Unit 3

Control Structures

Learning Outcomes

- Design algorithms that use decision making and implement them in Python.
- Design algorithms that use iteration and implement them in Python.
- Given an algorithm, approximate its running time.
- Find and fix bugs in small programs.
- Create algorithms for more complex problems.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Sections 4.2–4.7, 6.2–6.4, 1.3, Appendix A in *How to Think Like a Computer Scientist*.

Topic 3.1 Making Decisions

All of the code we have written so far has been pretty simple. It all runs from top to bottom, and every line executes once as it goes by. The process soon becomes boring. It's also not very useful.

```
**write** "Think of a number between 1 and 10."
**set** *guess* to 6.
**write** "Is your number equal to *guess*?"
**read** *answer*
**if** *answer* is "yes", **then**
    **write** "I got it right!"
**if** *answer* isn't "yes", **then**
    **write** "Nuts."
**write** "That's the end of the game."
```

Figure 3.1: Simplified guessing game

In the guessing game example from Figure 1.3, we need to decide if the user has guessed correctly or not and then take the appropriate action. Almost every program you write to solve a real problem is going to need to do this kind of thing.

There are a few ways to make decisions in Python. We'll only explore one of them here.

The `if` statement

The most common way to make decisions in Python is by using the `if` statement. The `if` statement lets you ask if some condition is true. If it is, the *body* of the `if` will be executed.

For example, let's simplify the guessing game example from Figure 1.3. In the simplified version, the user things of a number from 1 to 10 and the computer only takes one guess. Pseudocode for this game is shown in Figure 3.1.

Some Python code that implements Figure 3.1 can be found in Figure 3.2.

Here are two example executions of the program:

```
Think of a number between 1 and 10.
Is your number equal to 6?  no
Nuts.
That's the end of the game.
```

```
Think of a number between 1 and 10.
Is your number equal to 6?  yes
```

```
print "Think of a number between 1 and 10."
guess = 6
answer = raw_input("Is your number equal to " \
        + str(guess) + "? ")
if answer == "yes":
    print "I got it right!"
if answer != "yes":
    print "Nuts."
print "That's the end of the game."
```

Figure 3.2: Implementation of Figure 3.1

```
I got it right!
That's the end of the game.
```

As you can see from these examples, only one of the `print` statements inside of the `if` is executed. The `if` statement is used to make a decision about whether or not some code should be executed.

The *condition* is used to decide what to do. The two conditions in Figure 3.2 are `answer == "yes"` and `answer != "yes"`. These mean "`answer` is equal to `yes`" and "`answer` is not equal to `yes`," respectively. We will look more at how to construct conditions later.

The indented `print` statements are not executed when the `if` condition is false. These statements make up the *body* of each `if` statement. The last `print` is executed no matter what: it isn't part of the `if`.

In Python (unlike many programming languages), the amount of space you use is important. The only way you can indicate what statements are part of the `if` body is by indenting, which means you'll have to be careful about spacing in your program.

All block statements in Python (we'll be seeing more later) are indented the same way. You start the block and then everything that's indented after it is the body of the block. When you stop indenting, the block is over.

How much you indent is up to you, but you have to be consistent. Most Python programmers indent 4 spaces and all of the example code for this course is written that way.

Boolean Expressions

The expressions that are used for `if` conditions must be either true or false. In Figure 3.2, the first condition is `answer == "yes"`, and it evaluates to true when the value stored in `answer` is `"yes"`.

These conditions are called *boolean expressions*. The two *boolean values* are `True` and `False`. A boolean expression is any expression that evaluates to true or false.

To check to see if two values are equal, the `==` operator is used and `!=` is the not equal operator.

```
>>> if 4-1==3:
        print "Yes"

Yes
```

◈ You have to press an extra Enter in this example after the `print` statement. In the Python interpreter, the extra blank line is used to tell it you're done the block.

The less than sign (`<`) and greater than sign (`>`) do just what you'd expect. For `<`, if the left operand is less than the right operand, it returns true. There are also boolean operators less than or equal (`<=`), and greater than or equal (`>=`).

Note the difference between `=` and `==`. The `=` is used for variable assignment; you're telling Python to put a value into a variable. The `==` is used for comparison—you're asking Python a question about the two operands. Python won't let you accidentally use a `=` as part of a boolean expression, for this reason.

Functions and methods can also return `True` or `False`. For example, strings have a method `islower` that returns `True` if all of the characters in the string are lower case (or not letters):

```
>>> s="Hans"
>>> s.islower()
False
>>> s="hans"
>>> s.islower()
True
```

The `else` clause

In Figure 3.2, we wanted to take one action if the user answered `"yes"` and another if they answered anything else. It could also have been written in the following way:

```
if answer == "yes":
    print "I got it right!"
else:
    print "Nuts."
```

The `else` clause is executed if the `if` condition is *not* true.

In the `if` statement, you can specify an `else` clause. The purpose of the `else` is to give an "if not" block of code. The `else` code is executed if the condition in the `if` is false.

It is also possible to allow more possibilities with `elif` blocks. The `elif` is used as an "else if" option. In Figure 3.2, we could have done something like this:

```
if answer == "yes":
    print "I got it right!"
elif answer == "no":
    print "Nuts."
else:
    print "You must answer 'yes' or 'no'."
```

Here, the logic of the program changes a little. They have to answer "`yes`" or "`no`". If they answer anything else, they get an error message.

Any number of `elif`s can be inserted to allow for many possibilities. Whenever an `if`...`elif`...`elif`...`else` structure is used, only one of the code bodies will be executed. The `else` will only execute if *none* of the conditions are true.

Topic 3.2 Definite Iteration: `for` loops

We are still missing one major concept in computer programming. We need to be able to execute the same code several times (iterate). There are several ways to iterate in most programming languages. We will discuss two ways you can use in Python: `for` and `while` loops.

```
write "Enter a nonnegative integer:"
read $n$
set $factorial$ to 1
do this for $i$ equal to each number from 1 to $n$:
    set $factorial$ to $factorial \times i$
write $factorial$
```

Figure 3.3: Pseudocode to calculate factorials

```
num = int( raw_input("How high should I count? ") )
for i in range(num):
    print i,
```

Figure 3.4: Using a `for` loop in Python

The `for` loop

In some problems, you know ahead of time how many times you want to execute some code. For example, suppose we want to calculate *factorials*. If you haven't run across factorials before, "n factorial" is written "$n!$" and is the product of all of the number from 1 to n:

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n\,.$$

We can write a program to calculate factorials and we'll know that we have to do n multiplications. Figure 3.3 contains pseudocode to calculate factorials.

In Python, if you have a problem like this where you know before you start iterating how many times you'll have to loop, you can use a `for` loop.

Loops where you know how many times you're going to loop when you start are called *definite loops*. Well, they are in textbooks. Everybody else just calls them "`for` loops". Isn't Computing Science fun?

Before we try to implement a factorial program from Figure 3.3, let's explore the `for` loop a little. The program in Figure 3.4 uses a `for` loop to count as high as the user asks.

```
n = int( raw_input("Enter a nonnegative integer: ") )
factorial = 1

for i in range(n):
    factorial = factorial * (i+1)

print factorial
```

Figure 3.5: Calculating factorials with Python

The easiest way to construct a `for` loop is with the `range` function. When a `for` loop is given `range(x)`, the loop body will execute `x` times. Figure 3.4 will look like this when it's executed:

```
How high should I count? 12
0 1 2 3 4 5 6 7 8 9 10 11
```

Notice that the `range` starts from zero and counts up to `num` $-$ 1. If we wanted to count from one, we could have written the loop like this:

```
for i in range(num):
    print i+1,
```

We will have to do this when implementing the factorial algorithm since we need the values from 1 to n, not from 0 to $n-1$.

Here are the details of what happens when that loop runs: The given range, `range(num)`, will cause the loop to repeat `num` times; the range represents the integers `0`, `1`, `2`, ..., `num-1`. The loop variable (`i`) will be set to the first value in the range (`0`), and the loop body (the `print` statement) is executed. The body is then repeated for each of the other values in the range (`1`, `2`, ..., `num-1`).

Figure 3.5 contains a program program that calculates $n!$.

◈ For some strange historical reason, `i` is a common choice for the `for` loop variable if nothing else is appropriate. You should choose a descriptive variable name where possible, but for quick, short loops, `i` is a good default.

```
name = raw_input("What is your name? ")

while name=="":
    name = raw_input("Please enter your name: ")

print "Hello, " + name
```

Figure 3.6: Using a `while` loop in Python

Topic 3.3 Indefinite Iteration: `while` loops

If you don't know how many times you want the loop body to execute, the `for` loop is hard to work with. For example, in the guessing game in Figure 1.3, we just have to keep guessing until we get it right. That could be anywhere from 1 to 7 guesses. We will finally implement the guessing game in Topic 3.5.

In Python, you can do this with a `while` loop. To construct a `while` loop, you use a condition as you did in a `if` statement. The body of the loop will execute as many times as necessary until the condition becomes false.

For example, the program in Figure 3.6 will ask the user to enter his or her name. If a user just presses enter, the program will keep asking until the user provides a response. It looks like this when it's executed:

```
What is your name?
Please enter your name:
Please enter your name:
Please enter your name: Sherri
Hello, Sherri
```

When the `while` loop runs, these steps are repeated:

1. Check the value of the loop condition. If it's `False`, the loop exits, and the program moves on to the following code.
2. Run the loop body.

Basically, the `while` loop uses its condition to ask "should I keep going?" If so, it runs the loop once more and asks again.

When you use an indefinite loop, you have to make sure that the loop condition eventually becomes false. If not, your program will just sit there looping forever. This is called an *infinite loop*.

You'll write an infinite loop sooner or later. Press control-C to stop the Python interpreter when you get tired of waiting for infinity.

Topic 3.4 Choosing Control Structures

When you're starting to program, you may find it difficult to decide which control structure(s) to use to get a particular result. This is something that takes practice and experience programming. Once you figure out how to *really* work with these control structures, it become easier.

There aren't any rigid rules here. There are often many ways to do the same thing in a program, especially as things get more complicated. Below are a few guidelines.

Before you can choose a control structure, you need to have a pretty good idea what you want the computer to do: you need to have an algorithm in mind. Once you do, you can then think about how to get a program to do what you want.

- Just do it. Remember that most statements in Python (and most other programming languages) are executed in the order that they appear. So, a variable assignment, or `print` statement will execute right after the statement before (unless a control structure changes how things run).

  These statements tell the computer *what* you want to do. The control structures let you express *when* it will happen. (Sort of. Don't take that too literally.)

- Maybe do it. If you have some code that you want to execute only in a particular situation, then a conditional (`if` statement) is appropriate. An `if` will run its body *zero times or one time*. If you need to do similar things more than once, you should be looking at a loop.

  If the logic you need is "either this or that," then you should use `if` with an `else` clause. If you need to do "one of these things," then use the `if`...`elif`...`elif`...`else` form.

In any of these cases, you need to come up with a boolean expression that describes when it's appropriate to do each case. This again takes practice. The goal is to use the variables you have and the boolean operators to come up with something that is true *exactly* when you want the code to run.

- Do it several times. When you need to do something several times, you need a loop. Remember that the loops can execute zero, one, or more times, depending on exactly how you've expressed things (and the values of variables, and what the use types, and so on).

  Note that the task done in the body of the loop doesn't have to be *exactly* the same every time through. You can use the loop variable (in a `for` loop) and any other variables in your program to keep track of what should be done *this* time through the loop. For example, you might want to examine the loop variable to see if it has a particular property and print it to the screen if it does. To do this, you would use a `if` statement in the loop, and write a condition that expresses the property you're looking for.

- Do it this many times. If you know when you start the loop how many times it will run (for example, count up to a certain number, or run once for every item in a collection), you can use a `for` loop.

  For now, all of our `for` loops will loop over a `range` of numbers. In Topic 5.2, you will see that `for` can be used to loop over other items.

  Note that you don't need to know how many times you'll loop when you're *writing* the program. The size of the `range` can be an expression that's calculated from user input, or anything else. You just need to be able to figure this out when the program gets to the `for` loop.

  If you do a calculation and end up looping over `range(0)`, the body of the `for` loop will run zero times.

- Do it until it's done. There are often situations when you can't tell how many times to loop until you notice that you're done. These are usually of the form "keep doing this until you're done; you're done when *this* happens."

  In these cases, you probably need a `while` loop. A `while` loop is similar to an `if` in that you need to write a boolean expression that describes

```
print "Think of a number from 1 to 100."
smallest = 1
largest = 100
guess = (smallest + largest) / 2
print "My first guess is " + str(guess) + "."
```

Figure 3.7: Guessing game: first guess

when to “go”. Unlike a `if`, a `while` loop will execute the body repeatedly, until its condition is false.

When writing the condition for a `while` loop, you need to figure out how the computer will determine that you still have more work to do before the loop is finished. Often, this is “I need to take another step if. . . ”. The body of the `while` is then the code necessary to do a “step”.

Once again, finding the control structure to express what you want to do does take some practice, and there aren’t really any rules. But, hopefully the above will give you something to start with.

Topic 3.5 Example Problem Solving: Guessing Game

In Unit 1, we introduced the guessing game algorithm. It guessed a number from 1 to 100 that the user was thinking. Working from the pseudocode in Figure 1.3, we now have all of the tools we need to write a program implementing this algorithm.

As we saw in the first problem solving example in Topic 2.7, you shouldn’t try to write the whole program at once and just hope it will work. You should test as you write.

The program in Figure 3.7 starts the game and makes the first guess. This will let us test the expression to calculate `guess` first. We can change the initial values for `smallest` and `largest` and make sure `guess` is always halfway between.

Now that we can make one guess, we can combine this with an `if` statement to ask the user whether or not we’re right. This is similar to Figure 3.2. This is done in Figure 3.8.

```
print "Think of a number from 1 to 100."
smallest = 1
largest = 100
guess = (smallest + largest) / 2
answer = raw_input( "Is your number 'more', 'less'," \
        " or 'equal' to " + str(guess) + "? " )

if answer == "more":
    smallest = guess + 1
elif answer == "less":
    largest = guess - 1

print smallest, largest
```

Figure 3.8: Guessing game: get an answer

We can check this program and make sure our logic is right. Have we accidentally interchanged what should be done in the two cases? Suppose we're thinking of 80:

```
Think of a number from 1 to 100.
Is your number 'more', 'less', or 'equal' to 50? more
51 100
```

Now, the program will be guessing numbers from 51 to 100, which is the right range. On the other side, if we were thinking of 43,

```
Think of a number from 1 to 100.
Is your number 'more', 'less', or 'equal' to 50? less
1 49
```

Since 43 is in the range 1–49, we are still on the right track.

Now, we can put this into a loop and keep guessing until we get it right. By directly translating out pseudocode, we would get something like Figure 3.9.

But, if you try to run this program, you'll see an error like this:

```
Think of a number from 1 to 100.
Traceback (most recent call last):
  File "C:/Python23/guess3.py", line 4, in -toplevel-
    while answer != "equal":
NameError: name 'answer' is not defined
```

```
print "Think of a number from 1 to 100."
smallest = 1
largest = 100
while answer != "equal":
    guess = (smallest + largest) / 2
    answer = raw_input( "Is your number 'more', 'less'," \
            " or 'equal' to " + str(guess) + "? " )
    if answer == "more":
        smallest = guess + 1
    elif answer == "less":
        largest = guess - 1

    print smallest, largest

print "I got it!"
```

Figure 3.9: Guessing game: trying a loop

This happens because we have tried to use the value in the variable `answer` before ever putting anything into it. In Python, variables don't exist until you put a value in them with a variables assignment, using `=`. So, when we try to use the value in `answer` in the `while` loop's condition, the variable doesn't exist. Python doesn't have anything to use with the name `answer` so it generates a `NameError`.

This is a good example of problems that can come up when translating pseudocode into a programming language. There isn't always a nice, neat translation; you have to work with the language you're writing your program in. This is also part of the reason it's a good idea to write pseudocode in the first place: it's easier to work out the algorithm without fighting with the programming language.

In order to get around this, we have to get *something* in the variable `answer` before we try to use its value. We could copy the two statements that assign values to `guess` and `answer` outside of the loop, but that could be a little hard to work with later: if we have to fix the code, we have to fix two things instead of one.

The easiest thing to do is just put a dummy value in `answer` so the variable exists, but we're still sure the condition is false. This has been done

```
print "Think of a number from 1 to 100."
smallest = 1
largest = 100
answer = ""
while answer != "equal":
    guess = (smallest + largest) / 2
    answer = raw_input( "Is your number 'more', 'less'," \
            " or 'equal' to " + str(guess) + "? " )
    if answer == "more":
        smallest = guess + 1
    elif answer == "less":
        largest = guess - 1

    print smallest, largest

print "I got it!"
```

Figure 3.10: Guessing game: a working loop

in Figure 3.10.

Let's try this program. Suppose we're thinking of the number 43 and play the game:

```
Think of a number from 1 to 100.
Is your number 'more', 'less', or 'equal' to 50? less
1 49
Is your number 'more', 'less', or 'equal' to 25? more
26 49
Is your number 'more', 'less', or 'equal' to 37? more
38 49
Is your number 'more', 'less', or 'equal' to 43? equal
38 49
I got it!
```

There is still some extra output that we can use while testing the program. After each guess, it prints out the current range of numbers it's considering (the lines like `26 49`). Don't be afraid to `print` out extra stuff like this to help you figure out exactly what your program's doing while testing.

Notice that in Figure 3.10, we have a `if` block inside of a `while` loop.

```
print "Think of a number from 1 to 100."
# start with the range 1-100
smallest = 1
largest = 100
# initialize answer to prevent NameError
answer = ""
while answer != "equal":
    # make a guess
    guess = (smallest + largest) / 2
    answer = raw_input( "Is your number 'more', 'less'," \
            " or 'equal' to " + str(guess) + "? " )

    # update the range of possible numbers
    if answer == "more":
        smallest = guess + 1
    elif answer == "less":
        largest = guess - 1

print "I got it!"
```

Figure 3.11: Guessing game: final version

If you want to do that, or include a loop in a loop, or an `if` in an `if`, just increase the amount of indenting so the inside block is indented more.

Finally, we have a working guessing game program. A final polished version has been created in Figure 3.11.

You'll notice in Figure 3.11 that we have added some *comments* to the code. In Python, comments are on lines that start with a `#`.

Comments let you include information that is meant only for the programmer. The Python interpreter ignores comments. They are used only to make code easier to understand. You should be in the habit of writing comments on sections of code that briefly describe what the code does.

```
write "Think of a number between 1 and 100."
set $guess$ to 1
until the user answers "equal", do this:
    write "Is your number equal to or not equal to $guess$?"
    read $answer$
    set $guess$ to $guess + 1$
```

Figure 3.12: A much worse version of the guessing game

Topic 3.6 Running Time

In this section, we will explore how long it takes for algorithms to run. The running time of an algorithm will be part of what determines how fast a program runs. Faster algorithms mean faster programs, often much faster, as we will see.

The running time of algorithms is a very important aspect of computing science. We will approach it here by working through some examples and determining their running time.

The Guessing Game

When you were experimenting with the guessing game algorithm and program in Topics 1.4 and 3.5, you probably tried the game with a few different numbers. Hopefully, you noticed that this algorithm focuses in on the number you're guessing quite quickly. It takes at most seven guesses to get your number, no matter which one you're thinking of.

Suppose we had written the program from the pseudocode in Figure 3.12.

This algorithm starts at zero and guesses 1, 2, 3, …, until the user finally enters "equal". It does solve the "guess the number between 1 and 100" problem and it meets the other criteria in the definition of an algorithm.

What's different about this algorithm is the amount work it has to do to finish. Instead of at most seven guesses, this algorithm requires up to 100. Obviously, if we're trying to write a fast program, an algorithm that requires seven steps to solve a problem is much faster than one that takes 100.

Suppose we were writing a guessing game that guessed a number from 1 to n. The value of n could be determined when we write the program or by asking the user for the "size" of the game before we start. How many steps would each algorithm take if we modified it to guess a number from 1 to n?

The algorithm in Figure 3.12 would take up to n steps (if the user was thinking of n).

The number of guesses needed by the original algorithm from Figure 1.3 is a little harder to figure out. Each time the algorithm makes a guess, it chops the range from *smallest* to *largest* in half. The number of times we can cut the range 1–n in half before getting down to one possibility is $\lceil \log_2 n \rceil$.

◈ The notation $\lceil x \rceil$ means "round up". It's the opposite of the floor notation, $\lfloor x \rfloor$ and is usually pronounced "*ceiling*".

◈ The mathematical expression $\log_2 n$ is the "base-2 logarithm of n". It's the power you have to raise 2 to to get n. So, if we let $x = \log_2 n$, then it's always true that $2^x = n$.

Having a running time around $\log_2 n$ steps is good since it grows so slowly when n increases:

$$\begin{aligned} \log_2 1 &= 0\,, \\ \log_2 16 &= 4\,, \\ \log_2 1024 &= 10\,, \\ \log_2 1048576 &= 20\,. \end{aligned}$$

We could give this program inputs with $n = 1000000$ and it would still only take about 20 steps.

Why does this algorithm take about $\log_2 n$ steps? Consider the number of possible values that could still be the value the user is thinking of. Remember that this algorithm cuts the number of possibilities in half with each step.

Step	Possible values
0	$n = n/2^0$
1	$n/2 = n/2^1$
2	$n/4 = n/2^2$
3	$n/8 = n/2^3$
k	$n/2^k$

In the worst case, the game will end when there is only one possibility left. That is, it will end after k steps, where

$$\begin{aligned} 1 &= n/2^k \\ 2^k &= n \\ \log_2 2^k &= \log_2 n \\ k &= \log_2 n\,. \end{aligned}$$

So, it will take $\log_2 n$ steps.

Basically, any time you can create an algorithm like this that cuts the problem in half with every iteration, it's going to be fast.

◈ Remember: any time you have a loop that cuts the problem in half with each iteration, it will loop $\log n$ times. If you understand the above derivation, good. If you'd just like to take it on faith, that's fine too.

Repeated Letters

Now consider the algorithm in Figure 3.13. It will check a word (or any string, really) to see if any character is repeated anywhere. For example, the word "jelly" has a repeated "l"; the word "donuts" has no repeated letters. This algorithm works by taking leach letter in the word, one at a time, and checking each letter to the right to see if its the same.

For example, with the word "Lenny", it will make these comparisons:

L	is compared with	e, n, n, y	(none are equal)
e	is compared with	n, n, y	(none are equal)
n	is compared with	n, y	(equals the "n")
n	is compared with	y	(none are equal)
y	is compared with	nothing	

In the third line, it will compare the first and second "n" and notice that they are equal. So, this word has repeated letters.

You can find a Python implementation of this program in Figure 3.14. The only new thing in this program: you can get character `n` out of a string `str` by using the expression `str[n]`. This is called *string subscripting*. Note that the first character in the string is `str[0]`, not `str[1]`.

This program makes $n(n-1)/2 = n^2/2 - n/2$ comparisons if you enter a string with n characters. When measuring running time of a program,

```
write "Enter the word:"
read $word$
set $counter$ to 0
for all letters $letter\_a$ in the $word$, do this:
    for all letters $letter\_b$ to the right of $letter\_a$, do this:
        if $letter\_a$ is equal to $letter\_b$ then
            set $counter$ to $counter$+1
if $counter > 0$ then
    write "There are repetitions"
else
    write "No repetitions"
```

Figure 3.13: Algorithm to check for repeated letters in a word

```
word = raw_input("Enter the word: ")
counter = 0
length = len(word)

for i in range(length):
    # for each letter in the word...
    for j in range(i+1, length):
        # for each letter after that one...
        if word[i]==word[j]:
            counter = counter + 1

if counter>0:
    print "There are repeated letters"
else:
    print "There are no repeated letters"
```

Figure 3.14: Repeated letters Python implementation

we won't generally be concerned with the smaller terms because they don't change things too much as n gets larger (we'll ignore $n/2$ in the example).

We generally aren't concerned with the constant in front of the term (the $\frac{1}{2}$ on the n^2). So, we will say that the algorithm in Figure 3.13 has a running time of n^2.

You'd be right if you think that throwing away the $\frac{1}{2}$ is losing a lot of information. The difference between an algorithm that finishes in 100 or 200 seconds is significant. The problem is that it's too hard to come up with a factor like this that actually *means* something. The algorithm could easily run twice as fast or half as fast if it was implemented in a different programming language, using different commands in the language, on a different computer, and so on.
Bottom line: is the $\frac{1}{2}$ a big deal? Yes, but we don't worry about it when estimating running times.

If you are taking or have taken MACM 101, you might recognize all of this as the same thing that's done with big-O notation. We would say that the repeated letters algorithm has a running time of $O(n^2)$. If you haven't taken MACM 101, watch for the big-O stuff if you do.

Subset Sum

Let's consider one final example where the best known solution is *very* slow.

Suppose we get a list of integers from the user, and are asked if some of them (a subset) add up to a particular target value. This problem is known as "subset sum", since we are asked to find a *subset* that *sums* to the given value.

For example, we might be asked to find a subset of 6, 14, 127, 7, 2, 8 that add up to 16. In this case, we can. We can take the 6, 2, and 8: $6+2+8 = 14$. In this example, we should answer "yes".

If we used the same list of numbers, but had a target of 12, there is no subset that adds to 12. We should answer "no" in that case.

Some rough pseudocode for the subset-sum problem can be found in Figure 3.15. This algorithm will solve the problem. It simply checks every possible subset of the original list. If one sums to the target, we can answer "yes"; if none do, the answer is "no".

```
for every subset in the list:
    set $sum$ to to the sum of this subset
    if $sum$ is equal to $target$:
        answer "yes" and quit
answer "no"
```

Figure 3.15: Algorithm to solve the subset-sum problem.

n	$2^n \approx$	Approx. time
4	16	16 milliseconds
10	10^3	1 second
20	10^6	17.7 minutes
30	10^9	11.6 days
40	10^{12}	31.7 years

Figure 3.16: Running time of an exponential algorithm

What is the running time of this algorithm? It depends on the number of times the loop runs. If we have a list of n items, there are 2^n subsets, so the running time will be exponential: 2^n.

◈ More accurately, the running time is $n2^n$, since calculating the sum of the subset takes up to n steps.

An exponential running time is *very* slow. It's so slow that exponential running times are often not even considered "real" solutions. Consider Figure 3.16. It gives running times of a 2^n algorithm, assuming an implementation and computer that can do 1000 iterations of the loop per second.

As you can see from Figure 3.16, this algorithm can only solve subset-sum for the smallest of cases. Even if we find a computer that is much faster, we can only increase the solvable values of n by a few values.

Can we do better than the algorithm in Figure 3.15? Maybe. There are no known sub-exponential algorithms for this problem (or others in a large class of equally-difficult problems), but there is also no proof that they don't exist.

Number of "Steps"

We have avoided giving the exact definition of a "step" when calculating running times. In general, pick a statement in the *innermost* loop, and count the number of times it runs.

Usually, you can multiply together the number of iterations of the nested loops. For example, consider an algorithm like this one:

```
statement 1
**for** $i$ from 1 to $\log n$:
    statement 2
    **for** $j$ from 1 to $n/2$:
        statement 3
```

Here, "statement 3" is in the innermost loop, so we will count the number of times it executes. The first **for** loop runs $\log n$ times, and the second runs $n/2$ times. So, the running time is $(\log n) \cdot (n/2)$. We discard the constant factor and get a running time of $n \log n$.

Remember that when determining running time, we will throw away lower-order terms and leading constants. That means we don't have to count anything in "shallower" loops, since they will contribute lower-order terms. Similarly, we don't have to worry about how many statements are in the loops; that will only create a leading constants, which will be discarded anyway.

Summary

We have now seem algorithms that have running time $\log n$, n, n^2 and 2^n. See Figure 3.17 for a comparison of how quickly these times grow as we increase n. In the graph, 2^n has been excluded, because it grows too fast to see without making a much larger graph.

As you can see, the $\log_2 n$ function is growing *very* slowly and n^2 is growing quite fast.

Coming up with algorithms with good running times for problems can be very hard. We will see a few more examples in this course. A lot of computing scientists spend a lot of time working on efficient algorithms for particular problems.

For a particular algorithm, you can come up with programs that run faster or slower because of the way they are written. It's often possible to decrease the number of iterations slightly or make the calculations more efficient.

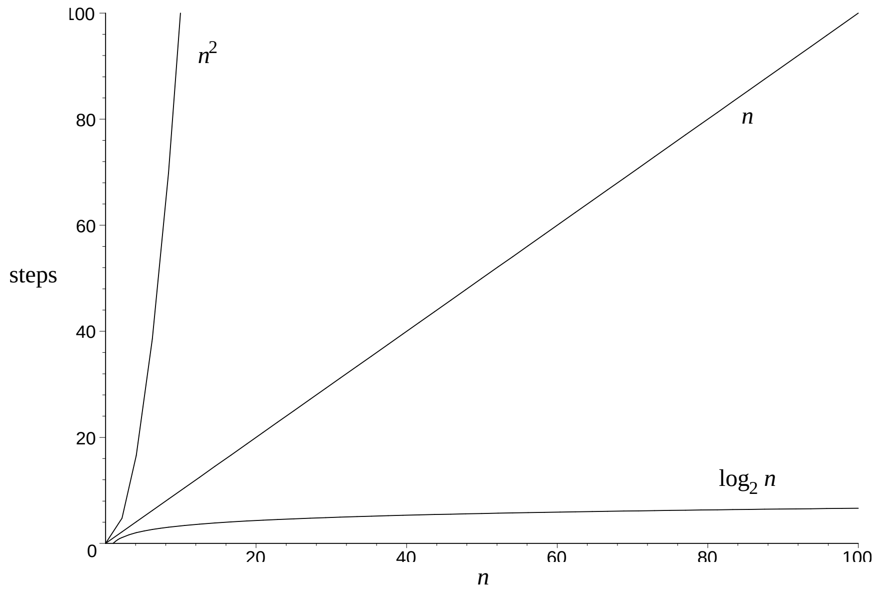

Figure 3.17: Graph of the functions $\log_2 n$, n, and n^2

But no matter how fast the program is you've written, a better algorithm will *always* win for large inputs. This is why most computing science courses spend so much time focusing on algorithms instead of the details of programming. The programming part is easy (one you learn how, anyway). It's coming up with correct, fast algorithms that's hard.

Running time is fundamental when it comes to studying algorithms. It is covered in CMPT 225 (Data Structures and Programming), CMPT 307 (Data Structures and Algorithms), and many other courses. The mathematical details of the analysis are covered in MACM 101 (Discrete Math I).

Topic 3.7 Debugging

Unfortunately, when you write programs, they usually won't work the first time. They will have errors or *bugs*. This is perfectly normal, and you shouldn't get discouraged when your programs don't work the first time. Debugging is as much a part of programming as writing code.

Section 1.3 and Appendix A in *How to Think Like a Computer Scientist* cover the topic of bugs and debugging very well, so we won't repeat too much here. You should read those before you start to write programs on your own.

Beginning programmers often make the mistake of concentrating too much on trying to fix errors in their programs without understanding what causes them. If you start to make random changes to your code in the hopes of getting it to work, you're probably going to introduce more errors and make everything worse.

When you realize there's a problem with your program, you should do things **in this order**:

1. Figure out where the problem is.
2. Figure out what's wrong.
3. Fix it.

Getting it right the first time

The easiest way to get through the first two steps here quickly is to write your programs so you know what parts are working and what parts might not be.

Write small pieces of code and **test them as you go**. As you write your first few programs, it's perfectly reasonable to test your program with every new line or two of code.

It's almost impossible to debug a complete program if you haven't tested any of it. If you get yourself into this situation, it's often easier to remove most of the code and add it back slowly, testing as you do. Obviously, it is much easier to test as you write.

Don't write your whole program without testing and then ask the TAs to fix it. Basically, they would have to rewrite your whole program to fix it, and they aren't going to do that.

As you add code and test, you should temporarily insert some `print` statements. These will let you test the values that are stored in variables so you can confirm that they are holding the correct values. If not, you have a bug somewhere in the code you've written and should fix it before you move on.

In the two example "Problem Solving" topics, 2.7 and 3.5, the program was written in small pieces to illustrate this approach.

Finding bugs

Unfortunately, you won't always catch every problem in your code as you write it, no matter how careful you are. Sooner or later, you'll realize there is a bug *somewhere* in your program that is causing problems.

Again, you should resist the urge to try to fix the problem before you know what's wrong. Appendix A of *How to Think Like a Computer Scientist* talks about different kinds of errors and what to do about them.

When you realize you have a bug in your program, you're going to have to figure out where it is. When you are narrowing the source of a bug, the `print` statement can be your best friend.

Usually, you'll first notice either that a variable doesn't contain the value you think it should or that the flow of control isn't the way you think it should be because the wrong part of an `if` is executed.

You need to work backwards from the symptom of the bug to its cause. For example, suppose you had an `if` statement like this:

```
if length*width < possible_area:
```

If the condition doesn't seem to be working properly, you need to figure out why. You can add in some `print` statements to help you figure out what's really going on. For example,

```
print "l*w:", length*width
print "possible:", possible_area
if length*width < possible_area:
    print "I'm here"
```

When you check this way, be sure to copy and paste the exact expressions you're testing. If you accidentally mistype them here, it could take a *long* time to figure out what has happened.

You'll probably find that at least one of the `print` statements isn't doing what it should. In the example, suppose the value of `length*width` wasn't what we expected. Then, we could look at both variables separately:

```
print "l, w:", length, width
```

If `length` was wrong, you would have to backtrack further and look at whatever code sets `length`. Remove these `print` statements and add in some more around the `length=`... statement.

Topic 3.8 Coding Style

Writing code that is *correct* and solves the problem isn't always enough. It's also important to write code that someone can actually read and understand.

It is often necessary for you or others to return to some code and add features or fix problems. In fact, in commercial software, most of the expense is in maintenance, not in the initial writing of the code.

It can be quite difficult to look at someone else's code (or even your own code after a few months) and figure out what's going on. In order to fix bugs or add features, you need to understand the logic and details of the code, otherwise you'll probably break more than you fix.

To help others (and yourself) understand the code you've written, it's important to try to make everything as clear as possible. There are no absolute rules in this, but there are some guidelines you can follow.

Comments

Comments can be used in your code to describe what's happening. In Python, the number sign (`#`, also called the hash or pound sign) is used to start a comment. Everything on a line after the `#` is ignored: it is there for the programmer only, and does not affect the way the program runs.

In your programs, you should use comments to explain difficult parts of the code. The comment should explain what is happening and/or why it needs to be done. This can include a description of the algorithm and purpose, if it's not immediately clear.

Often, when beginning programmers are told "comments are good," the results are something like this:

```
# add one to x
x = x + 1
```

Don't do that. Anyone reading your code should understand Python, and doesn't need the language explained to them. Comments that actually explain *how* or *why* are much more useful:

```
# the last entry was garbage data, so ignore it
count = count - 1
```

That comment will help somebody reading your code understand why it was necessary to decrease `count`.

It is often useful to put a comment at the start of each control structure, explaining what it does, or what the condition checks for. Here are some examples:

```
# if the user entered good data, add it in
if value >= 0:
    total = total + value
    count = count + 1

# search for a value that divides num
while num%factor != 0:
    factor = factor + 1
```

You can also put a comment at the top of a "section" of code. Look for chunks of code that do a specific task, and put a comment at the top that describes what that task is, and how it's done. For example,

```
# Get user input
# Ask for integers, adding them up, until the user
# enters "0".
```

The Code Itself

The way the code itself is written can make a huge difference in readability and maintainability.

Probably the easiest thing to do is to use good variable names. Variable names should describe what the variable holds and what it's for. Consider the first example above with poorly chosen variable names:

```
if x >= 0:
    y = y + x
    z = z + 1
```

It would take a lot of careful reading to figure out what this code actually does. With descriptive variable names, it's much easier, even without the comment:

```
if value >= 0:
    total = total + value
    count = count + 1
```

```
a = int(raw_input("Enter an integer: "))
b = 0
for i in range(a+1):
    if i>0:
        if a % i==0:
            b = b + 1
print "There are " + str(b) + " factors."
```

Figure 3.18: A program with poor style

There is a bit of a trade-off between the length of the variable name and how readable it is. On one hand, short variables names like `a`, `n1`, and `x` aren't very descriptive. But, variable names that are too long can be difficult to type and clutter code. The name `total_number_of_values_entered_by_user` may be very descriptive, but code using it would be unreadable. Perhaps `values_entered` would be better.

The spacing in your code is also important.

In Python, you are required to indent the blocks of code inside a control statement. In many other programming languages, this is optional, but still considered good practice. In Python, you should be consistent in your spacing: the standard style is to indent each block by four spaces.

Spacing *within* a statement can help readability as well. Consider these two (functionally identical) statements:

```
y = 100 / x+1
y = 100/x + 1
```

The spacing in the first one suggests that the `x+1` calculation is done first (as in `100/(x+1)`), but this is not the case. The order of operations in Python dictate that the expression is equivalent to `(100/x)+1`. So, the second spacing gives a more accurate first impression.

You can use space within lines, and blank lines in the code, to separate sections logically. This will make it easier to scan the code later and pick out the units.

Summary

Again, there are no absolute rules for coding style. It is overall a matter of opinion, but the guidelines above can be followed to point you in the right

```
# get user input and initialize
num = int(raw_input("Enter an integer: "))
count = 0

# check every possible factor (1...num)
for factor in range(1, num+1):
    # if factor divides num, it really is a factor
    if num%factor == 0:
        count = count + 1

# output results
print "There are " + str(count) + " factors."
```

Figure 3.19: Figure 3.18 with the style improved

direction.

When you are writing code, you should always keep in mind how easy it is to read and follow the code. Add comments and restructure code where necessary.

Consider the program in Figure 3.18. What does it do? How is it being done? Does the text displayed to the user in the last line help?

Now look at Figure 3.19. This program does the exact same thing, but has better style. Better variable names and a few comments make a big difference in how easy it is to understand. Even if you don't know what the `%` operator does in the `if` condition, you can probably figure out what the program does.

`a%b` computes the remainder of `a` divided by `b`. When it evaluates to zero, there is no remainder: `a` is evenly divisible by `b`.

There is another change in Figure 3.19 that is worth mentioning. The logic of the program was changed slightly to simplify it. The `if` in Figure 3.18 is only necessary to eliminate the case where the loop variable is zero: checking this case causes a division by zero error. We can change the `range` so the loop avoids this case in the first place.

This illustrates a final important aspect of coding style: the actual logic of the program. The first way you think of to accomplish something might not be the simplest. Always be on the lookout for more straightforward ways to do what needs to be done. You can often eliminate some logic in your program in favour of something that does the same thing in an easier way.

This, along with other aspects of coding style, takes experience. You will learn new methods and tricks to get things done in a program as you write more code, read more code, and take more courses. Be patient and keep your eyes open for new techniques.

- ▶ Go back to the code you wrote for the first assignment in this course. Can you easily understand how it works? Keep in mind that it's only been a few weeks: imagine coming back to it next year.
- ▶ Now, try to swap code with someone else in the course. Can you understand each other's code?

Topic 3.9 More About Algorithms

Now that you know about variables, conditionals, and loops, you have all of the building blocks you need to start implementing algorithms. (You'll still need to know some more about working with data structures before you can implement *any* algorithm. We will talk more about data structures in Unit 5.)

We have also said that coming up with an algorithm is generally much harder than implementing the algorithm with a programming language. Creating algorithms is something you'll practice over the next few years if you continue in computing science.

Binary Conversion

To get you started thinking about creating algorithms, we'll do one example here. In Topic 2.6, we talked about how to convert a binary value to decimal: each bit is multiplied by the next power of two and the results added.

But, how can we do the opposite conversion? If I give you the number 13, how can you determine its (unsigned) binary representation? (For the record, it's 1101.)

First, let's assume we're limited to 8-bit binary numbers. The 8-bit representation of the number 13 is shown in Figure 3.20, with the numeric value

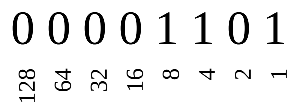

Figure 3.20: The number 13 represented with 8 bits

of each bit. We can check to see that this is the correct representation: $8 + 4 + 1 = 13$.

Now, back to the question of how we could come up with this. Let's try to get the eight-bit representation for the number 25.

First an easy question: do we need a 0 or 1 in the first position (the 128's position)? Obviously not, 128 is much bigger than 25, so adding in a 128 will make the whole thing too big. In fact, we can fill in the highest three bits this way. They are all larger than 25, so we'll definitely want 0's there:

000?????

Well, at least we're getting somewhere: we have the first few bits taken care of. Now, do we want a 0 or 1 in the next position (16)? According to the above reasoning, we *could* put a 1, but is that necessarily right? Suppose we don't: put a 0 in the 16's position. Then all of the bits we have left to fill are 8, 4, 2, and 1. These add up to 15, so there's *no way* we could get 25 out of that. We have to put a 1 in the 16's position:

0001????

For the rest of the positions, we can continue in a similar way. We have taken care of 16 of the number 25 we're trying to represent, so we have 9 left. Keep going down the line: for the 8's position, 8 is less than 9, so put a 1 in that position:

00011???

We now have 1 left to represent. We can't take a 4 or a 2, but will set the last bit to 1:

00011001

This has described a fairly simple algorithm to determine the binary representation of a number: if the bit we're looking at will fit in the number we have left, put a 1; if not, put a 0. Pseudocode for this algorithm is in Figure 3.21.

```
read $num$
for positions $p$ from 7, 6, ...0, do this:
    if $num < 2^p$, then
        set $binary$ to $binary$ + “0”
    otherwise,
        set $binary$ to $binary$ + “1”
        set $num$ to $num - 2^p$
write $binary$
```

Figure 3.21: Pseudocode to determine the 8-bit binary representation

This algorithm only works for numbers up to 255. If you give the algorithm a number any bigger than that, it will produce all 1's. It's possible to fix the algorithm by starting with bit $\lfloor \log_2 n \rfloor$, instead of bit 7.

So?

You should probably know how to convert a number to its binary representation, but that's not the point of this topic.

As you go on into computing science, development of algorithms is important. The point here is to give you an idea of how you can go about coming up with an algorithm. Start by trying to work out the problem by hand and try to recognize the common steps and decisions needed. Try to work this into pseudocode expressing a general method and test it on different values.

Once you have the pseudocode, you can then start working on a program that implements the algorithm you've developed.

Summary

At this point, we have seen all of the major building blocks of computer programs. Once you have loops and conditionals, you can combine them to tackle just about any problem.

Again with this material, you have to practice these ideas by writing programs before you'll really understand them.

Key Terms

- `if` statement
- condition
- boolean expression
- `for` loop
- `while` loop
- running time
- debugging