Practice Set P2c

2ITX0 2020–2021 Q2

The exercises in this set allow you to practice with the course material in the lab session. They are particularly suitable as a preparation for Assignment 2 and the final test.

Exercise 1: (Lower bound on sorting by comparison)

You have 10 items of unknown but distinct weights, and you want to sort them in order of increasing weight. You only have a balance that can compare two items by weight.

1. How much information (measured in bits) do you need to obtain to identify the sorted order, when all orders are equally likely? That is, how large is your (information theoretic) uncertainty? Give an exact result (in terms of elementary functions and constants), and a result rounded to three decimal places.

   Hint: The number of possible orders for the 10 items is $10! = 10 \times 9 \times \cdots \times 1$. Each of these could be the sorted order that you are looking for.

2. What is the amount of information (measured in bits) that you obtain from the balance for each comparison, when all outcomes are equally likely? Give an exact result (in terms of elementary functions and constants), and a result rounded to three decimal places.

   Hint: This is a bit of a trick question.

3. What is the lower bound on the number of comparisons needed to establish the sorted order in the worst case that follows from the preceding two answers?

   Note: When the outcomes of a comparison are not equally likely, then the average amount of information obtained is less than the number you found in Question 2 above. Recall the upper bound on the entropy (which is the average amount of information).

Exercise 2: (3-symbol information source)

Consider an information source that produces a random sequence of symbols chosen from the alphabet $\{ a, b, c \}$ with the following probabilities:

$P(a)$	$P(b)$	$P(c)$
$1/2$	$1/3$	$1/6$

1. What is the entropy of this source, measured in bits? Round your answer to three decimal places.

2. Consider the binary encoding of this source given by the following table:

a	b	c
0	10	11

   Encode the sequence $acbaba$.

3. Decode the sequence 1000111000.

4. Calculate the average length (in bits) per symbol of this encoding.
5. Would it be possible to define an encoding that achieves an average of $\sqrt{2}$ bits per symbol?
6. Now consider the binary encoding of this source given by the following table:

a	b	c
1	00	10

   Decode the sequence 0011100011.
7. Why is this second encoding less convenient than the first one?

Exercise 3: (5-symbol information source)
Consider an information source producing random messages from the set

$$\{\, a, b, c, d, e \,\}$$

The messages are independently distributed according to the following probability distribution:

```
{
    "a": 0.3,
    "b": 0.1,
    "c": 0.2,
    "d": 0.1,
    "e": 0.3
}
```

1. What is the entropy of this source, measured in bits? Round your answer to three decimal places.
2. Define a binary encoding of this source that uses, on average, no more that 2.2 bits per symbol.

Exercise 4: (The Universal Declaration of Human Right)
Download the text of *The Universal Declaration of Human Rights* (the title is a link to a specific text file to be used). We consider letters and whitespace only; all other characters are ignored. All contiguous whitespace is to be merged into a single blank. Upper case versions of letters are considered equivalent to their lower case version. Consider an information source that randomly emits letters and a blank (27 symbols), where the occurrence frequencies correspond to those observed in the given file.

1. What is the entropy of this source, in bits? Provide your answer in at least six decimal places.

   You can use *Tom's JavaScript Machine* with these programs (the titles are links to program files):

   (a) *Count Text*
   (b) *Convert Frequencies to Probabilities*

(c) *Calculate Entropy*

In the JavaScript Machine, you can

- paste input in the yellow input box,
- paste a program text into the green program box,
- click the Run button to execute the program, and
- find the output in the blue output box.

When enabling Advanced mode, you get a Copy to Input button above the blue output box. Using these tools, you can count symbol frequencies, convert them to probabilities, and calculate the entropy.

If you are up to it, you can combine the three programs into a single program to do all this in one run.

It is highly recommended that you first try these programs on Exercise 3.

2. Construct an optimal variable-length prefix-free binary code to encode each of the 27 symbols separately. You can use *Tom's JavaScript Machine* with these programs (the titles are links to program files):

   - *Huffman Assistant*
   - *Tree Encoder*

   Using the first tool, you can construct an optimal tree. With the second tool, you can extract an encoding table from that tree.

3. Calculate the average length (in bits) per symbol, when using the code found under 2. You can use *Tom's JavaScript Machine* with this program (the title is a link to a program file):

   - *Average Length Calculator*

Exercise 5: (Dutch Bank Account Numbers)
An old-style Dutch *bank account number* (DBAN) consists of nine decimal digits. In a DBAN, the digits d_i ($i = 1, \ldots, 9$, from left to right), are always such that

$$9d_1 + 8d_2 + 7d_3 + 6d_4 + 5d_5 + 4d_6 + 3d_7 + 2d_8 + d_9$$

is a multiple of 11.

1. Which of the following numbers are valid DBANs?

```
111111113
111111131
111111311
111113111
111131111
111311111
113111111
131111111
311111111
```

2. In a DBAN, the middle digit has become unreadable: $1234?6789$. What should that middle digit be?

3. Explain why changing a *single* digit in a DBAN will always be detectable.

4. Find two DBANs such that the first can be changed into the second by changing *two adjacent* digits. Note that the change is not restricted two swapping these adjacent digits. Explain why this shows that you cannot correct all one-digit errors in all DBAN.

Exercise 6: (Error correction)

Consider the following code. It encodes two message bits in a four-bit code word, by appending two error-control bits:

Message Bits		Check Bits	
a_1	a_2	a_3	a_4
0	0	0	0
0	1	1	1
1	0	0	1
1	1	1	0

1. Which four-bit words (not necessarily code words) are at Hamming distance one from two or more of the *code* words? That is, which words, when received, are not uniquely traceable to a *code* word by changing just one bit?

2. When the word 1011 is received, it is clear that there was a transmission error. Why? What is the most likely code word that was originally sent in that case? Why?

Exercise 7: (Symmetric-key crypto)

This exercise involves the use of GPG (*GNU Privacy Guard*). GPG is available for Windows, Mac, and Linux.[1] We recommend that you use GPG from the command line.[2]

1. Download *this encrypted file*. Inspect the file contents, to see that it is not some plaintext. Decrypt it using `AppliedLogic` as passphrase:

   ```
   gpg CN1.txt.asc
   ```

   GPG then prompts for the passphrase, and (if correct) writes the result to the file `CN1.txt` (stripping the `.asc`). There will be a warning that the "message was not integrity protected" (Exercise 8.1 will address integrity). Decryption should fail with any other passphrase (try it). This helps ensure *confidentiality* (that only persons with the passphrase can read the contents).

   Note that without options, GPG will try to decrypt the file and write the result to another file.

[1] For personal use, you can integrate GPG with various other programs, such as email clients.

[2] Windows: Right-click that START menu, select Command prompt. MacOS: Open the Terminal app.

2. Encrypt *The Universal Declaration of Human Rights* (cf. Exercise 4) using a symmetric cipher:

   ```
   gpg -a -c The_Universal_Declaration_of_Human_Rights.txt
   ```

   You need to choose a passphrase, and repeat it. The option `-a` (short for `--armor`) requests output as an ASCII text file, rather than a binary (text files are easier to email). The command `-c` (short for `--symmetric`) forces encryption using a symmetric-key cipher. The output will be written to the text file `The_Universal_Declaration_of_Human_Rights.txt.asc`.

3. Verify that you can decrypt the result of the preceding step, using the same passphrase:

   ```
   gpg -d The_Universal_Declaration_of_Human_Rights.txt.asc
   ```

   The command `-d` (short for `--decrypt`) requests decryption of the file, writing the result to standard output (the screen). With the option `-o file.txt`, output will be written to the indicated file.

Exercise 8: (Public-key crypto)
This exercise involves the use of GPG (*GNU Privacy Guard*).

1. Download the *public key for this course*, and import it into your local keyring:

   ```
   gpg --import pubkey-2ITX0.txt
   ```

   You can list the contents of your keyring with

   ```
   gpg --list-keys
   ```

   This can be shortened to `gpg -k`.

2. Download *this detached signature file*. Now, verify that the decrypted file `CN1.txt` that you obtained in exercise 7.1 was signed by us:

   ```
   gpg --verify CN1.sig.asc CN1.txt
   ```

   This helps ensure *authenticity* (that the file was indeed sent by us) and *integrity* (that no one tampered with its contents).

3. Create your own private-public key pair:

   ```
   gpg --gen-key
   ```

   You can stick to the defaults when answering the questions about kind of key, and key size. You may wish to consult this *GPG Quick Start*. Note that your private key will be protected by a passphrase of your choosing. If you forget that passphrase, then you can no longer sign messages, and read messages encrypted for you (with your public key).

4. Create a text file `file.txt` with some content and encrypt it for yourself:

```
gpg -a -r 'Your Name' -e file.txt
```

Replace `Your Name` by your name (the one you used when creating your key pair). Option `-r` (short for `--recipient`) sets the intended recipient. By the way, you can encrypt for multiple recipients together; e.g. add `-r 2itx0` for us. The command `-e` is short for `--encrypt`. The output is written to `file.txt.asc`. To check the contents, you can decrypt the result to the screen:

```
gpg -o- file.txt.asc
```

It is usually a good idea to encrypt files intended for others also for yourself as recipient (so that you can decrypt it later, in case the original gets lost).

5. Create an ASCII version of your public key, to allow sharing it:

   ```
   gpg -a -o pubkey.txt --export 'Your Name'
   ```

   You can share[3] your public key in `pubkey.txt`[4] with anyone, so that you can communicate information, while protecting *confidentiality*, *authenticity*, and *integrity* of all messages.

   Find others to exchange public keys: give them yours, and obtain theirs, importing those public keys into your keyring (cf. Exercise 8.1).

6. Create a text file `message.txt` with a text message. Sign and encrypt it:

   ```
   gpg -a -r 'Your Friend' -r 'Your Name' -s -e message.txt
   ```

   Option `-s` (short for `--sign`) includes a *(digital) signature* for the message. Mail it to your friend, and get confirmation of successful decryption. This message is integrity protected; so, your friend should not get a warning (cf. Exercise 7.1).

7. You can create a *detached signature* by

   ```
   gpg -a -b message.txt
   ```

   The command `-b` is short for `--detach-sign`. The detached signature will be written to the file `message.txt.asc`. You can verify it with

   ```
   gpg --verify message.txt.asc message.txt
   ```

Exercise 9: (Information security)

Why does it make sense to accompany an email message with a digital signature, even if the contents are *not* confidential and *not* encrypted? Also describe a realistic scenario.

[3] Via email, your personal web page, a public key server

[4] You can use a more informative name for this file. But anyone receiving it from you should import it into their keyring. After that, they can discard the file.