ECE 2400 Computer Systems Programming, Fall 2021
PA4: Handwriting Recognition Systems – Linear vs. Binary Search

School of Electrical and Computer Engineering
Cornell University

revision: 2021-11-16-10-08

1. Introduction

The fourth programming assignment will enable you to apply all of the concepts you have learned throughout the semester in the context of a simple machine learning system. You will leverage what you have learned on algorithms (e.g., iteration vs. recursion), data structures (e.g., vectors), complexity analysis, C++ basics (e.g., namespaces, references, exceptions, and dynamic allocation), and object-oriented programming (e.g., classes, member functions, constructors, operator overloading, the rule of three, data encapsulation, interface vs. implementation). You will apply this understanding to implement, test, and evaluate a handwriting recognition system that can classify handwritten numbers into ten classes, the digits from zero through nine, with high accuracy.

Research in the field of machine learning today has demonstrated remarkable success in a wide range of applications, including object detection and decision making for autonomous vehicles, natural language processing, and even world-class board game playing. Machine learning encompasses the design of software programs that detect and learn features from inputs before generalizing what they learn and applying it in new contexts. There are broadly three classic learning paradigms: (1) *supervised learning* where the learner is provided with a set of inputs together with the desired outputs; (2) *unsupervised learning* where the learner is only given training examples as input patterns with no associated output; and (3) *reinforcement learning* where the learner is only given evaluative output (e.g., reward for a certain action) to learn a mapping from states to actions to maximize the long-term reward.

In this assignment, you will implement a supervised learning model that classifies handwritten digits. This model has two main phases: *training* and *classification*. In the *training* phase, the model is provided with a large set of images, each with a label (e.g., '1', '7', '9') that indicates the corresponding digit. In the *classification* phase, the model is provided with new inputs that it has never seen before and predicts their label based on what it has learned in the training phase. You will use the classic MNIST database of handwritten digits. The database is composed of 70,000 examples of images each with 28×28 grayscale pixels. Each pixel has a value between 0 and 255 where 0 represents white, 255 represents black, and intermediate values represent intermediate levels of gray. The dataset is divided into a training dataset of 60,000 images and a classification dataset of 10,000 images for evaluation. Figure 1 shows a few images from the MNIST dataset. The digits were handwritten by several hundred different writers ranging from average high school students to Census Bureau employees. This means that the legibility of the handwritten digits varies significantly. The goal of the assignment is to design a system that can classify images of these handwritten digits with high accuracy into one of ten classes: the numbers zero through nine. You will start off by first implementing three classes: `VectorInt`, `Image`, and `VectorImage`. Then you will leverage these classes to construct two simple classification algorithms; one is based on linear search and the other is based on binary search. As in the previous assignments, we will leverage the CMake framework for building programs, the CTest framework for unit testing, GitHub Actions for continuous integration testing, and lcov for code coverage analysis.

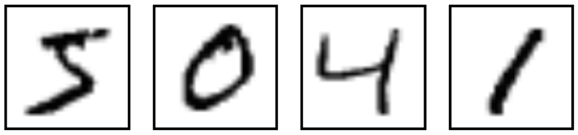

Figure 1: Four Example MNIST Images – Images include 28 × 28 grayscale pixels and a label. Each pixel has a value between 0 and 255 where 0 represents white, 255 represents black, and intermediate values represent intermediate levels of gray.

After your handwriting recognition systems are functional and tested, you will evaluate the accuracy and performance trade-offs between implementations. We will provide you with a list of optimizations that you can try out to further improve the performance. You will write a six-page report that includes your complexity analysis, a discussion of your optimizations, and a quantitative evaluation of the performance across all implementations. **You should consult the programming assignment logistics document for more information about the expectations for all programming assignments and how they will be assessed. While the final code and report are all due at the end of the assignment, we also require meeting an incremental milestone in this PA. Requirements specific to this PA for the incremental milestone and the final report are described at the end of this handout.**

This handout assumes that you have read and understand the course tutorials and that you have attended the discussion sections. To get started, log in to an `ecelinux` server, source the setup script, and clone your group's individual remote repository from GitHub:

```
% source setup-ece2400.sh
% mkdir -p ${HOME}/ece2400
% cd ${HOME}/ece2400
% git clone git@github.com:cornell-ece2400/groupid
% cd ${HOME}/ece2400/groupid/pa4-sys
% tree
```

Where `groupid` should be replaced with your group repository name. **You should never fork your individual remote repository! If you need to work in isolation then use a branch within your individual remote repository.** If you have already cloned your individual remote repository, then use `git pull` to ensure you have any recent updates before working on your programming assignment.

```
% cd ${HOME}/ece2400/groupid
% git pull
% tree pa4-sys
```

For this assignment, you will work in the `pa4-sys` subproject, which includes the following files:

• `CMakeLists.txt`	– CMake config script to generate Makefile
• `src/ece2400-stdlib.h`	– Header file for course standard library
• `src/ece2400-stdlib.cc`	– Source code for course standard library
• `src/sort-int.h`	– Header file for `sort_int`
• `src/sort-int.cc`	– Source code for `sort_int`
• `src/sort-int-adhoc.cc`	– Ad-hoc test program for `sort_int`
• `src/VectorInt.h`	– Header file for `VectorInt`
• `src/VectorInt.cc`	– Source code for `VectorInt`
• `src/VectorInt.inl`	– Inline source code for `VectorInt`

• `src/vector-int-adhoc.cc`	– Ad-hoc test program for `VectorInt`
• `src/Image.h`	– Header file for `Image`
• `src/Image.cc`	– Source code for `Image`
• `src/Image.inl`	– Inline source code for `Image`
• `src/image-adhoc.cc`	– Ad-hoc test program for `Image`
• `src/sort-image.h`	– Header file for `sort_image`
• `src/sort-image.cc`	– Source code for `sort_image`
• `src/sort-image-adhoc.cc`	– Ad-hoc test program for `sort_image`
• `src/VectorImage.h`	– Header file for `VectorImage`
• `src/VectorImage.cc`	– Source code for `VectorImage`
• `src/VectorImage.inl`	– Inline source code for `VectorImage`
• `src/vector-image-adhoc.cc`	– Ad-hoc test program for `VectorImage`
• `src/IHandwritingRecSys.h`	– Header file for the HRS interface class
• `src/HRSLinearSearch.h`	– Header file for `HRSLinearSearch`
• `src/HRSLinearSearch.cc`	– Source code for `HRSLinearSearch`
• `src/HRSBinarySearch.h`	– Header file for the HRS `HRSBinarySearch`
• `src/HRSBinarySearch.cc`	– Source code for the HRS `HRSBinarySearch`
• `src/digits.dat`	– Data file with selected test images
• `src/mnist-utils.h`	– Header file for MNIST-related utilities
• `src/mnist-utils.cc`	– Source code for MNIST-related utilities
• `test/sort-int-directed-test.cc`	– Directed test cases for `sort_int`
• `test/sort-int-random-test.cc`	– Random test cases for `sort_int`
• `test/vector-int-directed-test.cc`	– Directed test cases for `VectorInt`
• `test/vector-int-random-test.cc`	– Random test cases for `VectorInt`
• `test/image-directed-test.cc`	– Directed test cases for `Image`
• `test/image-random-test.cc`	– Random test cases for `Image`
• `test/sort-image-directed-test.cc`	– Directed test cases for `sort_image`
• `test/sort-image-random-test.cc`	– Random test cases for `sort_image`
• `test/vector-image-directed-test.cc`	– Directed test cases for `VectorImage`
• `test/vector-image-random-test.cc`	– Random test cases for `VectorImage`
• `test/hrs-linear-search-directed-test.cc`	– Directed test cases for `HRSLinearSearch`
• `test/hrs-binary-search-directed-test.cc`	– Directed test cases for `HRSBinarySearch`
• `eval/hrs-linear-search-eval.cc`	– Evaluation Program for `HRSLinearSearch`
• `eval/hrs-binary-search-eval.cc`	– Evaluation Program for `HRSBinarySearch`

The programming assignment is divided into the following steps. Complete each step before moving on to the next step.

- Step 1. Implement and test `sort_int` function
- Step 2. Implement and test `VectorInt` class using `sort_int`
- Step 3. Implement and test `Image` class using `VectorInt`
- Step 4. Implement and test `sort_image` function
- Step 5. Implement and test `VectorImage` class using `Image` and `sort_image`
- Step 6. Implement and test `HRSLinearSearch` class
- Step 7. Implement and test `HRSBinarySearch` class

We cannot stress enough how important it is to take an incremental design approach! You really must implement *and test* each step before trying to implement the next step. This means more than just adhoc testing. You must do thorough directed and random testing of each step *before* implementing the next step.

2. Implementation Specifications

The high-level goal for this programming assignment is to implement a handwriting recognition system with two classification algorithms. You will start off by implementing three data structures: `VectorInt`, `Image`, and `VectorImage` and two algorithms: `sort_int` and `sort_image`. `VectorInt` is a resizable vector data structure that stores integers and is very similar to what you have implemented in the second and third programming assignment except now you will be using the object-oriented programming paradigm in C++. `Image` uses a `VectorInt` to store pixels, and `VectorImage` is a vector that stores `Image`s. The `sort_int` algorithm sorts an array of integers while `sort_image` algorithm will sort an array of images based on their intensity. You will then leverage these data structures and algorithms to construct the two handwriting recognition systems: `HRSLinearSearch` and `HRSBinarySearch`.

Note that your implementations cannot use anything from the Standard C library except for the `printf` function defined in `cstdio`, the MIN/MAX macros defined in `climits`, the `NULL` macro defined in `cstddef`, and the `assert` macro defined in `cassert`. Your implementations cannot use anything from the Standard C++ library except for C++ I/O streams from `iostream`.

2.1. `sort_int` Algorithm

The `sort_int` function has the following interface:

- `void sort_int( int* a, int size );`

This function takes as input an integer array `a` with length `size` and sorts numbers in the array in an ascending order. You can copy over any of the sorting algorithms from the previous PA to use in your implementation. You can assume that `size` correctly reflects the size of the input array. Your algorithm must work correctly if `size` is zero, which means the input array pointer `a` may be a NULL pointer. The interface for `sort_int` is provided for you in `src/sort_int.h`. Write your implementation in `src/sort_int.c`.

2.2. `VectorInt` Data Structure

After `sort_int` is implemented and tested, you will then implement a resizable vector data structure that stores data that has type `int`. You are responsible for implementing each of the following functions:

```
VectorInt::VectorInt();
VectorInt::VectorInt( int* array, int size );

VectorInt::~VectorInt();
VectorInt::VectorInt( const VectorInt& vec );
VectorInt::VectorInt& operator=( const VectorInt& vec );

int  VectorInt::size() const;
void VectorInt::push_back( int value );
```

```
int  VectorInt::at( int idx ) const;
bool VectorInt::contains( int value ) const;
void VectorInt::sort();
int  VectorInt::find_closest_linear( int value ) const;
int  VectorInt::find_closest_binary( int value, int k ) const;
void VectorInt::print() const;
int  VectorInt::operator[]( int idx ) const;
```

The specification for these functions is as follows:

- `VectorInt::VectorInt()`
  The default constructor for `VectorInt`. It constructs an empty `VectorInt` by initializing all member fields in `VectorInt`.
- `VectorInt::VectorInt( int* arr, int size )`
  A non-default constructor that construct a `VectorInt` from an array of `int`s with the given `size`. This constructor should perform a deep copy, i.e., copy the values inside the array rather than copying the pointer. Modifying or deleting the input array after construction should have no effect on the constructed `VectorInt` object. You can assume that the input `size` is never greater than the actual size of the array `arr`. Construct an empty `VectorInt` if `size` is 0.
- `VectorInt::~VectorInt()`
  Destructor for `VectorInt`. It frees the memory allocated on the heap by `VectorInt`. Note that you should use the `delete` and `delete[]` operator instead of `free` as in C.
- `VectorInt::VectorInt( const VectorInt& vec )`
  Copy constructor that performs deep copy. It should copy the values stored in `vec`. Subsequent actions on `vec` should have no effect on the constructed `VectorInt`. *You must carefully handle the case when copying from an empty vector!*
- `VectorInt& VectorInt::operator=( const VectorInt& vec)`
  This overloads the assignment operator. Copy the values stored in `vec`. Note that if the current `VectorInt` is not empty, you need to free the memory allocated for it first. *It is very important that you carefully handle the case of self assignment! You must carefully handle the case when assigning from an empty vector!*
- `int VectorInt::size() const`
  Return the current number of elements in the vector. If the `VectorInt` is empty, this function should return 0.
- `void VectorInt::push_back( int value )`
  Push a new element with the given value `value` onto the end of the `VectorInt`. If there is not enough allocated space, dynamically allocate more memory to store both existing elements and the new element. Note that you should use the `new` operator instead of `malloc` as in C.
- `int VectorInt::at( int idx ) const`
  Return the value at the given index `idx` of the `VectorInt`. If the given index is out-of-bound, throw `ece2400::OutOfRange` with a useful error message.
- `bool VectorInt::contains( int value ) const`
  Search the `VectorInt` for the given value and returns `true` if the value is found and `false` otherwise. If the `VectorInt` is empty, then this function should just return `false`.

- `void VectorInt::sort()`

  Sort the internal array in ascending order. You should just call `sort_int` in `src/sort-int.h`.

- `int VectorInt::find_closest_linear( int value ) const`

  Perform a linear search of the `vector` for the value that is closest to the given value (`value`) and return the closest value. The distance between two integers x and y is defined as $|x - y|$. The difference between two integers can be larger than the maximum sized integer. For example, `INT_MAX` - `INT_MIN` is larger than `INT_MAX`. For this PA, we will consider it undefined behavior to store integers whose difference is larger tha `INT_MAX`. If there are multiple values that are equally (and minimally) close to `value`, return the one that has the smallest index in the internal array. If the `VectorInt` is empty, throw `ece2400::OutOfRange` with a proper error message.

- `int VectorInt::find_closest_binary( int value, int k ) const`

  Perform a binary search of the `vector` to find an index that has a value close to the given value (`value`), and then perform a linear search of K elements centered around the index determined by the binary search. To be more specific, you will search $K/2$ images backwards and $K/2$ images forwards from the final image found during the binary search. You will need to carefully handle the case where there are less than $K/2$ images either before or after the final image found during the binary search. Return the closest value from the linear search. The distance between two integers x and y is defined as $|x - y|$. The difference between two integers can be larger than the maximum sized integer. For example, `INT_MAX` - `INT_MIN` is larger than `INT_MAX`. For this PA, we will consider it undefined behavior to store integers whose difference is larger tha `INT_MAX`. If there are multiple values that are equally (and minimally) close to `value` within the linear search, return the smallest one. If the `VectorInt` is empty, throw `ece2400::OutOfRange` with proper error message. You must check that the `VectorInt` is sorted before doing the binary search, and throw `ece2400::InvalidArgument` with a proper error message if it is not sorted.

- `void VectorInt::print() const`

  Print the contents of the `VectorInt`. This function is used for your own debugging purpose. You can implement this function in any way you like. You do not need to test this function.

- `int VectorInt::operator[]( int idx ) const`

  This overloads the subscript operator. It should just return the value at the given index `idx` of the vector without any boundary check. Note that this is different from `at`, since `at` throws an exception when index is out-of-bound.

The functions vary in complexity and some may require just a few lines of code to implement. To give you an idea of how to use this class, here is a simple function that constructs a `VectorInt`, pushes back three values, gets the middle value, and then destructs the `VectorInt`:

```
#include "VectorInt.h"

int main( void )
{
  VectorInt vec;        // Declare a VectorInt on the stack
  vec.push_back( 11 );  // Push back 11
  vec.push_back( 12 );  // Push back 12
  vec.push_back( 13 );  // Push back 13
  int a = vec.at( 1 );  // int a now has 12
}
```

The interface for `VectorInt` is provided for you in `src/VectorInt.h`. Write the implementation of your member variables inside `src/VectorInt.h` and the implementation of each function inside of `src/VectorInt.cc`.

2.3. `Image` Data Structure

After `VectorInt` is implemented and tested, you will then use it to implement the `Image` class for storing small grayscale images. An `Image` uses `VectorInt` to store an array of integers. Each integer represents a pixel in grayscale and has an value within the range of [0, 255]. Lower numbers represent lighter shades (with 0 representing white), while higher numbers represent darker shades (with 255 representing black). Each `Image` object has a label associated with it. `Image` also has an `intensity`, which we define here as the sum of all pixels. You are responsible for implementing each of the following functions except for `print` and `display`, which we have already implemented for you:

```
Image::Image();
Image::Image( const VectorInt& vec, int ncols, int nrows );

int  Image::get_ncols() const;
int  Image::get_nrows() const;
int  Image::at( int x, int y ) const;
void Image::set_label( char l );
char Image::get_label() const;
int  Image::get_intensity() const;
int  Image::distance( const Image& other );
void Image::print() const;
void Image::display() const;

bool Image::operator==( const Image& rhs ) const;
bool Image::operator!=( const Image& rhs ) const;
```

Here is a brief specification for each member function of `Image` class.

- `Image::Image()`
  Default constructor for `Image`. This function constructs an empty `Image` by initializing all data members.
- `Image::Image( const VectorInt& vec, int ncols, int nrows )`
  Non-default constructor that constructs an `Image` from a `VectorInt` given the number of columns (`ncols`) and number of rows (`nrows`). Our `Image` class is only for storing small images, so `ncols` and `nrows` must not be larger than 128. If either dimension is larger than 128, throw `ece2400::InvalidArgument` with a useful error message. If the size of the vector does not match the number of columns and number of rows, throw `ece2400::InvalidArgument` with a useful error message.
- `int Image::get_ncols() const`
  Return the number of columns of the current `Image`. Return 0 if the current `Image` is empty.
- `int Image::get_nrows() const`
  Return the number of rows of the current `Image`. Return 0 if the current `Image` is empty.
- `int Image::at( int x, int y ) const`
  Return the value of the pixel at `x`-th column and `y`-th row. For example, if an `Image` is constructed from `{0,1,2,3}` with `ncols` and `nrows` both equal to 2, then `at(0,0)` returns 0, `at(1,0)` returns

1, `at(0,1)` returns 2, `at(1,1)` returns 3. If `x` or `y` is out-of-bound, throw `ece2400::OutOfRange` with proper error message.

- `void Image::set_label( char label )`
  Set the current label of the `Image` to the given character `label`.
- `char Image::get_label() const`
  Return the current label of the `Image`. If no `set_label` has been called, return `'?'`.
- `int Image::get_intensity() const`
  Return the intensity of the current `Image`. Note that intensity here is simply defined as the sum of all pixels. Since an `Image` cannot be larger than 128×128 and each pixel cannot be larger than 255, it should be possible to write this function without worrying about overflow.
- `int Image::distance( const Image& other )`
  Return the square of the Euclidean distance between this image and image `other`, which is just the sum of the difference between each pixel squared. For example, if image `a` has four pixels `{1,9,9,5}` and `b` has four pixels `{0,4,2,3}` then `distance(a,b)` should return $(1-0)^2 + (9-4)^2 + (9-2)^2 + (5-3)^2 = 55$. If the dimensions of the two images do not match, throw `ece2400::InvalidArgument` with a useful error message. Since an `Image` cannot be larger than 128×128 and each pixel cannot be larger than 255, it should be possible to write this function without worrying about overflow.
- `void Image::print() const`
  Print the label and intensity of the `Image`. We provide you with this function for use in your debugging.
- `void Image::display() const`
  Print the contents of the `Image`. We provide you with this function that makes a pretty grayscale picture based on the contents of the `Image`.
- `bool Image::operator==( const Image& rhs ) const`
  Overload the equal-to operator so that it compares the value of each pixel. Return `true` only if the each pixel in the right-hand-side image is the same as that in the current image. If the dimensions of the two images do not match, simply return `false`. Otherwise return `true`.
- `bool Image::operator!=( const Image& rhs ) const`
  Overload the equal-to operator so that it compares the value of each pixel. Return `false` only if the each pixel in the right-hand-side image is the same as that in the current image. If the dimensions of the two images do not match, simply return `true`. Return `false` if both images are empty.

The functions vary in complexity, and some may require just a few lines of code to implement. The interface for `Image` is provided for you in `src/Image.h`. Write the implementation of your member variables inside `src/Image.h` and the implementation of each function inside of `src/Image.cc`.

2.4. `sort_image` Algorithm

After `Image` is implemented and tested, you will implement the `sort_image` function which has the following interface:

- `void sort_image( Image* a, int size );`

This function takes as input an array `a` of `Images` with length `size` and sorts the `Images` in the array in an ascending order based on their intensity. You can copy over any of the sorting algorithms from the previous PA to use in your implementation, although you will need to modify the algorithm to call `Image::get_intensity` as oppropriate. The interface for `sort_image` is provided for you in `src/sort_image.h`. Write your implementation in `src/sort_image.c`.

2.5. `VectorImage` Data Structure

After `Image` and `sort_image` are implemented and tested, you will implement a `VectorImage` class that stores a vector of `Images`. `VectorImage` has the same member functions as `VectorInt`, except that it should operate on `Images` rather than `ints`. Note that `VectorImage::find_closest_linear` should use `Image::distance` to calculate the Euclidean distance between two images. `VectorImage::sort` should just call `sort_image`, and thus it sorts the images by their intensity.

`VectorImage::find_closest_binary` is similar to `VectorInt::find_closest_binary` with an important difference. We will sort the images based on intensity and then do the binary search based on intensity. This enables us to quickly find images with similar intensity. However, intensity, defined as the sum of all pixels, is not a very good feature to differentiate different digits. For example, it is very likely that a '6' and a '9' have very similar intensity. Therefore, after finding the index, we will do the linear search of K images using the Euclidean distance. To be more specific, you will search $K/2$ images backwards and $K/2$ images forwards from the final image found during the binary search. You will need to carefully handle the case where there are less than $K/2$ images either before or after the final image found during the binary search. Again, this final linear search should be based on the Euclidean distance *not* the intensity. You should return the image that has the smallest Euclidean distance within the K images. Figure 2 illustrates the overall approach: use binary search to quickly zoom in on images with similar intensity and then do a linear search to find the closest image for classification. The hope is that binary search will be faster since it only does a linear search over K images, but that the accuracy is still reasonable since this linear search is over K images with similar intensity (and thus probably the right digit).

2.6. `HRSLinearSearch` Handwriting Recognition System

The first handwriting recognition system you will implement is `HRSLinearSearch`, which uses a brute force linear search algorithm. You are responsible for implementing each of the following functions:

```
HRSLinearSearch::HRSLinearSearch();
void HRSLinearSearch::train( const VectorImage& vec );
Image HRSLinearSearch::classify( const Image& Image );
```

Here is a brief specification for each member function of `HRSLinearSearch`:

- `HRSLinearSearch::HRSLinearSearch()`
  Default constructor for `HRSLinearSearch`. Initialize your member variables if necessary.

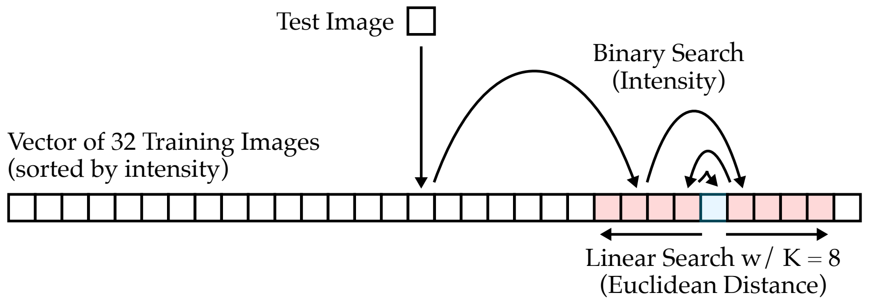

Figure 2: `VectorImage::find_closest_binary` **Example** – Example for 32 images and K = 8. Assumes vector images has already been sorted based on intensity. Binary search is based on intensity and quickly finds images with similar intensity as the test image. Linear search of 8 images is based on Euclidan distance and finds the closest match.

- `void HRSLinearSearch::train( const VectorImage& vec )`
  Train the HRS. For `HRSLinearSearch`, simply store a copy of the vector that contains the training images (`vec`). If you have implemented the assignment operator correctly, this should just be a one-line function.
- `Image HRSLinearSearch::classify( const Image& img )`
  Classify the given image. This function should search through the entire training set and return the image that has the smallest euclidean distance from the given image. You should just call `VectorImage::find_closest_linear`.

2.7. `HRSBinarySearch` Handwriting Recognition System

The second handwriting recognition system you will implement is `HRSBinarySearch`, which uses a mix of binary and linear search. The overall approach is to first sort the training images by their intensity. Then during classification we can use binary search to quickly find training images with similar intensity, and then do a linear search of K images using Euclidean distance. You are responsible for implementing each of the following functions:

```
HRSBinarySearch::HRSBinarySearch( int k );
void HRSBinarySearch::train( const VectorImage& vec );
Image HRSBinarySearch::classify( const Image& Image );
```

Here is a brief specification for each member function of `HRSBinarySearch`:

- `HRSBinarySearch::HRSBinarySearch( int k )`
  Default constructor for `HRSBinarySearch`. Initialize your member variables if necessary.
- `void HRSBinarySearch::train( const VectorImage& vec )`
  Train the HRS. For `HRSBinarySearch`, you need to store a copy of the vector that contains the training images (`vec`), and then sort the training images based on intensity. You should just call `VectorImage::sort`.
- `Image HRSBinarySearch::classify( const Image& img )`
  Classify the given image. This function should use a binary and linear search. You should just call `VectorImage::find_closest_binary`.

3. Testing Strategy

You are responsible for developing an effective testing strategy to ensure all implementations are correct. Writing tests is one of the most important and challenging aspects of software programming. Software engineers often spend far more time implementing tests than they do implementing the actual program.

Note that while there are limitations on what you can use from the Standard C/C++ library in your *implementations* there are no limitations on what you can use from the Standard C/C++ library in your *testing strategy*. You should feel free to use the Standard C/C++ library in your golden reference models and/or for random testing.

3.1. Ad-hoc Testing

To help students start testing, we provide one ad-hoc test program per implementation in `src/sort-int-adhoc.cc`, `src/vector-int-adhoc.cc`, `src/image-adhoc.cc`, `src/sort-image-adhoc.cc`, and `src/vector-image-adhoc.cc`. Students are encouraged to start compiling and running these ad-hoc test programs directly in the `src/` directory without using any build-automation tool (e.g., CMake and Make).

You can build and run the given ad-hoc test programs like this:

```
% cd ${HOME}/ece2400/netid/pa4-sys/src
% g++ -Wall -o sort-int-adhoc ece2400-stdlib.cc sort-int.cc sort-int-adhoc.cc
% ./sort-int-adhoc

% cd ${HOME}/ece2400/netid/pa4-sys/src
% g++ -Wall -o vector-int-adhoc ece2400-stdlib.cc \
    sort-int.cc VectorInt.cc vector-int-adhoc.cc
% ./vector-int-adhoc

% cd ${HOME}/ece2400/netid/pa4-sys/src
% g++ -Wall -o image-adhoc ece2400-stdlib.cc VectorInt.cc Image.cc image-adhoc.cc
% ./image-adhoc

% cd ${HOME}/ece2400/netid/pa4-sys/src
% g++ -Wall -o sort-image-adhoc ece2400-stdlib.cc Image.cc \
     sort-image.cc sort-image-adhoc.cc
% ./sort-image-adhoc

% cd ${HOME}/ece2400/netid/pa4-sys/src
% g++ -Wall -o image-adhoc ece2400-stdlib.cc sort-int.cc VectorInt.cc
%   Image.cc sort-image.cc VectorImage.cc vector-image-adhoc.cc
% ./vector-image-adhoc
```

The `-Wall` flag will ensure that g++ reports all warnings.

3.2. Systematic Unit Testing

While ad-hoc test programs help you quickly see results of your implementations, often too simple to cover most scenarios. We need a systematic and automatic unit testing strategy to hopefully test all possible scenarios efficiently.

In this course, we are using CMake/CTest as a build and test automation tool. For each implementation, we provide a directed test program that should include several test cases to target different categories and a random test program that should test that your implementation works for random inputs. **Unlike in the first three programming assignments, a great deal of tests have already been provided for you!** You can freely leverage the available tests to verify the functionality of your handwriting recognition systems. Remember however that your goal with respect to testing strategy is to convince yourself and the staff that your code is functional. If in order to convince yourself that your code is functional further tests may be needed (maybe just by copying and adjusting existing tests).

We **strongly** encourage students to take an incremental design approach. **Do not implement all of these functions before running your first test!** Instead, we recommend students implement and test each of the following substeps.

- Step 1. Implement and test `sort_int`
- Step 2. Implement and test `VectorInt`
  - Step 2a. Implement and test the default constructor, destructor, `push_back`, `size`, `at`
  - Step 2b. Implement and test the non-default constructor
  - Step 2c. Implement and test `contains`
  - Step 2d. Implement and test `sort`
  - Step 2e. Implement and test `find_closest_linear`
  - Step 2f. Implement and test `find_closest_binary`
  - Step 2g. Implement and test `operator[]`
  - Step 2h. Implement and test the copy constructor
  - Step 2i. Implement and test the assignment operator
- Step 3. Implement and test `Image`
  - Step 3a. Implement and test the default/non-default constructor, `get_ncols`, `get_ncols`, `at`
  - Step 3b. Implement and test `set_label`, `get_label`
  - Step 3c. Implement and test `get_intensity`
  - Step 3d. Implement and test `operator==`, `operator!=`
  - Step 3e. Implement and test `distance`
- Step 4. Implement and test `sort_image`
- Step 5. Implement and test `VectorImage`
  - Step 5a. Implement and test the default constructor, destructor, `push_back`, `size`, `at`
  - Step 5b. Implement and test the non-default constructor
  - Step 5c. Implement and test `contains`
  - Step 5d. Implement and test `sort`
  - Step 5e. Implement and test `find_closest_linear`
  - Step 5f. Implement and test `find_closest_binary`
  - Step 5g. Implement and test `operator[]`
  - Step 5h. Implement and test the copy constructor
  - Step 5i. Implement and test the assignment operator
- Step 6. Implement and test `HRSLinearSearch`
- Step 7. Implement and test `HRSBinarySearch`

Each substep should correspond to one or more directed test cases.

As in the previous programming assignment, we provide you a testing framework you should use for your directed and random testing. See the provided test programs in the `test` subdirectory for how to use this framework. The ECE 2400 standard library in `ece2400-stdlib.h` contains the following macros you should use to check the correctness of your implementations:

- `ECE2400_CHECK_FAIL()` – check program does not reach this point
- `ECE2400_CHECK_TRUE( expr_ )` – check `expr_` is always true
- `ECE2400_CHECK_FALSE( expr_ )` – check `expr_` is always false
- `ECE2400_CHECK_INT_EQ( expr0_, expr1_ )` – check `expr0_` equals `expr1_`

Before running the tests you need to create a separate `build` directory and use `cmake` to create the `Makefile` like this:

```
% cd ${HOME}/ece2400/groupid/pa4-sys
% mkdir -p build
% cd build
% cmake ..
```

Now you can build and run all unit tests for all implementations like this:

```
% cd ${HOME}/ece2400/groupid/pa4-sys/build
% make check
```

If you are failing a test program, then you can "zoom in" and run all of the unit tests for a single test program (e.g., directed tests for `sort_int`) like this:

```
% cd ${HOME}/ece2400/groupid/pa4-sys/build
% make sort-int-directed-test
% ./sort-int-directed-test
```

You can then "zoom in" to a specific test case by passing in the index of that test case like this:

```
% cd ${HOME}/ece2400/groupid/pa4-sys/build
% make sort-int-directed-test
% ./sort-int-directed-test 1
% ./sort-int-directed-test 2
```

3.3. Memory Leaks

Students are also responsible for making sure that their program contains no memory leaks or other issues with dynamic allocation. We have included a make target called `memcheck` which runs all of the test programs with Valgrind. Valgrind will force the test to fail if it detects any kind of memory leak or other issues with dynamic allocation.

You can check memory leaks and other issues with dynamic memory allocation for all your test programs like this:

```
% cd ${HOME}/ece2400/groupid/pa4-sys/build
% make memcheck
```

You can just check one test program (e.g. `vector-int-directed-test`) like this:

```
% cd ${HOME}/ece2400/groupid/pa4-sys/build
% make vector-int-directed-test
% valgrind --trace-children=yes --leak-check=full \
    --error-exitcode=1 --undef-value-errors=no ./sort-int-directed-test
```

Those are quite a few command line options to Valgrind, so we have created a ‘ece2400-valgrid‘ script. This script is just a simple wrapper which calls Valgrind with the right options.

```
% cd ${HOME}/ece2400/groupid/pa4-sys/build
% make sort-int-directed-test
% ece2400-valgrind ./sort-int-directed-test
```

3.4. Code Coverage

After your implementations pass all unit tests, you can evaluate how effective your test suite is by measuring its code coverage. The code coverage will tell you how much of your source code your test suite executed during your unit testing. The higher the code coverage is, the less likely some bugs have not been detected. You can run the code coverage like this:

```
% cd ${HOME}/ece2400/groupid/pa4-sys
% rm -rf build-coverage
% mkdir -p build-coverage
% cd build-coverage
% cmake ..
% make check
% make coverage
```

Note that these code coverage results will reflect *all* prior runs of the test and evaluation programs in the build directory. That is why in the above example, we do a fresh build in a separate `build-coverage` build directory.

If you want to drill down and explore the coverage of each line in a program you use use the elinks web browser like this:

```
% cd ${HOME}/ece2400/groupid/pa4-sys/build-coverage
% elinks coverage-html/index.html
```

Code coverage is just one more piece of evidence you can use to make a compelling case for the correct functionality of your implementations. It is not required that students achieve 100% code coverage. It is far more important that students simply use code coverage as a way to guide their test-driven design than to overly focus on the specific code coverage number.

4. Evaluation

In the first three PAs, students gained first-hand experience with test-driven design through implementing a compelling testing strategy. The final two PAs shift to focus more on incremental profiling and optimization.

4.1. Evaluating the Unoptimized HRS

Once you have verified the functionality of your handwriting recognition systems, you can evaluate their performance and accuracy with breakdowns for both the training phase and the classification phase. You can build and the evaluation programs like this:

```
% cd ${HOME}/ece2400/groupid
% mkdir -p pa4-sys/build-eval
% cd pa4-sys/build-eval
% cmake -DCMAKE_BUILD_TYPE=eval ..
% make eval
```

Note that we are working in a separate `build-eval` directory, and that we are using the `-DCMAKE_BUILD_TYPE=eval` command line option to the `cmake` script. to create optimized executables without any extra debugging information. **You must do your quantitative evaluation using an eval build. Using a debug build for evaluation produces meaningless results.**

The following runs a complete evaluation with 60K training images, 10K classification images, and K set to 1000.

```
% cd ${HOME}/ece2400/groupid/pa4-sys/build-eval
% make hrs-linear-search-eval
% ./hrs-linear-search-eval
% make hrs-binary-search-eval
% ./hrs-binary-search-eval
```

You can also specify the size of the number of training images, the number of classification images, and the size of the final linear search when using binary search on the command line for each evaluation program:

```
% ./hrs-linear-search-eval N M
% ./hrs-binary-search-eval N M K
```

Where *N* is the number of training images, *M* is the number of classification images, and *K* is the size of the final linear search when using the binary search. Note that, using a smaller number of training images and/or classification images is only for profiling and interative performance optimization. All accuracy results must use the full 60K training dataset and 10K classification dataset.

Since the implementation is still unoptimized at this point, the classification can be quite slow. For example, it might take over 20 minutes for the unoptimized version of `HRSLinearSearch`. Record the unoptimized training time, classification time, and total execution time for the report.

After recording the unoptimized performance, you will need to do some performance profiling to figure out where the bottleneck is and what to optimize. You will use `perf` to create a flame graph for each implementation. *You should only use `perf` if the execution time is about five minutes or less. If you use `perf` when the execution time is longer it will create a huge trace file which will fill up your home directory!* You can create the flame graph for the `HRSLinearSearch` with the following command:

```
% cd ${HOME}/ece2400/groupid/pa4-sys/build-eval
% perf record --call-graph dwarf ./hrs-linear-search-eval 60000 100
% perf script report stackcollapse | flamegraph.pl > graph-linear-unopt.svg
```

Notice that we only use 100 classification (test) images because using the full dataset to create the flame graph will create a huge trace file. 100 images is representative enough for iterative performance optimization. You can create the flame graph for the `HRSBinarySearch` with the following command:

```
% cd ${HOME}/ece2400/groupid/pa4-sys/build-eval
% perf record --call-graph dwarf ./hrs-binary-search-eval
% perf script report stackcollapse | flamegraph.pl > graph-binary-unopt.svg
```

Notice that for `HRSBinarySearch` you should probably use the full classification dataset because otherwise the ratio between the training time and classification time can be misleading. You can download the flame graph using VS Code.

Save these two flame graphs for your report. You should use your flame graph to determine what is the bottleneck in your implementation and what optimizations you may want to choose. **We highly recommend you do a `git commit` before you move on to the next step, so that if you want to reproduce the unoptimized results in the future, you can simply checkout to this commit without having to change your code back.**

4.2. Evaluating the Optimized HRS

We provide you with a list of optimizations that may improve performance. Note that any optimization must not change the provided interface of any class, although may add new public member functions if you wish. For example, you might want to add a public member function that performs a linear find closest over a subrange, or a public member function that perform a swap. Optimizations can only change the implementation of various classes. You are absolutely not required to implement all of these optimizations. Some optimizations may make a significant difference, while other operations may make no difference at all. You must think critically to choose what optimizations to implement! Start by examining the flame graph from the previous step. Identify what functions are taking a significant amount of time. Then you can try one of three things: (1) can you avoid calling the function at all? (2) Can you optimize the code inside the function? (3) If the code inside the function is simple you can potentially use inlining.

- *Choosing the Right Algorithm*

  Always start your optimizations by making sure you are using the right algorithm with an appropriate complexity class. For example, carefully choose an algorithm for `VectorInt::push_back`, `VectorImage::push_back`, `VectorInt::sort`, and `VectorImage::sort`. If you use quick sort, then carefully consider how you choose the pivot, since this can have a significant impact on performance in practice.

- *Choosing the Right Function*

  Sometimes an easy way to optimize your code is to simply choose to call a faster function which has the same behavior as a slower function. For example, while you might start by using `VectorInt::at()` (which does bounds checking) inside your Euclidean distance calculation, you might want to considering replacing these calls to `VectorInt::at()` with the `[]` operator (which does not do bounds checking) once you are sure your implementation is correct.

- *Accessing Member Fields Directly*

  Member functions can either use a different member function to access member fields, or they can access member fields directly. Using a different member function can sometimes improve readability by refactoring code, but directly accessing member fields can sometimes be more efficient. For example, using `push_back` in the copy constructor or assignment operator for `VectorInt` and `VectorImage` can be less efficient compared to dynamically allocating all of the necessary memory at once. Similarly, using `Image::at()` to access pixels within the `Image` member functions can be less efficient compared to directly accessing the private vector of pixels. Note that it is never acceptable to make member fields public to improve performance since this would break data encapsulation (and change the interface).

- *Function Inlining*

Inlining a function can eliminate the overhead of making a function call. Inlining essentially tells the compiler to replace a function call by copying the function body into the caller. To inline a function simply move it from the `.cc` file into the `.inl` file and add the `inline` keyword at the beginning of the function declaration. Note that inlining large functions can actually *reduce* performance, so only inline relatively small functions.

- *Constant References*

  Passing an object by value or returning a copy of an object will call the copy constructor and can be inefficient. You can also just pass or return a constant reference to the object. Note that you need to use the `const` keyword otherwise the data stored in the data structure can be mutated unintentionally. You will need to change the `.h` but technically this will not change the interface since a user should not be able to distinguish the difference between returning a copy and returning a const reference.

- *Memoization*

  In `Image::get_intensity`, the intensity is calculated each time `get_intensity` is called. You can instead calculate the intensity the *first* time `get_intensity` is called, save (or memoize) the intensity to an internal member field, and then simply reuse this saved value the *next* time `get_intensity` is called. Alternatively, you can calculate the intensity ahead of time in the constructor and simply return it in the `get_intensity` method. Similarly, `VectorInt::find_closest_binary` and `VectorImage::find_closest_binary` check if the internal array is sorted each time `find_closest_binary` is called. You can instead check if the array is sorted the *first* time `find_closest_binary` is called, save (or memoize) a flag stored as an internal member field indicating if the array is sorted, and then simply check this flag the *next* time `find_closest_binary` is called. Alternatively, you can update such a flag after calling `sort`. Note that if a user later uses `push_back` you will need to reset this flag. Since `get_intensity` and `find_closest_binary` are `const` you will need to use the `mutable` keyword so that a member variable can be modified in a `const` member function.

- *Efficient Swap*

  In `VectorImage::sort`, when you swap two images, you will have to copy the image multiple times, which can be slow. It is far more efficient to just swap the pointers inside the `VectorInt`. However, the interface does not provide you with access to the internal pointer. You can define a `VectorInt::swap` public member function that swaps a given vector with the current vector, and a `Image::swap` public member function that swaps a given image with the current image. Then you can use the `Image::swap` member function in your in your `VectorImage::sort`.

- *More Compact Image Storage*

  Since the value for each pixel is within [0,255], it is enough to store a pixel with just 8 bits. You can implement a `VectorByte` data structure that stores a vector of bytes (i.e., `unsigned char`) and then you can use this `VectorByte` data structure in the implementation of `Image`. Note you are not allowed to change the *interface* of `Image` so you will need to carefully convert from a `VectorInt` in the constructor and convert to an `int` in the return value for `Image::at`.

With proper optimizations, you should be able to reduce the execution time for the full classification dataset by 4–8$\times$. **For each optimization you try record the new training, classification, and total time for the report. Save the two final flame graphs for your most optimized implementations.**

5. Milestone and Report

This section includes critical information about the incremental milestone, final code submission, and the final report specific to this PA. **The programming assignment logistics document provides**

general details about the requirements for the milestone and final submission. You must actually read the document to ensure you know how we will access your milestone and final submission.

5.1. Incremental Milestone

While the final code and report are all due at the end of the assignment, we also require you to complete an incremental milestone, push and submit your code to GitHub on the date specified by the instructor. More specifically to meet the incremental milestone of this PA, you are expected to:

- Complete the implementation of `sort_int`
- Complete the implementation of `VectorInt`
- Pass all given directed and random tests for these implementations
- Consider adding a few of your own directed tests

Here is how we will be testing your milestone:

```
% mkdir -p ${HOME}/ece2400/submissions
% cd ${HOME}/ece2400/submissions
% rm -rf repo
% git clone git@github.com:cornell-ece2400/groupid

% cd ${HOME}/ece2400/groupid/pa4-sys
% mkdir -p build
% cd build
% cmake ..
% make check-milestone
```

5.2. Final Code Submission

Your code quality score will be based on the way you format the text in your source files, proper use of comments, deletion of instructor comments, and uploading the correct files to GitHub (only source files should be uploaded, no generated build files). To assist you in formatting your code correctly, we have created a make target that will autoformat the code for you. You can use it like this:

```
% cd ${HOME}/ece2400/groupid/pa4-sys
% mkdir -p build
% cd build
% cmake ..
% make autoformat
% git diff
# ... check all changes ...
% git commit -a -m "autoformat"
```

Note that the `autoformat` target will only work if you have already committed all of your work. This way you can easily use `git diff` to view the changes made by the autoformatting and commit those changes when you are happy with them. Since we provide students an automated way to format their code correctly, students have no excuse for not following the course coding conventions!

Note that students must remove unnecessary comments that are provided by instructors in the code distributed to students. Students must not commit executable binaries or any other unnecessary files. The `autoformat` target will not take care of these issues for you.

To submit your code you simply upload it to GitHub. Your code will be assessed both in terms of functionality and code quality. Your functionality score will be determined by running your code against a series of tests developed by the instructors to test its correctness. Here is how we will be testing your final code submission:

```
% mkdir -p ${HOME}/ece2400/submissions
% cd ${HOME}/ece2400/submissions
% rm -rf repo
% git clone git@github.com:cornell-ece2400/netid

% cd ${HOME}/ece2400/submissions/groupid/pa4-sys
% mkdir -p build
% cd build
% cmake ..
% make check
% make memcheck
% make eval
# ... run the eval programs ...
```

5.3. Final Report

The final report must be uploaded to Canvas. The date you upload your report will indicate how many slip days you are using for the assignment. For this PA, we require you to include five sections: introduction, complexity analysis, optimization, quantitative evaluation, conclusion, and work distribution. Your entire report must be no more than eight pages. Your report will include quite a few tables and plots.

The complexity analysis section of your report must include a table that includes the execution time of the training and classification phases as a function of N and K. The table should include the time complexity (in big-O notation) for the training and classification phases with N as the input variable. The table should include the execution time for training and classifying with N training images and M test images as a function of N, M, and K. Finally, the table should include the time complexity (in big-O notation) when training and classifying with N training images and M test images. In this situation, we have two input variables, so you should first analyze the time complexity with respect to N assuming M is a constant (i.e., we will classify a fixed number of images while the number of training images grows large), and then analyze the time complexity with respect to M assuming N is a constant (i.e., we have a fixed number of training images as the number of images we wish to classify grows large). See Table 1 for a template. Justify your table entries. You should discuss (at a high level) your chosen units. We recommend using number of images accessed in the classification dataset. Note that you don't need to explicitly discuss all six steps of complexity analysis and we are not looking for a rigorously formal proof, but you do need to be clear about the assumptions you made during analysis and provide some kind of compelling high-level argument.

The optimization section of your report should clearly discuss each optimization you evaluated. You must include a table in your report with one column for training time, one column for classification time, one column for total time, and one row for each optimization you tried. The first row should be the unoptimized results. You must include two flame graphs for your unoptimized implementations (one for linear search and one for binary search), and two flame graphs for your final optimized implementations (one for linear search and one for binary search).

			`HRSLinearSearch`	`HRSBinarySearch`
training	execution time	$T_K(N)$		
training	time complexity w.r.t. N	big-O		
classify one image	execution time	$T_K(N)$		
classify one image	time complexity w.r.t. N	big-O		
training + classify M images	execution time	$T_K(N, M)$		
training + classify M images	time complexity w.r.t. N	big-O		
training + classify M images	time complexity w.r.t. M	big-O		

Table 1: Template for Complexity Analysis Table

The quantitative evaluation section of your report should include the following five plots along with corresponding discussion and analysis.

- Plot of execution time of just the training phase on the y-axis vs. the size of the training dataset (N) on the x-axis. There should be two lines: one for linear search and one for binary search.
- Plot of execution time of just the classification phase on the y-axis vs. the size of the training dataset (N) on the x-axis. There should be two lines: one for linear search and one for binary search. Use the full classification dataset ($M = 10\text{K}$) and $K = 1000$.
- Plot of total execution time of the training plus classification phases on the y-axis vs. the size of the training dataset (N) on the x-axis. There should be two lines: one for linear search and one for binary search. Use the full classification dataset ($M = 10\text{K}$) and $K = 1000$. You should use this plot to determine the break-even point for N; i.e., how large does the training dataset need to be for one algorithm to always perform the best assuming we are doing 10K classifications?
- Plot of total execution time of the training plus classification phases on the y-axis vs. the size of the classification dataset (M) on the x-axis. There should be two lines: one for linear search and one for binary search. Use the full training dataset ($N = 60\text{K}$) and $K = 1000$. You should use this plot to determine the break-even point for M; i.e., how large does the classification dataset need to be for one algorithm to always perform the best assuming we have 60K training images?
- Plot of total execution time of the training plus classification phases on the y-axis vs. accuracy on the x-axis for the full training dataset ($N = 60\text{K}$) and full classification dataset ($M = 10\text{K}$). The plot should have one point for `HRSLinearSearch` and five points for `HRSBinarySearch` each for a different K. Make sure your label the points with the corresponding value of K. We recommend using K = 1, 10, 100, 1000, 10000.

The work distribution section should be a one paragraph description of which student did what work. It is perfectly fine if one students does more work; the key is to transparent and honest about the work distribution across students.

Acknowledgments

This programming assignment was created by Christopher Batten, Christopher Torng, Tuan Ta, Yanghui Ou, Peitian Pan, and Nick Cebry as part of the ECE 2400 Computer Systems Programming course at Cornell University. We also thank the curators of the MNIST database of handwritten digits.