CS100 Homework 4 (Fall Semester 2018)

Due time: 11:59 pm, December 6, 2018
Homework 4 comprises 3 problems. Important notes:

- The templates for all problems have been submitted to your gitlab project repositories. It is crucial that you take them as a starting point, they already implement the entire I/O functionality plus a substantial part of the interfaces. The sections that you should modify are clearly marked.
- As in the previous homework, we will use OJ for grading for unified compilation and testing. However, please beware the rules of using OJ! **Do not abuse OJ**, you should submit your solution **no more than 30 times**, and the **shortest interval between two submissions is 2** minutes. As in the previous homework, any violations of these rules will lead to a **(cumulative!) reduction of 10% of the total score**. It is implicit that you should compile and test your code locally on your machine before you start submitting your solution to OJ.
- Please also beware the **naming rule**. You should change your name in OJ under settings to your real Chinese name. No exceptions will be made this time! Please review the related posts on Piazza!
- The test examples are also uploaded to your gitlab accounts. For convenient testing, you may forward the test examples to your binary such that **`std::cin`** will directly read from the file. For example, suppose you call your binary main, and you want to test the output of 2.in. In your bash, simply type
  `./main < 2.in`
- Problem 3 may reuse the code from problem 2, so if you correctly solve problem 2, it will be of great help for problem 3 as you may simply copy-paste the code for the template matrix.
- All problems are to be implemented in a single file that can be copy-pasted to OJ.
- This homework depends at least on the C++-standard C++11, so please use a recent compiler. The standard may have to be provided to the compiler, as in
  `g++ -std=c++11 -stdlib=libc++ solution3.cpp -o main`

Homework 4 / Problem 1

Problem 1 introduces a datastructure which we denote a HyperVector. The idea of the HyperVector is to provide the benefits of a vector (which is efficient random-access to the data, i.e. efficient look-up of any element of the vector), while at the same time providing similar benefits than a list, which is a computational complexity for erase/insert/push-back operations that remains largely independent of the overall number of elements stored in the container. For the purpose of problem 1, the design is kept very simple, and only the push-back functionality is implemented. The HyperVector may hence be explained as follows:

- The data is internally stored using a **`std::vector< std::vector<T>* >`**, where T is a template parameter. In other words, it is a vector of vector-pointers.
- The internal vectors represent buckets of data and are allocated on demand. These buckets may at most hold **`VALUES_PER_BUCKET`** elements. This space is reserved as a new bucket is installed
- When pushing back a new element into the hyper-vector, the element will be added to the current bucket if it is not yet full, respectively added to a new bucket if there is no data in the container yet, or the previous bucket is full.
- This functionality prevents the actual data of the container from ever being copied around due to insufficient contiguous memory space, data does in fact not have to be contiguous in memory. On the other hand, efficient random access is still given, as an index into the hyper-vector can easily be transformed into an external and internal bucket index (notably by dividing by **`VALUES_PER_BUCKET`**, and considering the quotient and remainder or this operation.

As seen in class, having a container that permits random access allows for the application of the efficient **`std::sort`** algorithm. The latter however depends on the availability of STL random-access iterators into the elements of the container. An implementation of HyperVector is already provided, your task is to implement the random-access iterator for the hyper-vector, called **`HyperVectorIterator`**.

Hint: Your iterator needs to inherit from the STL iterator

```
std::iterator< std::random_access_iterator_tag, T >
```

and it needs to implement the following interface:

```
T & operator*();
T & operator->();
T & operator[](int n);

HyperVectorIterator<T> & operator++();
HyperVectorIterator<T> operator++(int);
HyperVectorIterator<T> & operator--();
HyperVectorIterator<T> operator--(int);
```

```
HyperVectorIterator<T> & operator+=( int n );
HyperVectorIterator<T> & operator-=( int n );
HyperVectorIterator<T> & operator+=( const
HyperVectorIterator<T> & that );
HyperVectorIterator<T> & operator-=( const
HyperVectorIterator<T> & that );

HyperVectorIterator<T> operator+( int n ) const;
HyperVectorIterator<T> operator-( int n ) const;
int operator+( const HyperVectorIterator<T> & that ) const;
int operator-( const HyperVectorIterator<T> & that ) const;

bool operator== ( const HyperVectorIterator<T> & that ) const;
bool operator!= ( const HyperVectorIterator<T> & that ) const;
bool operator< ( const HyperVectorIterator<T> & that ) const;
bool operator> ( const HyperVectorIterator<T> & that ) const;
bool operator<= ( const HyperVectorIterator<T> & that ) const;
bool operator>= ( const HyperVectorIterator<T> & that ) const;
```

Part of the homework task is to have a good intuition about what these functions are supposed to do, and research online if in doubt. The implementation is tested through a main function that is already provided. It simply reads an arbitrary number of random real numbers from the console and pushes them back into a hyper-vector. It then calls the sorting algorithm, and finally prints the sorted elements back to the console.

Input description: Files containing random numbers similar to the ones employed by OJ are already provided. For local testing you may forward them to **`std::cin`** by simply running

```
./main < 1.in
```

Ouput description: The program simply outputs the sorted numbers to the console. Each number is followed by a white-space, and the last whitespace is followed by and end-of-line character. The I/O functions are pre-implemented, so you should not encounter any problems here.

Homework 4/ Problem 2

Problem 2 is about the implementation of a template matrix class. The class should heavily rely on operators to enable convenient encoding of arithmetic operations (see main function in the template for examples). The exact interface is as follows:

```
Matrix( int rs, int cs, T val = 0 );
virtual ~Matrix();

Matrix<T> & resize( int rs, int cs, T val = 0 );

T         & operator()( int r, int c );
const T   & operator()( int r, int c ) const;
size_t      rows() const;
size_t      cols() const;

Matrix<T>   block( int r, int c, int h, int w ) const;
Matrix<T> & setBlock( int r, int c, int h, int w, T & val );
Matrix<T> & setBlock( int r, int c, const Matrix<T> & val );
Matrix<T> & setIdentity();
Matrix<T> & setConstant( const T & val );
Matrix<T> & setZero();

Matrix<T>   transpose() const;
Matrix<T> & transposeInPlace();

Matrix<T>   operator+ ( const Matrix<T> & op ) const;
Matrix<T> & operator+=( const Matrix<T> & op );
Matrix<T>   operator- ( const Matrix<T> & op ) const;
Matrix<T> & operator-=( const Matrix<T> & op );

Matrix<T>   operator* ( const T & op ) const;
Matrix<T>   operator* ( const Matrix<T> & op ) const;
Matrix<T> & operator*=( const T & op );

Matrix<T>   hadamard  ( const Matrix<T> & op ) const;
T           sum() const;
```

Part of the homework is to prove an intuitive understanding of what the above function are supposed to do. The matrix implementation will be tested in a pre-implemented main function that implements a fixed sequence of arithmetic operations testing a large part of the interface for correct functionality. It reads in a random matrix from the console starting with a first line in which the number of rows and columns are to be provided, and finishing with further lines each one containing the real numbers of that row in the matrix. The result of the arithmetic operations is again output on the console.

Input description: Files containing random numbers similar to the ones employed by OJ are already provided. The first line of the file contains the number of rows and columns in the matrix. For local testing you may forward them to **`std::cin`** by simply running

```
./main < 1.in
```

Ouput description: The program simply outputs the matrix results of the operations onto the console. The results are printed row-by-row for each matrix. Each number is followed by a white-space, and the last whitespace behind that last number in the row is followed by an end-of-line character. The I/O functions are pre-implemented, so you should not encounter any problems here. No empty lines are inserted between the matrices.

Homework 4/ Problem 3

Problem 3 consists of implementing a hyper-threaded kernel-convolution. Kernel convolutions are important in several domains of computer science, such as machine learning (deep learning, CNNs, https://en.wikipedia.org/wiki/Convolutional_neural_network) or image processing (https://en.wikipedia.org/wiki/Kernel_(image_processing)). In general terms, the operation consists of generating an output matrix from a given input matrix of similar size. Let o_{rc} be the value of the element in row r and column c in the output matrix. It is given as a result of convolving the values around the same element in the input matrix $\mathbf{A}$ with a small kernel matrix $\mathbf{K}$. Let $(2h+1)$ and $(2w+1)$ be the height and width of the kernel. The value of an element in the output matrix is then given as follows:

$$o_{rc} = \sum_{i=0:2h} \sum_{j=0:2w} \mathbf{K}(i,j) \cdot a_{r-h+i,c-w+j}$$

where a_{rc} denotes the element of the input $\mathbf{A}$ in row r and column c. The operation may also be rewritten in terms of an element-wise matrix multiplication (also called the Hadamard product):

$$o_{rc} = sum(\mathbf{K} * \mathbf{A}(r-h, c-h, 2h+1, 2c+1))$$

where *sum()* is a function that returns the sum of all the elements of the matrix parameter, * denotes the Hadamard product, and $\mathbf{A}(x,y,h,w)$ denotes the sub-block of $\mathbf{A}$ with top-left corner located at (x,y) and size hxw.

Your tasks: Add a generic template matrix class to hold the input and output matrices, as well as the kernel itself. The main function is already implemented, and it reads the input matrix from the console, along with an integer denoting a kernel-type. Depending on this integer, a certain kernel is then initialized (pre-coded). The main function also defines an output matrix of the same size than the input matrix. It finally starts four threads which compute the elements of the output matrix according to the above equation. Each thread notably takes care of 25 % of the output matrix. Each thread is provided an individual instance of

```
struct KernelApplicationJob {
  Matrix<float> * src;
  Matrix<float> * K;
  Matrix<float> * dst;
  size_t threadIndex;
  size_t numberThreads;
};
```

which passes the required information to the thread (the input/src matrix, the kernel matrix, and the output matrix/dst matrix). It furthermore communicates the number of threads, and a thread-index. The part of the image that is being dealt with in each particular

thread should notably be chosen as a function of the thread-index. The thread function to be implemented has the signature

```
void kernelApplicationThread( KernelApplicationJob & my_job );
```

Note: The output matrix contains a margin of width h or w for which values can not be computed since the corresponding field from the input matrix would exceed the image region. These values are simply set to 0! (Beware however that h and w can be different for each kernel).

Input description: Files containing random matrices similar to the ones employed by OJ are already provided. The first line contains the number of rows, columns, and the type of the kernel to be applied (a number between 0 and 5). For local testing you may forward the pre-arranged files to `std::cin` by simply running

```
./main < 1.in
```

Ouput description: The program simply outputs the matrix results after the convolution onto the console. The results are printed row-by-row for each matrix. Each number is followed by a white-space, and the last whitespace behind that last number in the row is followed by and end-of-line character. The I/O functions are pre-implemented, so you should not encounter any problems here.