# COMP61411 Exercises

## COMP61411 Exercises Week 1

### Week 1 Exercise 1

The first thing to do is to get some familiarity with *Mathematica* and its notebooks. If you are in a CS lab, make sure that you have some earphones. (N.B. The lab does not provide earphones, so use your own.)

Make sure you are in a Windows environment. (The labs for this course use a Windows installation of *Mathematica*. However these exercise notes have been prepared using a `linux` installation of *Mathematica*, so be alert for minor user interface differences.)

Go to `http://www.wolfram.com` . Find the 'LEARNING & SUPPORT' menu, and go to 'Videos & Screencasts' under 'Learning'. There are many videos there, but two in particular are worth viewing. Watch 'Hands-on Start to Mathematica' which is on the front page. Also watch 'Mathematica Basics' which is on the 'Getting Started with Mathematica' page from the 'Main Channels' menu on the right. These will give you a good grounding in working with *Mathematica*. You can see the duration of these tutorials displayed with the title. If you work with *Mathematica* alongside the tutorials, as they recommend, it will obviously take longer. Most of the other videos are too specialised to be worth watching at the beginning.

---

The suggested way of working is as follows. Download the 'Answer Notebook' from the 'General Resources' part of the `http://www.cs.manchester.ac.uk/~banach/COMP61411.Info` course website, or from Blackboard to your own filestore and save. In the title, change *MyName* to your actual name, change *MyRegNumber* to your actual registration number. When you start work on an exercise, use a scratch notebook of your own (and save it regularly as you work). It is probably advisable to break your working into *Mathematica* sessions which are not too long, to use different scratch notebooks for different blocks of work, and to restart *Mathematica* regularly — this stops the internal state of *Mathematica* from getting cluttered with old information. Feel free to make extensive use of the *Mathematica* Help system (e.g. there have been some minor changes in the names of some *Mathematica* functions compared with what is stated below; find their correct names via Help). When you have got the exercise working as you wish, copy and paste the relevant parts into your Answer Notebook, add some accompanying text to make the content clear, and save. To submit work as required below, create a pdf of your Answer Notebook, and submit **both the notebook and the pdf** according to the instructions provided in Blackboard.

**Submission deadlines are described later in this document. LATE SUBMISSION INCURS A PENALTY.**

### Week 1 Exercise 2

Download the notebook called 'van Tilborg's Cryptology Notebook (Updated)' from the COMP61411.Info website or Blackboard. This is a (rather large) notebook, that as well as being a notebook, is a textbook on cryptology — though one that dates back to 2000; many new things have happened since then. Open it, and open a scratch notebook too. Copy the `CaesarCipher` definition from Section 2.1.1 into your scratch notebook. Shift-Enter it to cause evaluation of the definition. Try `CaesarCipher["abcdefg",3]` and see that it comes out as `"defghij"`. Try the `Table[CaesarCipher[...]...]` example at the end of Section 2.1.1 in your scratch notebook and see that it comes out OK. Introduce[1] `ciphertext` as in that example unless you have already done so beforehand. Now introduce `Manipulate[CaesarCipher[ciphertext, n],n,1,26,1]`. You can enjoy playing with the controls for a while. Save and close your notebook. Reopen it. One of two things now happens. One possibility is that everything is OK — you can manipulate the `Manipulate` applet as before and it behaves as previously. A second possibility is that the `Manipulate`

[1] From now on, by 'introduce' I mean 'type or copy/paste it into your notebook and Shift-Enter it'.

applet has died in some sense, since it does not respond to manipulation, and instead of displaying the answer to the decryption problem, it just shows some of the `CaesarCipher` internal details. In such a case, go to the the ‘Evaluation’ menu and click ‘Evaluate Notebook’ — this will bring the notebook back to life.

## Week 1 Exercise 3

We are going to play with some aspects of a mini-Enigma. Unfortunately *Mathematica* isn’t very good at manipulating letters (i.e. `"a","b","c",...,"z"`) in as flexible a way as we would like, so we use code numbers for letters instead, so `"a"<->1,"b"<->2,"c"<->3,...,"z"<->26`.

Download the ‘PracticalHints.txt’ file from the COMP61411.Info website or Blackboard. This contains various helpful items in text form which you can just copy and paste into *Mathematica* notebooks. Introduce `RotI` — it is the definition of the permutation (on letter codes) of Enigma rotor I, as in the lecture notes. To help you visualise what is going on, introduce `Graph[RotI,VertexLabels->"Name"]`. Introduce `RotIinv` — it is the inverse of the rotor I permutation, but is incomplete — you must complete it in the obvious way. (The `Graph[...]` that you just created should help you.)

Introduce `lettercodes` (completing the missing bits, obviously). Now you can test your rotors. Introduce

```
ReplaceAll[ReplaceAll[lettercodes,RotI],RotIinv]
```

If all is well, you should get the value of `lettercodes` back. You can try `RotIinv` and `RotI` the other way round too if you wish.

Introduce `RefB` — it is the definition of the (involutive) permutation of Enigma reflector B, as in the lecture notes. You have to expand the definition in detail — the first swap has been written out explicitly as a pair of replacements, and you must do the rest yourself. With `lettercodes`, you can test your reflector. Introduce

```
ReplaceAll[ReplaceAll[lettercodes,RefB],RefB]
```

Since `RefB` is an involution, you should get the value of `lettercodes` back.

A one-rotor Enigma passes a letter through the rotor, through the reflector, and back through the rotor. See what happens to all the individual letters by introducing

```
Table[Replace[Replace[Replace[x,RotI],RefB],RotIinv],{x,1,26,1}]          (1)
```

which cycles through all the `lettercodes` using the iterator {`x,1,26,1`}.

Examine the output. It should be an involution once more. Work out all the pairs that are being swapped and introduce `EnigmaGuts` as the set of replacements that express these. In PracticalHints.txt you will find something to get you started.

Produce a table of the permutations of all the letters by introducing

```
Table[ReplaceAll[x,EnigmaGuts],x,1,26,1]          (2)
```

which again cycles through all the lettercodes using the iterator {`x,1,26,1`}.

Enigma works by not only permuting letters as we have seen, but my moving the rotors too. To allow for this, from PracticalHints.txt, introduce the `Enigma1` function

```
Enigma1[x_,n_] := Mod[((ReplaceAll[Mod[x+n,26,1],EnigmaGuts])-n),26,1]
```

You see that it takes lettercode `x`, displaces it by `n`, applies the `EnigmaGuts` permutation to it, and displaces it back (i.e. by `-n` this time). (N.B. We are modelling an Enigma without a plugboard, for reasons that should be clear from the lectures.)

For future reference, remember the notation (which you have seen in `CaesarCipher` already): `x_` and `n_` are the formal parameters, referred to as `x` and `n` on the right hand side of the definition, and `:=` is definitional equality, which says “don’t evaluate this right now, but remember it for the future, when I supply some actual parameters for `x_` and `n_`”.

Check what happens to the letter `"a"` when various displacements are applied by introducing

```
Table[Enigma1[1,n],{n,0,25,1}]          (3)
```

which cycles through the various permutations of the code of "a" using the iterator {n,0,25,1} —note that this starts at 0 and goes up to 25, unlike the lettercodes which run from 1 to 26— this is so that the undisplaced rotor position occurs at the beginning of the list.

As ever, Enigma permutations are involutions, so that decryption is the same process as encryption. Therefore if you apply Enigma1 to one of the encryptions just performed along with the correct displacement, you should get back the code for the letter "a". Try one or two cases.

Now systematise the preceding by combining all the cases into one expression. You will need an expression of the form MapThread[Enigma1,list1,list2], as in PracticalHints.txt. Figure out how MapThread works by consulting the help system. Figure out what list1 and list2 need to be, to recover the code for "a" in every instance.

**Week 1 Hand In**

Copy and save (and neatly comment) the following into your Answer Notebook **(10 marks total)**:

Q1 the table expression, marked (1) above, that defines the behaviour of the one-rotor machine, and its output;

Q2 the list of pairs swapped by the one-rotor machine, as revealed in the preceding output (write a list of pairs);

Q3 the EnigmaGuts permutation derived from the preceding output (write it out as a permutation);

Q4 the table expression, marked (2) above, and its output;

Q5 the table expression, marked (3) above, and its output;

Q6 the MapThread expression and its output.

**(The Answer Notebook and pdf for the Exercises for Week 1 and Week 2 should be uploaded to Blackboard no later than the end of Friday of Week 3.)**

# COMP61411 Exercises Week 2

## Week 2 Exercise 1

The week 1 exercises built a one rotor Enigma that could encipher a single lettercode at an arbitrary displacement. This week we build up to a more complete machine, culminating in a mini Turing Bombe.

Create EnigmaMachine[text_,key_]:=... by using the MapThread expression from week 1, but using a list of displacements generated from a starting value key_, using Table and Length. (Hint: what function of the iterator do you need to apply the Table function to, and what is it that you need the Length of?)

Since decryption is the same as encryption, you can test your EnigmaMachine definition. First evaluate EnigmaMachine[{1,2,3,4,5},28] (to see that you encrypt {1,2,3,4,5} to something sensible) and then evaluate EnigmaMachine[EnigmaMachine[{1,2,3,4,5},28],28]. If you have done this right, it should not matter that the key_ is bigger than 25 (only its value mod 26 counts). Verify that if you use different keys for encryption and decryption, you don't get back the original text.

The ciphertext {17,4,14,6,17,3,4,23,8,8,19,3,1,24,22,11,6,22,15} is believed to contain a crib for the plaintext {1,3,4,23,9,2,12,8}, (the latter being the encoding into lettercodes of that well-known word "acdwiblh"). Both are in PracticalHints.txt. Find the location of the crib and the length of the crib cycle. (If there is more than one crib, use the first.) What are the distances of the members of the crib cycle from the start of the plaintext? At this point, since there are only 26 different keys, it would be trivial to find the key by exhaustive search; but let's pretend that this is not the case.

Since Bombes work by following the fate of one letter through a crib cycle, it will be enough to use the previous Enigma1[x,n] function in building the Bombe. For simplicity we will build a special purpose Bombe that is hardwired to the length of the crib cycle. For simplicity, introduce cyfrag, the fragment of the cyphertext that aligns with the cribbed plaintext. Introduce

```
Bombe[plain_,cyfrag_,k_] := ...
```

part of whose definition is in PracticalHints.txt, and such that it has hardwired into it the correct number of, and displacements for, the plain/cipher text matches for the crib you found, programmed following the scheme suggested in PracticalHints.txt.

Run through all the possible keys by introducing

```
Table[Bombe[plain,cyfrag,k],{k,0,25,1}]
```

What is the key setting for the start position of the cribbed plaintext? Working backwards, what is the key setting for the start position of the ciphertext? Test your conclusion by deciphering the whole ciphertext with your derived key, using your `EnigmaMachine` built earlier. At this point you should see the cribbed plaintext at the relevant point in the middle of it. You can now re-encrypt the whole plaintext to recover the whole ciphertext.

**Week 2 Hand In**

Copy and save (or where appropriate just write) neatly commented, the following into your Answer Notebook **(10 marks total)**:

Q1 the definition of `EnigmaMachine` ;

Q2 the evaluations of `EnigmaMachine[{1,2,3,4,5},28]` and of `EnigmaMachine[EnigmaMachine[{1,2,3,4,5},28],28]` ;

Q3 the location of the start of the crib (within the plaintext), and the number of elements in the crib cycle;

Q4 the distances of the members of the crib cycle from the start of the plaintext;

Q5 the key setting revealed by the Bombe for the start of the cribbed plaintext, and the Bombe expression;

Q6 the key setting for the start of the whole cyphertext;

Q7 the decryption of the whole cyphertext.

Save your Answer Notebook containing the material for weeks 1 and 2, and generate a pdf from it.

**(The Answer Notebook and pdf for the Exercises for Week 1 and Week 2 should be uploaded to Blackboard no later than the end of Friday of Week 3.)**

When you have done the above, go to `http://williamstallings.com/Crypto/Crypto4e.html`, where the last link in the 'Chapter 5 — Advanced Encryption Standard' section is a really nice animation of the workings of AES. Watch it to improve your understanding of AES.

# COMP61411 Exercises Week 3

## Week 3 Exercise 1

This week we explore techniques related to RSA public key cryptography. Open the 'van Tilborg' notebook, and go to Appendix A. Work through from the beginning, familiarising yourself with the functions discussed there by going through the examples presented, and by trying some examples of your own. In particular you should become familiar with `IntegerQ`, `Divisors`, `Prime`, `PrimeQ`, `PrimePi`, `GCD`, `LCM`, `FactorInteger`, `ExtendedGCD`. The rest of A.2 can then be skipped.

Continue from A.3, familiarising yourself with `Mod`, `EulerPhi`, `DivisSum`, `CoPrimeQ`, `CoPrimes`. (N.B. some syntactic aspects have changed since the original printed book, but are corrected online.) Experiment with Euler's Theorem, as in the example directly following Theorem A.14. Experiment with examples of your own, including both some cases where `m` and `a` are coprime, and some cases where they are not (to see what happens) — cases in which `m=123...` and `a=111...` (with both strings of digits being of equal length) are fun. By identifying a fairly big example, in which both `m` and `a` are at least six digits long, notice the difference in the computation time needed to evaluate `Mod[a^EulerPhi[m],m]` and the time needed to evaluate `PowerMod[a,EulerPhi[m],m]` (which is mathematically the same thing) — the `PowerMod` form utilises optimisations built in to *Mathematica* for such calculations, and should always be used in preference to the `Mod` form.

Go on to check out Fermat's Little Theorem in the same way. Check that `EulerPhi` indeed gives the number of coprimes by comparing its output to the `Length` of the list generated by `CoPrimes`, as suggested at the end of Section A.3.2. In Section A.3.3 check out the `ExtendedGCD` function by recombining the multipliers generated by `ExtendedGCD` with the two original numbers in order to yield the `GCD` given by `ExtendedGCD`.

Experiment with the Chinese Remainder Theorem as described in Section A.3.4. Note that the function mentioned in Section A.3.4, `ChineseRemainderTheorem`, is now defunct, and has been replaced by function `ChineseRemainder` in the current version of *Mathematica*. (Moreover you are no longer required to read in any additional package since `ChineseRemainder` is now built in to the *Mathematica* core.) Choosing the mutually coprime moduli 13, 29, 64, determine the three basis numbers $u_1$, $u_2$, $u_3$, that leave remainders {`1,0,0`}, {`0,1,0`}, {`0,0,1`}, as done in the book. Check that `ChineseRemainder` acting on {`10,5,7`} yields the same number as $10u_1 + 5u_2 + 7u_3$ mod $13 * 29 * 64$.

## Week 3 Exercise 2

Turn to Chapter 8 of the book. At the beginning there is some material introducing functions like `PowerList` and entities like `GF[p]` which you might like to explore (although they are not needed for this week's exercises). For this you will need to read in the `FiniteFields` package. See PracticalHints.txt for the precise syntax. Again, this has changed from that in the printed book though the online notebook version has been corrected.

Turn to Section 8.1.2 on Diffie-Hellman key exchange. Work through Example 8.5 (Part 1). Using the function `Prime[k]`, which returns the `k`'th prime number, choose a fairly large prime (say six to ten digits long). Using your prime, invent a Diffie-Hellman example of your own. Ensure that the primitive element and exponents chosen are large enough that the `PowerMod` function needs to calculate at least one modulus operation.

Go to Section 8.2.1 on the ElGamal public key cryptosystem. Work through Example 8.5 (Part 2). Using the large prime from the previous task, invent an ElGamal example of your own. Again, ensure that all operations using `PowerMod` need to calculate at least one modulus operation.

Continue on to the ElGamal public key signature scheme. Work through Example 8.5 (Part 3). Using your large prime, invent a suitable example of your own.

## Week 3 Exercise 3

Turn to Chapter 9 of the book. At the beginning there is some material that revises parts of Appendix A. Go through this if you feel it will help you. In Section 9.1.2 the basic tools of RSA cryptography are set up. Work through these. Section 9.1.3 covers the use of RSA for privacy. Work through this, including the use of the Chinese Remainder Theorem for decryption. Section 9.1.4 briefly discusses the use of RSA for signatures. Familiarise yourself with this.

Now find two reasonably large primes of your own. Set up the RSA scheme using those. Choose an example to encrypt, and work out the details of RSA encryption and decryption for it. Choose an example to sign, and work out the details of RSA signing and verification for it. Finally, read Section 9.1.5 about securely signing excrypted messages. Set up a scheme in which Alice and Bob each have two RSA public/private key pairs such that one can be used for signing, and the other for encrypting. Choose a message for Alice to encrypt and sign, work through the details, and show how Bob can both decrypt the ciphertext and convince himself that it definitely came from Alice.

**Week 3 Hand In**

Copy and save (or where appropriate just write) neatly commented, the following into your Answer Notebook **(10 marks total)**:

Q1 two worked examples of Euler's Theorem including all the details (i.e. `m`, `a`, `EulerPhi[m]`, `PowerMod[a,EulerPhi[m],m]`) — one where the numbers are fairly small and one where the numbers are bigger;

Q2 an example of the working of `ExtendedGCD`, showing that recombining the multipliers with the two original numbers yields the GCD;

Q3 an example of the working of `ChineseRemainder` using moduli 13, 29, 64, determining $u_1$, $u_2$, $u_3$, and checking that `ChineseRemainder` acting on $\{$`10,5,7`$\}$ yields the same as $10u_1 + 5u_2 + 7u_3$ mod $13 * 29 * 64$;

Q4 the details of your Diffie-Hellman key exchange example;

Q5 the details of your ElGamal public key cryptosystem example;

Q6 the details of your ElGamal public key signature example;

Q7 the details of RSA encryption and decryption starting from your own primes;

Q8 the details of securely encrypting and signing a message from Alice to Bob, including the verification at Bob's end.

**(The Answer Notebook and pdf for the Exercises for Week 3, Week 4 and Week 5 should be uploaded to Blackboard no later than the end of Friday of Week 6.)**

# COMP61411 Exercises Week 4

## Week 4 Exercise 1

This week we explore Elliptic Curve cryptography.

We start with elliptic curves over the reals. Open the 'van Tilborg' notebook, and go to Chapter 10. Go to Section 10.2 and work through, displaying some elliptic curves over the reals. Note that the syntax for doing this has changed from the van Tilborg notebook. You no longer use `ImplicitPlot`. Instead you use `ContourPlot`. PracticalHints.txt shows the necessary syntax for this for the elliptic curve $y^2 = x^3 - 5x + 3$ over the reals — make sure you include the space between the `5` and the `x` to indicate multiplication. Note that `ContourPlot` has a lot of options, which you are welcome to make use of. You can explore these using the Help system if you wish.

Now add in the `Epilog->Line[{{-3,4},{4,-3}}]` option, and watch what happens as the coefficient of `x` in the equation varies between $-5$ and $-3$. You can do this using a suitable `Manipulate[...]`, which you should design. The lectures and notes talk about straight lines intersecting elliptic curves in three places. What happens with the given `Epilog` when the coefficient of `x` is $-3$? Why?

Try the `NSolve` example that follows, showing that it can find all the numerical solutions for the intersection of the line and the curve. Such intersections are an ingredient in the addition of points on elliptic curves over the reals, but they are not the whole story. Skip the rest of Section 10.2 for now, and go to Section 10.3, which explores addition of points on elliptic curves over the reals. Do the first example using `ContourPlot` and a suitable range for `y`, but omitting the `Epilog` completely. Now paste in the first Line from the `Epilog` and re-evaluate. Now paste in the second `Line` from the book and re-evaluate. Continue by pasting in successive elements of the `Epilog` (the various `Text` elements) and re-evaluate each time to see the final picture build up. This picture illustrates the addition of two distinct points in an elliptic curve over the reals. The second example illustrates the addition of a point to itself. Do that in the same way as the previous example. As a third example, repeat the first example, but letting the first `Line` be `{{-3,-2},{4,4}}` instead of `{{-3,-2},{4,5}}`. Having drawn the first `Line`, experiment with the placement of the second (vertical)

`Line` so that it intersects the curve and the first `Line` in the right way. Finally, experiment with the placement of a bullet so that it is reasonably close to the place that represents the addition of the two points of intersection of the first `Line` with the curve. Elliptic curves over the reals are not used in cryptography, but as you can see, they are at least easy to visualise, and they provide a jumping off point for elliptic curves over various kinds of Galois field.

## Week 4 Exercise 2

Now we move on to elliptic curves over a simple prime field $\mathbb{Z}_p$. Go back to Section 10.1, and work through the examples from the beginning. Additionally, try the `Table` example without the enclosing `Flatten` to see how the `Solve` systematically works through the various values of `x` in the given range, one by one. (Because we are working with a simple prime, the *Mathematica* `Mod` function and its close relatives (such as specifying a `Modulus` in other functions) are all we need to perform the necessary calculations.) When you get to Example 10.1 involving a Galois field of a higher prime power, skip it; we will return to this later.

Go to Section 10.2 and pick up just after the `NSolve` example done earlier. Do the `Solve` example that finds the points of intersection of the line $y = x-1$ with the $y^2 = x^3 - 5x + 3$ elliptic curve when working over $\mathbb{Z}_{11}$. Example 10.2 shows another approach to finding the points of intersection. Go through it. Now construct a similar example of your own. How do you do this, given that an arbitrarily chosen straight line through the curve will intersect it in non-integer points? The answer is to exploit the fact —to be illustrated shortly— that any two points on the curve can be 'added' to give a third point, even when we work over a Galois field. Above, we tabulated all the points on the $y^2 = x^3 - 5x + 3$ elliptic curve when working over $\mathbb{Z}_{11}$. Go back to that table, pick two distinct points on the curve from the table, and work out the straight line that runs through them as follows. Introduce `InterpolatingPolynomial[{{x1,y1},{x2,y2}},t]`, where `{x1,y1}` and `{x2,y2}` are the two points that you chose, and `t` is either a fresh variable, or one that you have `Clear`ed for reuse here. The output will be a linear expression for a line through the two selected points, which you can `Simplify` and/or `Expand` if you wish. Now that you have a suitable linear equation, rerun the `Solve` example that finds its points of intersection with the $y^2 = x^3 - 5x + 3$ elliptic curve when working over $\mathbb{Z}_{11}$ (make sure you `Clear` any variables needed that you have used before, or else you will get errors).

Now go to Section 10.3, just after Theorem 10.2, where you will find a `Module` called `EllipticAdd` that automates the addition of points on elliptic curves. There are a number of cases, depending on the relationship between the two input points, and whether either of them is the point at infinity, as in Definition 10.2 in the book. (You will see that the body of the `Module` is just a big `Which` statement — *Mathematica*-speak for a `switch` statement.) Introduce the version of `EllipticAdd` from PracticalHints.txt (where the concrete syntax has been fixed so that it will copy and paste without errors). Confirm that some of the examples immediately following the definition of `EllipticAdd` check out. Going back to the intersection problem for the two points that you chose on the $y^2 = x^3 - 5x + 3$ elliptic curve working over $\mathbb{Z}_{11}$, add them using the `EllipticAdd` function. What is the relationship of the answer you get now to the answer you got before?

The van Tilborg notebook next examines the order of the point `{9,4}` in the curve $y^2 = x^3 + 6x + 3$ over $\mathbb{Z}_{11}$. Confirm this calculation.

Finally we get to do a little cryptography with this stuff. We will do the elliptic curve analogue of Diffie-Hellman key exchange. In Diffie-Hellman using 'RSA-style' tools, the two secrets are *exponents* of a previously agreed element, whose values are hard to discern from a calculated exponential because the discrete logarithm problem is hard in a simple prime field. In elliptic curve Diffie-Hellman, the two secrets are *multiples* of a previously agreed element, whose values are hard to discern from a calculated scalar multiple because scalar division in an elliptic curve is hard. Go to Section 10.4. Near the beginning there is some material on scalar multiplication of a quantity via repeated additions and doublings. Make sure this is clear to you. Go to Example 10.6. It confirms that `{121,517}` lies on the curve $y^2 = x^3 + 100x^2 + 10x + 1$ over $\mathbb{Z}_{863}$. Then it confirms that the order of `{121,517}` is 432. It does so by introducing a recursive function that calculates repeated doublings of `{121,517}` and puts them in the list `P[i]`. Make sure you follow this. You should confirm the results by introducing a `Table` containing the doublings of `{121,517}`, working out the

IntegerDigits of 432, and combining the relevant entries from the Table using EllipticAdd. Now complete the Diffie-Hellman protocol by deriving QAlice, QBob, QA, QB, as in the example. Check that you get the same answer for QA and QB. Repeat the whole derivation with a primitive element of the elliptic curve $y^2 = x^3 + 100x^2 + 10x + 1$ over $\mathbb{Z}_{863}$ of your own choice. Thus you should: tabulate a range of points on the curve as you did above (you don't need to go all the way to 863); pick a point on the curve; check that its order is not too small (check up to order 10, that will be enough — finding the order of an element is in general hard); then complete the Diffie-Hellman protocol as previously.

## Week 4 Exercise 3

Finally we get to the real thing . . . elliptic curve cryptography over Galois fields of non-trivial extension degree. **This exercise is worth 2 marks** (in case you end up thinking it's just not worth all the trouble). It's probably best to begin by closing down *Mathematica* and restarting it, and working in a fresh notebook.

Introduce the FiniteFields package. Introduce z16 = GF[2,4]. This names the Galois field of characteristic 2 with extension degree 4 as z16. Introduce FullForm[%] to see the full details. Introduce FieldIrreducible[z16,x]. This displays the irreducible polynomial with respect to which all the calculations in z16 are performed. Note how the coefficients of the polynomial match up with the entries of the list included in the output to the preceding FullForm command; that's what that list in FullForm[%] is for. Introduce Characteristic[z16] and ExtensionDegree[z16] and FieldSize[z16] to confirm what you know already. Introduce dd = z16[{0,0,1,1}], a non-zero element of the field (obviously, it corresponds to the polynomial $x^2 + x^3$). Introduce dd+dd and dd−dd to illustrate that addition and subtraction in any field of characteristic 2 amount to the same thing, and that adding/subtracting anything to/from itself always gives zero. Introduce ee = z16[{1,1,0,0}]. Introduce dd ee and dd/ee to show that multiplication and division work in Galois fields of higher extension degree. Tabulate powers of dd from 0 to 15. Then tabulate them from 0 to 30. See that the same values just cycle round again once you go past 15.

Now we turn to elliptic curves over this stuff. In the previous exercise, the form of the elliptic curve equation mentioned only y on the left hand side, and only x on the right hand side. Although we didn't have to do it, this meant that one way of finding a solution in a finite field was to separately tabulate the left hand side and right hand side formulae separately, to find a common value, and then to try and get usable values of x and y that way. In the present context, the form of the elliptic curve equation that we must work with is $y^2 + xy = x^3 + a\,x^2 + c$. Note that this involves both x and y on the left hand side, so the previously suggested technique for searching for a solution will not work. At this juncture, inspired by our previous experience regarding addition and subtraction, we realise that if x and y are equal, then $y^2 + xy$ must evaluate to zero. So if we can find a value of x such that the right hand side, $x^3 + a\,x^2 + c$, evaluates to zero, then if we set y to the same thing, we get a solution to the equation. Set c to 0 and set a to ee. Now tabulate (dd^n)^3 + ee (dd^n)^2 for values of n say from 0 to 15. Do you see a value of zero in there? What power of dd does it correspond to? Set both x and y to this power of dd. Evaluate both y^2 + x y and x^3 + ee x^2 to check that they are both zero, and that a solution to the equation (with parameters ee for a and 0 for c) has indeed been found.

We need to be able to add points on the curve. For this introduce Z2mEllipticAdd from PracticalHints.txt. Also, clear P, initialise P[0] to the point {x,y}, set a and c, and define the P[i_] function, all as done in PracticalHints.txt. Now we can just follow the steps taken in the earlier exercise and do Diffie-Hellman on this stuff.

Set PDoubles to be the output of a tabulation of values of P[n] with n ranging from 0 to 31. Set QAlice to be the sum of PDoubles[[3]] and PDoubles[[1]]. Set QBob to be the sum of PDoubles[[4]] and PDoubles[[2]]. What multiples of {x,y} do these correspond to? What calculation with QAlice does Bob have to do in order to derive Bob's version of the common key QB? Do the calculation. What calculation with QBob does Alice have to do in order to derive Alice's version of the common key QA? Do the calculation. Do QA and QB agree?

**Week 4 Hand In**

Copy and save (or where appropriate just write) neatly commented, the following into your Answer Notebook **(10 marks total)**:

Q1 the `ContourPlot` and its output for the $y^2 = x^3 - 5x + 3$ elliptic curve example from Section 10.2, including the `Epilog->Line`, as the coefficient of `x` varies from `-5` to `-3`;

Q2 an answer to the question about straight lines intersecting elliptic curves in three places;

Q3 the `ContourPlot` and its output for the example of addition of two points on a curve over the reals, where the first line is given by `{{-3,-2},{4,4}}` and including the bullet that represents the answer;

Q4 the `Table` example (without the enclosing `Flatten`) that runs through the points of the $y^2 = x^3 - 5x + 3$ elliptic curve over $\mathbb{Z}_{11}$;

Q5 the `Solve` example that finds the points of intersection of the straight line you chose with the $y^2 = x^3 - 5x + 3$ elliptic curve over $\mathbb{Z}_{11}$;

Q6 the result of `EllipticAdd` with the curve $y^2 = x^3 - 5x + 3$ and the points defining the straight line you chose for the previous question, and the relationship between the answer here and the previous answer;

Q7 the `Table` containing the first 10-20 doublings of `{121,517}`, the `IntegerDigits` of `432`, and the confirmation that the relevant combination of these via `EllipticAdd` yields the point at infinity;

Q8 the details of the Diffie-Hellman protocol starting from the primitive element that you chose, including: proving that the point lies on the curve $y^2 = x^3 + 100x^2 + 10x + 1$ over $\mathbb{Z}_{863}$, deriving `QAlice`, `QBob`, `QA`, `QB`, as in the example, confirming that `QA` and `QB` are equal;

... and for the last 2 marks ...

Q9 the preliminary calculations with finite fields of characteristic 2 with non-trivial extension degree, including the basic manipulations with `dd` and `ee`, and including the tabulation of powers of `dd`;

Q10 the derivation of the solution of $y^2 + xy = x^3 + a\,x^2 + c$ for the given parameters, and including the tabulation of `PDoubles`;

Q11 the derivation of the Diffie-Hellman protocol for the case cited.

**(The Answer Notebook and pdf for the Exercises for Week 3, Week 4 and Week 5 should be uploaded to Blackboard no later than the end of Friday of Week 6.)**

# COMP61411 Exercises Week 5

## Week 5 Exercise 1

This week we build a simple emulation of the behaviour of the quantum key exchange protocols given in the lectures. This will be done via fairly straightforward functional programs in *Mathematica* acting on sequences of random bits.

First we do the Bennett and Brassard BB84 QKD protocol. We code the R and D bases using 0 and 1. Using `Table` and `Random[Integer]`, set `AliceBasis` to a random list of 0 and 1 of length 40. Similarly set `AliceData` and `BobBasis` to random lists of 0 and 1 of length 40. To model Bob's measurements of the data, set `BobData` to a list of 0 and 1 of length 40, such that at each position, if `AliceBasis` and `BobBasis` are equal, then `BobData` agrees with `AliceData`, else is random. To model the exchange of basis information, set `EqualBases` to 1 at each position where `AliceBasis` equals `BobBasis`, and 0 otherwise. Using a `For` loop and `Append`, set `AgreedDataAlice` to be the sublist of Alice's data selected on positions where `EqualBases` equals 1. Similarly, set `AgreedDataBob` to be the sublist of Bob's data selected on positions where `EqualBases` equals 1. Output `AgreedDataAlice` and `AgreedDataBob`. Using another `For` loop and `Random[Integer]`, partition `AgreedDataAlice` and `AgreedDataBob` into the set of agreed key bits and the set of agreed check bits: `AgreedKeyAlice`, `AgreedKeyBob`, `CheckDigitsAlice`, `CheckDigitsBob`, and output these four sequences.

Next we do a similar job on the Bennett B92 QKD protocol. We code the R and D bases using 0 and 1. Using `Table` and `Random[Integer]`, set `AliceBasis` to a random list of 0 and 1 of length 40. Since in B92, the basis for transmission is chosen according to the data, set `AliceBasis` to be a copy of `AliceData.` Set `BobBasis` to be a random list of 0 and 1 of length 40. To model Bob's measurements of the data, set `BobData` to a list of 0 and 1 of length 40, such that at each position, if `AliceBasis` and `BobBasis` are equal, then `BobData` agrees with `AliceData`, else is random. To model the determination of which data has been reliably received, set `ReliableData` to 1 at each position where, according to the information given in the notes, Bob can make a reliable decision about the bit that Alice sent, setting it to 0 otherwise. As previously, using a `For` loop and `Append`, set `AgreedDataAlice` to be the sublist of Alice's data selected on those positions in which `ReliableData` equals 1. Similarly, set `AgreedDataBob` to be the sublist of Bob's data selected on positions where `ReliableData` equals 1. Output `AgreedDataAlice` and `AgreedDataBob`. Using another `For` loop and `Random[Integer]`, partition `AgreedDataAlice` and `AgreedDataBob` into the set of agreed key bits and the set of agreed check bits: `AgreedKeyAlice`, `AgreedKeyBob`, `CheckDigitsAlice`, `CheckDigitsBob`, and output these four sequences.

Finally, we do the Eckert E91 QKD protocol. We code the R and D bases using 0 and 1 as before. This time there is going to be an evesdropper, `Eve`. To make the various effects show up more clearly, it is better to make the various lists longer, so throughout this experiment, use lists of length 60 or even 80. As before, using `Table` and `Random[Integer]`, set `AliceBasis` and `BobBasis` to random lists of 0 and 1. To model the data positions that `Eve` interferes with, set `EvesDrop` to a random list of 0 and 1. To model the exchange of basis information, set `EqualBases` to 1 at each position where `AliceBasis` equals `BobBasis`, and 0 otherwise. Since we know in advance when `Eve` will be evesdropping, set `ReliableData` to a table with 1 at each position where `EqualBases` is 1 and `EvesDrop` is 0. Now we model the measurement of entangled states. When the entangled data (which we do not represent, we only model its measurement) is not interfered with, Alice and Bob measure a random bit, but always the same random bit. When there is interference, the bits measured by Alice and Bob are uncorrelated random bits. Using a `For` loop, set `AliceData` and `BobData` to lists of random bits which must be the same at each position where `ReliableData` is 1. (Hint: *Mathematica* allows assignments of the form `x = y = val`, which sets both `x` and `y` to `val`.) Output `AliceData` and `BobData`, and see that they are not the same in general. Set `AgreedDataAlice` to be the sublist of Alice's data selected on those positions in which `EqualBases` is 1. Similarly, set `AgreedDataBob` to be the sublist of Bob's data selected on positions where `EqualBases` is 1. Output `AgreedDataAlice` and `AgreedKeyBob`, and see that they are not the same in general because of Eve's interference. As in the previous exercise, partition `AgreedDataAlice` and `AgreedDataBob` into the sets of agreed key and check bits: `AgreedKeyAlice`, `AgreedKeyBob`, `CheckDigitsAlice`, `CheckDigitsBob`, and output these four sequences. What do you see?

**Week 5 Hand In**

Copy and save (or where appropriate just write) neatly commented, the following into your Answer Notebook **(10 marks total)**:

Q1 the programs and outputs produced for your simulation of the BB84 QKD protocol, namely: `AliceBasis` `AliceData`, `BobBasis`, `BobData`, `EqualBases`, `AgreedDataAlice`, `AgreedDataBob`, (the code that produces, and the values of) `AgreedKeyAlice`, `AgreedDataBob`, `CheckDigitsAlice`, `CheckDigitsBob` ;

Q2 the programs and outputs produced for your simulation of the B 92 QKD protocol, namely: `AliceData`, `AliceBasis`, `BobBasis`, `BobData`, `AgreedDataAlice`, `AgreedDataBob`, (code etc., needed for) `AgreedKeyAlice`, `AgreedKeyBob`, `CheckDigitsAlice`, `CheckDigitsBob` ;

Q3 the programs and outputs produced for your simulation of the E91 QKD protocol, namely: `AliceBasis`, `BobBasis`, `EvesDrop`, `EqualBases`, `ReliableData`, `AliceData`, `BobData`, `AgreedData`

Alice, AgreedDataBob, AgreedKeyAlice, AgreedKeyBob, CheckDigitsAlice, CheckDigitsBob, and your comments about the last four pieces of data (lose a mark for no comment).

Save your Answer Notebook containing the material for weeks 3, 4 and 5, and generate a pdf from it.

**(The Answer Notebook and pdf for the Exercises for Week 3, Week 4 and Week 5 should be uploaded to Blackboard no later than the end of Friday of Week 6.)**