Assignment Brief

Part B & C

Part B: Dominoes

Dominoes is a game played with rectangular playing pieces, each of which is engraved with two symbols. The pieces are themselves called dominoes. The game proceeds by forming a line of dominoes, such that each domino is adjacent to another domino with a matching symbol.

This part of the assignment involves implementing a C++ class providing some functionality related to forming lines of dominoes, though not to the extent of creating a complete dominoes game. You will then generate performance data from your implementation, and then present and evaluate that data.

**Task B1 — Efficient Implementation**

There are three options of functionality to implement, at different difficulty levels Choose one of the options, and create a C++ class providing the required functionality. You should implement the class in the way that will have the most efficient time complexity at runtime (in the average case), relative to the total number of dominoes.

You are advised to make use of one or more of the standard containers provided by the C++ standard library, such as:

* std::pair
* std::vector
* std::list
* std::map
* std::unordered\_map

Clarifications:

* You are not expected to implement your own pointer-based data structures for this task, though may create your own structs/classes.
* Your solution should be single-threaded. Use of multiple threads is out-of-scope for this assignment.

**Task B2— Worst-case Guarantees**

For a higher grade, also implement the required functionality a second time, but this time in the way that you think will provide the most efficient ***worst-case*** time-complexity guarantees.

**Task B3 — Justifying Implementation Choices**

In your report, explain and justify the design of your implementation(s) from Tasks B1 and B2.  This should include stating the time-complexity guarantees of each of the container operations or significant library functions that are used in your code (and noting what those guarantees are relative to).

Your explanation should also include:

* Giving a time-complexity analysis of the constructor and each of the functions of your C++ class.
* Giving an overall time-complexity analysis of the entire process of constructing the line of dominoes.
* Distinguishing between ***average-case*** and ***worst-case*** complexity guarantees, when they differ (both for the individual operations, and for the overall analysis).

**Task B4 — Performance Measurement**

Using C++ standard-library timing facilities, write a C++ program that measures the time usage of your implementations from Tasks B1 and B2, relative to the total number of dominoes. Use your implementation(s) to generate performance data. Present the data in your report using table(s) **AND** graph(s).

Clarifications:

* Input data files containing data sets of varying sizes are provided to facilitate generating performance data (see Section 5.2). You should select data sets that are large enough to allow you to draw meaningful conclusions, but that can be processed in a viable amount of time.
* Ensure that ***all*** of your raw results data is included in tables.
* Present your graph(s) in a way that facilitates visually inferring the time complexity characteristics of the measured performance.
* If you have results from both Tasks B1 and B2, then you should (at least) have one graph on which both sets of results are plotted, to help a reader to visually compare the results.

**Task B5 — Evaluating Performance Results**

Evaluate the results from Task B4 in your report.

Clarifications:

* Ensure that you discuss what the results indicate regarding the time complexity of the implementation(s).
* If you gave a complexity analysis of the implementation(s) as a whole in Task B3, then you should argue whether the results support or contradict that analysis.

Part C: Dominoes Variation

This describes a variation of the dominoes problem, as well as a (rather convoluted) algorithm that can be used to provide a solution.

**Task C1 — Analysing the Convoluted Algorithm**

Analyse the time complexity of the convoluted algorithm, relative to the total number of dominoes.

**Task C2 — Implementing the Variant Solution**

Implement the required functionality specified in Section 5.4.1, using the convoluted algorithm from Section 5.4.2.

**Task C3 — Measuring the Convoluted Solution**

In a similar manner to Task B4, generate performance data from your implementation from Task C2, and present the data in your report using table(s) **AND** graph(s).

Clarification:

* You should (at least) have one graph on which the results from both Parts B and C are plotted, to visually compare the results.

**Task C4 [extension task] — Evaluating Performance Results**

Evaluate the results in your report. If you gave a complexity analysis of the algorithm for Task C1, then you should argue whether the results support or contradict that analysis.  You should also compare your results with those from Part B.

Dominoes Requirements

Assume we have a collection of dominoes where one of the symbols on each domino is coloured **blue**, and the other is coloured **red**. Each symbol appears on exactly two dominoes, once **blue** and once **red**. In this particular collection, it is possible to place all of the dominoes in a line such that the symbols on each domino match their neighbour. The dominoes start disordered, and the objective is to incrementally construct a complete line of dominoes with symbols that match their neighbours.

Here follows an example of a line of nine dominoes. Note however that your implemented solution should be able to process domino collections of arbitrary sizes (limited only by available RAM).



Note that to avoid the complications of graphics processing, throughout this assignment we will use unique text strings in the program code, instead of actual graphical symbols.

5.1 Functional Requirements for Part B

There are three variations of functional requirements for Part B, of increasing levels of difficulty. You only need to choose one of these options to implement.

**Basic Option: Coloured dominoes extending in one direction**

You are required to create a C++ class that can store the collection of disordered dominoes, and also incrementally compute and store the line of matching dominoes.

The class should provide a public constructor with two parameters.  The first parameter is for the starting domino, the second parameter is for the rest of the domino collection (unsorted). You may choose the types of the parameters and exactly how the dominoes are represented at this point. (E.g. The parameters could be file paths, input streams, strings or a data structure.)

The class should provide a public member function that determines the next domino to be placed in the line ***to the right***. This domino should be added to the line of dominoes stored in the class object, and the function should also return this domino.

The class should provide a public member function that determines if the line is completed (i.e. that all dominoes have been added).

The class should provide a public member function that displays the current line of dominoes (i.e. the starting domino and all dominoes added to its right thus far). You may choose the exact nature of 'displaying' here. (E.g. this could be directly to the standard output, or to a stream parameter, or returned as a string.)

**Intermediate Option: Coloured dominoes extending in both directions**

This variation is similar to the **'Basic'** option, except that the class should additionally provide a public member function for determining the next domino to be placed in the line ***to the left***. The class must allow for the line to be built by any sequence of intermixed left/right steps.

**Advanced Option: Uncoloured dominoes extending in both directions**

This variation is similar to the **'Intermediate'** option, except that the domino symbols are all the same colour (instead of being coloured **blue** and **red**).

Variant for Part C

Part C involves another variation of the functional requirements from Section 5.1. Unlike Part B, for Part C you are provided with an algorithm that can be used to implement a solution. Note that this algorithm is more complex than you are expected to have produced yourself for Part B.

**5.4.1 Functional Requirements Variation**

The requirements are similar to the **'Basic'** option, except that the requirement that the domino line be constructed ***incrementally*** is dropped. Therefore the function to determine the next domino to the right, and the function to check if the line is completed, are not required. The line is constructed by the class constructor, and the 'display' function only displays the completed line.

**5.4.2 The Convoluted Algorithm**

One (rather convoluted) algorithm satisfying these looser requirements can be implemented using the 'std::list' and 'std::pair' containers, as follows:

(1) Store the domino data (excluding the starting domino) as a list of pairs of symbols (call it List A).

(2) Start a new list to record the position of each **red** symbol (call it List P). This list will contain (Symbol,PositionNumber) pairs. Initially, just add a pair containing the **red** symbol from the starting domino, with position number 0.

(3) We now gradually compute the position numbers of every **red** symbol, and add these to List P. This can be achieved using a loop with a counter variable (call it 'distance') that starts at 1 and ***doubles*** at each iteration. The loop condition is that 'distance' is less than the total number of dominoes. The loop body behaves as follows:

1. Make a copy of List A (call it List B).
2. Create an empty list of symbol pairs (call it List C).
3. Sort List A based on the second symbol, and sort List B based on the first symbol.
4. Sort List P based on the symbol.
5. Use iterators to advance through Lists A, B and P, as follows. Let us refer to the current pairs on Lists A, B and P, as (a1, a2), (b1, b2) and (p1, p2), respectively. Then, until we reach the end of List B:
   1. If b1 = a2, append (a1, b2) to List C, and advance Lists A and B once.
   2. Else if b1 = p1, append (b2, p2+distance) to the ***front*** of List P, and advance Lists A and P once.
   3. Else if b1 > p1, advance List P once.
   4. Else if b1 > a2, advance List A once.

Note: Skip the if-conditions involving Lists A or P if the end of that list has already been reached.

1. Overwrite List A with List C.

(4) Sort List P based on the the position numbers.

(5) Now build the line of dominoes by iterating through List P, constructing the dominoes one by one.  For each domino, the **red** symbol can be copied from List P, and the **blue** symbol can be copied from the **red** symbol of the previous domino in the line (with the exception of the first domino, for which the **blue** symbol can be copied from the known starting domino).