# ECE 2400 Computer Systems Programming, Spring 2025
# PA5: Handwriting Recognition Systems – Tree vs. Table Search

School of Electrical and Computer Engineering
Cornell University

revision: 2025-04-25-20-48

## 1. Introduction

The fifth programming assignment will enable you to build off of the work you did in the previous programming assignment to continue to apply all of the concepts you have learned throughout the semester in the context of a machine learning system. You will continue to 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 also now leverage what you have learned on generic programming (e.g., templates), tree data structures, and table data structures. 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.

In this assignment, you will revisit the supervised learning models that classify handwritten digits developed in the previous programming assignment, and you will add three new models. Recall that these models have 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 (Figure 1). 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 a generic sorting algorithm before implementing four classes: `Vector<T>`, `Image`, `Tree<T,CmpFunc>` **or** `Table<T,HashFunc>`. Then you will leverage these classes to reimplement the two simple classification algorithms from the previous programming assignment (one based on linear search and the other based on binary search), before implementing two new classification algorithms (one based on tree search and the other based on table search). You will also have the opportunity to combine a Python GUI front-end with your C++ handwriting recognition back-end to create a complete system which will enable users to draw numbers which can then be (hopefully correctly) classified!

After your handwriting recognition systems are functional and tested, you will evaluate the accuracy and performance trade-offs between implementations. You should include all of the optimizations you explored in the previous programming assignment. You will write a report that includes your complexity analysis, a discussion of your optimizations, and a quantitative evaluation of the performance across all implementations. 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.

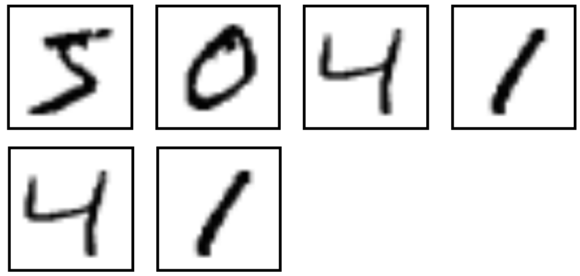


**Figure 1: Four Example MNIST Images –** Images include $28 \times 28$ grayscale pixels and a label. Each pixel has a value between 0 and 255 where 0 represents white, 255 represents black, and intermediate values represent intermediate levels of gray.

We assume that you have already cloned your pair repository. Use `git pull` to obtain the `pa5-sys` release code and to ensure you have any recent updates before working on your programming assignment.

```
% cd ${HOME}/ece2400/pair-xx
% git pull
% tree pa5-sys
```

For this assignment, you will work in the `pa5-sys` directory, which includes the following files:

```
|-- CMakeLists.txt
|-- eval
|   |-- CMakeLists.txt
|   |-- hrs-backend.cc
|   |-- hrs-binary-search-eval.cc
|   |-- hrs-frontend.py
|   |-- hrs-linear-search-eval.cc
|   |-- hrs-tree-search-eval.cc
|   |-- hrs-table-search-eval.cc
|-- include
|   |-- digits.dat
|   |-- ece2400-stdlib.h
|   |-- ece2400-stdlib.inl
|   |-- HRSBinarySearch.h
|   |-- HRSLinearSearch.h
|   |-- HRSTableSearch.h
|   |-- HRSTreeSearch.h
|   |-- IHandwritingRecSys.h
|   |-- Image.h
|   |-- Image.inl
|   |-- mnist-utils.h
|   |-- sort-directed-test.h
|   |-- sort.h
|   |-- sort.inl
|   |-- sort-random-test.h
|   |-- table-directed-test.h
|   |-- Table.h
|   |-- Table.inl
|   |-- table-random-test.h
|   |-- tree-directed-test.h
|   |-- Tree.h
|   |-- Tree.inl
|   |-- tree-random-test.h
|   |-- vector-directed-test.h
|   |-- Vector.h
|   |-- Vector.inl
|   |-- vector-random-test.h
|-- README.md
|-- scripts
|   |-- build.sh
|   |-- coverage.sh
|   |-- eval.sh
|   |-- format.sh
|   |-- memcheck.sh
|   |-- test.sh
|   |-- valgrind.sh
|-- src
|   |-- CMakeLists.txt
|   |-- ece2400-stdlib.cc
|   |-- HRSLinearSearch.cc
|   |-- HRSBinarySearch.cc
|   |-- HRSTreeSearch.cc
```

```
|   |-- HRSTableSearch.cc
|   |-- image-adhoc.cc
|   |-- Image.cc
|   |-- mnist-utils.cc
|   |-- sort-adhoc.cc
|   |-- vector-adhoc.cc
|   |-- tree-adhoc.cc
|   |-- table-adhoc.cc
|-- test
    |-- CMakeLists.txt
    |-- image-directed-test.cc
    |-- image-random-test.cc
    |-- sort-directed-test.cc
    |-- sort-random-test.cc
    |-- sort-int-directed-test.cc
    |-- sort-int-random-test.cc
    |-- sort-image-directed-test.cc
    |-- sort-image-random-test.cc
    |-- vector-directed-test.cc
    |-- vector-random-test.cc
    |-- vector-int-directed-test.cc
    |-- vector-int-random-test.cc
    |-- vector-image-directed-test.cc
    |-- vector-image-random-test.cc
    |-- tree-directed-test.h
    |-- tree-random-test.h
    |-- tree-int-directed-test.cc
    |-- tree-int-random-test.cc
    |-- tree-image-directed-test.cc
    |-- tree-image-random-test.cc
    |-- table-directed-test.h
    |-- table-random-test.h
    |-- table-int-directed-test.cc
    |-- table-int-random-test.cc
    |-- table-image-directed-test.cc
    |-- table-image-random-test.cc
    |-- hrs-linear-search-directed-test.cc
    |-- hrs-binary-search-directed-test.cc
    |-- hrs-tree-search-directed-test.cc
    |-- hrs-table-search-directed-test.cc
```

By the end of this programming assignment, you will have written many hundreds of lines of code, and you will be able to read and understand almost ten thousand lines of code which are included in this assignment spanning implementation, testing, and evaluation.

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

- Step 1. Implement and test generic `sort<T,CmpFunc>` function
- Step 2. Implement and test generic `Vector<T>` class
- Step 3. Implement and test `Image` class
- Step 4a. Implement and test generic `Tree<T,CmpFunc>` class if implementing `Tree`
- Step 4b. Implement and test generic `Table<T,HashFunc>` class if implementing `Table`
- Step 5. Implement and test `HRSLinearSearch` class
- Step 6. Implement and test `HRSBinarySearch` class
- Step 7a. Implement and test `HRSTreeSearch` class if implementing `Tree`
- Step 7b. Implement and test `HRSTableSearch` class if implementing `Table`

**Take an incremental design approach!** Implement *and test* each step before moving on to the next step. This means more than adhoc testing. Perform thorough directed and random testing of each step *before* beginning the next step.

## 2. Implementation Specifications

The high-level goal for this programming assignment is to implement a handwriting recognition system with four classification algorithms. Start by implementing a generic sorting algorithm (`sort<T,CmpFunc>`) which is basically a generic version of the sorting algorithm you implemented in the previous programming assignment. The sorting algorithm should work for any type `T` and also takes a comparison function object. Then implement a generic `Vector<T>` data structure which is basically a generic version of the resizable vector you implemented in the previous programming assignment. Your

Vector<T> will make use of sort<T,CmpFunc>. Image uses a Vector<int> to store pixels, and you can also now use Vector<Image> to create a vector that stores Images. You will implement a generic tree data structure (Tree<T,CmpFunc>) or a table data structure (Table<T,HashFunc>). You will then leverage these data structures to construct the four handwriting recognition systems: HRSLinearSearch, HRSBinarySearch, and HRSTreeSearch, or HRSTableSearch.

Your implementations cannot use anything from the Standard C library except for the printf function defined in stdio.h, the MIN/MAX macros defined in limits.h, the NULL macro defined in stddef.h, and the assert macro defined in assert.h. Your implementations cannot use anything from the Standard C++ library except for C++ I/O streams from iostream.

### 2.1. sort<T,CmpFunc> Algorithm

You will implement a generic sort algorithm that can sort an array of elements of any type T. This algorithm is similar to what you implemented in the previous programming assignment; the difference is that it should be generic across any type T, and that it takes a comparison function object (i.e., function pointer, functor, lambda) which should be used by your algorithm to compare different objects of type T. The interface for the generic sorting function is as follows:

```
template < typename T, typename CmpFunc >
void sort( T* a, int size, CmpFunc cmp );
```

Note how we use the CmpFunc template parameter to indicate the type of the comparison function cmp. More specifically, the comparison function cmp should be callable with the following function signature: bool cmp(const T& a, const T& b). The function should return true if object a is strictly less than b, and false otherwise. In this PA, there is no guarantee that comparison operators (e.g., operator<) have been defined for type T, so your sorting algorithm should use cmp(a,b) in places where you want to compare the value of a and b. For example, a < b is simply cmp(a,b), a > b is equivalent to cmp(b,a), and a == b is equivalent to !cmp(a,b) && !cmp(b,a). Your algorithm must work correctly if size is zero, which means the input array pointer a may be a nullptr.

The interface for sort<T,CmpFunc> is provided for you in src/sort.h. Write the implementation of your function inside of src/sort.inl. Note that since this is a templated data structure, all of the templated definitions must be placed in the .inl file not in a .cc file. We cannot compile the function template since we don't know the types of T nor CmpFunc yet! We can only compile a function template specialization.

### 2.2. Vector<T> Data Structure

Implement a generic resizable vector data structure that stores data of type T and differs from the previous programming assignment as follows: (1) it should be generic across any type T; (2) its find_closest member function is templated and takes a distance function object as an argument; and (3) its sort member function is templated and takes a comparison function object as an argument. Implement each of the following functions except for print which is already implemented:

```
Vector<T>::Vector();
Vector<T>::Vector( T* array, int size );

Vector<T>::~Vector();
Vector<T>::Vector( const Vector<T>& vec );
Vector<T>& Vector<T>::operator=( const Vector<T>& vec );
```

```
int      Vector<T>::size() const;
void     Vector<T>::push_back( const T& value );
const T& Vector<T>::at( int idx ) const;
T&       Vector<T>::at( int idx );
bool     Vector<T>::contains( const T& value ) const;

template <typename DistFunc>
T        Vector<T>::find_closest_linear( const T& value, DistFunc dist ) const;

template <typename DistFunc, typename CmpFunc>
T        Vector<T>::find_closest_binary( const T& value, int k,
                                         DistFunc dist, CmpFunc cmp ) const;

template <typename CmpFunc>
void     Vector<T>::sort( CmpFunc cmp );

void     Vector<T>::print() const;
const T& Vector<T>::operator[]( int idx ) const;
T&       Vector<T>::operator[]( int idx );
```

The specification for these functions is as follows:

- `Vector<T>::Vector()`

  The default constructor for `Vector<T>`. It constructs an empty `Vector<T>` by initializing all member fields in `Vector<T>`.

- `Vector<T>::Vector( T* arr, int size )`

  A non-default constructor that constructs a `Vector<T>` from an array of `T`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 `Vector<T>` object. You can assume that the input `size` is never greater than the actual size of the array `arr`. Construct an empty `Vector<T>` if `size` is 0.

- `Vector<T>::~Vector()`

  Destructor for `Vector<T>`. It frees the memory allocated on the heap by `Vector<T>`. Note that you should use the `delete` and `delete[]` operator instead of `free` as in C.

- `Vector<T>::Vector( const Vector<T>& vec )`

  Copy constructor that performs a deep copy. It should copy the values stored in `vec`. Subsequent actions on `vec` should have no effect on the constructed `Vector<T>`. *You must carefully handle the case when copying from an empty vector!*

- `Vector<T>& Vector<T>::operator=( const Vector<T>& vec)`

  This overloads the assignment operator. Copy the values stored in `vec`. Note that if the current `Vector<T>` 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 also carefully handle the case when assigning from an empty vector!*

- `int Vector<T>::size() const`

  Return the current number of elements in the vector. If the `Vector<T>` is empty, this function should return 0.

- void Vector<T>::push_back( const T& value )

  Push a new element with the given value value onto the end of the Vector<T>. 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 or new[] operator instead of malloc as in C.

- const T& Vector<T>::at( int idx ) const

  Return the value at the given index idx of the Vector<T>. If the given index is out-of-bound, throw ece2400::OutOfRange with a useful error message. Note that this is the "read" version of at since it is declared const and returns a const reference.

- T& Vector<T>::at( int idx )

  Return the value at the given index idx of the Vector<T>. If the given index is out-of-bound, throw ece2400::OutOfRange with a useful error message. Note that this is the "write" version of at since it returns a non-const reference. It enables writing items in the Vector<T>.

- bool Vector<T>::contains( const T& value ) const

  Search the Vector<T> for the given value and returns true if the value is found and false otherwise. If the Vector<T> is empty, then this function should just return false.

- template <typename DistFunc>
  T Vector<T>::find_closest_linear( const T& value, DistFunc dist ) const

  Perform a linear search of the vector for the value that is closest to the given value (value), as measured by the dist distance function object. The distance function object dist should be callable with the following function signature: int dist(const T& a, const T& b). The function should take two inputs of type T and return their distance as int. If multiple values have the same minimal distance, return the one that has the lowest index. As in the previous PA, is undefined if calculating the distance results in overflow. If the Vector<T> is empty, throw ece2400::OutOfRange with proper error message. Note that since this is a template member function in a template class, you need the following syntax to define Vector<T>::find_closest_linear:

  ```
  template <typename T>
  template <typename DistFunc>
  T Vector<T>::find_closest_linear( const T& value, DistFunc dist ) const
  {
    // ...
  }
  ```

- template <typename DistFunc, typename CmpFunc>
  T Vector<T>::find_closest_binary( const T& value, int k,
  DistFunc dist, CmpFunc cmp ) 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. The binary search should use the given cmp comparison function object, while the linear search should use the given dist comparison function. As in the previous PA, you will search $K/2$ items backwards and $K/2$ items forwards from the item found during the binary search. You will need to carefully handle the case where there are less than $K/2$ items either before or after the item found during the binary search. Return the closest value from the linear search. As in the previous PA, is undefined if calculating the distance results in overflow. If the Vector<T> is empty, throw ece2400::OutOfRange with proper error message. You must check that the Vector<T> is sorted before doing the binary search, and throw ece2400::InvalidArgument with a proper error message if it is not sorted.

```
template <typename T>
template <typename DistFunc, typename CmpFunc>
T Vector<T>::find_closest_binary( const T& value, int k,
                                  DistFunc dist, CmpFunc cmp ) const
{
  // ...
}
```

- `template <typename CmpFunc>`
  `void Vector<T>::sort( CmpFunc cmp )`
  Sort the internal array in an ascending order defined by the comparison function `cmp`. You must use the generic `sort<T,CmpFunc>` function developed in the previous step. Note that since this is a template member function in a template class, you need the following syntax in the definition of `Vector<T>::sort`:

  ```
  template <typename T>
  template <typename CmpFunc>
  void Vector<T>::sort( CmpFunc cmp )
  {
    ::sort( ... );
  }
  ```

  Notice now we need to explicitly use `::` so the compiler knows we want to call the generic global `sort` function and not the `sort` member function.

- `void Vector<T>::print() const`
  Print the content in the `Vector<T>`. This member function is used for debugging purposes. Note for this member function to be generic it needs to use the C++ `iostream` library (e.g., `std::cout`) instead of `printf`. This is because the type `T` may not even have a format specifier for `printf`. We provide you with this function.

- `const T& Vector<T>::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` as `at` throws an exception when index is out-of-bound. Note that this is the "read" version since it is declared `const` and returns a `const` reference.

- `T& Vector<T>::operator[]( int idx )`
  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` as `at` throws an exception when index is out-of-bound. Note that this is the "write" version since it returns a non-`const` reference. It enables writing items in the `Vector<T>`.

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 `Vector<int>`, pushes back three values, gets the middle value, and then destructs the `Vector<int>`:

```
int main( void )
{
  Vector<int> vec;       // Declare a Vector<int> 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 `Vector<T>` is provided for you in `src/Vector.h`. Write the implementation of your member variables inside `src/Vector.h` and the implementation of each function inside of `src/Vector.inl`. Note that since this is a templated data structure, all of the templated definitions must be placed in the `.inl` file not in a `.cc` file. We cannot compile the class template since we don't know the type `T` yet! We can only compile a class template specialization.

#### 2.3. `Image` Data Structure

After `Vector<T>` is implemented and tested, you will then use it to implement the `Image` class. `Image` uses `Vector<int>` 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`, `display`, and the << operator, which have been implemented for you:

```
Image::Image();
Image::Image( const Vector<int>& 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 ) const;
void Image::print() const;
void Image::display() const;

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

const int& Image::operator[]( int idx ) const;

std::ostream& operator<<( std::ostream& os, const Image& image );
```

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

- `Image::Image()`

  Default constructor for `Image`. This function constructs an empty `Image` by initializing all data members and setting the label to `'?'`.

- `Image::Image( const Vector<int>& vec, int ncols, int nrows )`

  Non-default constructor that constructs an `Image` from a `Vector<T>` given the number of columns (`ncols`) and number of rows (`nrows`). 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-bounds, 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. You can assume no overflow will occur for all arithmetic operations in this function.

- `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 $128 \times 128$ and each pixel cannot be larger than 255, it should be possible to write this function without worrying about overflow.

- `Image::print() const`
  Prints the label and intensity of the `Image`. We provide you with this function.

- `Image::display() const`
  Prints `Image` using the `print_pixel` helper function, which prints a block to the terminal that is a shade of grey determined. by the pixel value. We provide you with this function.

- `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 dimension of the two image does 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 dimension of the two image does not match, simply return `true`. Return `false` if both images are empty.

- `const int& Image::operator[]( int idx ) const`
  This overloads the subscript operator. It should just return the pixel value at the given index `idx` of the image without any boundary check. Note that this is different from `at` as `at` throws an exception when index is out-of-bound. Note that this is the "read" version since it is declared `const` and returns a `const` reference.

- `std::ostream& operator<<( std::ostream& os, const Image& image );`

Overloads the `<<` operator as a free function (not as a member function) to be able to display an `Image` using `std::cout`. Allows you to usee the `<<` operator to insert the imaege's intensity and label into the given `os` output stream. We provide you with this function.

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. Option 1: `Tree<T,CmpFunc>` Data Structure

Implement a generic tree data structure that stores data of type `T` whose ordering is established by `CmpFunc`. Each internal node in the binary search tree stores a value, a pointer to the left subtree, and a pointer to the right subtree. The value stored in any node should be greater or equal to any value stored in the left subtree and smaller than any value stored in the right subtree. Both subtrees should also recursively satisfy this property. You will need to define a nested struct or nested class which represents a node of the tree inside `Tree<T,CmpFunc>`. Your node should have the following fields: a field to store values of type `T`, a pointer to a node for the left subtree, and a pointer to a node for the right subtree. Note that this data structure is similar to a binary search tree, but with support for a find closest operation which does an exaustive search over a subtree of approximately $K$ "similar" items. You are responsible for implementing each of the following functions:

```
Tree<T,CmpFunc>::Tree( unsinged int k, CmpFunc cmp );

Tree<T,CmpFunc>::~Tree();
Tree<T,CmpFunc>::Tree( const Tree<T,CmpFunc>& tree );
Tree<T,CmpFunc>& Tree<T,CmpFunc>::operator=( const Tree<T,CmpFunc>& tree );

int       Tree<T,CmpFunc>::size() const;
void      Tree<T,CmpFunc>::add( const T& value );
bool      Tree<T,CmpFunc>::contains( const T& value ) const;
Vector<T> Tree<T,CmpFunc>::to_vector() const;

template <typename DistFunc>
T Tree<T,CmpFunc>::find_closest( const T& value, DistFunc dist ) const;

void Tree<T,CmpFunc>::print() const;
```

Here is a brief specification for each member function of `Tree<T,CmpFunc>` class:

- `Tree<T,CmpFunc>::Tree( int k, CmpFunc cmp )`
  Non-default constructor for `Tree<T,CmpFunc>`. Note that `k` is a parameter for `find_closest`, which corresponds to the number of elements for exhaustive search. `cmp` is the comparison function that this tree data structure should use to establish ordering between values of type `T`. This function constructs an empty `Tree<T,CmpFunc>` by initializing all data members, stores the value of `k`, and copies the comparator so that you can call it later.
- `Tree<T,CmpFunc>::~Tree()`
  Destructor for `Tree<T,CmpFunc>`. It frees the memory allocated on the heap by `Tree<T,CmpFunc>`. Note that you should use the `delete` and `delete[]` operator instead of `free` as in C. *Hint: You will likely need to use a private recursive helper function to implement this member function.*
- `Tree<T,CmpFunc>::Tree( const Tree<T,CmpFunc>& tree )`

Copy constructor that performs a deep copy. It should copy the values stored in `tree`. Subsequent actions on `tree` should have no side effect on the constructed `Tree<T,CmpFunc>`. *You must carefully handle the case when copying from an empty tree! Hint: You will likely need to use a private recursive helper function to implement this member function.*

- `Tree<T,CmpFunc>& Tree<T,CmpFunc>::operator=( const Tree<T,CmpFunc>& tree)`

  This overloads the assignment operator. Copy the values stored in `tree`. Note that if the current `Tree<T,CmpFunc>` 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 tree! Hint: You will likely need to use a private recursive helper function to implement this member function.*

- `int Tree<T,CmpFunc>::size() const`

  Return the current number of elements in the tree. If the `Tree<T,CmpFunc>` is empty, this function should return 0. *Hint: While you could use a private recursive helper function to implement this member function, you might want to consider using a constant time implementation.*

- `void Tree<T,CmpFunc>::add( const T& value )`

  Add a new node with the given value `value` to the `Tree<T,CmpFunc>`. The new node should be dynamically allocated on the heap. The binary search property for `Tree<T,CmpFunc>` should still hold after adding the new node. You will need to first traverse the tree to find the leaf node to which the new node should be added. We recommend you implement a recursive helper function to do so. If the value is already in the tree, this function should simply return and do nothing. Note that you should use the `new` operator instead of `malloc` as in C. *Hint: You will likely need to use a private recursive helper function to implement this member function.*

- `bool Tree<T,CmpFunc>::contains( const T& value ) const`

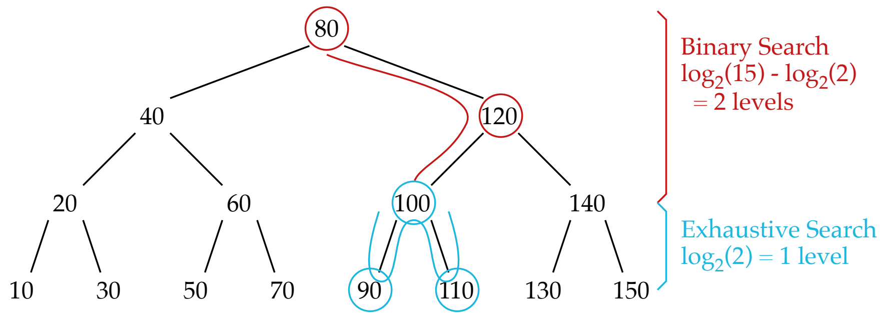
  Search the `Tree<T,CmpFunc>` for the given value and returns `true` if the value is found and `false` otherwise. If the `Tree<T,CmpFunc>` is empty, then this function should just return `false`. *Hint: You will likely need to use a private recursive helper function to implement this member function.*

- `Vector<T> Tree<T,CmpFunc>::to_vector() const`

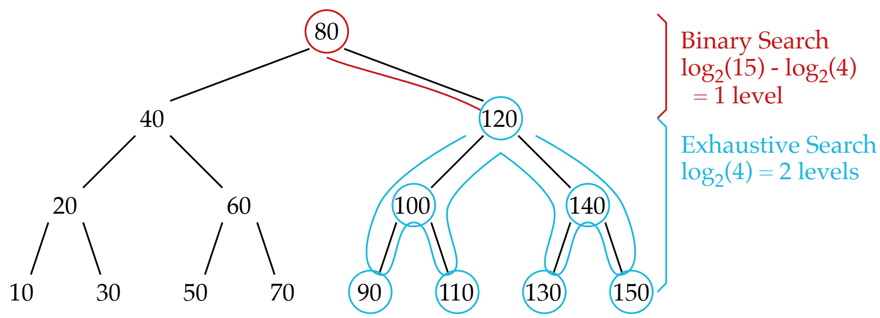
  Return a `Vector<T>` that contains all the items stored in the `Tree<T,CmpFunc>`. The `Vector<T>` should represent an in-order traversal of the tree (i.e., first add all values in the left subtree, then add the value of the current node, and finally add all values in the right subtree). *Hint: You will likely need to use a private recursive helper function to implement this member function. Avoid having this helper function return a `Vector<T>` since this can be slow. Consider having this helper function take a pointer to a `Vector<T>` as a parameter; simply push back on this single temporary `Vector<T>`.*

- `template <typename DistFunc>`
  `T Tree<T,CmpFunc>::find_closest( const T& value, DistFunc dist ) const`

  Search the `Tree<T,CmpFunc>` and return the value that has (approximately) the smallest difference (as defined by the `dist` distance function) from the given value. The distance function takes two inputs of type `T`, and returns the distance between them as an `int`, and has a function signature of `int dist(const T& a, const T& b)`. You should use a hybrid of binary search and exhaustive search (see Figure 2). You should perform a partial binary search to get down to a certain level in the tree and then perform an exhaustive search on the subtree. We recommend you first implement the exhaustive search by calling your recursive helper function from `Tree<T,CmpFunc>::to_vector` on the node you want to start your exhaustive search from, and then using `Vector<T>::find_closest_linear` on the vector that is returned. You will need to compute the number of levels for binary search and exhaustive search first. When doing so you can assume the tree is perfectly balanced, thus the total number of levels is $\log_2(N)$ where N




(a) Find Closest for 91 with K = 2




(b) Find Closest for 91 with K = 4

**Figure 2:** `Tree<T>::find_closest` **Example** – Example for balanced binary search tree with 15 integers. There are $\log_2(15)$ = 3 levels. (a) With K = 2, we do a binary search through $\log_2(15) - \log_2(2)$ = 2 levels, and then we do an exhaustive search through the remaining sub-tree which in this case has three nodes. (b) With K = 4, we do a binary search through $\log_2(15) - \log_2(4)$ = 1 level, and then we do an exhaustive search through the remaining sub-tree which in this case has seven nodes. The binary search uses intensity, while the exhaustive search uses Euclidean distance. Larger K improves accuracy while reducing performance.

is the number of elements in the tree. The number of levels for exhaustive search is $\log_2(K)$, and the number of levels for binary search is simply the total number of levels minus the number of levels for exhaustive search. We have provided you with a `ece2400::log2` function in `ece2400-stdlib.h`. If the `Tree<T,CmpFunc>` is empty, throw `ece2400::OutOfRange` with proper error message. You can assume no overflow will occur for all arithmetic operations in this function. *Hint: You will likely need to use one or more private recursive helper functions to implement this member function.* Note that this function is a template member function in a template class, so you need the following syntax in the function definition:

```
template <typename T, typename CmpFunc>
template <typename DistFunc>
T Tree<T,CmpFunc>::find_closest( const T& value, DistFunc dist ) const
{
  // ...
}
```

You need a distance function in order to call this member function. You can create a free function, functor, or lambda to calculate the distance. Distance for integers is the difference between the two values (make sure to return a positive number). Distance for images should be the Euclidean distance between the pixel arrays, just like the `Image::distance()` member function in Section 2.3. For the exhaustive search, start by calling your recursive helper function from Tree<T,CmpFunc>::to_vector. Once this is completely working, consider optimizations such as using a vector of pointers to images to avoid copying images; or implementing the exhaustive search directly as you traverse the tree without creating a temporary vector at all.

- void Tree<T,CmpFunc>::print() const
  Print the content in the Tree<T,CmpFunc>. This member 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. However, note that to make this member function generic you will need to use the C++ iostream library (e.g., std::cout) instead of printf. We have provided a commented-out version of our implementation of this function, which prints the tree with the root node on the left. You can leverage this implementation by replacing our references to the private member fields of the tree with the names of your private member fields.

### 2.5. Option 2: Table<T,HashFunc> Data Structure

You will implement a generic table data structure that stores data of type T. Internally, it stores data in a vector of vectors (i.e., Vector<Vector<T>>). Each Vector<T> corresponds to a bin that stores "similar" items. The hash function (type HashFunc) associated with the table data structure maps an object of type T to an integer in the range of 0 to INT_MAX. We can then map ranges of these integers to bins. Note that this data structure is similar to a hash table, but in a traditional hash table we only want a few items mapped to each bin (i.e., we *do not want* collisons). In this table data structure, we want *K* "similar" items to be mapped to a bin (i.e., we *do want* collisions of "similar" items). Then the table can support a find closest operation over a bin of *K* "similar" items. You are responsible for implementing each of the following functions:

```
Table<T,HashFunc>::Table( int k, HashFunc hash );

int       Table<T,HashFunc>::size() const;
void      Table<T,HashFunc>::add( const T& value );
bool      Table<T,HashFunc>::contains( const T& value ) const;
Vector<T> Table<T,HashFunc>::to_vector() const;

template <typename DistFunc>
T Table<T,HashFunc>::find_closest( const T& value, DistFunc dist ) const;

void Table<T,HashFunc>::print() const;
```

You do not need to implement the copy constructor, assignment operator, or the destructor because you will use your Vector<T> which already has those implemented. Here is a brief specification for each member function of Table<T,HashFunc> class:

- Table<T,HashFunc>::Table<T,HashFunc>( int k, HashFunc hash )
  Non-default constructor for Table<T,HashFunc>. This function constructs an empty Table<T,HashFunc> by initializing all data members and copies the hash function. The hash function hash should take a value of type T and hash it to an integer of type int in the range of 0 to INT_MAX. k corresponds to the load factor of the Table<T,HashFunc>. The load factor determines when you should increase the number of bins in your table. You should start with a single bin, and then every time the total number of elements in the table divided by the number of bins is greater than k, you should double the number of bins and rehash the table. For uniformly distributed data, k will roughly

correspond to the number of elements in each bin. Throw an `ece2400::InvalidArgument` if `k` is less than or equal to 0.

- `int Table<T,HashFunc>::size() const`

  Return the current number of elements in the table. If the `Table<T,HashFunc>` is empty, this function should return 0.

- `void Table<T,HashFunc>::add( const T& value )`

  Add the given value `value` to the `Table<T,HashFunc>`. You should first compute which bin the new value should be added based on using the hash function to map the given `value` to a positive integer, and then mapping ranges of integers to bins. After determing the bin for the new value, call `Vector<T>::push_back` on the appropriate `Vector<T>` to add the value to the table. Then, check the load factor of your table. If necessary, create a new `Vector<Vector<T>>` with twice as many bins as the current one and rehash the values in the current `Vector<Vector<T>>` into the new one.

- `bool Table<T,HashFunc>::contains( const T& value ) const`

  Search the `Table<T,HashFunc>` for the given value and returns `true` if the value is found and `false` otherwise. If the `Table<T,HashFunc>` is empty, then this function should just return `false`. You should first use the hash function to determine which bin will contain the given `value`, and then simply call `contains` on the corresponding `Vector<T>`.

- `Vector<T> Table<T,HashFunc>::to_vector() const`

  Return a `Vector<T>` that contains all the items stored in the `Table<T,HashFunc>`. The `Vector<T>` should add all items from the first bin before adding the items from the second bin.

- `template <typename DistFunc>`
  `T Table<T,HashFunc>::find_closest( const T& value, DistFunc dist) const`

  Search the `Table<T,HashFunc>` and returns the value that has (approximately) the smallest difference from the given value. You should first calculate which bin will contain the given `value`, and then simply call `find_closest` on the corresponding `Vector<T>`. If the corresponding `Vector<T>` is empty, then you should return a default-construced object of type `T`. If the `Table<T,HashFunc>` is empty, throw `ece2400::OutOfRange` with proper error message.

- `void Table<T,HashFunc>::print() const`

  Print the content in the `Table<T,HashFunc>`. This member 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. However, note that to make this member function generic you will need to use the C++ `iostream` library (e.g., `std::cout`) instead of `printf`. Consider printing the table such that it illustrates how the table is organized (i.e., print each bin a new line). We have provided you with a commented-out version of our implementation of this function that you can use to implement your own. You will need to replace the references to our private member fields with the names of your private member fields.

### 2.6. `HRSLinearSearch` Handwriting Recognition System

The first handwriting recognition system you will implement is `HRSLinearSearch`, which uses a brute force linear search algorithm. This system is almost identical to what you implemented in the previous programming assignment; the only difference is you should use `Vector<T>` instead of `VectorImage`. You are responsible for implementing each of the following functions:

```
HRSLinearSearch::HRSLinearSearch();
```

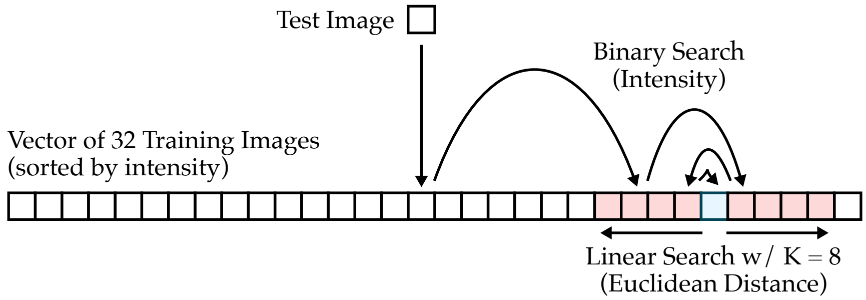



**Figure 3:** `VectorImage::find_closest_binary` **Example** – Example for 32 images and K = 8. Assumes vector images has already been sorted based on intensity. Binary search is based on intensity and quickly finds images with similar intensity as the test image. Linear search of 8 images is based on Euclidan distance and finds the closest match.

```
void HRSLinearSearch::train( const Vector<Image>& 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.
- `void HRSLinearSearch::train( const Vector<Image>& 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 `Vector<Image>::find_closest_linear` with a function object that calculates the Euclidean distance between images.

### 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 (see Figure 3). You are responsible for implementing each of the following functions:

```
HRSBinarySearch::HRSBinarySearch( int k = 1000 );
void HRSBinarySearch::train( const Vector<Image>& 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 Vector<Image>& 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 Vector<T>::sort with a function object that compares the intensity of two images.

- Image HRSBinarySearch::classify( const Image& img )

  Classify the given image (img). This function should use a binary and linear search. You should just call Vector<T>::find_closest_binary with a function object that calculates the Euclidean distance between images.

### 2.8. Option 1: HRSTreeSearch Handwriting Recognition System

The binary search system requires sorting the entire array during training. We can potentially improve the performance of sorting by incrementally sorting the training dataset as we add each training image using a binary search tree. Given this intuition, the third handwriting recognition system you will implement is HRSTreeSearch, which should use the Tree<Image,ImgCmpFunc> data structure implemented earlier in the programming assignment.

You are responsible for implementing each of the following functions:

```
HRSTreeSearch::HRSTreeSearch( int k = 1000 );
void HRSTreeSearch::train( const Vector<Image>& vec );
Image HRSTreeSearch::classify( const Image& Image );
```

Here is a brief specification for each member function of HRSTreeSearch:

- HRSTreeSearch::HRSTreeSearch( int k )

  Default constructor for HRSTreeSearch. Initialize the internal Tree<Image,ImgCmpFunc> data structure using the given k and a function that compares the intensity of images. We recommend declaring a functor as a private nested class inside HRSTreeSearch.

- void HRSTreeSearch::train( const Vector<Image>& vec )

  Train the HRS. For HRSTreeSearch, simply iterate through all training images (vec) and add each one to the internal Tree<Image,ImgCmpFunc> data structure.

- Image HRSTreeSearch::classify( const Image& img )

  Classify the given image. This function should simply call Tree<Image,ImgCmpFunc>::find_closest to return the image that has (approximately) smallest euclidean distance from the given image. You will need to pass in a function that calculates the Euclidean distance between images to Tree<Image,ImgCmpFunc>::find_closest.

Declaring a Tree type private data member in HRSTreeSearch requires the type of the image comparison function. We recommend using a functor declared as a private nested class inside HRSTreeSearch and declaring the data member like this:

```
Tree<Image,FunctorType> m_training_set;
```

### 2.9. Option 2: HRSTableSearch Handwriting Recognition System

Both the binary search and tree search systems rely on searching a smaller subset of the entire training dataset to improve performance. We can also use a hash table to group similar images together, and then do a linear search through a smaller subet of similar images to improve performance. Given this intuition, the fourth handwriting recognition system you will implement is HRSTableSearch, which should use the `Table<Image,ImgHashFunc>` data structure implemented earlier in the programming assignment. You are responsible for implementing each of the following functions:

```
HRSTableSearch::HRSTableSearch( int k = 1000 );
void HRSTableSearch::train( const Vector<Image>& vec );
Image HRSTableSearch::classify( const Image& Image );
```

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

- `HRSTableSearch::HRSTableSearch( int k )`
  Default constructor for `HRSTableSearch`. Initialize the internal `Table<Image,ImgHashFunc>` data structure using the given `k` and a function that generates hash values for images. We recommend declaring a functor as a private nested class inside `HRSTableSearch`. The hash function you should implement is:

  $$((\text{intensity} - 5,000)\%20,000) * 100,000$$

  This function distributes the images in our MNIST dataset into an approximately uniform distribution with values between 0 and `INT_MAX`. For more details on the hash function, see the end of this section.

- `void HRSTableSearch::train( const Vector<Image>& vec )`
  Train the HRS. For `HRSTableSearch`, simply iterate through all training images (`vec`) and add each one to the internal `Table<Image,ImgHashFunc>` data structure.

- `Image HRSTableSearch::classify( const Image& img )`
  Classify the given image. This function should simply call `Table<Image,ImgHashFunc>::find_closest` to return the image that has (approximately) smallest euclidean distance from the given image. You will need to pass in a function that calculates the Euclidean distance between images to `Table<Image,ImgHashFunc>::find_closest`.

Declaring a `Table` type private data member in `HRSTableSearch` requires the type of the image hash function. We recommend using a functor declared as a private nested class inside `HRSTableSearch` and declaring the data member like this:

```
Table<Image,FunctorType> m_training_set;
```

**Hash Function** – Hash function design is an important part of developing an application using a table data structure. Different applications require different behaviors out of their hash functions. In our application, we want the bins in our table to store many images of similar intensity so that we can search through all of them for the closest match to the image we are trying to classify. We would also like to have the bins have an approximately equal number of items in them. This will make execution time consistent no matter what bin we are searching and ensure each classification looks at about the same number of images to find the closest match. Because we know something about the kind of data we will be storing in our table ahead of time, we can design a hash function to fit these criteria. The intensities of the images in the MNIST dataset are distributed like this:

```
  4800 -   6400
  6400 -   8000 #
  8000 -   9600 ####
  9600 -  11200 ##########
 11200 -  12800 ###############
 12800 -  14400 #####################
 14400 -  16000 ###########################
 16000 -  17600 #################################
 17600 -  19200 #########################################
 19200 -  20800 ############################################
 20800 -  22400 #################################################
 22400 -  24000 ###################################################
 24000 -  25600 ###################################################
 25600 -  27200 ###################################################
 27200 -  28800 ################################################
 28800 -  30400 ############################################
 30400 -  32000 ######################################
 32000 -  33600 ################################
 33600 -  35200 ############################
 35200 -  36800 #######################
 36800 -  38400 ###################
 38400 -  40000 ###############
 40000 -  41600 ############
 41600 -  43200 #########
 43200 -  44800 ######
 44800 -  46400 #####
 46400 -  48000 ###
 48000 -  49600 ##
 49600 -  51200 ##
 51200 -  52800 #
 52800 -  54400 #
 54400 -  56000
```

The minimum hash value is 5,086 and the maximum hash value is 79,483. These are not pictured on the histogram above because each symbol represents 100 intensity values in that range.

Our hash function tries to change this bell-shaped distribution with values between 5,000 and 80,000 and a peak at about 25,000 into an approximately uniform distribution with values between 0 and INT_MAX. The first thing our function does is to subtract 5,000 from the intensity. This will slide the bell curve towards 0, so that its values range from 0 to 75,000 with a peak around 20,000. Next, we use the remainder operator with a numerator of 20,000. This slides the upper half of the bell down to 0, and confines all values to the range 0 to 20,000. We now have a roughly normal distribution of values between 0 and 20,000, so to stretch it out and take full advantage of the range from 0 to INT_MAX we multiply by 100,000. The histogram of the hashes of all the values in the MNIST dataset using our hash function looks like this:

```
         0 -   85899345 ##############################
  85899345 -  171798690 ##############################
 171798690 -  257698035 #############################
 257698035 -  343597380 ###########################
 343597380 -  429496725 #############################
 429496725 -  515396070 #############################
 515396070 -  601295415 ############################
 601295415 -  687194760 ############################
 687194760 -  773094105 ###########################
 773094105 -  858993450 #############################
 858993450 -  944892795 ###########################
 944892795 - 1030792140 ##############################
1030792140 - 1116691485 ############################
1116691485 - 1202590830 ##############################
1202590830 - 1288490175 ###############################
1288490175 - 1374389520 ###############################
1374389520 - 1460288865 ##############################
1460288865 - 1546188210 ###############################
1546188210 - 1632087555 ###############################
1632087555 - 1717986900 ################################
1717986900 - 1803886245 ################################
1803886245 - 1889785590 ###############################
1889785590 - 1975684935 ##############################
1975684935 - 2061584280 #########
```

You should feel free to experiment with your own hash function as a potential optimization to improve accuracy. If you do, make sure to be careful to avoid overflow and generating negative hash values!

## 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-adhoc.cc`, `src/vector-adhoc.cc`, `src/image-adhoc.cc`, `src/tree-adhoc.cc`, and `src/table-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 program for `sort` like this:

```
% cd ${HOME}/ece2400/pair-xx/pa5-sys
% scripts/build.sh
% cd build/src
```

```
% ./sort-adhoc
```

### 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 you realize further tests are needed (maybe just by copying and adjusting existing tests), then you should definitely write a few more.

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<T,CmpFunc>
- Step 2. Implement and test Vector<T>
  - 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 2e. Implement and test sort
  - Step 2d. 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 operator[]
  - Step 3c. Implement and test set_label, get_label
  - Step 3d. Implement and test get_intensity
  - Step 3e. Implement and test operator==, operator!=
- Step 4a. Implement and test Tree<T,CmpFunc>
  - Step 4aa. Implement and test the default constructor, destructor, add, size, contains
  - Step 4ab. Implement and test to_vector
  - Step 4ac. Implement and test find_closest
  - Step 4ad. Implement and test the copy constructor
  - Step 4ae. Implement and test the assignment operator
- Step 4b. Implement and test Table<T,HashFunc>
  - Step 4ba. Implement and test the default constructor, destructor, add, size, contains
  - Step 4bb. Implement and test to_vector
  - Step 4bc. Implement and test find_closest

- Step 5. Implement and test `HRSLinearSearch`
- Step 6. Implement and test `HRSBinarySearch`
- Step 7a. Implement and test `HRSTreeSearch`
- Step 7b. Implement and test `HRSTableSearch`

Each substep should correspond to a directed test case. Note that for some of these steps we provide both *generic* and *specialized* tests. For example, if you look in `sort-directed-test.h` you will see generic tests which are templated on the type `T` and the comparison function `CmpFunc`. This enables us to reuse the exact same test functions to test sorting an array of `ints` and to test sorting an array of `Images`. You will want build and run the specialized test programs (i.e., `sort-int-directed-tests` and `sort-image-directed-tests`).

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_`

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

```
% cd ${HOME}/ece2400/pair-xx/pa5-sys
% scripts/test.sh

Build Successful
Running tests...
<blah blah blah>
Tests failed
```

(Your tests will all fail initially.)

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/pair-xx/pa5-sys/build/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/pair-xx/pa5-sys/build/test
% ./sort-int-directed-test 1

test_case_one_element
<blah blah blah>
```

Your implementation of `find_closest` for both `Tree<T,CmpFunc>` and `Table<T,HashFunc>` may be slightly different than the staff implementation. These differences may not significantly impact the overall accuracy of your handwriting recognition systems, but might cause you to fail one or two of the test assertions in the directed testing. In this case, it is acceptable for you to carefully update the directed testing to better match your specific implementation. You might also consider adding some random testing for `find_closest` for both `Tree<T,CmpFunc>` and `Table<T,HashFunc>`.

### 3.3. Memory Leaks

Students are 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.

Check memory leaks and other dynamic memory allocation issues like this:

```
% cd ${HOME}/ece2400/pair-xx/pa5-sys
% scripts/memcheck.sh
```

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

```
% cd ${HOME}/ece2400/pair-xx/pa5-sys
% scripts/valgrind.sh build/test/sort-int-directed-test
```

valgrind.sh calls Valgrind with correct command line options so you don't need to remember them.

### 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/pair-xx/pa5-sys
% scripts/coverage.sh
```

The script will clean up any previous coverage data, create a fresh `build-coverage directory`, compile the project with code coverage flags, run the tests, and generate coverage reports. The coverage reports for your sorting implementations can be found at `build-coverage/*.cc.gcov`. Unexecuted lines are marked #####. Lines marked with * contain some unexecuted basic blocks.

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 use code coverage as a way to guide their test-driven design than to obsess over specific code coverage numbers.

## 4. Evaluation

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 the evaluation programs like this:

```
% cd ${HOME}/ece2400/pair-xx/pa5-sys
% ./scripts/eval.sh
```

Take a peek inside `eval.sh` to see what it does! Notice that it calls for a "release" build (as opposed to a "debug" build which is slower) and then calls two evaluation programs: `hrs-linear-search-eval` and `hrs-binary-search-eval`. You can specify the size of the number of training images `N`, the number of classification images `M`, and the size of the final linear search when using binary search `K` on the command line for each evaluation program:

```
% cd ${HOME}/ece2400/pair-xx/pa5-sys/build/eval
% ./hrs-linear-search-eval N M
% ./hrs-binary-search-eval N M K
```

See the complete usage text by forcing an error message:

```
  % ./hrs-linear-search-eval -help
or
  % ./hrs-binary-search-eval -help
```

Finally, note that using a smaller number of training images and/or classification images is only for profiling and interactive performance optimization. All accuracy results must use the full 60K training dataset and 10K classification dataset.

You should leverage your insights from the previous programming assignment to apply similar optimizations that you feel are worthwhile to these handwriting recognition systems. You may need to do some performance profiling to figure out where the bottleneck is and what to optimize. You can 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/pair-xx/pa5-sys/build/eval
% perf record --call-graph dwarf ./hrs-linear-search-eval
% perf script report stackcollapse | flamegraph.pl > graph-linear.svg
```

While you are encouraged to use flame graphs to guide your optimization, you do not need to include them in your report unless you think it would be useful in your quantitative evaluation section. Unlike the previous programming assignment, you do not need to record incremental performance after every optimization. You should focus on analyzing the final optimized performance results in your report. You should optimize the linear and binary search systems so that they can achieve the performance targets used in the previous programming assignment.

## 5. Putting It All Together

Recall that complete systems often include a frontend written in a productivity-level language (e.g., Python) and a backend written in an efficiency-level language (e.g., C/C++). We have provided you a Python GUI frontend which allows you to classify real handwritten digits using one of the five available backends (`HRSLinearSearch`, `HRSBinarySearch`, `HRSTreeSearch` or `HRSTableSearch`. To use this Python GUI frontend, first make sure that you have built the `hrs-backend` target in the build system (preferably an evaluation build since it will be faster!):

```
% cd ${HOME}/ece2400/pair-xx/pa5-sys
% scripts/eval.sh
```

Copy the compiled backend program to the evaluation directory:

```
% cd ${HOME}/ece2400/pair-xx/pa5-sys
% cp build-eval/hrs-backend eval/hrs-backend
```

Then launch the GUI frontend (`hrs-frontend.py`) using Python:

```
% cd ${HOME}/ece2400/pair-xx/pa5-sys/eval
% python hrs-frontend.py
```

The frontend only works if you use a remote access option that supports Linux applications with a GUI. If you are using VS Code and you want to experiment with the GUI, then you will need to use either X2Go, MobaXterm, or Mac Terminal with XQuartz.

## 6. Milestone and Report

This section includes critical information about the incremental milestone, final code submission, and the final report specific to this PA.

### 6.1. 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<T,CmpFunc>`
- Complete the implementation of `Vector<T>`
- Complete the implementation of `Image`
- Pass all given directed and random tests for these implementations
- Consider adding a few of your own directed tests

### 6.2. 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/pair-xx/pa5-sys
% ./scripts/format.sh
% 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.

#### 6.3. 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. Your entire report must be **no more than seven pages**. You will have to use this Overleaf template to generate your pdf:

`https://tinyurl.com/2400-sp25-pa5temp`

## 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.