Discrete Mathematics through the eyes of a Python programmer

Johan Nordlander
johan.nordlander@ltu.se
Dept. of Computer Science and Electrical Engineering
Luleå University of Technology

The purpose of this note is to establish some additional links between programming concepts and the different areas of Discrete Mathematics studied in the course MAM200. The exposition roughly follows the structure of Grimaldi's text book (Discrete and Combinatorial Mathematics, 5th ed.) chapters 1-6, with an emphasis on areas most relevant in MAM200.

Prerequisites assumed in this document are programming skills and understanding comparable to a passed grade in the course Introduction to Programming (SMD180). The favored programming language will be Python, although many of the techniques and idioms shown should be transferable to other modern languages without too much difficulty.

1. Fundamental Principles of Counting

In Python as well as in other imperative languages, combinatorial problems can easily be formulated in terms of for-loops. The rule of sum, for example, simply means running the respective loops in sequence:

```
for d in democrats:
    print "Possible president:", d
for r in republicans:
    print "Possible president:", r
```

The rule of product, on the other hand, corresponds to running *nested* loops:

```
for d in democrats:
    for r in republicans:
        print "Possible final candidates:", d, "against", r
```

The four major counting formulas listed in table 1.11 can be straightforwardly expressed as nested loops over integer ranges. Here follows an example that prints all possible ways to select 3 out of the N first non-negative integers, if order is relevant and repetitions are allowed.

```
for i1 in range(0,N):
    for i2 in range(0,N):
        for i3 in range(0,N):
            print i1, i2, i3
```

Should repetitions not be allowed, a test must be added around the **print** statement:

```
for i1 in range(0,N):
    for i2 in range(0,N):
        for i3 in range(0,N):
            if i1 != i2 and i2 != i3:
                print i1, i2, i3
```

If order is irrelevant we may choose to print only those combinations that occur in increasing order:

```
for i1 in range(0,N):
    for i2 in range(i1+1,N):
        for i3 in range(i2+1,N):
            print i1, i2, i3
```

Repetitions can be allowed in such a combination by slightly adjusting the inner ranges:

```
for i1 in range(0,N):
    for i2 in range(i1,N):
        for i3 in range(i2,N):
            print i1, i2, i3
```

Larger selections can be obtained by rewriting the examples with an appropriate number of nested loops. Moreover, working with integers is actually not a limitation here, since other types of objects can always be accessed by using the integers as list indices:

```
print democrats[i1], democrats[i2], democrats[i3]
```

Should the result need to be stored in a data structure rather than simply being printed, the simplest solution is to replace the print statement in the examples above with an update to an accumulating variable, as in

```
result = result + [ (i1, i2, i3) ]
```

For the more adventurous, the notion of *list comprehensions* might be an interesting alternative. See the Python tutorial, section 5.1.4, for more information on this feature.

Exercises

1.1 Complete text book exercise 1.3.34 as a pair of Python programs.

1.2 Implement the code in text book exercise 1.4.20 in Python and determine the result, both experimentally and formally.

2. Fundamentals of Logic

The connection between logic, proofs and programming is a very rich and interesting field that goes far byond the scope of an introductory course in discrete mathematics. But instead of illustrating this point here by introducing loads of material not covered in the text book, we will limit our ambitions

to simply emphasizing the obvious: the similarities between logical truth values and Python's *boolean* type and its operations.

In Python terms, a logical statement is an *expression* of boolean type, such as $3 < 7$. Variables can be used in such expressions, although it should be noted that every variable will stand for some concrete value at run-time. Computing with open statements in the sense of section 2.4 is also possible in Python, but the techniques required will not be described in this note.

Of the logical connectives in section 2.1, Python directly supports conjunction and disjunction in terms of the built-in primitives **and** and **or**, as well as negation via the primitive **not**. Other connectives can easily be encoded as functions by use of the **if** statement. The implication arrow, for example, may be defined as follows:

```
def implies( a, b ):
    if a:
        return b
    else:
        return True
```

To express the logical statement $x \geq 0 \wedge y \geq 0 \rightarrow x \star y \geq 0$ in Python one simply writes

```
implies( x >= 0 and y >= 0,  x*y >= 0 )
```

Notice, though, that the Python interpreter will merely check whether the implication statement holds for the *particular* values assigned to x and y at run-time. To check the validity of the statement in general, one has to run it for every possible combination of values for x and y – a daunting task even considering the limited range of Python's integers!

Alternatively, one may attempt to prove the statement by logical means (as we do in chapter 4), but it should be noted that this is an activity that is quite different from repeated evaluation of boolean expressions. Whether computers can be helpful in constructing proofs as well is a question best deferred to a later course.

Exercises

2.1 Write a Python program that prints the truth table for text book exercise 2.1.13. The table should contain a row for each possible combination of values for variables p, r and s (q should be constantly true) and columns for the final result as well as each relevant boolean subexpression.

2.2 For each logical implication in text book exercise 2.3.3, write a Python program that verifies its validity. Truth tables do not have to be printed here, the programs should instead simply print *INVALID!* in case a combination of variable values is found for which the implication is false.

3. Set Theory

The most immediate connection between set theory and programming is the use of sets as a data structure. Like lists, sets are collections of data elements. However, in contrast to lists, sets cannot contain duplicated elements, and any apparent order of the elements in a set is irrelevant. If these two characteristics of sets are given special consideration, though, lists can very well be used to *represent*

sets. As we shall see, Python actually provides sets as a built-in datatype, but an encoding in terms of lists may nevertheless be interesting and provide useful insights in the nature of sets.

Easy things first: the Python expression

x **in** *A*

checks whether *x* is an element of list *A*. This membership test works just as well in cases where *A* is supposed to represent a set. Moreover, list concatenation can be seen as a form of set union operation. For example, if *A* and *B* are lists representing sets, the value of *A*+*B* is a list representing the union of *A* and *B*.

Set intersection, relative complement (asymmetric set difference) and subset test can all be implemented as variations on a common theme:

```
def intersection(A, B):
    result = []
    for x in A:
        if x in B:
            result = result + [x]
    return result

def difference(A, B):
    result = []
    for x in A:
        if x not in B:
            result = result + [x]
    return result

def isSubset(A, B):
    result = True
    for x in A:
        if x not in B:
            result = False
    return result
```

The operations seens so far do not depend on the special characteristics of sets; i.e., they work just as well even if the lists involved contain duplicates, and they even preserve element order where this is appropriate. However, the *size* of a set represented as a list is obvously different from the list's length if the list contains duplicates. Hence we need to define size computations using a bit more caution:

```
def size(A):
    count = i = 0
    while i < len(A):
        if A[i] not in A[i+1:]:
            count = count + 1
        i = i + 1
    return count
```

Likewise, testing whether two sets are equal cannot be done using the == operator on the list representations, because equality on lists takes both element order and possible duplicates into account. Instead we follow the text book and define set equality as mutual subset containment:

```
def equalSets(A, B):
    return isSubset(A, B) and isSubset(B, A)
```

Other set operations can be defined in a similar manner.

It should be noted, though, that the purpose of the examples given above is purely pedagogical. For practical programming with sets, Python provides a built-in datatype which handles removal of duplicates and independence of element ordering automatically. In particular, for arguments of this built-in set type, the standard operations *len*() and == behave just like our functions defined above.

The Python Tutorial offers an introduction to the use of sets in section 5.4, with more information to be found in the Python Library Reference, section 5.15.2.

Exercises

3.1 Continuing the encoding of sets as lists above, implement the operation *symmetric difference* (see definition 3.5).

3.2 Using Python's built-in set type and operations, write programs that compute the results of each of the set expressions in text book exercise 3.2.1. In place of the *complement* operation, use *relative complement* with the given universe as the first argument. See definitions 3.7 and 3.8 for more information.

3.3 Extend Python's built-in support for sets by implementing a function that computes the *Cartesian product* of two sets (definition 3.11).

4. Integers and Mathematical Induction

The concept of mathematical induction is an important tool for establishing properties about iterating programs. The text book illustrates this point for several examples where iteration is expressed using a while-loop. However, the connection between induction and programming is probably even more apparent when iteration is expressed in terms of *recursion*.

Here is a reformulation of example 4.11 of the text book, now using recursion instead of a loop.

```
def pow( x, y, n ):
    if n == 0:
        return x
    else:
        return y * pow( x, y, n-1 )
```

Using this definition, the logical statement we wish to prove can be captured quite succinctly:

S(n): for all $x, y \in R$, $pow(x, y, n) = xy^n$

Proving S(n) for all $n \geq 0$ using induction is straightforward. First the base case:

S(0): for all $x, y \in R$, $pow(x, y, 0) = xy^0 = x$

This fact is easily established if we check the definition of *pow* and run the if-statement by hand with n replaced by 0.

Then follows the induction step. That is, assuming

S(k): for all $x, y \in R$, $pow(x, y, k) = xy^k$

we need to show

S(k+1): for all $x, y \in R$, $pow(x, y, k+1) = xy^{k+1}$

Again by checking the definition of *pow*, but now with n replaced by k+1, we realize that the second branch of the if-statement must be taken. That is, we know that

$pow(x, y, k+1) = y \star pow(x, y, k+1-1) = y \star pow(x, y, k)$

But according to our assumption above (the induction hypothesis), $pow(x, y, k) = xy^k$. So we actually have

$y \star pow(x, y, k) = y \star xy^k = xy^{k+1}$

which establishes our result.

The proof given above is arguably very simple, involving only equalities between expressions and the semantics of an if-statement. In comparison, the proof of example 4.11 in the text book takes up considerably more space, mainly because it needs to reason about program points in a flowchart as well as distinguishing between initial and final values of free variables.

It is illustrative to check whether merely encapsulating the given while-loop in a function will help us reduce the complexity somewhat. So let us define *powloop*() as follows:

```
def powloop( x, y, n):
    while n < 0:
        x = x ⋆ y
        n = n - 1
    return x
```

Using this definition, we may indeed formulate the logical statement we wish to prove in the same succinct way as we did above. That is,

$S_2(n)$: for all $x, y \in R$, $powloop(x, y, n) = xy^n$

The sad news is that we will not be able to prove this statement for all $n \geq 0$ using induction. For the induction step, we would need to show

$S_2(k+1)$: for all $x, y \in R$, $powloop(x, y, k+1) = xy^{k+1}$

which means replacing n with k+1 in *powloop* and concluding that the loop will run at least once. Then we would need to somehow refer to the effect of running the loop k times. Turning to our induction hypothesis, we do have some knowledge about the result of calling *powloop* with k as an argument:

$S_2(k)$: for all $x, y \in R$, $powloop(x, y, k) = xy^k$

However, no such call is made from within *powloop* itself, so this knowledge is actually useless! Our only way forward will in fact be to resort to the more complex statement formulation used in the text book.

What this exercise shows us is that the proof technique of mathematical induction is intimately connected to programming with recursion. While it is true that recursive programs can sometimes be more challenging to understand on an intuitive level than their looping counterparts, pay-off time definitely comes when understanding is required in terms of formal proofs as well.

Exercises

4.1 Rewrite the code segment in text book exercise 4.1.22 as recursive function *sum* along the lines of the example above, and prove that for all $n \geq 0$ and for all $x, y \in R$, $sum(x, y, n) = x + ny$.

4.2 Reformulate Euclid's *gcd* algorithm as a recursive Python function (see Fig. 4.11).

4.3 Prove that for all $m \geq n > 0$, $gcd(m,n)$ divides m. Note: you will need to use *strong* mathematical induction here.

5. Functions and Relations

There exists two fundamentally different views of what a function is in the computer science literature. According to one view, a function is a "recipe" for computing a result on basis of some input value; i.e., a function is an algorithm. This view is quite natural for a programmer, as it coincides exactly with the notion of a function found in most programming languages (including Python).

However, if we ignore the question of *how* a function produces its output, it also makes sense to view a function simply as a relation between input and output values. This is the view predominant in discrete mathematics as well as in many areas within computer science where the semantics of functions is studied. Applying such a function to some input value then just means looking up the output value corresponding to the given input.

To illustrate how the two views relate to each other, we will use Python to express them both. Let us first define a simple function in the standard Python way – a "recipe" for computing a result.

```
def funny( x ):
    if x in range(0,5):
        return x*x+7
    if x in range(5,10):
        return x*5
```

The function *funny* will return an integer result for each input value in the range 0 to 9. By running the function by hand for each valid input value we may conclude that the input-output relation it defines is the set { (0, 7), (1, 8), (2, 11), (3, 16), (4, 23), (5, 25), (6, 30), (7, 35), (8, 40), (9, 45) }.

This information is obviously an interesting fact about *funny*, which we may use in proofs and in other similar circumstances where the meaning of functions is in focus. However, we may also go further and claim that the function *funny* actually *is* its input-output relation. That is, now that we know all input and output values, we may choose to simply skip the "recipe" and redefine *funny* as a data structure representing all possible input-output value combinations. In Python this can be expressed as a list of pairs:

funny = [(0, 7), (1, 8), (2, 11), (3, 16), (4, 23), (5, 25), (6, 30), (7, 35), (8, 40), (9, 45)]

Whether this is practically feasible in general depends, of course, on the size of the relation. For example, we would not want to attempt this trick for a function that is defined for all integer input values. On the other hand, a function represented as an input-output relation will be independent of the computation times required by the underlying algorithm, so this alternative representation can be used to trade off CPU time against memory usage, if desired.

Anyway, whereas applying the original function *funny* above to some argument *arg* is simply a matter of writing *funny*(*arg*), the alternative version can only be "applied" if we conduct a search through the defined list:

```
def apply(function, arg):
    result = None
    for (input,output) in function:
        if input == arg:
            result = output
    return result
```

That is, applying the new *funny* to *arg* is written *apply*(*funny*, *arg*). If the result of this expression is *None*, it must be because *arg* is not in the domain of *funny*. Note that the same result would be obtained if the original function *funny* were applied to the same *arg*.

However, knowing that the number of pairs with *arg* in the left position might zero, can we really be sure that the only alternative is exactly *one* such pair? Looking back at the way *funny* was constructed, this must be our conclusion. That is, for a given input value, our original algorithm *funny* must either have returned *None*, or its result must have been exactly one integer.

But for an arbitrary set of integer pairs, it might of course be that a certain value appears in the left position of more than one pair. Such a set will not correspond to a function, though, but rather to a more general relation between integers. Here is an example that captures the relation *m divides n* for integers *m* and *n* in the range 1 to 6.

divides = [(1,1), (1,2), (1,3), (1,4), (1,5), (1,6), (2,2), (2,4), (2,6), (3,3), (3,6), (4,4), (5,5), (6,6)]

Notice that our function *apply* above will still work in this generalized setting, by rather arbitrarily selecting the last pair it finds. An equally ad hoc solution would be to go for the first element:

```
def apply(function, arg):
    for (input,output) in function:
        if input == arg:
            return result
    return None
```

However, for a general relation it makes more sense to search for *all* pairs that match a certain value on the left (like returning all possible "output" values for a given input):

```
def findright(relation, left):
    right = []
    for (m,n) in relation:
        if m == left:
            right = [n] + right
    return right
```

Or we might turn the information flow upside down and search for all pairs that match a certain value on the right (like asking which input values would result in a certain output):

```
def findleft(relation, right):
    left = []
    for (m,n) in relation:
        if n == right:
            left = [n] + left
    return left
```

These last four definitions describe, in a very concrete form, the difference between programming with functions and programming with relations. In common with most other programming languages, Python is biased towards functions and has built-in support for representing them in algorithmic form. However, there are languages – Prolog is one example – where relations play an equally central role. In either case, the characteristics and uses of both functions and relations should be clear to any serious programmer.

We end this section with a quick illustration of how function composition can be directly expressed in Python, thanks to its support for so called *lambda abstractions*. A lambda abstraction is an expression that denotes an anonymous function with just a return statement. Such an expression might be useful when we want to refer to a function by itself, and not to its result when applied to arguments. For example, consider the following use of function *g*:

```
def g(x, y, z):
    return x+y*z
preferred_function = g
...
print preferred_function(1,2,3)
```

As an alternative, that does not require a separate function definition, we may also write

```
preferred_function = lambda x, y, z:  x+y*z
...
print preferred_function(1,2,3)
```

A lambda expression can be used as any other value, in particular, it can be the return value of another function. This is what we need in order to define function composition as a Python function *compose*.

```
def compose( g, f ):
    return ( lambda x:  g( f(x) ) )
```

The names *g* and *f* are both parameters standing for some specific but unknown functions. Note especially how the actual values of *g* and *f* will be "remembered" by the function value returned by *compose*.

Now, in the context of the following definitions:

```
def alpha( a ):
    return a + 100

def beta( b ):
    return b*3
```

we may write

```
>>> h = compose(alpha, beta)
>>> h(1)
103
>>> compose(beta, alpha)(3)
309
```

The last interpreter command may look a little funny, but it is in fact quite logical: *compose*(*beta*, *alpha*) is an expression that evaluates to an anonymous function, which in turn is applied to 3.

Exercises

5.1 Write a Python function that takes a list of pairs as an argument and determines if the argument constitutes a function or a general relation.

5.2 For each text book exercise 5.1 (a-d), provide a hint towards the correct answer by means of a Python program that constructs the specified relation and applies the outcome of exercise 5.1 to investigate wheter the relation is a function or not. Note: it may be wise to limit the size of these relations to something much smaller than infinity, but still big enough to give some credibility to the test.

5.3 Implement the functions *f*, *g*, and *h* in text book exercise 5.6.7, and in the context of these functions as well as function *compose* above, capture each composition in exercise 5.6.7 (a) as a Python expression. Try out the result on a few input values and verify the correctness of the output by hand.

6. Finite State Machines

State machines capture the idea of a computer that bases its behavior not only on current input data, but also on the history of previous input. A *state* is an abstraction of certain histories that can be regarded as equivalent; this way only a state identifier needs to be memorized, not an entire input histo-

ry. State machines are frequently used in the implementation of such diverse applications as games, user interfaces, network protocols and text analysis programs.

A simple way to code a state machine in Python is to designate one global variable as the state, and set up a transition function that is called each time a new input arrives. The transition function is supposed to define a behavior for each possible combination of state and input values, where a behavior essentially specifies two things: an output value and a next state to remember.

As an illustration we will look at a Python encoding of text book example 6.20. It uses three states and a common alphabet for input and output values that just consists of the integers 0 and 1.

```
state = 's0'

def transition( input ):
    global state
    if state == 's0':
        if input == 0:
            print 0
            state = 's0'
        elsif input == 1:
            print 0
            state = 's1'
    elsif state == 's1':
        if input == 0:
            print 0
            state = 's0'
        elsif input == 1:
            print 0
            state = 's2'
    elsif state == 's2':
        if input == 0:
            print 0
            state = 's0'
        elsif input == 1:
            print 1
            state = 's2'
```

Each time a transition is supposed to be taken, *transition* should be called with the current input as argument. If all input is given at once, the transition function can be called repeatedly:

```
for i in inputs:
    transition(i)
```

The declaration **global** *state* found inside *transition* is a Python detail that is gives the enclosing function rights to overwrite the global variable *state*. Without this declaration, the first assignment to *state* inside *transition* would be interpreted as the creation of a new local variable. Note that overwriting a global variable with a new value is not the same thing as destructively modifying parts of it – in the former case the actual value bound to a name changes, whereas in the latter case the bound value (which must be the address of a data structure) stays the same.

The state machine encoding shown above is very explicit, but also a bit spacious. An alternative encoding uses the fact that both the prescribed output and the successor state of a transition can be captured in two-dimensional tables, indexed by the current state and the current input. Such a variant of example 6.20 is shown below.

```
state = 's0'

next  =  {  's0' : [ 's0', 's1' ],
            's1' : [ 's0', 's2' ],
            's2' : [ 's0', 's2' ]   }

output = {  's0' : [ 0, 0 ],
            's1' : [ 0, 0 ],
            's2' : [ 0, 1 ]   }

def transition( input ):
    global state
    print output[state][input]
    state = next[state][input]
```

Which alternative that is most approproie usually depends on the situation. One advantage of the explicit version with nested if-statements is that it may easily be extended with more customized output actions. In any case it is important to fully understand the connection between state diagrams, table representations and the two encoding principles shown.

Exercises

6.1 Implement the state machine in text book exercise 6.2.8 as two Python programs, one in each of the two styles described above. Use these implementations to answer questions 6.2.8.(a-b).

6.2 Construct the state machine requested in text book exercise 6.3.1 (a), and implement this machine as a Python program.