C Final Test

Imperial College London

Department of Computing

Gogen

~~Monday 7 June 2021~~
~~09:00 to 12:00~~
~~THREE HOURS~~
~~(including 10 minutes planning time)~~

- The maximum total is **100 marks**: 50 for each of the two parts.
- **Important:** TEN MARKS will be deducted from solutions that do not compile. Comment out any code that does not compile before you submit.
- Your code needs to compile and work correctly in the test environment which will be the same as the lab machines.
- You will need to complete Part A to solve Questions 3, 4, 5 of Part B.
- The files can be found directly under the `gogen` directory inside your gitlab repository.
- ~~Push the final version of your code to `gitlab` before the deadline, and then go to `LabTS`, find the final/correct commit and submit it to `CATe`.~~
- **Important:** You should only modify files `letter_mask.c` and `gogen.c`. All other files are read-only, and are going to be overwritten by our autotester.

Overview

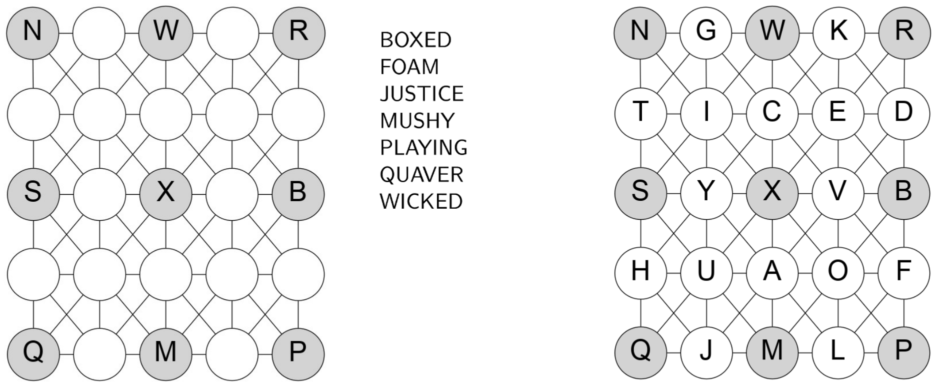

Figure 1: Gogen puzzle consisting of 5 × 5 board and list of words (left) and solution (right)

As shown on the left of Figure 1, a Gogen puzzle consists of a 5 × 5 board and a list of words. As shown on the right of Figure 1, the aim of Gogen is to insert the 25 letters 'A' to 'Y' ('Z' is not used) into the cells on the board such that (a) each letter appears in one (and only one) board cell, and (b) by moving one step (horizontally, vertically or diagonally) at a time, it is possible to spell out each of the words.

When solving Gogen puzzles, it helps to keep track of the possible board cell locations for each letter using a 5 × 5 bitmask known as a *letter mask.* In the above, the initial masks for the letter 'X' (said to be *fixed* because it has only one possible location), the letter 'E' (said to be *free* because it has more than one possible location) and the letter 'R' (also fixed) are shown in Figure 2.

The 1-neighbourhood of a mask describes the set of board cells that are at most one cell away from the cells marked in the mask. For example, the 1-neighbourhoods of 'X', 'E' and 'R' are shown in Figure 3.

Because 'X' and 'E' appear consecutively in the word BOXED, we know 'E' appears in the 1-neighbourhood of 'X'; likewise, because 'E' and 'R' appear consecutively in the word QUAVER, we know 'E' appears in the 1-neighbourhood of 'R'. Thus, by intersecting (and-ing) the mask for 'E', the mask for the 1-neighbourhood of 'X' and the mask for the 1-neighbourhood of 'R', we can compute an updated mask for 'E' as shown in Figure 4. With only one possible location, 'E' now becomes fixed.

Proceeding in this manner, it is possible to narrow down the possible locations for each letter. If a point is reached where all letters are fixed, the puzzle is solved. If some letters remain stubbornly free, it becomes necessary to recursively investigate the remaining options.

```
00000   01010   00001
00000   11111   00000
00100   01010   00000
00000   11111   00000
00000   01010   00000
```

Figure 2: Letter masks for 'X', 'E', 'R'

```
00000   11111   00011
01110   11111   00011
01110   11111   00000
01110   11111   00000
00000   11111   00000
```

Figure 3: 1-neighbourhood masks for 'X', 'E', 'R'

```
00000
00010
00000
00000
00000
```

Figure 4: Mask for 'E' after intersecting 1-neighbourhoods of 'X' and 'R'.

Part A: Data Structure – Bitmask for Letter Mask

We represent a letter mask using the 25 most significant bits of a `uint32_t`. The 7 least significant bits remain unused and should always have value 0. For example, the bitmasks for 'X', 'E' and 'R' from the Gogen game in Figure 1 are:

	Binary Number
'X' :	00000 00000 00100 00000 00000 0000000
'E' :	01010 11111 01010 11111 01010 0000000
'R' :	00001 00000 00000 00000 00000 0000000

Table 1: Representation of bitmasks for letters 'X', 'E' and 'R'.

For Part A the following files, functions and macros are of interest:

1. `letter_mask.h` contains:
   - definition for `letter_mask_t` which represents a letter mask
   - macros you might find useful:
     - `#define HEIGHT 5` the number of columns in the letter mask
     - `#define WIDTH 5` the number of rows in the letter mask
     - `#define NUM_LETTERS 25` the number of letters the game is using
     - `#define NUM_BITS 32` the number of bits in `uint32_t`
     - `#define MSB(x) (x >> 31)` to get the most significant bit of a 32-bit number
     - some constants that represent letter masks for specific configurations
   - declarations for the functions you should implement
2. `letter_mask.c` contains:
   - `print_bitmask(...)` prints a letter mask nicely (including the unused bits)
   - empty stubs of the functions you should implement

Important: Be aware that when working with 32-bit numbers (like `uint32_t`) shifts of more than 31 places have undefined behaviour.

Part A Tasks

Testing frequently, complete the following functions in file `letter_mask.c`:

1. `bool get_bit_value(letter_mask_t mask, int row, int col);`

   This function returns the value of the bit at the given row and column coordinates. Rows and columns are zero indexed. You should use assertions in your code to ensure the coordinates are valid.

   [5 Marks]
2. `void set_bit_value(letter_mask_t *mask, int row, int col, bool value);`

   This function sets the value of the bit at the given coordinates to the given value. You should use assertions in your code to ensure the coordinates are valid.

   [5 Marks]

3. `void set_all_bits(letter_mask_t *mask, bool value);`

   This function sets the values of all bits in the bitmask to the value given.

   [5 Marks]

4. `bool is_free_letter(letter_mask_t mask);`

   This function returns true if the given mask has at least two bits set and false otherwise.

   [10 Marks]

5. `bool get_fixed_letter_pos(letter_mask_t mask, int *row, int *col);`

   If the mask has exactly one bit set, this function sets the output parameters `row` and `col` to its coordinates and returns true, otherwise the function returns false.

   [10 Marks]

6. `void intersect_neighbourhoods(letter_mask_t *fst, letter_mask_t *snd);`

   This function updates both `fst` and `snd` by intersecting (and-ing) each of them with the 1-neighbourhood of the other. For example, calling this function with pointers to `fst` and `snd` whose masks are shown in Figure 5 should change them as shown in Figure 7.

```
00110          00000
00000          00000
00000          00000
01000          00000
00000          10000
```

Figure 5: Letter mask for `fst` (left) and `snd` (right).

```
01111          00000
01111          00000
11100          00000
11100          11000
11100          11000
```

Figure 6: 1-neighbourhood masks for `fst` (left) and `snd` (right).

```
00000          00000
00000          00000
00000          00000
01000          00000
00000          10000
```

Figure 7: Bitmasks for `fst` (left) and `snd` (right) after calling the function.

Hint 1: You might find it useful to create helper methods for shifting the 5 × 5 bitmask up, down, left, and right.

Hint 2: Remember to always set the 7 (least significant) unused bits to 0.

[15 Marks]

Part B: Algorithms – Solving Gogen Puzzle

We represent a Gogen board as a 2-dimensional (5×5) array of characters, where each empty position is marked by the '.' character. You can see an example of this in the `gogen-challenge.txt` file. The file also contains the number of words that form the puzzle and then the words themselves. The list of words is stored in memory as a `NULL` terminated array of strings. You can check the implementation of `parse_gogen_file(...)` for details on how the file is parsed.

The Gogen puzzle solving algorithm repeatedly uses the adjacent letters in the words to restrict the possible locations for a letter on the board. The possible locations are tracked using letter masks that are updated based on the board and the 1-neighbourhood intersection of adjacent letters in the words. Once a letter mask becomes fixed (exactly one bit set), the letter can be set on the board.

For Part B the following files and functions are of interest:

1. `gogen.h` contains:
   - definition for `board_t` which represents a board
   - declarations for the functions you should implement
2. `gogen.c` contains:
   - `print_board(...)` and `print_words(...)` are utility functions to be used during debugging
   - `copy_board(...)` can be used to copy a source board to a destination board
   - `main(void)` can be used for solving Question 5.
   - empty stubs of the functions you should implement

Important: A correct implementation for Part A is needed for Questions 3, 4 and 5 of Part B.

Part B Tasks

Testing frequently, complete the following functions in file `gogen.c`:

1. `bool get_letter_position(board_t board, char letter, int *row, int *col);`

   This function searches in a row-by-row fashion for the first occurrence of character `letter` in a given board. If the character is found, the function should return true and set the output parameters `row` and `col` to the row and column number (indexed from 0). If the character cannot be found, the function should return false.

   [7 Marks]

2. `bool valid_solution(board_t board, char **words);`

   This function returns true if the given board represents a solution to the Gogen puzzle with the given list of words. Read the game's description from the Overview Section to ensure all rules are checked.

   [10 Marks]

(See next page for Question 3)

3. `void update(board_t board, letter_mask_t masks[NUM_LETTERS]);`

   This function takes as parameters a partially completed board and an array of letter masks representing the possible positions of each letter from 'A' to 'Y'. The function mutually updates the board and letter masks by applying the following rules for each letter `ch`.

   (a) If `ch` is found in the board at position (r, c), then set all bits in the corresponding letter mask to be false with the exception of the bit at coordinates (r, c), which should be set to true.

   (b) If `ch` is not found on the board, then update the corresponding letter mask by marking each bit at coordinates where a letter exists on the board to false.

   (c) If `ch` is not found on the board and the previous step left the letter mask in a fixed position (exactly one bit set), then set the corresponding cell in the board to `ch`.

   For example, given the board shown on the left in Figure 1, the code on the left would update the masks for 'X' and 'E' as can be seen on the right:

```
letter_mask_t masks[NUM_LETTERS];
for (int i = 0; i < NUM_LETTERS; i++)
    set_all_bits(&masks[i], true);
update(board, masks);
```

```
X:  00000          E:  01010
    00000              11111
    00100              01010
    00000              11111
    00000              01010
```

   [15 Marks]

4. `bool solve_board(board_t board, char **words);`

   This function attempts to find a solution to a given Gogen puzzle. If a solution can be found, parameter `board` should contain the completed board and the function should return true. Otherwise the function should return `false`. A high-level overview of one possible approach is to perform the following steps:

   (a) Create an array of 25 masks (one for each letter 'A' to 'Y'), set all of their bits to true and use the `update(...)` function to initialise them.

   (b) Use the adjacent letters from each word with the `intersect_neighbourhoods(...)` and `update(...)` functions to refine the letter masks of free letters. Perform this step as long as the letter masks still have free letters and the number free letters is decreasing. You might find it useful to create a utility function that counts the number of free letters from an array of letter masks.

   (c) If the number of free letters does not decrease after iterating over all the words, we take a guess and fix a free letter on one of its possible positions. We recursively attempt to solve the board updated with this guess. If the guess fails, we backtrack and attempt to fix the free letter at another possible position. Once all possible positions are exhausted for a letter we can return false as the puzzle has no possible solution.

   [15 Marks]

5. Find the solution to the Gogen game from `gogen-challenge.txt`. Paste it in `gogen.c` as a comment above the `main` function.

   [3 Marks]

Building and Testing

The folder `gogen` contains both a `Makefile` (for use with the command line and a text editor) and a `CMakeLists.txt` (for use with CLion, if you wish to use an IDE).

Targets: Both the `Makefile` and the `CMakeLists.txt` specify 5 targets:

1. `letter_mask` runs the `main` function provided to you in `letter_mask.c`. You may use this to debug your code.
2. `gogen` runs the `main` function provided in `gogen.c`. You may use this to debug your code.
3. `testA` runs all the tests for Part A.
4. `testB` runs all the tests for Part B.

If using `make`, enter the `gogen` folder and build a specific target (e.g. `make gogen`, `make testA`) or `make all` to build all the aforementioned targets.

You can now run each target from the `gogen` folder. You should at least run the following commands, as this is what **the auto-tester will run**:

1. `./testA`
2. `./testB`

If using CLion, import your project using: `File` → `New CMake Project from Sources` and open the `gogen` folder as: `Open Existing Project`.

You can now build and run each target by selecting its configuration from: `Run` → `Edit Configurations` and then running it with `Run` → `Run`.

Important note to Windows users: As your code needs to compile in the test environment, we suggest you ssh on a lab machine and test your code there as well. A common issue is detecting the end of line using only '\r' as delimiter. You should use '\n' as well.

Important: In case you accidentally overwrite any file, you can revert it to a previous version using: `git checkout <git-commit-hash> -- <file-name>`

Good luck!