
Armstrong = 0
while n > 0:
  ndiv = n // 10
  digit = n -ndiv10 * 10
  Armstrong += digit ** m
  n = ndiv10
```

This will not work because we have know the value of m before the loop starts. A similar loop is necessary to determine m but we must be careful not to destroy n in the process.

```
m = 0
copyOfn = n
while n > 0:
  ndiv = n // 10
  m += 1
n = copyOfn
# followed by Armstrong code
```

```
digits : array[1..100] of number

get num from user
set cnt = 0
while num <> 0 do
  begin block
     set cnt = cnt + 1
     set digits[cnt] = remainder of num/10
     set num = integer part of num/10
  end block
set total = 0
for i ranges from 1 to cnt do
  set total = total + digits[i]\^cnt
if num = total then display num is an Armstrong number
```

———— □

Problem 1.5 Display the first 15 Armstrong numbers

```
set number = 0
set num = 1
repeat
  set cnt = 0
  while num <> 0 do
    begin block
      set cnt = cnt + 1
      set digits[cnt] = remainder of num/10
      set num = integer part of num/10
    end block
  set total = 0
  for i ranges from 1 to cnt do
    set total = total + digits[i]^cnt
  if num = total then
    begin block
```

```
      display num
      set number = number + 1
    end block
  set num = num + 1
until number = 15
```

Problem 1.6 Find all the $n < 10000$ with nondecreasing digits
Examples of such numbers are $11, 12, \ldots, 19, 22, 23, \ldots, 29, 33, 34, \ldots, 39$.

Problem 1.7 Find all the $n < 1000$ with nondecreasing digits that also have squares with nondecreasing digits
Example of such numbers are $12^2 = 144, 13^2 = 169, 15^2 = 225, 16^2 = 256, 37^2 = 1369$.

Problem 1.8 Convert an integer in base 10 to another base
Given a number in base 10*, convert it to a base supplied by the user. The user supplied base can be anything between* 2 *and* 16.

Solution— Converting between bases
If there was a limit on the size of the base 10 number, then a collection of if statements would have sufficed. That is:

```
get Num from user
get Base from user

if Num > Base^n then
    begin block
        display integer part of Num/(Base^n)
        set Num = remainder of Num/(Base^n)
    end block
    .
    .
    .
if Num > Base^0 then
    begin block
        display integer part of Num/(Base^0)
        set Num = remainder of Num/(Base^0)
    end block


where Base^n is Base to the power of n
```

——— □

Solution— 2
This can easily be rewritten as

```
get Num from user
get Base from user


for n ranges from x to 0 do
    begin block
        if Num > Base^n then
            begin block
```

```
                display integer part of Num/(Base^n)
                set Num = remainder of Num/(Base^n)
            end block
    end block
```

———— □

However, we are still restricted by the size of x and $2^10(1024)$ is much smaller than $9^10(3486784401)$

Solution— 3

We know that we can convert a number from base ten to any base by repeatedly dividing by the base. So we have

```
get Num from user
get Base from user

repeat
    display remainder Num/Base
    set Num = integer part of Num/Base
until Num = 0
```

———— □

Solution— 4

As a while loop

```
get Num from user
get Base from user

while (Num <> 0) do
    begin block
        display remainder Num/Base
        set Num = integer part of Num/Base
    end block
Implement a binary search algorithm that displays
the index of the desired value and the number of
lookups made.Implement a bubble sort algorithm.Write a program that allows the user
to play blackjack against the computer.
```

———— □

Use a array of unique random numbers to ”shuffle” the cards.d

Problem 1.9 Cash register

When a cash purchase is made, the cashier normally has to give the customer change. The customer does not want a bulky wallet, so it is up to the cashier to give the customer the smallest number of notes and coins. Write an algorithm to assist the cashier in determining which notes and coins to give the customer.

Answer

What do we need? The total of the purchase made. The amount tendered.

What do we know? The following notes and coins are available to us: R200, R100, R50, R20, R10, R5, R2, R1, 50c, 20c, 10c, 5c, 2c, 1c

What do we want to do? We want to minimize the number of coins and notes used, so we want to always use the biggest (in value) notes and coins possible. So we start with the biggest denomination and work our way down, each time calculating how many of that denomination to return.

Solution— 1

```
get purchase Total from user
get Amount tendered from user
set Change = Amount -Total

if Change >= R200 then
    begin block
        display R200 x integer part of Change/R200
        set Change = remainder of Change/R200
    end block
if Change >= R100 then
    begin block
        display R100 x integer part of Change/R100
        set Change = remainder of Change/R100
    end block
if Change >= R50 then
    begin block
        display R50 x integer part of Change/R50
        set Change = remainder of Change/R50
    end block
if Change >= R20 then
    begin block
        display R20 x integer part of Change/R20
        set Change = remainder of Change/R20
    end block
if Change >= R10 then
    begin block
        display R10 x integer part of Change/R10
        set Change = remainder of Change/R10
    end block
if Change >= R5 then
    begin block
        display R5 x integer part of Change/R5
        set Change = remainder of Change/R5
    end block
if Change >= R2 then
    begin block
        display R2 x integer part of Change/R2
        set Change = remainder of Change/R2
    end block
if Change >= R1 then
    begin block
        display R1 x integer part of Change/R1
```

```
93          set Change = remainder of Change/R1
94      end block
95  if Change >= 50c then
96      begin block
97          display 50c x integer part of Change/50c
98          set Change = remainder of Change/50c
99      end block
100 if Change >= 20c then
101     begin block
102         display 20c x integer part of Change/20c
103         set Change = remainder of Change/20c
104     end block
105 if Change >= 10c then
106     begin block
107         display 10c x integer part of Change/10c
108         set Change = remainder of Change/10c
109     end block
110 if Change >= 5c then
111     begin block
112         display 5c x integer part of Change/5c
113         set Change = remainder of Change/5c
114     end block
115 if Change >= 2c then
116     begin block
117         display 2c x integer part of Change/2c
118         set Change = remainder of Change/2c
119     end block
120 if Change >= 1c then
121     begin block
122         display 1c x integer part of Change/1c
123         set Change = remainder of Change/1c
124     end block
```

——— □

Solution— 2

The algorithm above solves the problem, but is not very elegant. It would be nice if we could find a way of using a loop rather than writing so many if statements. We notice that the pattern 5xx, 2xx, 1xx is repeated. In fact, if we had a R500 note, we would have had the same pattern 5xx, 2xx, 1xx repeated 5 times, decreasing by a factor 10 every time. So we will use this to to implement a more elegant solution. In order to make it easier for reader to follow, it will be assumed that all amounts are in cents instead of Rands and cents. (Converting all amounts to cents can easily be achieved multiplying by 100.)

```
125 get purchase Total from user
126 get Amount tendered from user
127 set Change = Amount -Total
128
129 set One = 50000
130 set Two = 20000
131 set Three = 10000
```

```
132
133 repeat
134    if (Change >= One) and (One < 50000) then
135        begin block
136            if One >= 500 then display R(One/100) x integer part of Change/One
137            else display (One)c x integer part of Change/One
138            set Change = remainder of Change/One
139            set One = One/10
140        end block
141    if (Change >= Two) then
142        begin block
143            if Two >= 200 then display R(Two/100) x integer part of Change/Two
144            else display (Two)c x integer part of Change/Two
145            set Change = remainder of Change/Two
146            set Two = Two/10
147        end block
148    if (Change >= Three) then
149        begin block
150            if Three >= 100 then display R(Three/100) x integer part of Change/Three
151            else display (Three)c x integer part of Change/Three
152            set Change = remainder of Change/Three
153            set Three = Three/10
154        end block
155 until Change = 0
```

——— □

Solution— 3

The algorithm can be rewritten using a while loop.

```
156 get purchase Total from user
157 get Amount tendered from user
158 set Change = Amount -Total
159
160 set One = 50000
161 set Two = 20000
162 set Three = 10000
163
164 while Change <> 0) do
165    begin block
166        if (Change >= One) and (One < 50000) then
167            begin block
168                if One >= 500 then display R(One/100) x integer part of Change/One
169                else display (One)c x integer part of Change/One
170                set Change = remainder of Change/One
171                set One = One/10
172            end block
173        if (Change >= Two) then
174            begin block
175                if Two >= 200 then display R(Two/100) x integer part of Change/Two
176                else display (Two)c x integer part of Change/Two
```

```
177             set Change = remainder of Change/Two
178             set Two = Two/10
179         end block
180     if (Change >= Three) then
181         begin block
182             if Three >= 100 then display R(Three/100) x integer part of Change/Three
183             else display (Three)c x integer part of Change/Three
184             set Change = remainder of Change/Three
185             set Three = Three/10
186         end block
187   end block
```

□

Solution— 4

The algorithm in Solutions 2 and 3 still uses multiple if statements. We can reduce this by recognizing another pattern.

R200.00 R100.00 R50.00 R20.00 R10.00 R5.00 R2.00 R1.00 R0.50 R0.20 R0.10 R0.05 R0.02 R0.01

We can obtain the next denomination by halving the current one. Every third time we multiply by 2/5. We continue to use only cents and the algorithm now becomes:

```
188 get purchase Total from user
189 get Amount tendered from user
190 set Change = Amount -Total
191
192 set Denomination = 200000
193 set Cnt = 1
194
195 while (Change <> 0) do
196   begin block
197     if (Change >= Denomination) then
198         begin block
199             if Denomination >= 100 then display R(Denomination/100) x integer part of Change/D
200             else display (Denomination)c x integer part of Change/Denomination
201             set Change = remainder of Change/Denomination
202         end block
203     if Cnt is divisible by 3 then set Denomination = 2*Denomination/5
204     else set Denomination = Denomination/2
205     set Cnt = Cnt + 1
206   end block
```

□

Solution— 5

We can also use our knowledge of arrays to solve the problem. We use a two dimensional array to store the number of each denomination needed. First we initialize the array with the denominations and we zero the number for each.

Money is an array[1..2][1..14] of number

```
get purchase Total from user
get Amount tendered from user
set Change = Amount -Total

set Denomination = 20000

for Index ranges from 1 to 14 do
    begin block
        set Money[1][Index] = Denomination
        set Money[2][Index] = 0
        if Index is divisible by 3 then set Denomination = 2*Denomination/5
        else set Denomination = Denomination/2
    end

set Cnt = 1

while (Change <> 0) do
    begin block
        if Change >= Money[1][Cnt] then
            begin block
                set Money[2][Cnt] = integer part of Change/Money[1][Cnt]
                set Change = remainder of Change/Money[1][Cnt]
            end block
        set Cnt = Cnt + 1
    end block

set Cnt = 1

while (Money[1][Cnt] >= 100) do
    begin block
        if Money[2][Cnt] > 0 then display R(Money[1][Cnt]/100) x Money[2][Cnt]
        set Cnt = Cnt + 1
    end block

repeat
    if Money[2][Cnt] > 0 then display (Money[1][Cnt])c x Money[2][Cnt]
    set Cnt = Cnt + 1
until Cnt > 14
```

———— □

Problem 1.10 Determine if a number N is prime
Given a number (N), write an algorithm to determine if the number is prime or not.

What is a prime number?

It is a number that is divisible only by itself and 1.

How would we check to see if a number is prime or not? If a number is divisible by any number other than 1 or itself, then that number is not prime.

Solution— Algorithm for primes by trial division

In plain English

Get the number to check

Assume the number is prime

Test divisibility of the number by all potential factors between 1 and the number itself

If the number is divisible by any of these numbers, then it is not prime, otherwise it is prime

In pseudo code

```
get Num from user
get IsPrime = True
for PFactor ranges from 2 to Num-1 do
  begin block
     if Num divisible by PFactor then set IsPrime = False
  end block
if IsPrime = True then display Num is prime
else display Num is not prime
```

———— □

Solution— Slightly improved prime checker

If we look at the factors of non-prime numbers, we notice that they are symmetrical around the square root of the number. That is, for say 36, the factors are 1×36, 2×18, 3×12, 4×9, 6×6, 9×4, 12×3, 18×2, and 36×1. Therefore we only need to check each potential factor up to the square root of the number. Also, as soon as we find a factor for the number, we can stop checking because the number is not prime.

In plain English

Get the number to check

Check all potential factors until the factor is greater than the square root of the number or we find a factor that divides into the number without a remainder

In pseudo code

```
repeat loop

get Num from user
set PFactor = 1
repeat
  set PFactor = PFactor + 1
until (PFactor > square root of (Num)) or (Num divisible by PFactor)
if PFactor <= square root of (Num) then display Num is not prime
else display Num is prime


while loop

get Num from user
set PFactor = 2
```

```
while (PFactor <= square root of (Num)) and (Num not divisible by PFactor)
  begin block
     set PFactor = PFactor + 1
  end block
if PFactor <= square root of (Num) then display Num is not prime
else display Num is primeWrite a program that will convert and angle from
decimals to degrees minutes and seconds.Write a program that will convert a number from
digits to its English equivalent. For example 105 becomes
one hundred and five.GCD
```

———— □

Problem 1.11 Greatest common divisor—GCD
Write and algorithm to find the GCD (Greatest Common Divisor) of two numbers, x and y.

Solution— 1

What is the GCD?

It is the biggest number that divides into both x and y without leaving a remainder.

This means that the biggest possible GCD is at most the smaller of the two numbers.

```
get x from user
get y from user

if x > y then set Smaller = y
else Smaller = x

for PFactor ranges from 1 to Smaller do
    begin block
        if (x divisible by PFactor) and (y divisible by PFactor) then GCD = PFactor
    end block
display GCD
```

———— □

Solution— GCD version 2
We can reduce the execution time of the algorithm by first checking if the smaller number is the GCD and then checking all numbers less then it and stopping as soon as we find a factor.

```
get x from user
get y from user

if x > y then set Smaller = y
else Smaller = x

set GCD = Smaller

While not((x divisible by GCD) and (y divisible by GCD)) do
    set GCD = GCD -1
display GCD
```

The condition for the while statement can also be written as: While (x not divisible by GCD) or (y not divisible by GCD) do ———— □

Solution— 3
Two millennia and three centuries ago Euclid presented us with an even more elegant solution. He showed that

```
GCD(x, y) = GCD(y, remainder of x/y) if y > 0
GCD(x, y) = x if y = 0

In pseudo code this is:

get x from user
get y from user

while (y > 0) do
    begin block
        set Tmp = remainder of x/y
        set x = y
        set y = tmp
    end block
display GCD = x
```

———— □

Problem 1.12 GCD of a list of numbers
Find the GCD of any list of numbers.

Solution— GCD of a list numbers
The GCD of three or more numbers can be found by finding the GCD of pairs of numbers. That is:

```
GCD (x, y, z) = GCD (x, GCD (y, z)) or GCD (x, y, z) = GCD ( GCD (x, y), z)
```

———— □

Solution— GCD of a list numbers
If can also be found using the same strategy as used in Solutions 1 and 2. Solution 2, for three numbers would then be

```
get x from user
get y from user
get z from user

if (x > y) and (z > y) then set Smallest = y
if (x > z) and (y > z) then set Smallest = z
if (y > x) and (z > x) then set Smallest = x


set GCD = Smaller

While not((x divisible by GCD) and (y divisible by GCD) and (z divisible by GCD)) do
    set GCD = GCD -1
display GCD
```

———— □

Problem 1.13 Convert marks into letter grades

Write a program that will convert a mark as a given as a percentage to grade symbol, e.g. 80% becomes an 'A'.

Problem 1.14 Guess the number
Write a program that will generate a random integer, in the range, starting with $[1..10]$ *and allow the user to guess the number.*

After each guess the program tells the user if the guess was to high or too low. After 12 or so unsuccessful guesses the game is lost.

Introduce levels so that each time the user guesses the number correctly, a new number is generated in from a larger range. Each time the user fails to guess the correct number within the allowed number of guesses, they get demoted to a lower level.

Problem 1.15 Lowest common denominator—LCM
Find the LCM (Lowest Common Denominator) of two numbers, x and y.

What is the LCM?

It is the smallest number that is divisible by both x *and* y*. That is, into which both* x *and* y *will divide without leaving a remainder.*

What do we know about the LCM?

The LCM must be greater than or equal to the larger of x *and* y*. The LCM must be less than or equal to the product of the* x *and* y*.*

Solution— LCM 1

```
get x from user
get y from user

if x > y then set Greater = x
else set Greater = y


for PLCM ranges from (x*y) to Greater do
    if (PLCM is divisible by x) and (PLCM is divisible by y) then set LCM = PLCM
display LCM
```

——— □

Solution— 2
Solution 1 will always check all numbers between the larger (of x and y) and the product (of x and y). We can also rewrite the algorithm in such a way that it will only check numbers up to and including the LCM and then stop.

```
get x from user
get y from user

if x > y then set Greater = x
else set Greater = y

set PLCM = Greater
set LCM = 1

```

```
repeat
    if (PLCM is divisible by x) and (PLCM is divisible by y) then set LCM = PLCM
    set PLCM = PLCM + 1
until (LCM is divisible by x) and (LCM is divisible by y)
display LCM
```

Note: LCM is used to terminate the loop because the value of PLCM is changed after assigning it to LCM. Also, LCM is initialized otherwise it has no definite value until the condition for the if statement is satisfied. ——— □

Solution— LCM 3

We know that if one number is divisible by another number, then the first number must be a multiple of the second number. That is, 100 is divisible by 10. 100 is a multiple of 10. 10 x 10 = 100. Only multiples of 10 will be divisible by 10 (10, 20, 30, ...). We therefore do not even need to check numbers between 10 and 20, or 20 and 30, etc. Solution 2 can therefore be improved as follows:

```
get x from user
get y from user

if x > y then set Greater = x
else set Greater = y

set PLCM = Greater
set LCM = 1

repeat
    if (PLCM is divisible by x) and (PLCM is divisible by y) then set LCM = PLCM
    set PLCM = PLCM + Greater
until (LCM is divisible by x) and (LCM is divisible by y)
display LCM
```

——— □

Solution— LCM 4

Solution 3 can be rewritten using a while loop.

```
get x from user
get y from user

if x > y then set Greater = x
else set Greater = y

set PLCM = Greater

while not ((PLCM is divisible by x) and (PLCM is divisible by y)) do
    set PLCM = PLCM + Greater

display PLCM
```

Note: We are checking the loop termination condition before changing PLCM, so we do not need to make a copy of it. When the loop terminates, PLCM will be the LCM. Also, the not

can be moved into the bracket, changing the condition from not((PLCM is divisible by x) and (PLCM is divisible by y)) to (PLCM is not divisible by x) or (PLCM is not divisible by y)
——— □

Problem 1.16 Solve a set of n linear equations
Write a program that will solve a set of n linear equations in n unknowns. Write a program that allows the user to perform matrix operations, such as matrix addition, subtraction, and multiplication. Write a program that generates two random arrays (reuse your old code) of length n and m respectively. Merge the two arrays into one by alternately selecting elements from each.Write a program that takes two sorted arrays and produces a single sorted array.

Problem 1.17 Merge two sorted lists
Given arrays sorted n ascending order, merge them to produce one sorted array.

Solution— Merge two sorted lists

```
379 arrone : array [1..n] of number
380 arrtwo : array [1..m] of number
381 arrthree : array [1..(m+n)] of number
382
383 procedure merge
384   begin block
385     set cnt1 = 1
386     set cnt2 = 1
387     set cnt3 = 1
388     while (cnt1 <= n) and (cnt2 <= m) do
389        begin block
390          if arrone[cnt1] < arrtwo[cnt2] then
391            begin block
392              set arrthree[cnt3] = arrone[cnt1]
393              set cnt1 = cnt1 + 1
394            end block
395          else
396            begin block
397              set arrthree[cnt3] = arrtwo[cnt2]
398              set cnt2 = cnt2 + 1
399            end block
400          set cnt3 = cnt3 + 1
401        end block
402     if cnt1 > n then
403       begin block
404         for i ranges from 0 to m-cnt2 do
405           set arrthree[cnt3+i] = arrtwo[cnt2+i]
406       end block
407     else
408       begin block
409         for i ranges from 0 to n-cnt1 do
410           set arrthree[cnt3+i] = arrone[cnt1+i]
411        end block
412   end block
```

——— □

Problem 1.18 Find all the prime numbers from 2 to N.

There are many useful variations of this problem.

Problem 1.19 Find the first N primes

Solution— Find primes
What is a prime number?

It is a number that is divisible only by itself and 1.

How would we check to see if a number is prime or not?

If a number is divisible by any number other than 1 or itself, then that number is not prime.

What do we know/have?

We have already constructed several algorithms to check whether a number is prime or not. All we need to know now is use one of these algorithms to check all numbers and count each prime until we have N of them.

Algorithm

In plain English

Get N, the number of primes the user wishes to find Check all numbers from 2 up until we have found N primes

In plain English, refine

```
Get N, the number of primes the user wishes to find

Start with 2
Repeat
  Assume the number is prime
  Test divisibility of the number by all possible numbers
  between 1 and the number itself
  If the number is divisible by one of these numbers, then
    it is not prime,
  Otherwise
    it is prime,
    count it
  Until we have found N primes
```

In slightly more exact pseudo code

```
repeat loop

get N from user
set Count = 0
set Num = 2
repeat
  set PFactor = 1
  repeat
    set PFactor = PFactor + 1
  until (PFactor > square root of (Num)) or (Num divisible by PFactor)
```

```
  if (PFactor <= square root of (Num)) then display Num is not prime
  else
    begin block
      display Num is prime
      set Count = Count + 1
    end block
until Count = N
```

Another more refined version

```
while loop

get N from user
set Count = 0
set Num = 2
while Count < N do begin block
    set PFactor = 2
    while (PFactor < square root of (Num)) and
          (Num not divisible by PFactor) do
      set PFactor = PFactor + 1

    if (PFactor <= square root of (Num)) then
      display Num is not prime
    else begin block
        display Num is prime
        set Count = Count + 1 end block end block
```

———— □

Problem 1.20 Numeric integration by summing rectangles
Given a function $y = f(x)$*, an interval* $[x_i, x_{i+1}]$*, and a number* n *of intervals such that* $i = 0, 1, \ldots, n-1$.

Find the approximate area under the curve by calculating and summing the areas of each rectangle of width $x_i - x_{i+1}$ and height $f(x_i)$.

Solution— Integration
We slice the area under the graph into n rectangles and calculate the area of each rectangle. Summing the areas of each rectangle will yield an approximation for the total area under the graph.

If we substitute a value for x into the equation, we get the corresponding y value. This $y = f(x_i)$ value represents the height of the rectangle. Using x_i and x_{i+1} we can calculate the width. Thus we can calculate the area of the rectangle. If we sum the areas of all these rectangles that are in the interval $[a, b]$ then we get the approximate area under the graph. It should be clear that the sum of these rectangles is less than the area under the curve.

```
get f from user
get a from user
get b from user
get n from user

set Area = 0
for Tmp ranges from a to b in steps of (b -a)/n do
```

```
466   set Area = Area + f(Tmp)*width
467 display Area
```

———— □

We can improve this approximation by using the second ordinate for the value of the height, and following the same procedure. This approximates the actual area under the curve from above. The average of these two values will provide a much better approximation of the integral. Without repeating the whole process the following idea achieves the same result.

Problem 1.21 Numeric integration by summing trapezia[3]

This problem is often posed as finding an approximation to the definite integral

$$\int_a^b f(x)dx.$$

Find the approximate area under the curve by calculating and summing the areas under each trapezium bounded by the x-axis, by the two ordinates x_i–$f(x_i)$, x_{i+1}–$f(x_{i+1})$, and the line $(x_i, f(x_i))$–$(x_{i+1}, f(x_{i+1}))$. Figure1.1 shows the ith trapezium. The function $f(x)$ is given. and

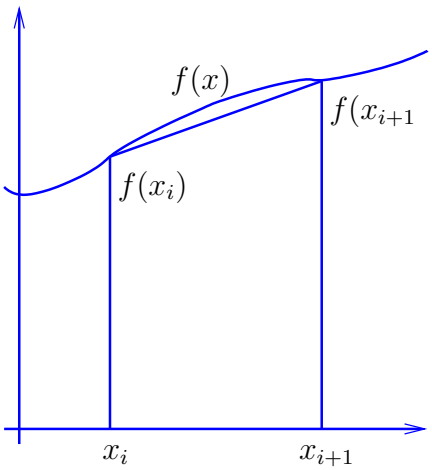

Figure 1.1: A trapezium.

the bounds of the area to be approximated are given as the closed interval $[a, b]$.

[3] If you do not know Latin you might be inclined to spell the plural of trapezium as *trapeziums*. Some people talk of *trapezoids*.

2 Binomial coefficients and Pascal's triangle

2.1 Binomial coefficients

Consider the binomial $(x+y)^n$ for $n=0$. Since $(x+y)^0=1$, the terms in x and y disappear and we write down a single 1. Putting $n=1$

$$(x+y)^1 = 1x + 1y,$$

and we get a 1 and another 1. When $n=2$ the equation becomes

$$(x+y)^2 = 1x^2 + 2xy + 1y^2$$

giving 1, 2, and 1 again. Next for $n=3$ the equation becomes

$$\begin{aligned}(x+y)^3 &= (x+y)(1x^2+2xy+1y^2)\\ &= 1x^3+2x^2y+1xy^2\\ &= \quad +1x^2y+2xy^2+1y^3\\ &= 1x^3+3x^2y+3xy^2+1y^3,\end{aligned}$$

yielding the coefficients 1, 3, 3, and 1. In general we write

$$(x+y)^n = \sum_{r=0}^{n} \binom{n}{r} x^r y^{n-r} \tag{2.1}$$

giving the coefficients as the numbers $\binom{n}{r}$. For now we know that $\binom{3}{0}=1$, $\binom{3}{1}=3$, $\binom{3}{2}=3$, $\binom{3}{3}=1$. One of the tasks we set is to calculate these coefficients in various ways. Before doing this we look at a property that we can derive from Equation 2.1 by setting both x and y to 1,

$$2^n = \sum_{r=0}^{n} \binom{n}{r} 1^r 1^{n-r}$$

So

$$\sum_{r=0}^{n} \binom{n}{r} = 2^n \tag{2.2}$$

Pascal's triangle is an arrangement of the binomial coefficients in a table. We will call the number in Row n and Column r, $C(n,r)$. The table is easy to calculate. Assume that all the values in Column 0, $C(i,0)=1$, for $i=1,2,\ldots$ and the values in Column 1 are In column 1

	Columns	
Rows	$r-1$	r
$n-1$	$C(n-1,r-1)$	$C(n-1,r)$
n	$C(n,r-1)$	$C(n,r) = C(n-1,r-1) + C(n-1,r)$

the values are given by $C(i,1) = i$ for $i = 1, 2, \ldots$. This makes $C(1,0) = 1$ and $C(1,1) = 1$, Consider a portion of the table of numbers of the binomial coefficients in a triangle. It is named after Blaise Pascal. It has been studied for centuries and is still the focus of vigorous attention.

The rows of Pascal's triangle are conventionally enumerated starting with row zero, and the numbers in odd rows are usually staggered relative to the numbers in even rows. A simple construction of the triangle proceeds in the following manner. On the zeroth row, write only the number 1. Then, to construct the elements of following rows, add the number directly above and to the left with the number directly above and to the right to find the new value. If either the number to the right or left is not present, substitute a zero in its place. For example, the first number in the first row is 0 + 1 = 1, whereas the numbers 1 and 3 in the third row are added to produce the number 4 in the fourth row."

n	$\binom{n}{0}$	$\binom{n}{1}$	$\binom{n}{2}$	$\binom{n}{3}$	$\binom{n}{4}$	$\binom{n}{5}$	$\binom{n}{6}$	$\binom{n}{7}$	$\binom{n}{8}$	$\binom{n}{9}$	$\binom{n}{10}$	$\binom{n}{11}$
0	1											
1	1	1										
2	1	2	1									
3	1	3	3	1								
4	1	4	6	4	1							
5	1	5	10	10	5	1						
6	1	6	15	20	15	6	1					
7	1	7	21	35	35	21	7	1				
8	1	8	28	56	70	56	28	8	1			
9	1	9	36	84	126	126	84	36	9	1		
10	1	10	45	120	210	252	210	120	45	10	1	
11	1	11	55	165	330	462	462	330	165	55	11	1

Problem 2.1 Display Pascal's triangle

Design an algorithm that will display a Pascal triangle. Use recursion to calculate the terms.

$$b(n,k) = b(n-1,k-1) + b(n-1,k) b(n,0) = 1 b(n,n) = 1$$

Solution— A recursive algorithm

Recursive function to calculate terms

```
468 function pascal(n,k)
469   begin block
470     if (k = 0) or (k = n) then return set pascal = 1
471     else return set pascal = pascal(n-1, k-1) + pascal(n-1, k)
472   end block
```

——— □

Solution— Binomial coefficients using addition

——— □

Problem 2.2 Display Pascal's triangle
Pascal's triangle must be presented in a tabular form.

Problem 2.3 Display Pascal's triangle as a triangle
Display Pascal's triangle in the form

```
                    1
                  1   1
                1   2   1
              1   3   3   1
            1   4   6   4   1
          1   5  10  10   5   1
        1   6  15  20  15   6   1
```

By convention the first row is called 'Row 0'. The sum of its coefficients is 1. The sum of the coefficients of Row 4 is 15, in general the sum of the binomials in row r is 2^r.

Solution— Displaying Pascal's triangle

```
get num from user
for n ranges from 0 to num do
  begin block
    for k ranges from 0 to num do
      begin block
        display pascal(n, k)
      end block
  end block
```

———— □

Problem 2.4 Perfect numbers
Design an algorithm that will check if a number supplied by the user is a perfect number.

A perfect number is a number, n, such that:

$$n = f_1 + f_2 + \ldots + f_k$$

where the f_i, $i = 1, 2, \ldots, k$ are the factors of n.

In other words: If a number is equal to the sum of all its factors, excluding itself, then that number is a perfect number.

Algorithm

Solution— Perfect numbers
This function finds all the factors of a number, places them in an array and returns the number of factors found.

```
factors : array [1..1000] of numbers

function fact (num)
  begin block
    set cnt = 0;
    for i ranges from 1 to num do
      begin block
        if num is divisible by i then
```

```
          begin block
            set cnt = cnt + 1
            set factors[cnt] = i
          end block
      end block
    return set fact = cnt
  end block
```

The main algorithm

```
get num from user
count = fact (num)
set total = 0
for i ranges from 1 to count -1 do
   set total = total + factors(i)
if total = num then display num is a perfect number
```

———— □

Problem 2.5 Display the first 15 perfect numbers

Solution— Display the first 15 perfect numbers

```
set count = 0
set num = 1
repeat
  count = fact (num)
  set total = 0
  for i ranges from 1 to count -1 do
    set total = total + factors(i)
  if total = num then
    begin block
      set count = count + 1
      display num
    end block
  set num = num + 1
until count = 15
```

———— □

Problem 2.6 Prime numbers again

Use an array to construct an efficient algorithm to find prime numbers.

Solution— Prime numbers with an array

We know that if n is divisible by d, then n must also be divisible by all of d's factors, e.g. 16 is divisible by 8. It is also divisible by 8's factors, 1, 2, 4. This means that when we check for factors of a number we can ignore compound numbers and consider only the prime divisors of n.

———— □

Let's find the first 1000 prime numbers

Algorithm

```
primes : array [1..1000] of numbers
```

```
517
518 set num_of_primes = 1
519 set primes[num_of_primes] = 2
520 set number_to_test = 3
521 repeat
522   set cnt = 1
523   while (num_to_test is not divisible by primes[cnt]) and (primes[cnt] < sqrt(num_to_test)) do
524     begin block
525       set cnt = cnt + 1
526     end block
527   if num_to_test is not divisible by primes[cnt] then
528     begin block
529       set num_of_primes = number_of_primes + 1
530       set primes[number_of_primes] = num
531     end block
532   set num_to_test = num_to_test + 1
533 until num_of_primes = 1000
534 for i ranges from 1 to 1000 do
535   display primes[i]
```

Problem 2.7 2

Find all prime numbers between Num1 and Num2.

Solution— Getting primes

What is a prime number? It is a number that is divisible only by itself and 1.

How would we check to see if a number is prime or not? If a number is divisible by any number other than 1 or itself, then that number is not prime.

What do we know/have? We have already constructed several algorithms to check whether a number is prime or not. All we need to know now is use one of these algorithms to check each number between Num1 and Num2.

Algorithm

In plain English

```
536 Get Num1, the lower boundary of the range
537 Get Num2, the upper boundary of the range
538 For each number between Num1 and Num2
539    Assume the number is prime
540    Test divisibility of the number by all potential factors between 1 and the number itself
541    If the number is divisible by any of these numbers, then it is not prime, otherwise it is prime
```

In pseudo code

```
542 repeat loop
543
544 get Num1 from user
545 get Num2 from user
546 for Num ranges from Num1 to Num2 do
547   begin block
548     set PFactor = 1
549     repeat
```

```
550       set PFactor = PFactor + 1
551     until (PFactor > square root of (Num)) or (Num divisible by PFactor)
552     if PFactor <= square root of (Num) then display Num is not prime
553     else display Num is prime
554   end block
```

while loop

```
555 get Num1 from user
556 get Num2 from user
557 for Num ranges from Num1 to Num2 do
558   begin block
559     set PFactor = 2
560     while (PFactor < square root of (Num)) and (Num not divisible by PFactor)
561       begin block
562         set PFactor = PFactor + 1
563       end block
564     if PFactor <= square root of (Num) then display Num is not prime
565     else display Num is prime
566   end block University of the Western Cape
```

———— □

3 A part of James' notes in their original format

Computer Science I Problem Solving

James Connan jconnan@uwc.ac.za

Academic Responsibility

Academic dishonesty will not be tolerated. Students suspected of academic dishonesty will be referred to the Proctor's office.

Course Objectives

Read, understand and solve problems
Problem solving techniques
Understanding of sequence, selection and repetition control structures
Design algorithms to solve problems
Express algorithms in pseudo code or structured diagrams
Improved analytical thinking and problem solving skills

What does Computer Scientists do?

Computer Scientists use computers to solve problems
Often new software needs to be developed to solve problems
Software development requires problem solving and programming skills

What is Problem Solving?

Problem Solving is a process
The first step in Problem Solving is understanding and exploring the problem
Next a strategy for solving the problem must be found
Use the strategy to solve the problem
Reflect on the solution

What are Algorithms?

Algorithms describe methods by which tasks are to be accomplished
An Algorithm is a set of instructions
When an Algorithm is executed a task is accomplished

Example Algorithms

Knitting pattern (e.g. Jersey)
Assembly Instructions (e.g. Models, furniture, etc.)

Recipe (e.g. Cake)
Pattern (e.g. Dress, jacket, etc.)
Musical Score (e.g. Beethoven, Bach, etc.)

Syntax, Semantics and Logic

When designing algorithms there are certain common errors that must be avoided. These are errors with syntax, semantic and logic. Syntax rules state how symbols in a language are used. In the sentence 'sleeve the seams' or the expression '2+=3' valid English and mathematical symbols are used, but the way they are combined does not result in a valid sentence or expression. Semantic rules examine the meaning of symbols. The sentence 'the elephant ate the peanut' is a valid English sentence, but 'the peanut ate the elephant' makes no sense.
Logic rules check if the solution properly describes the process. If an incorrect formula is used or an invalid conclusion is made then the algorithm will fail.

Stepwise/Top-down Refinement of Algorithms

When constructing an algorithm, it is a good idea to start of by describing the solution in the most general terms. So, designing an algorithm to make a cup of coffee, the first iteration yields:

1.boil water
2.put coffee in cup
3.add water to cup

Each step is now examined and were sensible further refinements are made.

1.boiled water

This can be further refined by saying:

1.1.fill kettle
1.2.switch on kettle
1.3.wait until boil
1.4.switch off kettle

2.put coffee in cup

Can be expanded to:

2.1.open coffee jar
2.2.measure coffee
2.3.put coffee in mug
2.4.close jar

3.add water to cup

Becomes

3.1.pour water from kettle into mug until mug full

Again the algorithm is checked to see if there are any steps that need further refinement.

1.boil water
1.1.fill kettle
1.1.1.put kettle under tap
1.1.2.turn on tap

1.1.3.wait until kettle full
1.1.4.turn off tap
1.2.switch on kettle
1.3.wait until boil
1.3.1.wait until kettle whistles
1.4.switch off kettle
2.put coffee in cup
2.1.open coffee jar
2.1.1.take jar off shelf
2.1.2.remove lid
2.2.measure coffee
2.3.put coffee in mug
2.4.close jar
2.4.1.replace lid
2.4.2.replace jar on shelf
3.add water to cup
3.1.pour water from kettle into mug until mug full

There are no now further sensible refinements that can be made to the algorithm, so the algorithm is complete.

Sequential Algorithms

The coffee-making algorithm is an example of a sequential algorithm.
Sequential algorithms have the following characteristics:
1.Steps are executed one at a time
2.Each step is executed exactly once: none are repeated or omitted
3.The steps are executed in the same order in which they are written
4.Termination of the last step implies termination of the algorithm

Trace Tables

Trace tables are used to verify that the algorithm performs the desired task. All variables are written down in such a way as to serve as headers for columns that will be used to track them as the algorithm is executed. The algorithm is then executed one line at a time and changes to all variables are noted. It may be desirable to keep track of any output produced by the algorithm as well.

Conditionals

Sequential algorithms allows us to solve many problems, but sometimes the solution to an problem can not be expressed in a sequential way. There may also be instructions that we only want to execute under certain conditions. In the coffee making algorithm, for example, if the kettle is already full we do not need to fill it. The conditional statements we have are as follows:

If condition then statement
The statement will only be executed if the condition holds.

If condition then statement1 else statement2
If the condition holds statement1 is executed and if it does not hold statement2 is executed.

Case variable of

condition1 : statement1
condition2 : statement2
condition3 : statement3
Case statements can be used to replace multiple if statements. If the variable matches condition1 statement1 will be executed. Same condition2 etc.

Looping Constructs

Solutions often include some form of repetition. The same action is repeated over and over with few or no changes. If the objective is to work out the average of some numbers then the number must be read in and totalled one at a time until all numbers have been added. The total is then divided by the number of numbers to obtain the average. There are three looping structures available to us. These are the for-loop, the while-loop and the repeat-loop.

The for-loop

for variable ranges from start value to end value do
statement

The value of variable is set to start value. The statement is then executed. The value of variable is then incremented and statement is executed again. This is repeated until the value of variable exceeds end value. At this stage the loop exits. It is also possible to set start value higher than end value and decrement variable each iteration until it reaches end value.

The while-loop

while condition execute do
statement

As long as the condition is valid the while-loop will continue executing the statement. This means that statement must change something to allow condition to become untrue or the loop will never terminate. There are examples where it is desirable to have a loop that never terminates, but in most cases this is not true.

The repeat-loop

repeat
statement
until condition

The statement will be executed until the condition is met. If the statement does not change something to allow the condition to become true then the loop will never terminate.

Some rules of thumb for loops
1.If we know exactly how many times we want a loop to be executed, it is normally desirable to use a for-loop.
2.The main difference between a while-loop and repeat-loop is that with a while-loop the condition is checked before the loop is entered while with a repeat-loop the condition is checked after the first iteration of the loop.
3.repeat-loops and while-loops are usually interchangeable. In order to change from the one to the other the condition needs to be negated.

Statement Blocks

So far we have only referred to single statements. If we have multiple statements we can encap-

sulate them with a begin block and end block. These statements then become a single unit.

start block
statement
statement
statement
end block

begin ... end ...

Nested Loops

Loops within loops

Arrays

A lot of problems requires the use of lists as part of the solution. One way to handle lists is to use arrays. The syntax is as follows:

arrname : array [start..end] of type

where, arrname is the name of the array,
start..end is the range of the array,
and type is the type of data that can be stored in the array

Examples

```
num : array [1..3] of number

          1 2 3
          ---
num --> | | | |
          ---
```

This array provides us with a variable, num, that has three 'slots' of type number in which we can store values. These 'slots' are accessed by using num[1], num[2] and num[3].

word : array[1..15] of character

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
– – – – – – – – – – – – – – –
word –> | | | | | | | | | | | | | | |
– – – – – – – – – – – – – – –

This array provides us with 15 'slots' of type character. They are accessed by using word[1], word[2], ... , word[15]

num : array [5..8] of number

5 6 7 8
- - - -

num –> | | | | |
- - - -

This provides us with four 'slots' referenced using num[5], num[6], num[7] and num[8]

set arr[2] = 7

This will store the value 7 in the position two of the array.

set arr[i] = 5

This will store the value 5 in position i of array arr. The value i must be within the range of the array. That means that the value of i must be between start and end.

set arr[i+1] = 5

Same as above, except that i+1 must now be within the range of the array.

set arr[5] = arr[7]

This will set the value of the 5'th element of the array equal to the value of the 7'th element.

set num = arr[3]

This will set the value of num equal to the value stored in position 3 of array arr.

Arrays can have more than one dimensions. That is:

num : array [1..3][1..5] of number

1 2 3 4 5
- - - - -
num –> 1 | | | | | |
- - - - -
2 | | | | | |
- - - - -
3 | | | | | |
- - - - -

This visualization is often used to make it easier to understand what multiple (in this case 2) dimensional arrays are. We now have 15 'slots' and each 'slot' is identified by two numbers. num[1,1] is the first element in the array and num [3,5] is the last. There are now two indexes that need to be used to reference each element and each must be within the boundaries defined for its range. It is important to remember that the visualization above is exactly that, just a visualization. The array could also have been visualized as follows:

1,1 1,2 1,3 1,4 1,5 2,1 2,2 2,3 2,4 2,5 3,1 3,2 3,3 3,4 3,5
— — — — — — — — — — — — — — —
num –> | | | | | | | | | | | | | | |
— — — — — — — — — — — — — — —

We are not restricted to two dimensions.

num : array [1..3][4..6][1..5][1..2][3..5] of number

This is a five dimensional array. We can store 3x3x5x2x3 = 270 elements in this array. Each element is referred to by 5 indexes. Hence, num [i,j,k,l,m] refers to one element in the array. Each of the indexes i,j,k,l,m must be within its defined range.

Arrays, combined with the looping structures we have already encountered, are very powerful tools to help us solve problems.

Functions and Procedures

We have already encountered block statements. When we have a block of statements that perform a particular task it is often convenient to label this block in a special way that allows us to refer to it again at a later time. For example:

The following algorithm calculates and displays x to the power of y.

```
set ans = 1
for i ranges 1 to y do
set ans = ans * x
display ans
```

We can rewrite this as

```
procedure pow(x,y)
begin block
set ans = 1
for i ranges from 1 to y do
set ans = ans * x
display ans
end block
```

We can now call this procedure from another algorithm

```
get num from user
get power from user
pow (num,power)
```

We can also write this as a function

```
function pow(x,y)
begin block
set ans = 1
for i ranges from 1 to y do
set ans = ans * x
return set pow = ans
end block
```

The difference between a procedure and function is that a procedure can only act on data/variables while a function can return a value.

get num from user
get power from user
set result = pow(x,y)
display result

Recursion

A function or procedure is said to be recursive when it calls itself. It is possible to rewrite the power function above as a recursive function.

function pow(x,y)
begin block
if y = 0 then return set pow = 1
else return set pow = pow(x,y-1) * x
end block

When writing a recursive function the first thing to do is to find the base case. For example:

Write an algorithm to work out n!

1! = 1
2! = 1 x 2
3! = 1 x 2 x 3
4! = 1 x 2 x 3 x 4
etc

we can rewrite this as

1! = 1
2! = 1! x 2
3! = 2! x 3
4! = 3! x 4
etc

in general

n! = (n-1)! x n

the base case is

when n = 1 then n! = 1

Thus

```
function fact(n)
begin block
```

```
  if n = 1 then return set fact = 1
  else return set fact = fact (n-1) x n
end block
```

Write a recursive algorithm that will add two numbers.Roots

Problem 3.1 Roots of a quadratic equation
Write an algorithm that will find and classify the roots of the quadratic equation

$$ax^2 + bx + c = 0$$

then

$$x = (-b + -sqrt(b^2 - 4ac))/2a$$

Notes:

if $a = 0$ then $x = -c/b$
if $b = 0$ then $x = \sqrt{(} - 4ac)/2a$
if $c = 0$ then $x = -b/a$

We need to ensure that $\sqrt{x}$ exists, i.e. that the *argument*, of the square root function x is positive.

Remember that the square root of something is always non negative, in other words it is greater or equal to zero.[1]

Algorithm

```
get a from user
get b from user
get c from user

if a = 0 then
  begin block
    if c <> 0 then display x = -b/c
    else
       begin block
         if b <> 0 then display x = 0
         else display a=b=c=0
       end block
  end block
if a <> 0 then
  begin block
    if b^2 > -4ac then display x = (-b +- sqrt(b^2 -4ac))/2a
    else display roots are not real
  end block
```

Write a program that searches through an array sequentially and returns the index of the desired value and the number of lookups made.Write a program that will find the sum of the first n terms of a Tailor series.Write a program that will convert a temperature from Celsius to Fahrenheit and vice versa.Write a program that generates n unique random numbers.

[1] Don't let this definition of square root mislead you: `The square root of a number is that number that when it is squared gives the number.` Since the $(-10)^2$ is 100 it leads you to believe that the square root of 100 is -10 or 10. The $\sqrt{100} = 10$, it is not -10.

Problem 3.2 Make a list of 10 different random numbers
Design an algorithm to will construct an array of 10 unique random whole numbers.

Solution— Make a list of 10 different random numbers

```
numbers : array [1..10] of number

function insert (num, cnt)
  begin block
    set found = false
    for i ranges from 1 to cnt do
      if numbers[i] = num then set found = true
    if found <> true then
      begin block
        set cnt = cnt + 1
        set numbers[cnt] = num
      end block
    return set insert = cnt
  end block
```

```
set cnt = 0
repeat
  set x = random
  cnt = insert(num, cnt)
until cnt = 10

Write a program tat will allow the user to calculate the sum,
dot (http://en.wikipedia.org/wiki/Dot_product) product
or cross product (http://en.wikipedia.org/wiki/Dot_product) of tho vectors.
```

□

Problem 3.3 Create a list of 10 random integers in a tighter range
Now redesign your the algorithm to construct an array of 10 unique random whole numbers in the range $[A, B)$, for example, let the range be numbers from the set $\{17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28\}$ represented by the range $[17, 29)$. Note that (1) the range excludes 29, and that there are enough different numbers to enable the construction of at least 10 different numbers.

Problem 3.4 Create a list of 10 random integers
Now redesign your the algorithm to select 10% of the numbers from an array of N unique random whole numbers in the range $[A, B)$, for example, let the range be numbers from the set $\{1, 2, 3, \ldots, 100\}$ denoted by the range $[1, 101)$. Note that (1) the range excludes 101, and that there are enough different numbers to enable the construction of at least 10% of the numbers.

Problem 3.5 Water flow
Design an algorithm that will perform water flow rate calculations given that

Volume of water (kl)	*Cost (R/kl)*
0 to 80.0	*0.90*
80.1 to 120.0	*1.15*
120.1 or more	*1.25*

Algorithm

```
get volume from user
if volume > 120 then set cost = (volume-120)x1.25 + 40x1.15 + 80x0.90
if volume > 80 and volume <=120 then set cost = (volume-80)x1.15 + 80x0.90
if volume <= 80 then set cost = volume x 0.90
display volume water costs cost
```

3.1 Dates and days

Calculations with dates and clocks are common applications of modular arithmetic.

Problem 3.6 Given the year, calculate if it is a leap year or not.
If the year is divisible by 400 *it is a leap year. If it is divisible by* 100 *and not by* 400 *it is not a leap year. The years* 1700, 1800 *and* 1900 *are not leap years, but* 1600, *and* 2000 *are leap years. Of the remaining years, every year that is not divisible by* 100 *but is divisible by* 4 *is a leap year. For example* 2008, 2012, 2016 *are leap years.*

Solution— Calculating leap years
The leap year status is determined by some if statements.

```
def isLeapyear(year):
  if year % 400 == 0:
    return True
  if year % 100 == 0:
    return False
  if year % 4 == 0:
    return True
  else:
    return False
```

A shorter version is:

```
def isLeapyear(year):
  return year % 400 == 0 or year % 4 == 0 and year % 100 != 0
```

———— □

Problem 3.7 Number of days in the months
Calculate the number of days in the months of a given the year.

Solution— Days in a month

```
daysinmonth = [30 + (month + month//8 ) % 2 for month in range(1, 13)]
\end{lslisting}
gives
\begin{lslisting}
[31, 30, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

All that still needs to be done is to adjust for February. What we do is to replace **daysinmonth[1]** by **28 + int(isLeapyear(year))]. The** \lstinline**int**(True)! is 1 and **int(False)** is 0. The complete code for the days in the months follows.

```
def daysInmonth(year):
  days = [30 + (month + month//8 ) % 2 for month in range(1, 13)]
```

```
  days[1] = 28 + int(isLeapyear(year))
  return days
```

———— □

Problem 3.8 Day of the week
A popular application of modulo arithmetic concerns calculating the day of the week. Given the Gregorian calendar date determine the day of the week.

Solution— Day of the week
One start by counting the difference in days by calculating the number of years, months and days from a base date for which you know the day of the week, and then simply getting the day of the week using *modulo* 7. We call this number the *daydate*.

It does not matter what the starting date is, if the days counted are correct then calculate daydate for today and get **today = daydate % 7**. Since we know what day of the week today is, henceforth **today** represents that day of the week. Suppose today is a Tuesday, and **today**= 3, then 3 is a Tuesday, 4 is a Wednesday, 5 is a Thursday, 6 is a Friday, 0 is a Saturday, 1 is a Sunday, 0 is a Monday. Now we calculate daydate, given the parameters *year*, *month*, *day*

```
def daydate(year, month, day):
  mdays = [30 + (m + m//8 ) % 2 for m in range(1, 13)]
  mdays[1] = 28 + int(isLeapyear(year))
  days = 1
  mdays = [0]+mdays[:-1]

  for m in range(month):
    days += mdays[m]
    mdays[m] = days

  days = day + mdays[month-1]
  years = year -1600 # Our base date is 1601-01-01
  days += years * 365
  leapcenturies = ((years -1) // 100 -16)//4 + 1 # Add leap years
  days += leapcenturies
  weekdays = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
  return weekdays[days % 7]
```

———— □

Problem 3.9 Calculate the Gregorian calendar date of Easter Sunday.
The rules for determining the date of Easter or Passover are quite easy. During 325 ACE in Nicaea the first ecumenical council of the catholic church determined the date of Easter as the first Sunday after the full moon after March 21—the spring equinox in the Northern hemisphere. The calculations were based on the Julian calendar and since 1582 and 1752 in the United Kingdom and the United States of America on the Gregorian calendar. The date of Easter varies between March 22 and April 25.

Solution— The Gauss 1812 algorithm for calculating Easter

The algorithm of Gauss for calculating Easter

```

a = year %19
b = year % 4
```

```
c = year % 7
k = floor (year/100)
p = floor ((13 + 8k)/25)
q = floor (k/4)
M = (15 −p + k −q) % 30
N = (4 + k −q) % 7
d = (19a + M) % 30
e = (2b + 4c + 6d + N) % 7
Gregorian Easter is 22 + d + e March or d + e −9 April
if d = 29 and e = 6, replace 26 April with 19 April
if d = 28, e = 6, and (11M + 11) % 30 < 19, replace 25 April with 18 April
For the Julian Easter in the Julian calendar M = 15 and N = 6 (k, p, and q are unnecessary)
```

——— □

Solution— Easter in a spreadsheet
The following spreadsheet formula calculates the date of Easter Sunday.

```
=ROUND(DATE(A1,4,1)/7 + MOD(19*MOD(A1,19) -7, 30)*14%, 0)*7 -6
```

——— □

Problem 3.10 Birthdays on the same day
Suppose that a room is occupied by some people. If there is only one person in the room nobody will share his birthday so the probability q that there is nobody to share his birthday is $q_1 = \frac{365}{365} = 1$. When there are two people in the room the probability that no two people share a birthday becomes $q_2 = \frac{365\times364}{365^2} = 0.99\overline{72602739}$.[2] In general if there are n people in the room the probability that no pair of people share a birthday becomes

$$q_n = \frac{\prod_{r=1}^{n} 365 - r + 1}{365^n}.$$

The problem is to discover how many people must be in the room for the probability that there are two people that have a birthday on the same day are in the room, i.e. find an n such that $p = 1 - q_n > 0.5$.

Solution— Birthdays on the same day
Simply calculate the running fraction q_n to discover at which n it becomes less that a half.
——— □

[2] Notice how we write the repeating decimal $0.99\ 72602739\ 72602739\ldots$.

4 Problems with solutions in python

4.1

Problem 4.1 $S = \sum_{r=1}^{N} r$
Suppose that the integer N is given. Construct an algorithm to add all the integers from 1 up to and including N, i.e. calculate the sum S,

$$S = \sum_{r=1}^{N} r = 1 + 2 + \ldots + N.$$

Solution— $S = \sum_{r=1}^{N} r$
We first initialize a variable called `S` to zero and subsequently add the values of `r` running from 1 to N one by one.

```
N = 10
S = 0
for r in range(1, N+1):
  S += r
print ("Sum␣of␣integers␣from␣1␣to␣%d␣is␣%d" % (N, total))
```

A much tidier solution, more in the spirit of Python style, is:

```
N = 10; S = sum([r for r in range(1, N+1)])
```

The `for r in range(1,N+1)` generates each `r` which is then agglomerated into a list by the surrounding brackets `[...]`. The function `sum` is built in, and yields the sum of the numbers in a list.

If you were Carl-Fried Gauss, at the age of about ten years, you would no doubt have noticed that the sums

$$1, 2, \ldots, N-1, N$$

$$+$$

$$N, N-1, \ldots, 2, 1$$

add up to

$$N+1, N+1, \ldots, N+1,$$

and that half of this is the required sum, and further you would have seen that there are N of these $N+1$ terms and you have given the answer in a flash as

$$\frac{N \times (N+1)}{2}.$$

Gauss would have programmed the requested result as

```
N = 10; S = N*(N+1)/2
```

Notice that despite the division by two, the result is a whole number. ——— □

Problem 4.2 Sum a list of integers
Suppose an array of numbers is given in the list L, e.g.

```
L = [17, 23, 29, 31, 37, 41, 47]
```

Sum the numbers in the list. You may assume that all the entries in the list are integers.

Solution— Sum the numbers in a list
Suppose L is ready to use, then the simple Python answer is

```
S = sum(L)
```

——— □

Problem 4.3 Sum numbers stored in a file
On the other hand it is more likely that the numbers reside in some file, "with each number on its own line",[1] and where the file is displayed by an editor as:

```
17
23
29
31
37
41
47
```

Solution— Sum numbers stored in a file
Let us now read the list of numbers and add them one by one.

```
f = open ("listofintegers.text", 'r')
L = f.readlines()
f.close()
S = 0
for r in L:
  S += int(r)
print ("Sum␣of␣integers␣in␣the␣list␣is␣%d" % S)
```

The **f.readlines()** bunches the substrings separated by newline characters together into one string. This string still has to be converted into an integer by the function **int(r)** before it can be summed.
——— □

Solution— Sum numbers stored in a file—Python style
A more Pythonic way to program this follows

```
print ("Sum␣of␣integers␣in␣the␣list␣is␣%d" %
       sum([int(r) for r in open("lines.text",'r').readlines()]) )
```

Since the number r is read in as a string from the file we turn it into an integer using the built in `int` function so that it may be summed. ——— □

[1] In actual fact it means that the file has each integer terminated by the newline character '**\n**'. So the list in our example is stored in a file as the single string **"17\n23\n29\n31\n37\n41\n47\n51\n53\n"**. Check that the file has altogether 27 characters. The file does not end with a character for the end of the file.

Problem 4.4 Sum numbers recursively
The next version of the $\sum_{r=1}^{N} r$ problem sets it in a new angle. Sum the numbers 1 to N using a recursive function called by invoking ***add(N)****. A recursive function is defined in terms of itself.*

Solution— Sum numbers recursively
The idea is that when called as **add(1)**, the function yields the sum 1. When **add(N)** is called with a value of N exceeding 1, the result is defined as the sum of that value of N added to the value of **add(N-1)**. At each step the function calls itself, but with an argument that is smaller by 1. These contractions of the arguments yield what is known as a *contraction mapping* which eventually turns the argument into 1 which will then be added to 2, ...until finally N is added. So, first test if N is 1 and otherwise return $N + add(N - 1)$.

```
def add(N):
  if N == 1:
    return N
  else:
    return N + add(N-1)

print (add(10))
```

——— □

Problem 4.5 Add using inc and dec
Add the two positive integer arguments A and B with ***add(A,B)*** *given the two functions* ***inc(N)*** *which returns $N + 1$ and* ***dec(N)*** *which returns the value of $N - 1$.*

Solution— Add using inc and dec
The idea is to add B to A in the following manner. First check if B is zero, in which case the value of A is returned. Otherwise return the value of the **add** of A increased by 1, and B decreased by 1. In other words return the value of **add(inc(A), dec(B))**. This continues until the second argument becomes a zero when the value of the first argument is returned.

```
def add(A, B):
  if B == 0:
    return A
  else:
    return add(inc(A), dec(B))

print (add(3, 7))
```

Notice how the second argument is contracted until it becomes zero, at which stage the first argument has been incremented by one enough times to have reached $A + B$. ——— □

Problem 4.6 Multiply without $\times$-operator
Show how to multiply two positive integers numbers using addition only. The function ***mult(A, B)*** *must yield the value of $A \times B$.*

Solution— Multiply with addition
The idea is to add the value of A repeatedly until it has been added B times. The function returns zero when the value of B is zero, and returns the value of A when B is one, otherwise the function returns **mult(A, B -1) + A**. We have multiplied without using addition. The Python code follows

```
def mult (A, B):
  if B == 0:
```

```
    return 0
  elif B == 1:
    return A
  else:
    return add(mult(A, dec(B)), A)

print (mult(3,7))
```

———— □

Problem 4.7 Recursive exponentiation
Show how to exponentiate a positive integer to a positive integer power using multiplication only.

Solution— Recursive exponentiation
We will call the function **power(A, B)** and it will yield the value of A^B which is $A \times A \times A \ldots A$, B times. The idea is to multiply the value of A repeatedly until it has been multiplied B times. The function returns 1 when the value of B is zero, and returns the value of A when B is one, otherwise the function returns **power(A, B -1) * A**. We have raised A to the power B using multiplication. The Python code follows

```
def power (A, B):
  if B == 0:
    return 1
  elif B == 1:
    return A
  else:
    return mult(power(A, dec(B)), A)

print (power(3,7))
```

———— □

These recursive solutions have practical bounds—they just do not work on a computer with a small memory stack when the arguments get large. Our **power** function will soon run out of memory on even the largest of machines.

Problem 4.8 More efficient exponentiation
*Calculate **power(A, B)**= A^B more efficiently.*

Solution— Using logarithms to calculate exponents
The simple answer is to use logarithms. Since $\log A^B = B \log A$, we write $A^B = \exp(B \log A)$ and get the result quite quickly. The snag is that the accuracy of the log and exp is usually limited on computers to about 7 digits for single precision or to 18 digits for double precision—Python uses double precision reals. This may be a problem when doing exponentiation in cryptographic calculations where numbers with precisions of 256 or 512 binary digits are used.[2] We give a version of this function

```
def power (A, B):
  from math import log, exp
  return exp(B*log(A))

print (power(3, 100))
```

[2]Since $\lceil \log_{10} 2^5 12 / \log_{10} \rceil = 107$, 512 bits occupy 107 decimal digits.

Python has a built-in operator for exponentiation, so we could simply have programmed

```
print (3**100)
```

Python also has a built-in function for exponentiation, so the following is also a good way

```
print (pow(3,100))
```

———— □

Problem 4.9 An efficient method for exponentiation
What is an efficient way of doing multiple digit exponentiation in Python, without the advantage of the built-in operator or function called ***pow(A, B)****?*

Solution— A fast method for exponentiation
There are several fast methods. We will show a fast method that uses an interesting trick for exponentiating A^B where both numbers are positive integers. We must calculate $A^B = A \times A \times \ldots \times A$, B times, requiring $B-1$ multiplications. Suppose we were asked to calculate A^{32}. This could have been done in less multiplications by

$$\begin{aligned} A^2 &= A \times A, \\ A^4 &= A^2 \times A^2, \\ A^8 &= A^4 \times A^4, \\ A^{16} &= A^8 \times A^8, \\ A^{32} &= A^{16} \times A^{16}. \end{aligned}$$

Instead of 31 multiplications 5 are sufficient. What do we do when B is not a power of 2? If $B = 33$ the sequence of multiplications becomes

$$\begin{aligned} A^2 &= A \times A, \\ A^4 &= A^2 \times A^2, \\ A^8 &= A^4 \times A^4, \\ A^{16} &= A^8 \times A^8, \\ A^{32} &= A^{16} \times A^{16}. \end{aligned}$$

At some stage one more multiplication by A is needed, since $A^{33} = A^{32} \times A$. If B were 34 then one more multiplication by A^2 is needed.

The table shows the number of multiplications needed for calculating some powers of A
The power is calculated as follows: start with a **product** of 1. Set **mul** to **A**. If B is odd then multiply **product** by **mul** and replace B by $B//2$. Square and replace **mul *= mul**. Stop when $B = 0$.

```
def power (A, B):
  mul = A
  product = 1
  while B != 0:
    if B % 2 == 1:
      product *= mul
    mul *= mul
    B //= 2
```

Table 4.1: Number of multiplications needed for A^n. Note: Table has not been checked for correctness.

Power of A	Number of multiplications	How
A^2	1	$A \times A$
A^3	2	$A \times A \times A$
A^4	2	$A^2 \times A^2$
A^5	3	$A^2 \times A^2 \times A$
A^6	3	$A^2 \times A^2 \times A^2$
A^7	2	$A^2 \times A^2 \times A^3$
A^8	3	$A^4 \times A^4$
A^9	4	$A^4 \times A^4 \times A$
A^{10}	4	$A^4 \times A^4 \times A^2$
A^{11}	5	$A^4 \times A^4 \times A^3$
A^{12}	4	$A^4 \times A^4 \times A^4$
A^{13}	4	$A^4 \times A^4 \times A^5$
	$\vdots$	$A^4 \times A^4 \times A^7$
A^{16}	4	$A^8 \times A^8$
A^{32}	5	$A^{16} \times A^{16}$
A^{33}	6	$A^{32} \times A$
A^{34}	6	$A^{32} \times A^2$

```
  return product

print (power(3,7))
```

This is a very efficient algorithm but Python's built-in **pow(A, B)** has a much better performance. How does Python do it? ——— □

Problem 4.10 $N!$ **recursively**
Calculate the factorial function, $N! = 1 \times 2 \times 3 \ldots \times N$ *using recursion.*

Solution— $N!$ **recursively**
When the argument of **factorial(N)** is 1 or 0 return a 1 otherwise the factorial of N is **N * factorial(N-1)**.

```
def factorial (N):
  if N < 2:
    return 1
  else:
    return N * factorial(N-1)

print (factorial(6))
```

——— □

Problem 4.11 $N!$ **iteratively**
Calculate the factorial function, $N! = 1 \times 2 \times 3 \ldots \times N$ *without using recursion.*

Solution— $N!$ **iteratively**
The non-recursive solution of $N!$ applies the definition directly. Build the product that we have named **product** starting from 1. Multiply this by each value $r \in [2..N] \equiv [2..N+1)$.

```
1 def factorial (N):
2   product = 1
3   if N > 1:
4     for r in range(2, N+1):
5       product *= r
6   return product
7
8 print (factorial(6))
```

In Line 5 the "times and becomes" operator "**product *= r**" requires less typing than "**product = product * r**", so it is preferred. ——— □

Problem 4.12 Binomial coefficients
Calculate "choose r out of n objects",

$$\binom{n}{r} = {}_nC^r = \frac{n!}{r!(n-r)!}$$

We will discuss binomial coefficients later in Section 2.1.

Solution— Binomial coefficients using factorials
A very slow way of calculating $\binom{n}{r}$ is by following the formula slavishly, i.e. divide $n!$ by $r!(n-r)!$.

```
1 def factorial (n):
2   product = 1
3   if n > 1:
4     for r in range(2, n+1):
5       product *= r
6   return product
7
8 def binom(n, r):
9   return factorial(n)/(factorial(r)*factorial(n-r))
10
11 print (binom(52,4))
```

——— □

Solution— Binomial coefficients using some mathematics
Since $n! = (n \times n-1 \times \ldots \times n-r+1) \times (n-r)!$, first dividing $n!$ by $(n-r)!$ yields

$$n^{\underline{r}} \quad = \quad n \times n-1 \times \ldots n-r+1$$

The symbol $n^{\underline{r}}$ is called a falling factorial and is an abbreviation for $n \times n-1 \times \ldots n-r+1$. What remains, is to calculate $n^{\underline{r}}/r!$. Noticing that the one of the numbers in the product $n \times n-1$ is divisible by 2 and next seeing that one of the factors in $n \times n-1 \times n-2$ must be divisible by 3 leads us to the idea of calculating the binomial coefficient as an accumulating product. First put **binom = 1** and then letting n decrease by 1 in each iteration, and r starting at 0 increase by 1 in each iteration accumulate the binomial product as

```
1   binom = n
2   for r in range(2, R+1):
3     n -= 1
4     binom = binom * n // r
```

This method uses $r-1$ multiplications and divisions and the product never gets bigger than the result. The method in Solution 4.1 gets the factorials with close to $2n$ multiplications and one division. Some simple arithmetic provides a method that works faster for any $n > r$.

```
def binom(n, r):
  binom = n
  for k in range (2, r+1):
    n -= 1
    binom = binom * n // k

  return binom
```

———— □

Another very interesting solution based solely on additions is presented in Solution2.1

Problem 4.13 Counting digits in numbers
Count the total number digits of each factorial from 1 *to* 100000.

The ultimate Pythonic way to write the factorial function is to use a generator. The generator yields values one by one, discarding the previous values.

Solution— $N!$ with a generator
Our generator for **factorials** is a simple variation of the iterative version of the function. When control arrives at the **yield** command the calculation is suspended and the environment is saved so that it can be restored on reentry.

```
def factorials (N):
  product = 1
  num = 1
  while num <= N:
    yield product
    num += 1
    product *= num

n = 1000
count = 0
for f in factorials(n):
  count += len(str(f))

print (count)
```

———— □

Many problems lend themselves to recursive solutions. These recursive solutions often lend themselves to be readily turned into iterative solutions, e.g. factorials, Fibonacci numbers and sorting. Let us consider those together with their more practical iterative solutions.

Problem 4.14 Fibonacci numbers
The sequence of numbers $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$ *are called the Fibonacci numbers. Starting with* 0 *and* 1 *next number in the sequence is the sum of the previous two.*

Solution— Fibonacci numbers recursively
Fibonacci numbers are defined recursively as $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ so the recursive code is pretty easy.

```
def F (N):
    if N == 0:
        return 0
    elif N == 1:
        return 1
    else:
        return F(N-1) + F(N-2)

print (F(50))
```

———— □

Problem 4.15 Fibonacci numbers
There is no need to implement Fibonacci numbers recursively and the recursive code runs excessively slowly. Create an iterative solution.

Solution— Fibonacci numbers iteratively
If N is zero return a zero; if N is 1 return a 1, and otherwise use two variables to hold the oldest and most recent values and return their sum after updating them.

```
def F (N):
  if N <= 0:
    return 0
  n = 0
  ancient = 0
  old = 1
  while n < N:
    ancient, old = old, ancient + old
    n += 1
  return ancient

from time import time
for n in range (2, 101):
  start = time()
  print("F(%d) = %d, took %.5f s." % (n, F(n), time() -start))
```

This code runs in linear time, much faster than the recursive solution that runs in exponential time. ———— □

Problem 4.16 A generator for Fibonacci numbers
Program a generator version of Fibonacci based on the iterative solution.

Solution— A generator for Fibonacci numbers
The trick is to figure out where to put the **yield** command.

```
def F (N):
  n = 0
  ancient = 0
  old = 1
  while n <= N:
    yield ancient
    ancient, old = old, ancient + old
    n += 1
  return ancient
```

```
10
11 from time import time
12 for n in range (2, 101):
13   start = time()
14   print("F(%d) = %d, took %.5f s." % (n, F(n), time() -start))
```

———— □

Problem 4.17 Sorting a list
Construct a program to sort a list L of uniform—all the elements are numbers or all of them are strings—items from smallest to largest. The sorting algorithm works as follows. Remove the first element of the list and then concatenate the collection of the elements smaller than it, the selected element, and the collection of all those greater or equal to it.

Solution— Sorting a list

```
1 def sort(L):
2   if not L:
3     return L
4   it, rest = L[0], L[1:]
5   smaller = [item for item in rest if item < it]
6   greaterEqual = [item for item in rest if item >= it]
7   return sort(smaller) + [it] + sort(greaterEqual)
8
9 from random import random
10
11 N = 100
12 L = [int(random()*N) for x in range(N)]
13 print (sort(L))
```

———— □

Recursive solutions are sometimes easier to conceive. An example whose iterative solution is far from obvious is the towers of Hanoi problem. The recursive solution is easy.

Problem 4.18 The towers of Hanoi
Program the towers of Hanoi problem recursively. Three pins are presented; one has a stack of disks piled on top of one another from the smallest on top to the largest at the bottom. The other two pins are empty. The three pins are called '`from`', '`help`' and '`to`'. The pins are stacked on the `from` tower. The problem is to move the disks one-by-one such that a larger disk never lies on top of a smaller disk from tower to tower until all the disks are in the correct order on the `to` tower.

Solution— The towers of Hanoi
Assume that there are N disks to be manipulated. The idea of the solution is very simple. How can the bottom disk be moved? It can only be moved if nothing lies on top of it *and* the pin you are moving it to is clear. How can this be brought about? Obviously $N-1$ pins need to be moved to the `help` disk. Once these disks are out of the way the largest disk—lying on the `from` pin—can be moved to the `to` pin. What remains to be done? The $N-1$ disks on the `help` pin must be moved onto the biggest disk now lying on the `to` pin. In all the recursive procedures above the problem was reduced to doing something for one item, and then repeating the procedure for $N-1$ items. So we have solved the towers of Hanoi problem. The next step is

the most difficult. What parameters will the procedure need? The value N seems to be essential. Each separate tower needs its own parameter since our procedure must know at each step from where, to where a disk must be moved and which tower remains.[3] We will call the procedure **hanoi** and give it four parameters.

```
def hanoi (N, from, help, to):
  if N == 1:
    print ("move␣disk␣from␣%s␣to␣%s" % (from, to))
  else:
    hanoi (N-1, from, to, help)
    print ("move␣disk␣from␣%s␣to␣%s" % (from, to))
    hanoi (N-1, help, from, to)

hanoi (3, "From", "Help", "To")
```

———— □

Polynomials are interesting for many reasons. They form the basis of our number system. They may be used to approximate functions as accurately as needed. Polynomials are useful because they are easy to differentiate and integrate. They play an important role in algebra.

Problem 4.19 Evaluating a polynomial
A univariate polynomial of degree n in x may be written as

$$P_n(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1} + a_nx^n.$$

Univariate polynomials are usually referred to simply as polynomials. By writing the same terms the other way round

$$P_n(x) = a_nx^n + a_{n-1}x^{n-1} + \ldots + a_2x^2 + a_1x + a_0$$

we get a different view of the same polynomial and in this form putting the variable $x = 10$ and restricting the coefficients to the decimal digits $d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the polynomial

$$P_n(10) = d_n10^n + d_{n-1}10^{n-1} + \ldots + d_210^2 + d_110 + d_0$$

represents the decimal number written as

$$d_nd_{n-1} \ldots d_2d_1d_0$$

Solution— Evaluating a polynomial
Suppose the coefficients are stored in an array a. Write code for evaluating the polynomial $P_n(x)$, called by **P(n, a, x)**. The value of n may be determined by inside the function using **n = len(a)**, but the meaning of the call is clearer if the n is passed as an argument. It may be coded as follows.

```
def P(n, a, x):
  pol = a[0]
  xpowern = 1
  for i in range(1, n+1):
    xpowern *= x
    pol = pol + a[i]*xpowern
```

[3] Perhaps it is possible to can get away with naming only the `from` and `to` towers in the list of parameters.

```
  return pol

from time import time
a = [1, 4, 6, 4, 1]
n = 4
for x in range(0, 11):
  start = time()
  print("P%d(%f) = %f, took %.5f s." % (n, x, P(n,a,x), time() -start))
```

Another way of writing the polynomial is to number the coefficients differently as

$$P_n(x) = a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x^2 + a_{n-1}x + a_n.$$

Rewrite the code corresponding to the array written with the coefficient of the x^n first. ——— □

Solution— Faster evaluation of a polynomial

```
def Pn(n, a, x):
  pol = 0
  xpowern = 1
  for i in range(n, -1, -1):
    pol = pol + a[i]*xpowern
    xpowern *= x
  return pol

from time import time
a = [1, 4, 6, 4, 1]
n = 4
for x in range(0, 11):
  start = time()
  print("P%d(%f) = %f, took %.5f s." % (n, x, Pn(n,a,x), time() -start))
```

——— □

These obvious first attempts both use $2n$ multiplications. Rewrite the code to use n multiplications.

Solution— Horner's rule for polynomial evaluation
The insight is to start at the back and do only one multiplication per iteration. The idea is called Horner's rule and uses only n multiplications for a polynomial of degree n.

```
def P(n, a, x):
  pol = a[n]
  for k in range(n-1, -1, -1):
    pol = a[k] + pol*x
  return pol

from time import time
a = [1, 4, 6, 4, 1]
n = 4
for x in range(0, 11):
  start = time()
  print("P%d(%f) = %f, took %.5f s." % (n, x, P(n,a,x), time() -start))
```

———— □

Problem 4.20 Representation of numbers by polynomials
A number in base ten, e.g. 1234567 *may be represented as the array* ***digits = [1, 2, 3, 4, 5, 6, 7]****. Use the polynomial algorithm to find the value of the number.*

Solution— Representation of numbers by polynomials

```
def P(n, a, x):
  pol = a[n]
  for k in range(n-1, -1, -1):
    pol = a[k] + pol*x
  return pol

from time import time
digits = [1, 2, 3, 4, 5, 6, 7]
n = 6
for x in range(0, 11):
  start = time()
  digits.reverse()
  print("P%d(%f) = %f, took %.5f s." % (n, x, P(n,digits,x), time() -start))
```

———— □

Problem 4.21 Addition of polynomials
Given two polynomials, $P_n(x)$ *and* $Q_n(x)$ *write a program to add them, i.e. write code to produce*

$$R_n(x) = P_n(x) + Q_n(x).$$

Problem 4.22 Subtraction of polynomials
Given two polynomials, $P_n(x)$ *and* $Q_n(x)$ *write a program to subtract them, i.e. write code to produce*

$$R_n(x) = P_n(x) - Q_n(x).$$

Problem 4.23 Multiplication of polynomials
Given two polynomials, $P_n(x)$ *and* $Q_n(x)$ *write a programmed to multiply them. i.e. write code to produce*

$$R_n(x) = P_n(x) \times Q_n(x).$$

Problem 4.24 Division of polynomials
Given two polynomials, $P_n(x)$ *and* $Q_n(x)$ *write a programmer to divide them. i.e. write code to produce*

$$R_n(x) = P_n(x)/Q_n(x).$$

Problem 4.25 Polynomials representing numbers
The polynomial over the variable b *with coefficients conveniently called digits* $d_i \in [0..b) = \{0, 1, ..., b-1\}$*, may be used to represent any number in base* b

$$P_n(b) = d_n b^n + d_{n-1} b^{n-1} + \ldots + d_2 b^2 + d_1 b + d_0$$

Write code to convert a number $P_n(b)$ *in base* b *to a number* $Q_n(r)$ *in base* r *with digits called* $s_i \in [0..c) = \{0, 1, \ldots, c-1\}$*. with the same value. The problem is to convert the polynomial* P

$$P = d_n b^n + d_{n-1} b^{n-1} + \ldots + d_2 b^2 + d_1 b + d_0$$

into the polynomial Q

$$Q = s_n r^n + s_{n-1} r^{n-1} + \ldots + s_2 r^2 + s_1 r + s_0,$$

so that $P = Q$.

Solution— Polynomials representing numbers
This turns out to be easy. We are given the polynomial P and its coefficients and are asked to extract the "digits", s_i from it. Since $P = Q$, we can do arithmetic on Q as the proxy of P, so, first copy Q into P. Consider Q,

$$Q = s_n r^n + s_{n-1} r^{n-1} + \ldots + s_2 r^2 + s_1 r + s_0,$$

All the terms of Q excepting s_0 are divisible by r, so it is clear that we can calculate s_0 by

$$s_0 = Q \mod r$$

and then subtract—the digit—s_0 from Q, divide the result with r and replace Q with the result

$$Q = \frac{Q - s[0]}{r} = s_n r^{n-1} + s_{n-1} r^{n-2} + \ldots + s_2 r + s_1.$$

In Python we do this with the two lines

```
s[0] = Q % r
Q //= r
```

The steps start repeating. At this stage, all the terms of Q excepting s_1 are divisible by r, so it is calculated

$$s_1 = Q \mod r$$

and then subtract the digit s_1 from Q, divide the result with r and replace Q with that

$$Q = \frac{Q - s[1]}{r} = s_n r^{n-2} + s_{n-1} r^{n-3} + \ldots + s_3 r + s_2.$$

In Python we do this with the two lines

```
s[1] = Q % r
Q //= r
```

The process continues while $Q \neq 0$. The code becomes

```
s = [0]*(n + 1)
i = 0
while Q != 0:
    s[i] = Q % r
    Q //= r
    i += 1
```

——— □

5 Some problems from various sources

Problem 5.1 Flatten a binary search tree rightwise
You are given an unbalanced binary search tree. Flatten it into a binary search tree with only right children.

Problem 5.2 Flatten a binary search tree leftwise
You are given an unbalanced binary search tree. Flatten it into a binary search tree with only left children.

Problem 5.3 Balance a binary search tree
Balance a binary search tree by rightwise flattening and then turn this into a balanced tree.

Problem 5.4 Find the sum of the numeric keys in a path from the root to a given node in a binary search tree

Problem 5.5 Find an item in a sorted list that has been cut once
Suppose you have a sorted list of length n that has been rotated by k places.

Problem 5.6 Find kth non-repeated char

Problem 5.7 Find nth largest number in a stream of numbers

Problem 5.8 Reverse a linked list

Problem 5.9 Reverse a doubly-linked list

Problem 5.10 Find Pythagorean triples
Find the sum of all Pythagorean integer triples a, b, c with $c^2 = a^2 + b^2$ where $0 < a \leq N$, $0 < b \leq N$ and $0 < c \leq N$ and $N \leq 1000$.

Problem 5.11 Given the time give the angle in radians between the short hand and the long hand of an analogue clock

Problem 5.12 Program Snakes and ladders

Problem 5.13 Merge three sorted lists

Problem 5.14 Merge n sorted lists

Problem 5.15 Find nearest common ancestor of two nodes in a binary search tree

Problem 5.16 Find common numbers in two sets
Given a set of 100 *integers and another set of* 1000000 *integers. Find their intersection.*

Problem 5.17 Find the number of days between two dates

Problem 5.18 Is a given undirected graph a tree or not?

Problem 5.19 Is a given undirected graph a tree or not?

Problem 5.20 Find common data from two very large files that do not fit into memory simultaneously

Problem 5.21 Print an n-ary tree in breadth first order

Problem 5.22 Checking if two binary search trees are similar or not

Problem 5.23 Use command line code to get the common data from three files

```
cat f1 f2 | sort | uniq -d > tmpfile;
cat f3 tmpfile | sort | uniq -d
```

Problem 5.24 You have a jar with balls uniquely numbers from 1 to 100. You have removed 99 of these balls. What is the number of the remaining ball?

Problem 5.25 Find telephone numbers like 021-959-3004, or 084-455-3885 in all .html files
The find below finds numbers of the form +27-123-123-1234.

```
find -name *.html | xargs grep "<\s[+]{0,1}\d{1,2}[-]\d{3}[-]\d{3}[-]\d{4}>/"
```

Problem 5.26

Problem 5.27

1. The ancient Babylonians had a fast formula for calculating the square root of any number. This algorithm was described by Heron of Alexandria in his book called Metrica near the dawn of the Christian era about 100 BCE–100 ACE.

   Apply Heron's formula to calculate $\sqrt{A}$:
   "(1) Start with a guess g.
   (2) If it g^2 is close enough to A, then use g as the square root.
   (3) Otherwise $g \leftarrow \frac{g+A/g}{2}$.
   (4) Repeat from step (2)."

   In books on numerical mathematics Heron's formula to calculate $\sqrt{A}$ is stated as:
   Initialize $x_0 = 1$, and
   iterate $x_{n+1} = (x_n + A/x_n)/2$

until $|x_{n+1}^2 - A| < \varepsilon$.
Note that careful application of this formula in a program does not require subscripts.

2. Find the first repeated element in an array.

3. Find the all the repeated elements in an array.

4. Find the largest and and second largest elements in a list of integers.

5. Swap the largest and smallest elements in a list of integers.

6. Reverse the words of a sentence.

7. Convert decimal number to Roman number.

8. Remove the n-th element from a linked list.

9. Remove the last element from a linked list.

10. Remove the first element from a linked list.

6 Conclusion

One of my favourite books on programming Jon Bentley's **?** *Programming Pearls* is stacked with examples.

Bibliography

John Louis Bentley. *Programming Pearls*. Addison-Wesley, 1986.

Thomas H Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

Thomas H Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 3rd ed. edition, 2009.

Thomas H Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 4th ed. edition, 2022.

Marvin Zelkowitz. *PL/I Programming with PLUM*. Paladin, 1976.

7 AppendixA

7.1 The Standard International (SI) symbols for numbers

Prefix	Symbol	1000^m	10^n	Decimal	Short scale	Long scale	Since
quetta	Q	1000^{10}	10^{30}	1 and thirty zeros	Nontillion	Sextrillion	2022
ronnna	R	1000^{9}	10^{27}	1 and twenty seven zeros	Octillion	Quintrillion	2022
yotta	Y	1000^{8}	10^{24}	1000000000000000000000000	Septillion	Quadrillion	1991
zetta	Z	1000^{7}	10^{21}	100000000000000000000	Sextillion	Trilliard	1991
exa	E	1000^{6}	10^{18}	1000000000000000000	Quintillion	Trillion	1975
peta	P	1000^{5}	10^{15}	1000000000000000	Quadrillion	Billiard	1975
tera	T	1000^{4}	10^{12}	1000000000000	Trillion	Billion	1960
giga	G	1000^{3}	10^{9}	1000000000	Billion	Milliard	1960
mega	M	1000^{2}	10^{6}	1000000	Million		1873
kilo	k	1000^{1}	10^{3}	1000	Thousand		1795
hecto	h	$1000^{2/3}$	10^{2}	100	Hundred		1795
deca	da	$1000^{1/3}$	10^{1}	10	Ten		1795
		1000^{0}	10^{0}	1	One	—	
deci	d	$1000^{-1/3}$	10^{-1}	0.1	Tenth		1795
centi	c	$1000^{-2/3}$	10^{-2}	0.01	Hundredth		1795
milli	m	1000^{-1}	10^{-3}	0.001	Thousandth		1795
micro	µ	1000^{-2}	10^{-6}	0.000001	Millionth		1960
nano	n	1000^{-3}	10^{-9}	0.000000001	Billionth	Milliardth	1960
pico	p	1000^{-4}	10^{-12}	0.000000000001	Trillionth	Billionth	1960
femto	f	1000^{-5}	10^{-15}	0.000000000000001	Quadrillionth	Billiardth	1964
atto	a	1000^{-6}	10^{-18}	0.000000000000000001	Quintillionth	Trillionth	1964
zepto	z	1000^{-7}	10^{-21}	0.000000000000000000001	Sextillionth	Trilliardth	1991
yocto	y	1000^{-8}	10^{-24}	0.000000000000000000000001	Septillionth	Quadrillionth	1991
ronto	y	1000^{-9}	10^{-27}	0. twenty six zeros 1	Octtillionth	Quintillionth	1991
quecto	y	1000^{-10}	10^{-30}	0. twenty nine zeros 1	Nonillionth	Sextillionth	1991

Contents