SUBMISSION GUIDELINES

Video guidance on submitting homework on Gradescope:

Submit through Gradescope website

Submit through Gradescope mobile app Links to an external site

On Gradescope, go to the class homepage → Assignments to submit your pdf and code files.

In Homework 2, we have 4 questions consisting of two types:

1. **Short answers**. For Q1, please write your answers in a single pdf file, then upload it to the assignment *Homework 2 - Q1 (Short Answers Only)*.
    - **Submission**: while submitting your pdf file, please **specify on which page your answer is for each question** (see the video guidance above). It helps us grade quicker and more consistently.
2. **Programming**. For Q2, Q3, and Q4, you need to submit a .cpp file for each question.
    - **Submission**: please save your function as a separate .cpp file using **the exact function name** as its filename (be careful about upper/lowercase), then upload it to the assignment for the corresponding question on Gradescope.
    - **Function part only**: you only need to keep the required function without any main function, input, or output. You can create more functions as needed, as long as the provided function signature can work as expected.
    - Make sure your code satisfies all the requirements and is robust to corner cases.
    - Do your best to optimize your algorithms (if no specific algorithm is required) rather than use brute force.
    - **Code style**: your code will be graded by both autograder and TA. Therefore, please make sure that:
        - Your code is **clean enough**. Use blank lines, indent, and whitespace as needed, and delete all the sentences that are no longer needed.

    - Use **understandable variable names**, such as *old_array* and *new_array* rather than *a*, *b*, and *c*.
    - Write necessary **comments** for your key steps. Your code should let others easily understand what it is doing.
- Just in case the auto-grading script does not include the STL templates you've used (even though it is almost impossible), please comment in the first line of your code to illustrate what STLs you've used. No need for your submission to "#include" any STL libraries or "use" any namespace, but feel free to do it.

Note:

1. Be honest with yourself. The cost of plagiarism is always much more than expected as it consists of not only the external outcomes (e.g., failed assignments and courses) but also the moral burden on your conscience, which you will have to bear. Grading will be generous as long as you have tried your best.

2. If you have any questions about homework or in-class content, feel free to put your questions in the Discussions module on Canvas or show up during the instructor's or TA's office hours. Try everything as best you can instead of trying to find a shortcut.

Good luck! :D

1. (10 pts) The C++ statement below declares an array and a pointer:

```
int arr_1[10], *arr_2;
```

Determine if each expression below is **valid or not**, and **briefly explain why**.

a. `arr_1 = arr_1 + 5;`
b. `arr_2 = arr_2 + 10;`
c. `arr_2 = &arr_1;`
d. `arr_2 = arr_2 - arr_1;`
e. `arr_1[2] = arr_1 - arr_2;`

Hint: "validity" means that the expression can be interpreted and executed by the C++ compiler – not necessarily that it makes sense to the programmer.

2. (10 pts) A group of kids are lining up to learn ballroom dancing. From the first person to the last, each kid was given a number to uniquely identify them. The first half of the kids are all wearing blue shirts, and the second half are all in orange. The line can be represented as:

   $L_0, L_1, L_2, \ldots, L_{n-1}, L_n$

   Now, they need to be reordered so that each kid in blue will be somehow paired with another one in orange. One way to do that is to pair the first person in blue with the last one in orange, pair the second person in blue with the second last one in orange, and so on and so forth. The reordered line can be represented as:

   $L_0, L_n, L_1, L_{n-1}, L_2, L_{n-2}, \ldots$

   Imagine there are thousands of children (how spectacular!) – that would be hard to do manually. Can you write a function named **`shuffle`** that rearranges a singly-linked list as shown above?

   - Function signature:

     ```
     void shuffle(Node* head)
     ```

   - Example 1:
     - Input list: *1, 2, 3, 4, 5, 6, 7, 8*
     - Expected list after shuffle: *1, 8, 2, 7, 3, 6, 4, 5*
   - Example 2:
     - Input list: *1, 2, 3, 4, 5, 6, 7, 8, 9*
     - Expected list after shuffle: *1, 9, 2, 8, 3, 7, 4, 6, 5*
   - Example 3:
     - Input list: *1, 4, 3, 2*
     - Expected list after shuffle: *1, 2, 4, 3*

   Note:

   a. You **must** use the following class to define a node. Insert this class into your code:

      ```
      class Node {
      public:
            int val;
            Node *next;
            Node() : val(0), next(NULL) {}
      ```

```
        Node(int x) : val(x), next(NULL) {}
        Node(int x, Node *next) : val(x), next(next) {}
};
```

b. The count of the blue group may be equal to that of the orange group or be 1 greater than that. The extra person is to be left at the end of the list (i.e., paired with the teacher).
c. You may not expect the numbers to be sorted or have any particular meaning.
d. You may only change the pointers, but not the values of nodes.
e. The head of the list should remain the same. Our grading code will use the same **head** pointer as the input, so you do not need to return anything from your function.

3. (15 pts) Program a class `CircularQueue` that implements the queue data structure with a circular C-style array. It should store floating-point numbers. Use the lecture material as a guide.

   Methods you must implement:

   - `CircularQueue(int n)` // initialize a queue that **can keep n elements**
   - `CircularQueue(const CircularQueue& c)` // copy constructor
   - `~CircularQueue()` // destructor
   - `bool enQueue(float value)` // insert a value into the queue back
   - `float deQueue()` // remove value from the queue front
   - `bool isFull()` // evaluate if the queue is full
   - `bool isEmpty()` // evaluate if the queue is empty
   - `float getLast()` // get the current rear element of the queue
   - `float getFront()` // get the current front element of the queue

   The copy constructor must implement the **deep copy**, so that each queue has its own independent underlying array. The destructor must correctly **free up** the allocated memory.

   - Example program:
     - CircularQueue q(3.0);
     - q.enQueue(1.0); q.enQueue(2.0);
     - CircularQueue p(q);
     - q.enQueue(3.0); p.deQueue(); p.deQueue();
   - Expected commands and effects after this program is run:
     - q.isFull(); // true
     - p.isEmpty(); // true
     - q.getLast(); // 3.0
     - q.getFirst(); // 1.0

   Note:

   - You **cannot** use the STL to implement your queue: it needs to be built on a dynamically-allocated C-style array.
   - If an operation cannot be carried out, throw the std::logic_error exception.

4. (15 pts) Suppose a matrix is represented with a nested STL vector:

```
vector< vector<double> >
```

Given an input matrix, write a function `isInvertible` to determine if it is **invertible**.

- Function signature:

```
bool isInvertible(vector< vector<double> >& mat)
```

- Example 1:

  Input: *[[1, 2, 3], [4, 5, 6], [5, 7, 8]]*

  Output: *True*

- Example 2:

  Input: *[[1, 0, 0], [0, 1, 0], [0, 0, 0]]*

  Output: *False*

Note:

- There are many ways to check if a matrix is invertible. For example, using Gaussian elimination (you could reuse your code for the excursion!) or checking if its determinant is 0.
- Non-square matrices are not invertible. Neither are empty ones.
- Remember to guard against corner cases!