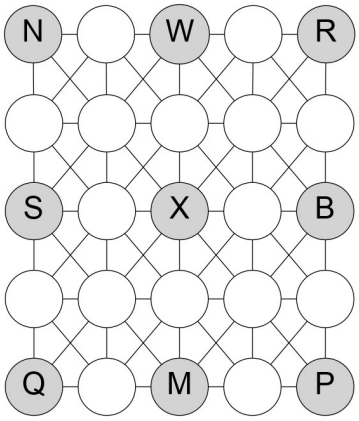
MSc (Computing) 2020-2021
C/C++ Laboratory Examination (Timed Remote Assessment)
Imperial College London

Tuesday 12 January 2021, 09h00 – 11h00

The solution often turns out more beautiful than the puzzle.

Richard Dawkins



- ☞ You are advised to use the first 10 minutes to read through the questions.
- ☞ You may work locally on your own computer or connect to DoC lab computers and work on them.
- ☞ You must add to the pre-supplied header file **gogen.h**, pre-supplied implementation file **gogen.cpp** according to the specifications overleaf. You are also pre-supplied with a header **common.h**, as well as functions for manipulating masks in header file **mask.h**, and a main program in **main.cpp**.
- ☞ You will find source files **gogen.cpp**, **gogen.h**, **common.h**, **mask.h** and **main.cpp**, and data files **board-easy.txt**, **board-medium.txt**, **board-hard.txt**, **words-easy.txt**, **words-medium.txt**, **words-hard.txt**, **solution-easy.txt**, **solution-wrong1.txt** and **solution-wrong2.txt** in the **skeleton.zip** file associated with the exercise. If one of these files is missing please alert one of the examiners by email to **wjk@imperial.ac.uk** or **fp910@imperial.ac.uk**.
- ☞ Submit your **gogen.cpp** and **gogen.h** files into the CATe system **before** the end of the exam. You should not (and cannot) submit any other files.
  LATE SUBMISSIONS WILL NOT BE ACCEPTED.
- ☞ You are advised to **save your work regularly** and to **make regular submissions of your work into CATe**.
- ☞ No communication with any other student or persons other than the examiners is permitted. You are bound by the Honour Code for Timed Remote Assessments.
- ☞ On page 7, there are hints that you should find helpful.
- ☞ **This question paper consists of 10 pages.**

Problem Description

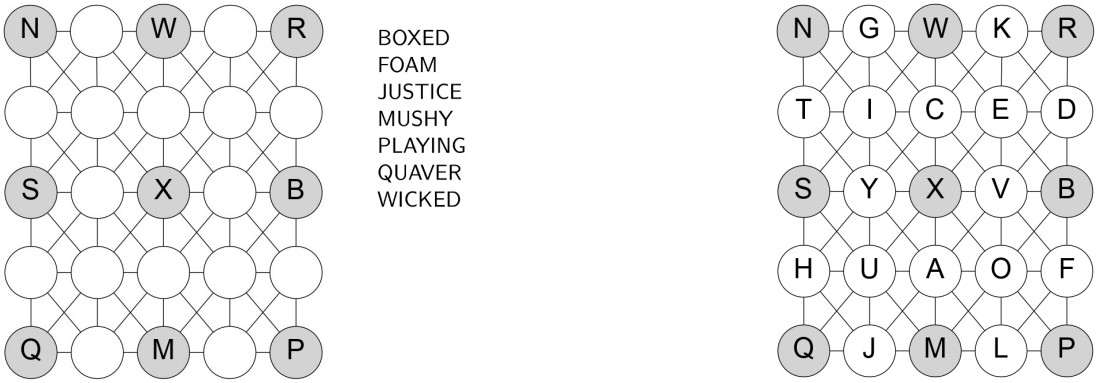


Figure 1: Gogen puzzle consisting of 5×5 board and list of words (left) and solution (right)

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 an array of Boolean values known as a *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, respectively (0=false, 1=true):

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

The *d-neighbourhood* of a mask describes the set of board cells that are up to d cells away from those cells marked in the mask. For example, the 1-neighbourhoods of 'X', 'E' and 'R' are, respectively:

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

Because 'X' and 'E' appear consecutively in the word BOXED, we know 'E' appears in the 1-neighbourhood of 'X'[1]; 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:

```
00000
00010
00000
00000
00000
```

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.

[1] We also know that 'X' appears in the 1-neighbourhood of 'E' but this is not insightful on this occasion.

Pre-supplied functions and files

You are supplied with a main program in **main.cpp**, and several data files. In particular:

- **board-easy.txt**, **board-medium.txt** and **board-hard.txt** contain the initial boards for Gogen puzzles of easy, medium and hard difficulties respectively. The contents of **board-easy.txt** are:

```
N.W.R
.....
S.X.B
.....
Q.M.P
```

- **words-easy.txt**, **words-medium.txt**, and **words-hard.txt** contain the corresponding word list for each puzzle. Each word appears on a separate line.
- **solution-easy.txt** contains a valid solution to the easy Gogen puzzle, while **solution-wrong1.txt** and **solution-wrong2.txt** contain invalid solutions to the easy Gogen puzzle.

You are also supplied with the beginnings of the header file **gogen.h** (for your function prototypes) and the beginnings of the implementation file **gogen.cpp** (for your function definitions).

The file **gogen.cpp** includes the definition of several pre-supplied functions:

```
char **load_board(const char *filename);
bool save_board(char **board, const char *filename);
void print_board(char **board);
char **load_words(const char *filename);
void print_words(char **words);
void delete_board(char **board);
void delete_words(char **words);
```

- `load_board(...)` reads a Gogen board from a file and returns a two-dimensional array of characters describing the board. Memory for the board is allocated on the heap; it can be freed later using the function `delete_board(...)`.
- `save_board(...)` outputs a Gogen board to a file. It will be used for saving your solutions to the easy, medium and hard puzzles.
- `print_board(...)` prints the contents of the two-dimensional array of characters describing the Gogen board in an appropriate format.
- `load_words(...)` reads a word list from a file and returns a NULL-terminated array of character pointers, each of which points to one word. Memory for the words is allocated on the heap; it can be freed later using the function `delete_words(...)`.
- `print_list(...)` prints out the word list.

The header file **common.h** simply defines two constants `HEIGHT` and `WIDTH`, both as 5. You are also supplied with a header file **mask.h** which contains the definition of a Mask class. The section "More About Using Masks" found towards the end of this document will help you understand how to use masks.

Specific Tasks

1. Write a Boolean function `get_position(board, ch, row, column)` which searches for the first occurrence of character `ch` in a given board when searched in a row-by-row fashion. If the character is found, the function should return true and set the output parameters `row` and `column` to the row and column number (indexed starting from 0). If the character cannot be found, the function should return false and `row` and `column` should both be set to -1.

   For example, given the board shown on the left in Figure 1, the code:

   ```
   char **board = load_board("board-easy.txt");
   int row, column;
   if (get_position(board, 'B', row, column))
     cout << "'B' found in row " << row << ", column " << column << endl;
   else
     cout << "'B' could not be found" << endl;
   ```

   should display the output

   ```
   'B' found in row 2, column 4
   ```

   Similarly, the code

   ```
   if (get_position(board, 'O', row, column))
     cout << "'O' found in row " << row << ", column " << column << endl;
   else
     cout << "'O' could not be found" << endl;
   ```

   should display the output

   ```
   'O' could not be found
   ```

2. Write a Boolean function `valid_solution(board, words)` which returns true if the given board represents a solution to the Gogen puzzle with the given list of words.

   For example, given the words in **words-easy.txt**, and a file **solution-easy.txt** containing the following lines:

   ```
   NGWKR
   TICED
   SYXVB
   HUAOF
   QJMLP
   ```

   Then the code:

   ```
   char **solution = load_board("solution-easy.txt");
   char **words = load_words("words-easy.txt");
   cout << "Solution in 'solution-easy.txt' is "
        << (valid_solution(solution, words) ? "valid" : "invalid")
        << "!" << endl;
   ```

   should display the output

   ```
   Solution in 'solution-easy.txt' is valid!
   ```

3. (a) Write a (void) function `update(board, ch, mask)` which mutually updates the board and a mask for letter `ch` according to the following rules:
   - If `ch` is found in the board at position (r, c), then set every entry in `mask` to be false with the exception of element (r, c), which should be set to true, and return from the function.
   - For every cell in the board that is marked with a letter (that is not `ch`), set the corresponding element in `mask` to false.
   - If there is one (and only one) cell with the value true in `mask` then set the corresponding cell in the board to `ch`.

   For example, given the board shown on the left in Figure 1, the code:

```
Mask x; // by default all bits set to all true
update(board, 'X', x);
cout << "Mask for 'X'" << endl;
x.print();
```

   should result in the output

```
Mask for 'X'
00000
00000
00100
00000
00000
```

   Likewise, given the same board, the code:

```
Mask e; // by default all bits set to all true
update(board, 'E', e);
cout << "Mask for 'E'" << endl;
e.print();
```

   should result in the output:

```
Mask for 'E'
01010
11111
01010
11111
01010
```

   (b) Write a (void) function `neighbourhood_intersect(one, two)` which modifies masks `one` and `one` by intersecting each with the 1-neighbourhood of the other. For example, the code:

```
Mask x, e;
update(board, 'X', x);
update(board, 'E', e);
neighbourhood_intersect(x, e); // from BOXED
cout << "After neighbourhood intersection, mask for 'E' is " << endl;
e.print();
```

   should result in the output

```
After neighbourhood intersection, mask for 'E' is
00000
01110
01010
01110
00000
```

4. Write a Boolean function `solve_board(board, words)` which attempts to find a solution to a given Gogen puzzle. If a solution can be found, parameter `board` should contain the completed board. Otherwise the function should return `false`. A high-level overview of one possible approach is:

   - Create an array of 25 masks (one for each letter 'A' to 'Y') and use the `update(...)` function to initialise them.
   - Then repeatedly use the adjacent letters appearing in each word with the `neighbourhood_intersect(...)` and `update(...)` functions to refine the masks of free letters as much as possible.
   - Finally, if it necessary, recursively investigate the remaining options for the free letters. This should not be necessary with the "easy" puzzle, but will be needed for the "medium" and "hard" puzzles.

   For example, the code:

```
char **board = load_board("board-easy.txt");
char **words = load_words("words-easy.txt");

if (solve_board(board, words)) {
  cout << "Board solved! Solution: " << endl;
  print_board(board);
  save_board(board, solution_filename);
} else
  cout << "Board could not be solved" << endl;
```

   should result in the following output:

```
Board solved! Solution:
[N]--[G]--[W]--[K]--[R]
 | \/ | \/ | \/ | \/ |
 | /\ | /\ | /\ | /\ |
[T]--[I]--[C]--[E]--[D]
 | \/ | \/ | \/ | \/ |
 | /\ | /\ | /\ | /\ |
[S]--[Y]--[X]--[V]--[B]
 | \/ | \/ | \/ | \/ |
 | /\ | /\ | /\ | /\ |
[H]--[U]--[A]--[O]--[F]
 | \/ | \/ | \/ | \/ |
 | /\ | /\ | /\ | /\ |
[Q]--[J]--[M]--[L]--[P]
```

   For full credit for this question, your function – or helper function if you choose to use one – must be recursive.

(The four parts carry, respectively, 20%, 30%, 20% and 30% of the marks)

What to hand in

Place your function implementations in the file **gogen.cpp** and corresponding function declarations in the file **gogen.h**. Use the file **main.cpp** to test your functions. You may wish to create a **makefile** which will compile your submission into an executable file entitled **gogen**.

Submit your **gogen.cpp** and **gogen.h** files into CATe regularly. **You do not need to (and indeed are unable to) hand in any other files**.

Hints

1. You will save time if you begin by studying the main program in **main.cpp**, the pre-supplied implementation file **gogen.cpp**, the pre-supplied header files **gogen.h**, **common.h** and **mask.h** and the given data files **board-easy.txt**, **board-medium.txt**, **board-hard.txt**, **words-easy.txt**, **words-medium.txt**, **words-hard.txt**, **solution-easy.txt**, **solution-wrong1.txt** and **solution-wrong2.txt**.
2. Do not spend too much time decoding the operation of the Mask class in **mask.h**. Your time will likely be better spent in understanding how to use it effectively, as described in the section "More About Using Masks" below.
3. If you pass a mask to a function and expect the function to modify it, remember to pass the mask by reference. This particularly applies to **Question 3**.
4. Feel free to define any of your own helper functions which would help to make your code more elegant. This applies to **all questions**.
5. You should feel free to exploit the answer to any earlier questions when answering a question. This particularly applies to **Questions 2, 3 and 4**.
6. The standard header `<cctype>` contains some library functions that you may find useful when answering **Questions 2 and 3**. For example, `isalpha(char ch)` returns a non-zero value if ch is an alphabetic character.
7. For **Question 4**, a brute force recursive solution approach that avoids using masks will almost surely fail to terminate in reasonable time owing to the sheer number of possible combinations.
8. Your solution to **Question 4** is required to be **recursive**, i.e. the function (or a helper function if you chose to use one) should call itself. You are permitted to add default arguments if you need to (but this is not strictly necessary).
9. Try to attempt all questions. If you cannot get one of the questions to work, try the next one.

More About Using Masks

Creating a mask

You are supplied with a Mask class in **mask.h**. To create a Mask, simply declare:

```
Mask m;
```

This will create a mask with all entries set to true:

```
11111
11111
11111
11111
11111
```

If you wish to create a mask with all entries set to false, use:

```
Mask m(false);
```

Modifying the elements in a mask

One way to set all the entries in a mask is to use the `set_all(...)` method, e.g.:

```
Mask m;
m.set_all(false);
```

Individual elements of masks may be modified or retrieved using the `set_element(...)` and `get_element(...)` methods respectively, e.g.

```
Mask m(false);
m.set_element(2,2);
```

results in `m` having value:

```
00000
00000
00100
00000
00000
```

You can also use square brackets to access the elements in the mask, as if Mask were simply a two dimensional array, e.g.

```
Mask m(false);
m[2][2] = true;
if (m[0][0]) {
   ...
}
```

Printing masks

You can use the `print()` method to print a mask using 0 for false and 1 for true, e.g.

```
Mask m(false);
m[2][2] = true;
m.print();
```

will output:

```
00000
00000
00100
00000
00000
```

Finding the d-neighbourhood of a mask

The `neighbourhood(int d)` method returns the d-neighbourhood of a mask. By default $d = 1$. Thus the code:

```
Mask m(false);
m[2][2] = true;
Mask nbr = m.neighbourhood();
nbr.print();
```

will output:

```
00000
01110
01110
01110
00000
```

Counting the number of bits set to true in a mask

The `count()` method returns the number of bits set to true in a given mask. Thus:

```
Mask m(false);
m[2][2] = true;
Mask nbr = m.neighbourhood();
cout << nbr.count() << endl;
```

will output:

```
9
```

Finding the first bit with a particular value in a mask

The `get_position(bool value, int &row, int &col)` method scans a mask in a row-wise fashion looking for the first mask element having a given `value` (true or false). If such an element is found, `row` and `col` are set to the row and column of the found element respectively, and the method returns true. Otherwise the method returns false and `row` and `col` are both set to -1.

Thus the code:

```
Mask m(false);
m[3][4] = true;
int r,c;
m.get_position(true, r, c)
```

results in `r` having the value 3 and `c` the value 4.

Intersection and union operations on masks

Masks may be intersected in an elementwise fashion in number of different ways. One way is to use the `intersect_with(...)` method:

```
Mask one, two; // all values true by default
one[2][2] = false;
two[3][4] = false;
two[3][0] = false;
one.intersect_with(two);
```

results in `one` having the value:

```
11111
11111
11011
01110
11111
```

The * and *= operators have also been overridden to denote intersection operations, so:

```
one *= two;
```

or

```
one = one * two;
```

will have the same effect.

Likewise `union_with(...)` will perform an elementwise union, for which the + and += operators have also been overridden.

Flipping all bits in a mask

The not (!) operator has been overridden to return a mask with all bits flipped. Thus:

```
Mask m(false);
m[2][2] = true;
(!m).print();
```

outputs:

```
11111
11111
11011
11111
11111
```