Encrypting a Picture in StarLogo

1 Introduction

In this exercise we will be working with pictures. In addition to drawing pictures (with the turtles or by other methods available in StarLogo), we will be programming an algorithm to encrypt and decrypt the pictures that you have drawn.

An *algorithm* is a set of instructions, like a recipe. Computers can only do exactly what they are told, so when you program a computer you are giving it an algorithm, or set of instructions, to follow.[1]

> Do this.
>
> Do that.
>
> If the grass is purple, do a somersault;
>
> Otherwise, do jumping jacks.

Encryption means substituting something in for something else. For example, to encrypt a sentence you would substitute new letters for each letter of the sentence. This is like using a code to encode a sentence, only when you encode a sentence you replace whole words or phrases with other words or phrases. If you encode a picture you might replace a house with a boat or a tree with a cow. If you encrypt a picture you might break the picture into a bunch of little squares, or pixels, and substitute a new pixel in for each original pixel. Just like a code tells you what to replace each word or object with, a *cipher* tells you what to replace each letter or pixel with.[2]

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
F H J L N P R T V X Z B D G I K M O Q S U W Y A C E
```

```
V TIKN CIU NGXIC DFST JFDK!
```

[1]When you use a computer program that someone else has created, like a word processor, you are not giving the computer an algorithm. Instead, the people who programmed your word processor already gave the computer the algorithms that it needs to do stuff like save or spell check your file, so it is still has a set of instructions to follow.

[2]In math a code is something a little different, so people who do this stuff for a living may refer to everything as encryption, to avoid confusion.

Encryption is very important in making sure that computers are secure. For example, if you don't want your parents or siblings reading your email to your friends, you couldn't lock the email up in a box. Instead you would have to encrypt it. Similarly, if governments or businesses don't want other governments or businesses to know their military or trade secrets, they will encrypt any emails that discuss such matters.

Encryption works differently on computers than on regular text, since data (text, pictures, etc.) are stored on computers in a special format. Everything on a computer is represented by some *binary* number.[3] Binary numbers are numbers written in *base 2*.

2 Base n: What does this mean?

Usually we write numbers in base 10. That is, any number, say 23579, has a ones digit (9), a tens digit (7), a hundreds digit (5), and so on. If we wrote a decimal, 2.3849, we would also have a tenths digit (3), a hundreths digit (8), and so on. We find a pattern when we look at exponents of ten:

$$\begin{aligned} 1 &= 10^0 \\ 10 &= 10^1 \\ 100 &= 10^2 \\ \frac{1}{10} &= \frac{1}{10^1} = 10^{-1} \\ \frac{1}{100} &= \frac{1}{10^2} = 10^{-2} \end{aligned}$$

and so on. Now we can write

$$\begin{aligned} 23579 &= (2 \times 10^4) + (3 \times 10^3) + (5 \times 10^2) + (7 \times 10^1) + (9 \times 10^0) \\ &= 20000 + 3000 + 500 + 70 + 9 \end{aligned}$$

and

$$\begin{aligned} 2.3849 &= (2 \times 10^0) + (3 \times 10^{-1}) + (8 \times 10^{-2}) + (4 \times 10^{-3}) + (9 \times 10^{-4}) \\ &= 2 + 0.3 + 0.08 + 0.004 + 0.0009. \end{aligned}$$

Since ten is the base of the exponent, we say that these two numbers are written in base 10.

[3] You will find out why this is a little later this week.

We can do the same thing with two as the base of the exponent. For example,

$$\begin{aligned}
1 &= 1 \times 2^0 = 1 \text{ (base 10)} \\
10 &= (1 \times 2^1) + (1 \times 2^0) = 2 + 0 = 2 \text{ (base 10)} \\
11 &= (1 \times 2^1) + (1 \times 2^0) = 3 \text{ (base 10)} \\
100 &= (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) = 4 \text{ (base 10)} \\
101 &= (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 5 \text{ (base 10)}.
\end{aligned}$$

Question 1 *What is the binary number 10011101 equal to in base 10?*

Notice that the digits of a binary number can only be 0 or 1, that is, natural numbers less than 2. Just like a normal base 10 number can only have 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9 as digits.

3 Modular Arithmetic

In order to encrypt our pictures, we will also need to know a little bit about modular arithmetic. If a and b are any two numbers, we say that

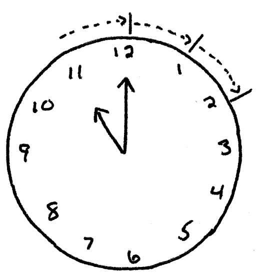
$$a = b \text{ modulo } n$$

or that

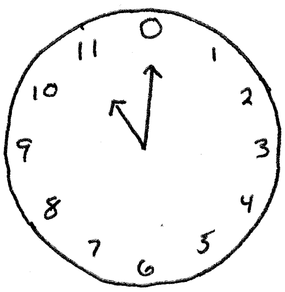
$$a = b \pmod{n}$$

if, in base n, the n^0 digit of a (the ones digit of a) is the same as the n^0 digit of b (the ones digit of b). For example, $12 = 2 \pmod{10}$ since they both have the same ones digit (2).

You do modular arithmetic whenever you make a calculation involving the time of day. For example, if it is 11:00, and you need to be home in three hours, then you will need to be home by 2:00. Another way of saying this is $11 + 3 = 2 \pmod{12}$.



Usually in modular arithmetic we would write 0 instead of 12. This is just a naming convention. $0 = 12 \pmod{12}$ and $12 = 0 \pmod{12}$.



Let's do some examples in base 2.

$$\begin{aligned} 0+1 &= 1 \pmod 2 \\ 1+1 &= 2 = 0 \pmod 2 \\ 11+1 &= 100 = 0 \pmod 2 \\ 1+1+1 &= 11 = 1 \pmod 2 \end{aligned}$$

Every number is equal to either 0 or 1 modulo 2, so we can write out a complete addition table for numbers modulo 2.

+ (mod 2)	0	1
0	0	1
1	1	0

Now we can define a new operation between binary numbers. If a and b are two binary numbers, we want to add each digit of a to the corresponding digit of b modulo 2. For example, if $a = 1100$ and $b = 1010$, then we will first look at the ones digits, and find that $0 + 0 = 0 \pmod 2$. Next we will look at the twos digits, and we see that $0 + 1 = 1 \pmod 2$. From the fours digits we see that $1+0 = 1 \pmod 2$, and from the eights digits we see that $1 + 1 = 0 \pmod 2$. So the result of applying this new operation to a and b is the binary number 0110 (which is the same as 110). We will call this new operation *xor*, and we will write a xor $b = 0110$.

Question 2 *Check that* 10110010 *xor* 01100101 = 11010111.

Question 3 *What is 10011111 xor 11100110?*

Question 4 *What is 11100110 xor 1111001?*

4 Encryption keys

When computers encrypt a binary number, they xor it with another binary number, called a *key*. This changes the original binary number into something meaningless. Only someone who knows the key can decrypt the number to get the original message back. Remember that everything on a computer is represented as some binary number, so now we have a way to encrypt both emails and pictures.

Question 5 *If you know the key that was used to encrypt a binary number, and you know the encrypted number, how would you get the original (unencrypted) number back?*

Suppose that you want to encrypt email to a friend in Australia. You can now do this, so long as you both have the encryption key; you just have to agree on a key. If you agree on a key through a phone conversation or unencrypted email, your parents could overhear the conversation or intercept the email and learn what your key is. Then they could also decrypt and read your friend's and your email. That is not so good. The two of you could meet and whisper the key to each other, but since you live so far apart, that would be rather difficult.

To solve this problem, we will again use modular arithmetic. You and your friend will first agree upon two numbers, b and p. It won't matter if anyone else knows these numbers, so you can do this through unencrypted email. The number p must be a prime number, and the number b must share no common factors with $p-1$.[4] Next you will both choose another number, say you choose the number x and your friend chooses the number y. You each keep your number secret, even from each other. You will each need to calculate $b^{\text{(your number)}} \pmod p$. For example, you will calculate $b^x \pmod p$, and your friend will calculate $b^y \pmod p$. The fourth step is to send the result of your calculations to each other. You send the result $b^x \pmod p$ to your friend, and your friend sends you $b^y \pmod p$.

some primes	
2	23
3	29
5	31
7	37
11	41
13	43
17	47
19	53

Now if you were not working modulo p, you could take a logarithm and find out what the number y that your friend chose is, and your friend could

[4] To explain why we need these conditions would require some complicated math. The short statement is that the set of integers modulo p must be be a field, and the number b must be a generator of this field. See any beginning text on abstract algebra, such as Thomas Hungerford's *Abstract Algebra: An Introduction*, for an explanation of what this means.

do the same to find out what x you chose. In fact, your parents could read your emails to each other and figure out what both x and y are. However when you are computing these numbers modulo p, it suddenly becomes very difficult to take a logarithm.

Question 6 *Let p be 7 and let b be 5. Make a table of the values of 5^1, 5^2, 5^3, and 5^4. Make another table of the values of $5^1 \pmod 7$, $5^2 \pmod 7$, $5^3 \pmod 7$, and $5^4 \pmod 7$. If you know a value from this second table, aside from guessing and checking can you find a way to figure out what exponent was used?*

Imagine if p and b were much larger. For example, $p = 27! + 1$ is bigger than 1×10^{28}, that is, 1 followed by 28 zeros. We can choose b to be 9973. This is also prime, so b will not divide $p - 1 = 27!$ and $27!$ will not divide b. Thus 9973 is a legitimate choice for b. 9973 is not nearly as large as p, but if we compute 9973^8 we already have a number that is almost 900 times larger than p. If we were to make a table of $b^x \pmod p$, we would find that as x increased from 1 to 2 to 3 and so on, the value of $b^x \pmod p$ would increase and decrease pretty erratically. Usually if $b^x > b^y$ we know that $x > y$, but when we are working modulo p, knowing that $b^x > b^y \pmod p$ doesn't tell us which of x and y is bigger.

The result of all of this is that your parents, or anyone else eavesdropping on your email, will not be able to figure out the numbers x and y that you and your friend have chosen, even if they know p, b, $b^x \pmod p$, and $b^y \pmod p$. Now you can calculate $(b^y)^x \pmod p$, and your friend can calculate $(b^x)^y \pmod p$. It turns out that $(b^y \pmod p)^x \pmod p$ is equal to $(b^y)^x \pmod p$ for any number x (try this with $b = 5$ and $p = 7$), so you and your friend will end up with the same number, $k = b^{xy} \pmod p$. Your parents won't know x or y, so they won't be able to calculate k. Now you and your friend can use k as your encryption key. This method of determining a secure encryption key is called Diffie-Hellman key exchange, and we will be using it in our encryption algorithm.

5 On to the program

Now we are ready to begin programming. Create a new project (select `New` from the `File` menu). Find the `Procedures` window. We will be doing most of the work in this window, and then checking that our procedures work using some of the other user interface windows.

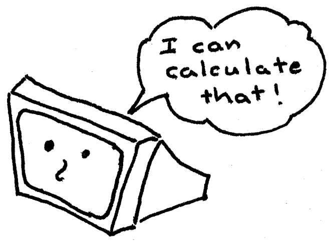
5.1 Computing the key

First we need a procedure to compute the encryption key. This will require some variables to store the values of b, p, x, b^y, and k. When programming, it is a good idea to label variables with words instead of just letters, so that if you look at your program again a year later you can still tell what each variable does. So we will call these variables, respectively, `base`, `prime`, `my-number`, `your-number`, and `key`. To let StarLogo know that we will be using these variables, on the first line of the `Procedures` window type

```
globals [base prime my-number your-number key]
```

and press enter two or three times to move down a couple of lines. Now we want to set the values of these variables, and perform the calculation to find the value of the key. When we declare a global variable, StarLogo automatically creates a command that lets us set the value of that variable, so StarLogo has created the commands `setbase`, `setprime`, and so on, which we will be using. To begin our first procedure, type

```
to make-key
  setprime 9973
  setbase 281
  setmy-number 11
  setyour-number ((281 ^ 18) mod prime)
  setkey ((your-number ^ my-number) mod prime)
end
```



The procedure is called `make-key`. First it sets the value of the variable `prime` to 9972. Next it sets the value of the variable `base` to 281. When we come to the next line, you should replace 11 with another number of your choosing. A good number would be between 2 and 20. For the time being you can leave the value of `your-number` at 281^{18}. When you actually encrypt or decrypt a picture, you will need to be told this number by the person you are sending the picture to or receiving the picture from. On the next line, the command `(your-number ^ my-number)` tells StarLogo to compute $\text{your-number}^{\text{my-number}}$. The command `mod` then takes this number modulo `prime`. The result of this entire computation is the value assigned to `key`.

5.2 The picture

In order to encrypt a picture, we want to represent the picture using numbers. Fortunately StarLogo already does this, in a sense.

The pictures that we will be working with are in the `Graphics` window. The `Graphics` window is divided into a bunch of squares, called patches. You can change the size of the patches and how many of them are in the window by selecting `Settings` from the `Edit` menu. Each patch is colored a particular color, which is represented as a number from 0 to 140. For example, black is 0, white is 9, red is 15, orange is 25, yellow is 45, green is 55, and blue is 105. These numbers are in base 10, just like our key is currently. StarLogo has a command that will take two base 10 numbers, convert them to binary, xor them, and convert the result back to base 10, however. This command is `bitxor`. We will be using it soon.

Consider each patch as an eight digit binary number that is equal to the color of the patch (which we know in base 10). We could form a big binary number by concatenating the 8-digit binary number associated to each patch. We could then xor this number with the binary form of the key, and this would encrypt our picture. This is essentially what we will be doing, however we will not be working with binary numbers themselves.

Why use an eight-digit binary number for each patch? We need enough digits so that we can use any possible color, up to 140. We also don't want to use more digits than we need. 140 is greater than 128, which is 2^7, so we need at least seven binary digits. 140 is less than $256 = 2^8$, however, so we do not need any more than eight binary digits.

So each patch is represented by an eight digit chunk of the binary number representing the entire picture. When we xor this with the key, the result for a particular patch will only depend on the corresponding eight digits in the binary form of the key, since xor operates just on digits. This means that we can split the key up into eight binary digit chunks, each of which will go with one patch in the picture. Since $2^8 = 256$, if we compute `key mod 256`, the value of our key (in base 10) modulo 256, we will find the base 10 value of the first eight-binary-digit chunk of the key.

We need to do a little arithmetic to find the second chunk of the key. The value of $b \pmod p$ is the same as the remainder when b is divided by p. That is,

$$b/p = (\text{the greatest integer less than b / p}) + (b \pmod p).$$

In StarLogo, this first term is represented by `int (base / prime)`. If we compute this, we have essentially cut off the first eight digits of the binary form of the key. Now if we take this new number modulo p, we will have found the second chunk of the key. We can repeat this process as long as there are chunks of key left to find the value of.

Question 7 *Suppose our key were 1249. Using the method described above, find the values in base 10 of each chunk of the key. Now convert 1249 to binary. Break the binary key up into eight digit pieces, and separately convert each piece back to base 10 (as if each were just an eight digit binary number). Do you get the same values for the chunks?*

The following procedure will create a turtle for each chunk and store the value of that chunk of the key as the color of the turtle.

```
to make-key-turtles
  if not (key = 0)
   [crt 1
    osetc-of (count-turtles - 1) (key mod 256)
    setkey (int (key / 256))
    make-key-turtles]
end
```

This is an example of what is called a *recursive* procedure. A recursive procedure is one that calls itself. It must have an exit condition, that is, in order to avoid having the procedure just call itself over and over without stopping, we need a condition that tells the procedure when it is done and doesn't need to call itself again. This is what the `if` statement is for. The expression `not (key = 0)` is called a conditional statement; it is either true or false. If it is true, then the commands enclosed in square brackets are run. If the conditional statement is false, then the commands in square brackets are ignored. That is, the if statement tells the computer to run some commands only if the conditional statement is true. One of the commands that gets ignored if the conditional statement is false is `make-key-turtles`, which calls the procedure again, so we check to see that there are still blocks left in the key before trying to call the procedure again.

The command `crt 1` stands for `create-turtles 1`, which tells StarLogo to create one turtle. Each turtle is numbered as it is created, with the first turtle having the ID number 0, the second turtle having the ID number 1, and so on. The command `osetc-of (count-turtles - 1) (key mod 256)` tells StarLogo that we want to set the color of the most recent turtle (the turtle with ID number one less than the total number of turtles created so far) to the value `key mod 256`. Recall that this is the value in base 10 of the first chunk of the key. The next line cuts this first chunk off of the key. Now when we call this function again, if we still have chunks of the key left to process then it will create a new turtle that will have as its color the value of the next chunk in the key.

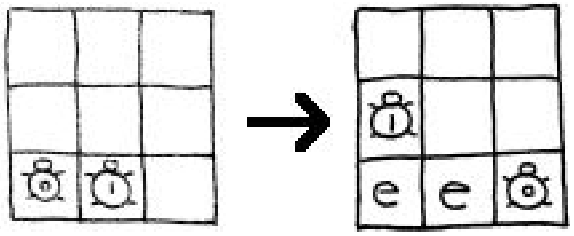
5.3 Encrypting the picture

To encrypt a picture, we will send turtle number 0 (the first turtle) to the bottom left patch in the picture. We will send turtle number 1 (the second turtle) to the patch immediately to the right of turtle 0. Turtle 2 will go to the patch to the right of turtle 1, and so on. If we come to the end of a row of patches, we would like the next turtle to go to the leftmost patch in the next row up.

Once all of the turtles are in place, they can each xor their color with the color of the patch that they are on, and re-color their patch with the result. Since we split the key up into eight binary digit chunks, this will have the same result as xoring the binary number representing the entire picture with the entire binary key.

We might run into a problem, however. What if there are fewer turtles than patches? You could encounter the same problem when encrypting email to your friend in Australia. It would be inconvenient to try to make the key exactly the same length as the email or the picture, since you would need a different key for each email or picture. Likewise it would be inconvenient to make emails and pictures only as large as the key. We need a way of expanding the key to fill in the missing digits.

To extend the key, we will repeat it as many times as we need to fill in the extra digits for the picture or email. Note that for the last repetition we may only need part of the key. In our case, when the turtles finish encrypting the first batch of patches, we will send turtle 0 to the first unencrypted patch, send turtle 1 to the right of turtle 0, and so on. We will then instruct the turtles to encrypt this next batch of patches, and we will repeat this process as many times as we need to in order to encrypt all of the patches in the picture.



The result will be the following procedure.

```
to encrypt
  repeat stop-after
   [setxy counter-x counter-y
    setcounter 0
    repeat (min count-turtles patches-left)
     [setx-of counter (counter-x + counter)
      if ((counter-x + counter) > screen-edge)
       [sety-of counter (counter-y + 1)
        setx-of counter ((counter-x + counter) - screen-size)]
      setcounter (1 + counter)
      wait 0.1]
    stamp (pc bitxor color)
    setcounter-x (1 + xcor-of (counter - 1))
    setcounter-y (ycor-of (counter - 1))
    if (counter-x > screen-size)
     [setcounter-x (counter-x - screen-size)
      setcounter-y (1 + counter-y)]
    setpatches-left (patches-left - counter)]
end
```

This is a long procedure, so it will take some explanation. On the first line, `repeat` tells StarLogo that we want to repeat a set of instructions multiple times. The exact number of times that we will repeat this loop is the value of `stop-after`, a global variable that we will need to declare and set the value of. During each iteration of this loop, the turtles will be performing one batch of encryptions.

What are we telling StarLogo to do repeatedly? The first thing that we tell it to do is to move all of the turtles to a specific patch. The coordinates of the patch are contained in the variables `counter-x` and `counter-y`, which are also global variables that we will need to declare and set the values of.

Once all of the turtles are on this base patch, we want to move individual turtles to their correct positions. This will be done in the next loop.

Before we begin the second, inner `repeat` loop, notice that we are setting the value of the variable `counter` to 0. `counter` is a global variable that we will be using within this next loop, so we will have to declare `counter`.

Each time through this inner loop we will move one turtle from the base patch to its correct station. We have `count-turtles` number of turtles that we need to make sure get to the proper patches, so we will be repeating this loop `count-turtles` number of times.[5] Within this loop we use the `if` statement again. We also use the commands `setx-of` and `sety-of`, which set the x and y positions, respectively, of the turtle with the given ID number. The last command, `wait 0.1`, tells StarLogo to pause a moment before carrying out any more instructions. This allows us to see the turtles as they move. If we remove this command, the turtles would whiz along the screen very fast (try this once you have the entire program working!).

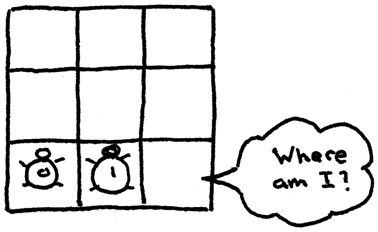
Question 8 *How do the commands in this loop make the turtles go to the proper patches?*

Once the turtles are all in place, we can tell all of them to encrypt their patches with a single command. This is done in the line `stamp (pc bitxor color)`. The command `stamp` tells StarLogo that each turtle should set the color of the patch that it is on to the color immediately following the command. The color that we want each turtle to set its patch to is computed in `(pc bitxor color)`: `pc` gives the color of a turtle's patch, and `color` gives the turtle's own color.

Before repeating the big loop and encrypting the next batch of patches, we need to change the values of `counter-x` and `counter-y`, the variables that tell the turtles what base patch they are beginning their encryption at. We want to set the values of these variables equal to the x and y position of the first unencrypted patch.

Question 9 *How do the four lines after the `stamp` command do this?*

[5]On the last batch we may not need to move all of the turtles, only as many as there are patches left. To make sure we move the correct number of turtles each time, we use whichever is the smaller (minimum) number of `count-turtles` and `patches-left`.



5.4 Some final details

In the procedure encrypt, we used a bunch of global variables that we still need to declare and to give initial values to. To declare these variables, return to the top of the Procedures window. On the line immediately below the previous variable declaration, type the following line.

```
globals [stop-after counter-x counter-y counter patches-left]
```

We will create a new procedure to set the initial values of these new variables.

```
to reset
  setcounter-x (0 - screen-edge)
  setcounter-y (0 - screen-edge)
  setpatches-left (screen-size * screen-size)
  ifelse ((patches-left / count-turtles) -
int (patches-left / count-turtles)) = 0
   [setstop-after (patches-left / count-turtles)]
   [setstop-after (1 + int (patches-left / count-turtles))]
end
```

The bottom left patch of the screen has position (-screen-edge, -screen-edge) (the top right corner has position (screen-edge, screen-edge)). The number of patches to a side is screen-size, which is equal to 2 times screen-edge + 1 (for the position 0), and the picture is a square, so the total number of patches is screen-size * screen-size. All of them are unencrypted before we start.

The number of batches of encryption that we will need to do will be the total number of patches to encrypt divided by the number of turtles doing the encrypting, rounded up. StarLogo does not have a command to round up to the nearest whole number, so we have to test if patches-left

`/ count-turtles` minus the largest whole number less than it is zero (which would be the case if the number of turtles evenly divided the number of patches). If this is the case, then we can stop repeating after that many times. If not, we have to add one to the largest whole number less than the number of patches divided by the number of turtles to find the number of batches to run.

Each time we encrypt a picture, we would like to reset all of these variables. To do this, we can modify the `encrypt` procedure by adding a line to call the `reset` procedure at the beginning. The beginning of the modified `encrypt` procedure will then look like this.

```
to encrypt
  reset
  repeat stop-after
```

We can also consolidate the two procedures that contribute to making the key and putting the key in the form used in the procedure `encrypt`. We can make another new procedure, `setup`, which calls the first two procedures that we made. We may also want to make sure that the `Graphics` window is clear when we begin. This can be done by telling StarLogo to `ca`, or clear all.

```
to setup
  ca
  make-key
  make-key-turtles
end
```

6 The user interface

Now we have a program to encrypt and decrypt pictures. How do we use it? We can call our procedures from the `Command Center` window. We can also create buttons to run our procedures in the `Interface` window.

Click on the `Interface` window to make it active. A menu with some pictures should appear (usually directly below the `Interface` window). The second selection from the left shows a blue button: click on this. You will have to click in the `Interface` window and drag the mouse to make a box where you want your button. Next, a dialog box will appear to help you create the button.

On the top line, labeled `Name`, type in the name that you want to appear on the button. On the next line, labeled `Logo Instruction`, type in the name of the procedure that you want to run when the button is clicked. For example, a button named "setup" could run the procedure `setup`. if the name of the button and the name of the procedure are different, click on the circle next to `Show Name` to make sure that the name of the button will appear on the button. Now click `OK`, and the button will appear in the `Interface` window.

You can click near the button and drag the mouse to create a box encompassing the button to make little squares appear at each corner of the button. If you click in the middle of the button and drag the mouse to a different location in the `Interface` window, the button will move to the new location. If you click on one of the corner squares and drag you can resize your button.

Create a second button for the `encrypt` procedure. This procedure will actually both encrypt and decrypt pictures (try this!), so you might want to give this second button a name that indicates that.

7 Conclusion

You can create a picture by typing turtle commands into the `Command Center` or by using the paint pallet in the `Interface` menu. Once you have created a picture, you can save it by choosing `Export` from the `File` menu, and then choosing just `Patches`. After you have set StarLogo up to encrypt pictures, by clicking on your `setup` button, you can import your picture back into the `Graphics` window by choosing `Import` from the `File` menu. You can also import your friends' pictures to decrypt. If you are encrypting your own picture, be sure to export it again after it has been encrypted!

8 The complete program

```
globals [base prime my-number your-number key]
globals [stop-after counter-x counter-y counter patches-left]

to make-key
  setprime 9973
```

```
  setbase 281
  setmy-number 11
  setyour-number ((281 ^ 18) mod prime)
  setkey ((your-number ^ my-number) mod prime)
end

to make-key-turtles
  if not (key = 0)
   [crt 1
    osetc-of (count-turtles - 1) (key mod 256)
    setkey (int (key / 256))
    make-key-turtles]
end

to encrypt
  reset
  repeat stop-after
   [setxy counter-x counter-y
    setcounter 0
    repeat (min count-turtles patches-left)
     [setx-of counter (counter-x + counter)
      if ((counter-x + counter) > screen-edge)
       [sety-of counter (counter-y + 1)
        setx-of counter ((counter-x + counter) - screen-size)]
      setcounter (1 + counter)
      wait 0.1]
    stamp (pc bitxor color)
    setcounter-x (1 + xcor-of (counter - 1))
    setcounter-y (ycor-of (counter - 1))
    if (counter-x > screen-size)
     [setcounter-x (counter-x - screen-size)
      setcounter-y (1 + counter-y)]
    setpatches-left (patches-left - counter)]
end

to reset
  setcounter-x (0 - screen-edge)
  setcounter-y (0 - screen-edge)
```

```
  setpatches-left (screen-size * screen-size)
  ifelse ((patches-left / count-turtles) - int (patches-left /
count-turtles)) = 0
   [setstop-after (patches-left / count-turtles)]
   [setstop-after (1 + int (patches-left / count-turtles))]
end

to setup
  ca
  make-key
  make-key-turtles
end
```