

# Assignment 2

---

## Introduction

This assignment contains three problems:

- Problem 1 involves the implementation of four cryptographic programs, used for hashing, message authentication and authenticated encryption (AE) and decryption.
- Problem 2 involves the conception and implementation of a new function on a 2-3-4 tree (introduced in workshop 9).
- Problem 3 involves answering a number of theory questions in different situations.

As with the first assignment, you may find different problems and parts more or less difficult and may choose to only answer some parts of each problem. Marks may not be distributed in order of difficulty.

# Problem 1: Cryptography Introduction

In this assignment, you will implement some of the basic building blocks of cryptography. Cryptography is an area of computer science which deals with various aspects of information security. Specifically, you will build cryptographic primitives which provide assurances for integrity, authenticity and confidentiality for some data that you might wish to send or store. These primitives are the cryptographic hash function, message authentication code (MAC) system, and authenticated encryption and decryption schemes.

This assignment is purely an academic exercise. You **must not** use your implementations of these primitives to seriously attempt to secure data. Programming production-ready cryptographic libraries is extremely difficult and may have fatal consequences if done incorrectly. Stick to using peer-reviewed, battle-tested libraries if you ever need to use cryptography outside of a learning environment.

## The Sponge Construction



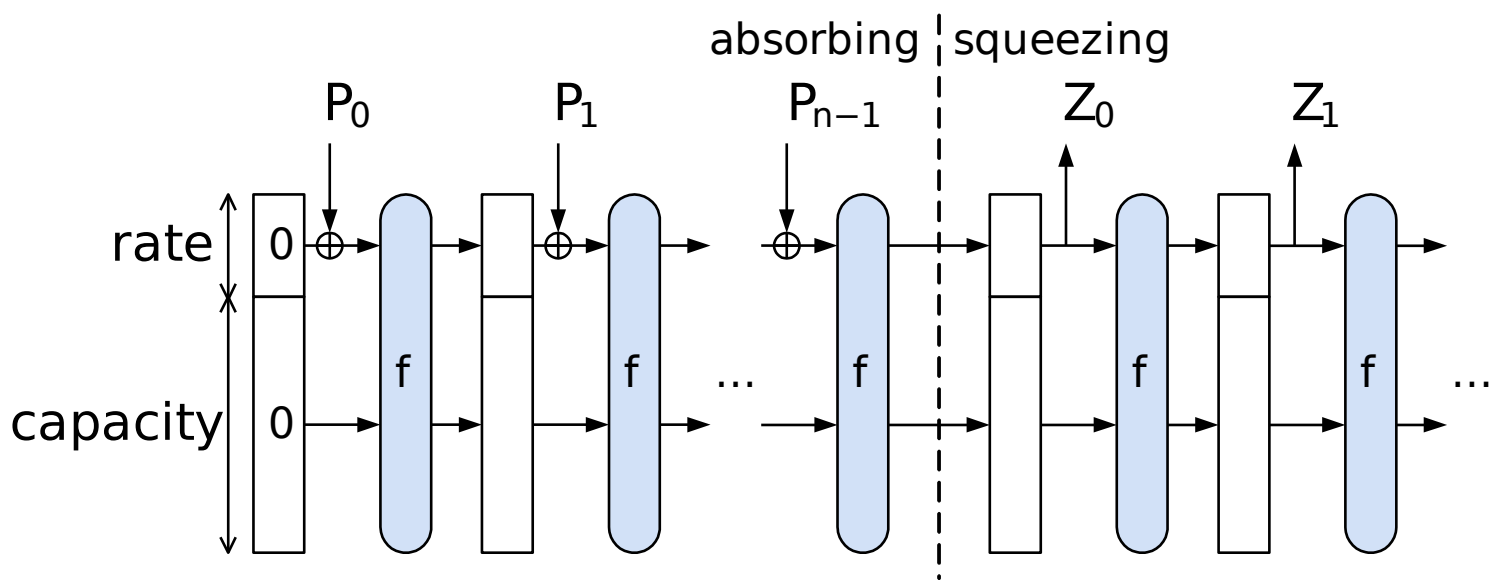
The primitives you implement here will be used in the three sets of programs you create.

All three cryptographic primitives that you will implement are based on a data structure known as the cryptographic sponge. A sponge's state is simply an array of bytes, with a set of operations that need to be supported. For this assignment, you will use a sponge with a 48 byte (384 bit) state.

A cryptographic sponge has some internal state. This internal state is split into two segments:

- The *rate*, which is the part of the sponge which absorbs some amount of the input each round by XOR-ing it with the current contents of that part of the sponge and
- The *capacity*, which is the remainder of the state and is not modified directly by the input each round.

Between each absorption, the state is run through a bijective function which typically acts on the entire state in some way. Following this absorbing phase, after all input data has been absorbed by the sponge, a squeezing phase typically begins, where data is extracted from the sponge's state, again running this function on the sponge. A diagram is shown below modified from Wikipedia [1] which shows this process, *rate* is the *rate* as defined above, *capacity* is the *capacity* as defined above,  $f$  is our bijective function,  $P$  is our input, split into  $n$  segments of size *rate*, and  $Z$  is our output, split into segments of size *rate*.



Note that the demarcation phase occurs immediately following the full absorption of  $P$  and is not shown in the diagram above, it is used to help with padding messages where their length isn't an exact multiple of the rate `r` and to help distinguish between messages of different length.

For this assignment, the first program you will have to write is the `hash` program. The scaffold provides the interface already as well as a function to permute the values, `permute_384`.

The operations to be supported are:

- `void sponge_init(sponge_t *sponge)` which initialises a sponge by zeroing its state,
- `void sponge_read(uint8_t *dest, sponge_t const *sponge, uint64_t num)` which copies the first `num` bytes from the sponge's state into the `dest` buffer,
- `void sponge_write(sponge_t *sponge, uint8_t const *src, uint64_t num, bool bw_xor)` which writes the first `num` bytes from the `src` buffer into the first `num` bytes of the sponge's state either by
  - bit-wise XOR with the state's existing first `num` bytes if `bw_xor` is `true`, or else
  - by overriding the state's first `num` bytes.
- The remaining bytes of the state should be unaltered.
- `void sponge_demarcate(sponge_t *sponge, uint64_t i, uint8_t delimiter)` which bit-wise XORs the `delimiter` into the state's `i`th byte, and finally
- `void sponge_permute(sponge_t *sponge)`, which applies a bijective function.

$$f : \{0, 1\}^{384} \rightarrow \{0, 1\}^{384}$$

(also known as a 384-bit *permutation*) to the state.

You should provide implementations of these operations in the `src/sponge.c` file. You must use the

`permute_384` function found in `include/permutation.h` to implement the `sponge_permute` operation.

For the purposes of this assignment, the provided `permute_384` implements a permutation chosen uniformly randomly from the set of all possible 384-bit permutations. It is fast to compute the result of applying `permute_384` to an input bit-string of length 384 bits. It is also fast to compute its inverse (remembering such an inverse permutation exists because permutation functions are bijective). You should keep this fact in mind when analysing the security of the sponge-based cryptographic primitives, but we do not explicitly make use of the inverse permutation in our implementation.

## Cryptographic Hash Function

A cryptographic hash function  $H$  is used to provide a check for data integrity against **non-deliberate** corruption. The hash function


$$H : \{0, 1\}^* \rightarrow \{0, 1\}^n$$
$$H(m) = h$$

takes an input message  $m$  of arbitrary length and outputs a fixed-length hash value  $h$  (sometimes also called a digest) of  $n$  bits. There is a technical definition for how a cryptographic hash function should behave, but the key properties that you should know are:

- uniformity: for a randomly chosen message  $m$ ,  $H(m)$  is distributed uniformly among  $\{0, 1\}^n$ ,
- avalanching: a small change in the input should result in an uncorrelated and completely different-looking digest.

One example of how a hash function might be used is to check for stray radiation causing a bit-flip in your computer's hard drive. If you store the message  $m$  and its hash  $h = H(m)$  on the hard drive, then later you can retrieve  $m'$  and  $h'$  and calculate  $H(m')$ . If  $H(m') = h'$ , then there is a high probability that non-deliberate corruption has not occurred. Conversely if  $H(m') \neq h'$ , then some corruption has happened.

## Sponge-based hash function

 This section is about the first program you'll create, the `hash` program.

You should implement the function `hash` in `src/crypto.c` using the sponge construction. This function should accept an input message of specified length and output a digest of specified length.

We begin by defining `RATE = 16` in `src/crypto.c`. The sponge's rate is the maximum number of bytes that can be read from or written to the sponge's state using `sponge_read` or `sponge_write` before the sponge must be permuted via `sponge_permute`. After zeroing the sponge's state with `sponge_init`, the hash function proceeds in three stages:

## Absorbing phase

The input message is ‘absorbed’ into the sponge by alternating between writing subsequent blocks of size `RATE` bytes from the message (setting `bw_xor` to `true`) into the sponge, and permuting the sponge’s state. In the last block, there will be between zero and `RATE - 1` (inclusive) bytes left to write; you should just write these remaining  $r$  bytes into the sponge without permuting immediately after.

## Demarcation phase

The two constant bytes `DELIMITER_A` and `DELIMITER_B` are defined in `src/crypto.c`. You should use `sponge_demarcate` to absorb `DELIMITER_A` into the state’s  $r$ ’th byte (zero-based indexing, where  $r$  was defined in the absorbing phase). You should then demarcate with `DELIMITER_B` into the state’s  $(RATE - 1)$ ’th byte. Finally, permute the sponge’s state.

## Squeezing phase

We can now ‘squeeze’ a hash value from the state. You should alternate between reading `RATE` bytes from the sponge’s state using `sponge_read` and permuting the sponge’s state. In the last block, you may have fewer than `RATE` bytes to read; just read that number of bytes from the sponge.

## Implementation and testing

You should view the comments in `include/crypto.h` for descriptions of the inputs and outputs to `hash`, and the comments in `src/crypto.c` for some examples demonstrating the correct number of writes and reads to the sponge’s state that you need to be doing. Note that output-based testing will be less useful for all tasks in this programming assignment, so you may find `gdb` more useful in ensuring functions are called in the correct order with the arguments you expect. You may test whether your program is working by clicking the `Mark` button on Ed, similar to the first assignment. You should be able to observe how changes to single characters in the input file lead to completely different looking hash values, demonstrating the avalanching property.

You can make the `hash` program by running `make hash`.

Input for `hash` is in the form:

```
./hash <digest length in bytes> <input file>
```

Where:

- `<digest length>` is the length the output hash will be in bytes.
- `<input file>` is the file to hash the contents of.

The output is a string representing the hash of the file in hexademical encoding, e.g.

```
./hash 32 tests/panda_1.txt
```

# Message Authentication Code (MAC) System

**i** This section is about the second program you'll create, `mac`.

We saw in the previous section how a cryptographic hash function can be used to provide a check for data integrity by detecting non-deliberate corruption with high probability. In the face of a sophisticated attacker however, this is not enough. In our hard drive example, an attacker with unnoticed access to our machine could just alter our stored file, recalculate its hash and overwrite our stored digest. Then, we would not be able to detect this kind of tampering since the stored hash would match the hash that we calculate for the tampered file.

The key to solving this problem is to use a key! The key  $K$  and message  $m$  are inputs to the message authentication code (MAC) algorithm

$$\begin{aligned} \text{MAC} : \{0, 1\}^k \times \{0, 1\}^* &\rightarrow \{0, 1\}^n \\ \text{MAC}(K, m) &= t. \end{aligned}$$

where  $k$  is the length of the key  $K$  in bits, the message  $m$  is of arbitrary length and the output *authentication tag* is of length  $n$  bits. In the hard drive example, we would generate a random key and use it in the MAC algorithm, storing the message and resulting tag on the hard drive, but keeping the key stored separately in some secure environment. Our MAC algorithm should be designed such that any tampering from an attacker who does not possess the key can be detected with high probability. If an attacker alters the stored message or tag in any way, recalculating the MAC on the message with our secure secret key should result in a tag mismatch, which alerts us to a tampering attempt.

For your implementation of the `mac` function in `src/crypto.c`, you will see `CRYPTO_KEY_SIZE = 32` defined in `include/crypto.h` indicating the use of 32-byte (256-bit) keys. You should begin by absorbing the input key into a zeroed sponge. Since `RATE` divides `CRYPTO_KEY_SIZE` exactly, this should be a simple matter of calling `sponge_write` (with `bw_xor` set to `true`) for the first key block, then permuting the sponge, then writing the 2nd key block into the sponge, then performing one more permutation. Let this phase be known as the keying phase.

The rest of the MAC algorithm then proceeds with an absorbing, demarcation and squeezing phase as described for the hash function. In essence, you will be calculating  $\text{MAC}(K, m) = H(K||m)$ , where  $||$  in this notation represents concatenation (rather than a logical OR in C).

## Implementation and testing

You should view the comments in `include/crypto.h` for descriptions of the inputs and outputs to `mac`, and the comments in `src/crypto.c` for some hints on how to implement this approach. You may find the sponge diagram useful in approaching this task.

You can make the `mac` program by running `make mac`.

Input for `mac` is in the form:

```
./mac <tag length in bytes> <key file> <message file>
```


Where:

- `<tag length in bytes>` is the length the output MAC authentication tag will be in bytes.
- `<key file>` is the binary file containing the key, two are provided for you in `tests/key_1.key`, `tests/key_2.key`.
- `<message file>` is the file to generate the MAC of.

The output is a string representing the MAC authentication tag of the file in hexadecimal encoding, e.g.

```
./mac 32 tests/key_1.key tests/panda_1.txt  
b31bd00e9663482c0466d63923330d587dd82bde3549ad47952d688f73025959
```

## Authenticated Encryption (AE) Scheme

 This section is about the final pair of programs you'll create, `encr` and `decr`

Say that you and your equally security-conscious friend have managed to generate and store a key  $K$  that only both of you hold. Using the MAC algorithm, you can now send and receive messages between yourselves over the internet, and be almost completely assured that they were not tampered with in transit by someone without the key. However, there is one major issue: anyone over the network can still read your messages! Clearly this is not ideal if you don't want anyone but your friend to know about your amazing study routine for COMP20007. Therefore, you will implement a scheme for authenticated encryption (AE) which provides a check for data integrity and authenticity, and also provides confidentiality. This prevents outsiders from reading your precious messages.

## Encryption

The encryption routine

$$\text{AuthEncr} : \{0, 1\}^k \times \{0, 1\}^l \rightarrow \{0, 1\}^n \times \{0, 1\}^l$$
$$\text{AuthEncr}(K, m) = (t, c)$$

takes a key  $K$  of length  $k$  bits and a plaintext message  $m$  of length  $l$  bits as input, and outputs an authentication tag  $t$  of length  $n$  bits and the encrypted ciphertext  $c$  also of length  $l$  bits.

You should implement the `auth_encr` function in `src/crypto.c`. The scheme is almost identical to

the MAC algorithm. However, we make a slight alteration to the absorbing phase. Immediately after every call to `sponge_write` during the absorbing phase (but not the keying phase!), you should `sponge_read` an identical number of bytes into the ciphertext buffer.

## Decryption

The decryption routine

$$\text{AuthDecr} : \{0,1\}^k \times \{0,1\}^n \times \{0,1\}^l \rightarrow \{0,1\}^l \times \{0,1\}$$
$$\text{AuthDecr}(K, t, c) = \begin{cases} (m, 0) & \text{if authentication tag is valid} \\ (m, 1) & \text{otherwise} \end{cases}$$

takes a key  $K$  of length  $k$  bits, a tag  $t$  of length  $n$  bits and a ciphertext message  $c$  of length  $l$  bits as input. It should output the decrypted plaintext, and an integer 0 if the tag is valid for this message, or 1 otherwise. Given the description of the encryption routine, you should be able to implement decryption in `auth_decr` in `src/crypto.c`. You will have to use `sponge_write` with `bw_xor` set to `false` during the absorbing phase.

## Implementation and testing

You should view the comments in `include/crypto.h` for descriptions of the inputs and outputs to `encr` and `decr` and the comments in `src/crypto.c` for some hints on how to implement this approach. You may find the sponge diagram particularly useful in approaching the `decr` task.

You can make the `encr` and `decr` programs by running `make encr` and `make decr` (respectively) or simply `make all` to make both.

Input for `encr` is in the form:

```
./encr <tag length in bytes> <key file> <plaintext file> <optional: concatenated tag+ciphertext file>
```

Where:

- `<tag length in bytes>` is the length the output MAC authentication tag will be in bytes.
- `<key file>` is the binary file containing the key, two are provided for you in `tests/key_1.key`, `tests/key_2.key`.
- `<plaintext file>` is the file to encrypt.
- `<optional: concatenated tag+ciphertext file to write>` is an optional argument which if given is the file the binary output of the tag and ciphertext are written to.



Note that `decr` reads a file in the binary encoding, so the `stdout` output will not work as an input, the optional file must be used.

The output is a string representing the MAC authentication tag of the file in hexademical encoding,



followed by a colon and the encrypted ciphertext, also in hexademical encoding e.g.

```
./encr 32 tests/key_1.key tests/panda_2.txt  
16f0e981d627acd926f05ec95430db76c7a178a8d2fe8ebaa28eafaec3030c:4f63f7696ebc0326c741484094f575b8dd
```

For `decr`, input is in the form:

```
./decr <tag length in bytes: must match tag length used for encryption> <key file> <concatenated tag
```

Where:

- `<tag length in bytes: must match tag length used for encryption>` is the length of the output MAC authentication tag used during encryption.
- `<key file>` is the binary file containing the key, two are provided for you in `tests/key_1.key`, `tests/key_2.key`. For decryption to succeed, the key must match the key used for encryption.
- `<concatenated tag+ciphertext file>` is the (optional) output file from `encr`, containing the MAC authentication tag and ciphertext generated. Two pairs of encrypted outputs are provided for you, `tests/encr-key_1-panda_1.bin` and `tests/encr-key_1-panda_2.bin`, which are `panda_1` and `panda_2` encrypted with `key_1` (respectively), and `tests/encr-key_2-panda_1.bin` and `tests/encr-key_2-panda_2.bin`, which are `panda_1` and `panda_2` encrypted with `key_2` (respectively).

The output will either be a note about successful decryption, e.g.

```
./decr 32 tests/key_1.key tests/encr-key_1-panda_1.bin  
Decryption successful. Printing decrypted file below dashes:  
-----  
A panda eats: shoots and leaves.
```

or an error indicating something went wrong:

```
./decr 32 tests/key_1.key tests/encr-key_2-panda_1.bin  
Error: invalid tag. Wrong decryption key used, or tampering or corruption has occurred.
```

[1] <https://en.wikipedia.org/wiki/File:SpongeConstruction.svg>

---

## Problem 1: Cryptography using the Sponge Construction

*This code slide does not have a description.*

---

## Problem 2: Finding the Median in a 2-3-4 Tree

This problem looks at an addition to the 2-3-4 tree of a new function `findMedian`.

There are four written parts and one programming part for this problem.

For a set of  $n + 1$  inputs in sorted order, the median value is the element with  $\frac{n}{2}$  values both above and below it.

### Part A

For the first part, assume the 2-3-4 tree is unmodified, write pseudocode in `written-problem.txt` for an algorithm which can find the median value.

### Part B

For the second part, assume you are now allowed to keep track of the number of descendants during insertion, write pseudocode in `written-problem.txt` to update the number of descendants of a particular node. You may assume other nodes have been updated already.

### Part C

For the third part, write pseudocode in `written-problem.txt` for an efficient algorithm for determining the median.

### Part D

For the fourth part, determine and justify the complexity of your efficient approach in Part C in `written-problem.txt`.

### Part E

For the final part, a modified version of the 2-3-4 tree from the workshop solutions has been provided. Your task is to modify the existing `insertTree` function to implement your Part B algorithm and to implement the `findMedian` function according to your Part C approach. Your program should construct a 2-3-4 tree for a given set of numbers given on `stdin`, and output the median value and the level (starting from the root as level 0) which this value was found on in the 2-3-4 tree.

You may assume for simplicity that the number of elements inserted into the tree is always odd.

---

## Problem 3: Hacking Merge Sort

You are a hacker that managed to infiltrate the central server of a company. With some effort, you were able to find their sorting function, which uses Mergesort. Here is how it was implemented, shown in pseudocode:

```
function MERGESORT( $A[0..n - 1]$ )
  if  $n > 1$  then
     $\text{mid} \leftarrow \lfloor n/2 \rfloor$ 
     $B[0..\text{mid} - 1] \leftarrow A[0..\text{mid} - 1]$ 
     $C[0..n - \text{mid} - 1] \leftarrow A[\text{mid}..n - 1]$ 
    MERGESORT( $B[0..\text{mid}]$ )
    MERGESORT( $C[0..\text{mid}]$ )
    MERGE( $B, C, A$ )

function MERGE( $B[0..p - 1], C[0..q - 1], A[0..p + q - 1]$ )
   $i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$ 
  while  $i < p$  and  $j < q$  do
    if  $B[i] \leq C[j]$  then
       $A[k] \leftarrow B[i]; i \leftarrow i + 1$ 
    else
       $A[k] \leftarrow C[j]; j \leftarrow j + 1$ 
     $k \leftarrow k + 1$ 
  if  $i == p$  then
     $A[k..p + q - 1] \leftarrow C[j..q - 1]$ 
  else
     $A[k..p + q - 1] \leftarrow B[i..p - 1]$ 
```

Your plan is to change this sorting algorithm to perform a kind of Denial-of-Service attack. The idea is to keep the algorithm correct, but make it substantially slower. To do this, you modified the Mergesort function by injecting a single line of code, shown in blue below:

```
function MERGESORT( $A[0..n - 1]$ )
  if  $n > 1$  then
     $\text{mid} \leftarrow \text{HASH}(\lfloor n/2 \rfloor)$ 
     $B[0..\text{mid} - 1] \leftarrow A[0..\text{mid} - 1]$ 
     $C[0..n - \text{mid} - 1] \leftarrow A[\text{mid}..n - 1]$ 
    MERGESORT( $B[0..\text{mid}]$ )
    MERGESORT( $C[0..\text{mid}]$ )
    MERGE( $B, C, A$ )
```

In other words, the variable "mid" now receives the output of a hash function called "HASH", which in turn receives the original value of "mid" as its argument. The Merge function does not change.

Write, in pseudocode, an implementation of HASH that forces Mergesort to become a quadratic algorithm in the worst case, in other words:

$$\Theta(n^2)$$

Mergesort should still return a valid sorted output. Assume that the basic operation is *to compare one element with another*. You can upload a figure or a PDF as the answer if you prefer.

**Question 2** *Saved May 11th 2022 at 7:51:06 pm*

Explain why your implementation forces Mergesort to become a quadratic algorithm in the worst case. You can give an example to show your reasoning. You can upload a figure or a PDF as the answer if you prefer.

**Question 3** *Saved May 11th 2022 at 7:54:55 pm*

Now assume the basic operation is *to read an element from memory*. What is the best case complexity in your implementation? Give a proof of your best case complexity bound. You can upload a figure or a PDF as the answer if you prefer.

---

# Requirements

The following implementation requirements must be adhered to:

- You must write your implementation in the C programming language.
- You must write your code in a modular way, so that your implementation could be used in another program without extensive rewriting or copying. This means that the tree and cryptography operations are kept together in separate .c files, with their own header (.h) files, separate from the main program.
- Your tree code should be easily extensible to different trees. This means that the functions for modifying parts of your tree should take as arguments not only the values required to perform the operation required, but also a pointer to the tree.
- **Your implementation must read the input file once only.**
- Your program should store strings in a space-efficient manner. If you are using malloc() to create the space for a string, remember to allow space for the final end of string '\0' (NULL).
- A full Makefile is not provided for you - the Makefile provided with the scaffold will need to be modified if you make changes to the structure of the program. The Makefile should direct the compilation of your program. To use the Makefile, make sure it is in the same directory as your code, and type `make hash`, `make mac`, `make encr`, `make decr` for the first programming task and `make findMedian` for the second task. You must submit your Makefile with your assignment.

Hint: If you haven't used `make` before, try it on simple programs first. If it doesn't work, read the error messages carefully. A common problem in compiling multifile executables is in the included header files. Note also that the whitespace before the command is a tab, and not multiple spaces. It is not a good idea to code your program as a single file and then try to break it down into multiple files. Start by using multiple files, with minimal content, and make sure they are communicating with each other before starting more serious coding.

---

# Programming Style

Below is a style guide which assignments are evaluated against. For this subject, the 80 character limit is a guideline rather than a rule - if your code exceeds this limit, you should consider whether your code would be more readable if you instead rearranged it.

```
/** *****
 * C Programming Style for Engineering Computation
 * Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au) 13/03/2011
 * Definitions and includes
 * Definitions are in UPPER_CASE
 * Includes go before definitions
 * Space between includes, definitions and the main function.
 * Use definitions for any constants in your program, do not just write them
 * in.
 *
 * Tabs may be set to 4-spaces or 8-spaces, depending on your editor. The code
 * Below is ``gnu'' style. If your editor has ``bsd'' it will follow the 8-space
 * style. Both are very standard.
 */

/**
 * GOOD:
 */

#include <stdio.h>
#include <stdlib.h>
#define MAX_STRING_SIZE 1000
#define DEBUG 0
int main(int argc, char **argv) {
    ...

/**
 * BAD:
 */

/* Definitions and includes are mixed up */
#include <stdlib.h>
#define MAX_STING_SIZE 1000
/* Definitions are given names like variables */
#define debug 0
#include <stdio.h>
/* No spacing between includes, definitions and main function*/
int main(int argc, char **argv) {
    ...

/** *****
 * Variables
 * Give them useful lower_case names or camelCase. Either is fine,
 * as long as you are consistent and apply always the same style.
```

```

* Initialise them to something that makes sense.
*/

/**
* GOOD: lower_case
*/

int main(int argc, char **argv) {

    int i = 0;
    int num_fifties = 0;
    int num_twenties = 0;
    int num_tens = 0;

    ...
/**
* GOOD: camelCase
*/

int main(int argc, char **argv) {

    int i = 0;
    int numFifties = 0;
    int numTwenties = 0;
    int numTens = 0;

    ...
/**
* BAD:
*/

int main(int argc, char **argv) {

    /* Variable not initialised - causes a bug because we didn't remember to
    * set it before the loop */
    int i;
    /* Variable in all caps - we'll get confused between this and constants
    */
    int NUM_FIFTIES = 0;
    /* Overly abbreviated variable names make things hard. */
    int nt = 0

    while (i < 10) {
        ...
        i++;
    }

    ...

/** *****
* Spacing:
* Space intelligently, vertically to group blocks of code that are doing a
* specific operation, or to separate variable declarations from other code.
* One tab of indentation within either a function or a loop.

```



```

* Spaces after commas.
* Space between ) and {.
* No space between the ** and the argv in the definition of the main
* function.
* When declaring a pointer variable or argument, you may place the asterisk
* adjacent to either the type or to the variable name.
* Lines at most 80 characters long.
* Closing brace goes on its own line
*/

```

```

/**
* GOOD:
*/

```

```

int main(int argc, char **argv) {

    int i = 0;

    for(i = 100; i >= 0; i--) {
        if (i > 0) {
            printf("%d bottles of beer, take one down and pass it around,"
                " %d bottles of beer.\n", i, i - 1);
        } else {
            printf("%d bottles of beer, take one down and pass it around."
                " We're empty.\n", i);
        }
    }

    return 0;
}

```

```

/**
* BAD:
*/

```

```

/* No space after commas
* Space between the ** and argv in the main function definition
* No space between the ) and { at the start of a function */
int main(int argc,char ** argv){
    int i = 0;
    /* No space between variable declarations and the rest of the function.
    * No spaces around the boolean operators */
    for(i=100;i>=0;i--) {
        /* No indentation */
        if (i > 0) {
            /* Line too long */
            printf("%d bottles of beer, take one down and pass it around, %d
bottles of beer.\n", i, i - 1);
        } else {
            /* Spacing for no good reason. */

            printf("%d bottles of beer, take one down and pass it around."
                " We're empty.\n", i);
        }
    }
}

```

```

}
}
/* Closing brace not on its own line */
return 0;}

/** *****
 * Braces:
 * Opening braces go on the same line as the loop or function name
 * Closing braces go on their own line
 * Closing braces go at the same indentation level as the thing they are
 * closing
 */

/**
 * GOOD:
 */

int main(int argc, char **argv) {

    ...

    for(...) {
        ...
    }

    return 0;
}

/**
 * BAD:
 */

int main(int argc, char **argv) {

    ...

    /* Opening brace on a different line to the for loop open */
    for(...)
    {
        ...
        /* Closing brace at a different indentation to the thing it's
        closing
        */
    }

    /* Closing brace not on its own line. */
    return 0;}

/** *****
 * Commenting:
 * Each program should have a comment explaining what it does and who created
 * it.
 * Also comment how to run the program, including optional command line
 * parameters.

```

```

* Any interesting code should have a comment to explain itself.
* We should not comment obvious things - write code that documents itself
*/

/**
 * GOOD:
 */

/* change.c
 *
 * Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au)
 13/03/2011
 *
 * Print the number of each coin that would be needed to make up some
 change
 * that is input by the user
 *
 * To run the program type:
 * ./coins --num_coins 5 --shape_coins trapezoid --output blabla.txt
 *
 * To see all the input parameters, type:
 * ./coins --help
 * Options::
 *   --help           Show help message
 *   --num_coins arg   Input number of coins
 *   --shape_coins arg Input coins shape
 *   --bound arg (=1)  Max bound on xxx, default value 1
 *   --output arg      Output solution file
 *
 */

int main(int argc, char **argv) {

    int input_change = 0;

    printf("Please input the value of the change (0-99 cents
 inclusive):\n");
    scanf("%d", &input_change);
    printf("\n");

    // Valid change values are 0-99 inclusive.
    if(input_change < 0 || input_change > 99) {
        printf("Input not in the range 0-99.\n")
    }

    ...

/**
 * BAD:
 */

/* No explanation of what the program is doing */
int main(int argc, char **argv) {

```

```

/* Commenting obvious things */
/* Create a int variable called input_change to store the input from
the
* user. */
int input_change;

...

/** *****
* Code structure:
* Fail fast - input checks should happen first, then do the computation.
* Structure the code so that all error handling happens in an easy to read
* location
*/

/**
* GOOD:
*/
if (input_is_bad) {
    printf("Error: Input was not valid. Exiting.\n");
    exit(EXIT_FAILURE);
}

/* Do computations here */
...

/**
* BAD:
*/

if (input_is_good) {
    /* lots of computation here, pushing the else part off the screen.
    */
    ...
} else {
    fprintf(stderr, "Error: Input was not valid. Exiting.\n");
    exit(EXIT_FAILURE);
}

```

Some automatic evaluations of your code style may be performed where they are reliable. As determining whether these style-related issues are occurring sometimes involves non-trivial (and sometimes even undecidable) calculations, a simpler and more error-prone (but highly successful) solution is used. You may need to add a comment to identify these cases, so check any failing test outputs for instructions on how to resolve incorrectly flagged issues.

---

## Additional support

Your tutors will be available to help with your assignment during the scheduled workshop times. Questions related to the assignment may be posted on the Ed discussion forum, using the folder tag Assignments for new posts. You should feel free to answer other students' questions if you are confident of your skills.

A tutor will check the discussion forum regularly, and answer some questions, but be aware that for some questions you will just need to use your judgment and document your thinking. For example, a question asking whether your answer is correct will not be answered; you must perform tests and see whether your solution is appropriate for the problem.

If you have questions about your code specifically which you feel would reveal too much of the assignment, feel free to post a private question on the discussion forum.

---

## Submission

Your C code files (including your Makefile and any other files needed to run your code) should be submitted through Ed to this assignment. Your programs must compile and run correctly on Ed. You may have developed your program in another environment, but it still must run on Ed at submission time. For this reason, and because there are often small, but significant, differences between compilers, it is suggested that if you are working in a different environment, you upload and test your code on Ed at reasonably frequent intervals.

A common reason for programs not to compile is that a file has been inadvertently omitted from the submission. Please check your submission, and resubmit all files if necessary.

The theoretical analysis for Problem 2 should be submitted with your other files in the *Problem 2: Finding the Median in a 2-3-4 Tree* slide by filling in the text file `written-problem.txt` or as a PDF, referenced in `written-problem.txt` if more complex figures are required.

For Problem 3, submit your answers directly on the "Answer" box below each question. You are allowed to upload figures or PDF files to help explaining your answers for each question but this is not required.

---

# Assessment

There are a total of 20 marks given for this assignment.

Your C programs should be accurate, readable, and observe good C programming structure, safety and style, including documentation. Safety refers to checking whether opening a file returns something, whether mallocs do their job, etc. The documentation should explain all major design decisions, and should be formatted so that it does not interfere with reading the code. As much as possible, try to make your code self-documenting, by choosing descriptive variable names. The remainder of the marks will be based on the correct functioning of your submission.

Note that marks related to the correctness of your code will be based on passing various tests. If your program passes these tests without addressing the learning outcomes (e.g. if you fully hard-code solutions or otherwise deliberately exploit the test cases), you may receive less marks than is suggested but your marks will otherwise be determined by test cases.

For the cryptography problem, given the nature of the errors we expect to see, not passing the test cases may lead to some lost marks, but some partial marks may be awarded for answers which are correct but have minor errors (e.g. where the structure is correct but the wrong variable was used). Though as usual, it is best to endeavour to pass the test cases as this will lead to the least surprise when marking is returned.

## Mark Breakdown

Problem	Mark Distribution
Problem 1	2 Marks (Hash) + 2 Marks (MAC) + 2 Marks (Encr) + 3 Mark
Problem 2	1 Mark (Part A) + 1 Mark (Part B) + 1 Mark (Part C) + 1 Mark (Part D)
Problem 3	1 Mark (Question 1) + 2 Marks (Question 2) + 2 Marks (Ques

Marks will be given based on test case passing, a maximum of one mark may be deducted across the programming exercises for serious code style or structural errors. Note that not all marks will represent the same amount of work, you may find some marks are easier to obtain than others. Note also that the test cases are similarly not always sorted in order of difficulty, some may be easier to pass than others.

---

# Plagiarism

This is an individual assignment. The work must be your own.

While you may discuss your program development, coding problems and experimentation with your classmates, you must not share files, as this is considered plagiarism.

If you refer to published work in the discussion of your experiments, be sure to include a citation to the publication or the web link.

“Borrowing” of someone else’s code without acknowledgment is plagiarism. **Plagiarism is considered a serious offence at the University of Melbourne.** You should read the University code on Academic integrity and details on plagiarism. Make sure you are not plagiarising, intentionally or unintentionally.

You are also advised that there will be some programming-related questions in the final examination. Students who do not program their own assignments will be at a disadvantage for this part of the examination.



---

## Late policy

The late penalty is 20% of the available marks for that project for each day (or part thereof) overdue. Requests for extensions on medical grounds will need to be supported by a medical certificate. Any request received less than 48 hours before the assessment date (or after the date!) will generally not be accepted except in the most extreme circumstances. In general, extensions will not be granted if the interruption covers less than 10% of the project duration. Remember that faculty servers are often heavily loaded near project deadlines, and unexpected outages can occur; these will not be considered as grounds for an extension.

Students who experience difficulties due to personal circumstances are encouraged to make use of the appropriate University student support services, and to contact the lecturer, at the earliest opportunity.

Finally, we are here to help! There is information about getting help in this subject on the LMS. Frequently asked questions about the project will be answered on Ed.