TIC1001—Introduction to Computing and Programming I
National University of Singapore
School of Continuing and Lifelong Education

Practical Examination 2

12 November 2020

Time allowed: 1 hour 30 mins

Instructions (please read carefully):

1. This is an **OPEN-BOOK** exam. You are allowed to reference materials in printed form, or softcopy materials on your current PC. No other devices are allowed.
2. This practical exam consists of **ONE** question with **THREE** parts.
3. The maximum score of this test is **10 marks**. Note that the number of marks awarded for each question **IS NOT** correlated with the difficulty of the question.
4. You are advised to attempt all questions. Even if you cannot solve a question correctly, you are likely to get some partial credit for a credible attempt.
5. You are provided with a template `pe2-template.cpp` and some supplementary files to work with and test your own code for correctness on VS Code.
6. Please note that while sample executions are given, it is **not sufficient to write programs that simply satisfy the given examples**. Your programs will be tested on other inputs and they should exhibit the required behaviours as specified by the problems to get full credit. There is no need for you to submit test cases.
7. Please note that it shall be your responsibility to ensure that your solution is **submitted correctly on Examplify by the end of the examination**. Examplify will stop accepting answers the moment the timer runs out, so ensure you have copied your answers in before that.
8. Marks will also be awarded for **good coding style** like suitable variable names, proper indentations, etc., in addition to the correctness of the program.

ALL THE BEST!

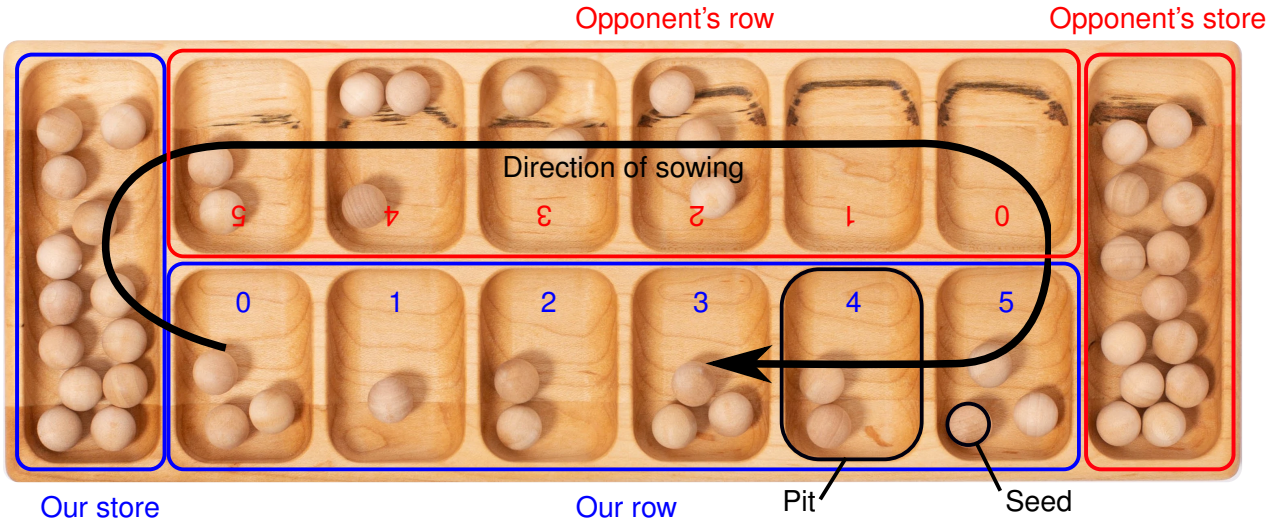
Question 1: Mancala [10 marks]

Background: Mancala is a generic name for a family of two-player turn-based strategy board games played with small stones, beans, or seeds and rows of holes or pits in the earth, a board or other playing surface. The objective is usually to capture all or some set of the opponent's pieces.

Source: Wikipedia

For this PE, we will be modelling and implementing a simple two-row Mancala game. This will be done in several parts, cumulating in the final implementation.

Note that the difficulty of each part is not dependent on the order nor marks. You are advised to read through all parts before deciding the order to attempt.



Our game of Mancala is played on a board with two rows: one for each player. On the left of each row, is a player's store. Each row consists of a number of pits, each containing zero or more seeds. To make things easy, the pit closest to the store is numbered pit 0. Note the opponent row is numbered from his perspective.

Players take turns sowing the seeds. On our turn, a pit is chosen from our row and all the seeds are scooped up. The seeds are then dropped, one-by-one, into every pit on the left. If there are still seeds remaining, one is dropped into our store and the dropping continues on the opponent's row. We continue dropping seeds in a clock-wise direction until all the seeds are dropped. Note that the opponent's store is skipped.

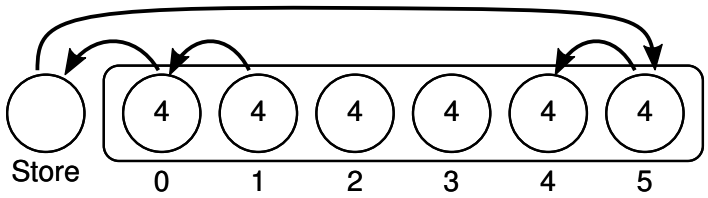
Each row of the Mancala board is modelled using a `vector` of `int`, with each index representing the pits and the value indicating the number of seeds in the pit. The size of the vector is not fixed, and depends on the size of the board being played. The store is not included in the vector.

A. We will being by modelling the sowing process with only one row. The function

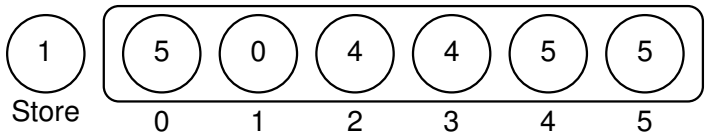
```
int sow(vector<int> & row, int pit)
```

that takes as inputs a representation of our row, and a pit. It updates the row based on sowing from the given pit. Since there is only one row, the sowing only loops around our row. The function returns the number of seeds dropped in our store during the sowing process.

For example, suppose our row has 6 pits with 4 seeds each, and we sow from pit 1.



At the end of the sowing, we will end up with four seeds in pits 2 & 3, five seeds in pits 0, 4 & 5, zero seeds in pit 1, and one seed in our store.



Provide an implementation for the function `sow`. [4 marks]

Sample tests:

Contents of `row` before	Function call	Return	Contents of `row` after
`{4, 4, 4, 4, 4, 4}`	`sow(row, 1)`	`1`	`{5, 0, 4, 4, 5, 5}`
`{5, 0, 4, 4, 5, 5}`	`sow(row, 5)`	`0`	`{6, 1, 5, 5, 6, 0}`
`{8, 8, 8, 8}`	`sow(row, 0)`	`2`	`{1, 9, 10, 10}`

B. We will implement a function to help us with the final task. The function

```
vector<int> slice(const vector<int> v, int start, int end)
```

takes as inputs a vector, and two integers. It returns a "slice" of the vector, which is a vector that contains elements from the index `start` up to but **excluding** index `end`.

Provide an implementation for the function `slice`. You may assume that $0 \leq start < end \leq n$ where n is the number of elements of the vector. [3 marks]

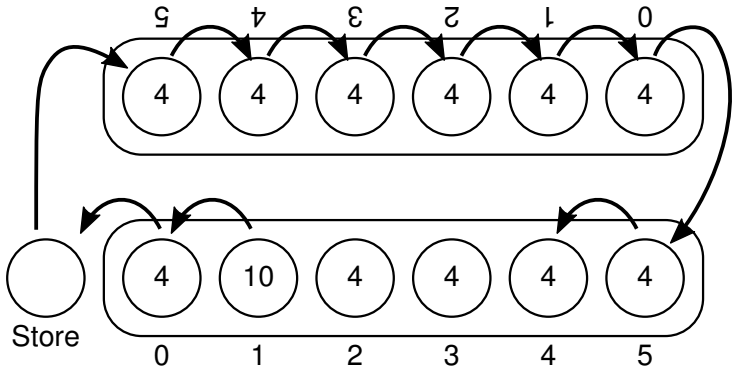
Sample tests:

Function call	Return
`slice({1, 2, 3, 4, 5, 6}, 2, 5)`	`{3, 4, 5}`
`slice({1, 2, 3, 4, 5, 6}, 0, 6)`	`{1, 2, 3, 4, 5, 6}`
`slice({1, 2, 3, 4, 5, 6}, 3, 4)`	`{4}`

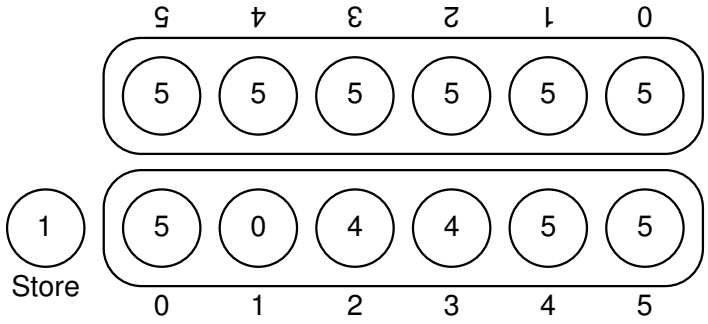
C. Now we are ready to work with two rows. The function

```
int sow2(vector<int> & our_row, vector<int> & their_row, int pit)
```

now takes in two rows, ours and the opponents, and performs the sowing action from the given pit. This time, the sowing continues across the opponent's row and back to our row, skipping the opponent's store.



Note that the opponent's row is index from his perspective. Sowing from pit 1 results in the following state:



To help with our implementation, a function `concat` has been defined. `concat` takes in two vectors, concatenates them into and returns a new vector. For example, `concat({1, 2, 3}, {4, 5, 6})` will return a vector `{1, 2, 3, 4, 5, 6}`.

Using the functions `concat`, `sow` and `slice` which were defined in the earlier parts A and B, provide an implementation for the function `sow2`.

[3 marks]

Sample tests:

Contents before	Test function	Output	Contents after
`opp = {4, 4, 4, 4, 4, 4}` `our = {4, 10, 4, 4, 4, 4}`	`sow2(our, opp, 1)`	`1`	`opp = {5, 5, 5, 5, 5, 5}` `our = {5, 0, 4, 4, 5, 5}`
`opp = {3, 4, 5}` `our = {9, 8, 7}`	`sow2(our, opp, 0)`	`2`	`opp = {4, 5, 7}` `our = {1, 9, 8}`

— E N D O F P A P E R —