Sorting
Due Monday, February 9, 2015, at 9pm

goal

In this project you will implement mergesort in C++.

Sorting algorithms are generic—you need to be able to compare keys and swap entries, but the basic form of the algorithm is the same whether we are sorting `long int` arrays or `double` arrays or `unsigned char` arrays. There are several ways you could implement generic sorting algorithms.

You could, for instance, doing something kludgy:

1. You could clone your code and make the necessary changes for different data types. However, this rapidly becomes a nightmare to support—if you find a bug or add a feature in the mergesort for `double`, then you need to do the same in your mergesort for `float`, `unsigned char`, &c.

2. You could have a single set of procedures, and somehow automagically edit the source as needed to create sorting procedures for other types, say, using `sed` scripts.

There are several better ways you can write generic algorithms in C++:

1. You could write your code so that it is agnostic about the type of data being sorted. This is what the C library sorting routines (e.g., `qsort()`) do. The user supplies a function to compare whatever is being sorted and the size of the objects being sorted. The library code calls the user-supplied comparison function, and rearranges objects during the sort by the library code since it knows the size of the objects it needs to move.

   This moves the burden of supplying type-specific code to the user. The user now potentially needs to keep track of a multitude of different comparison functions. When we look at C++ classes in later projects we will see how C++ simplifies things using *virtual functions*. This is a form of *polymorphism*, which refers in general to approaches for specifying a common interface to a wide variety of different objects.

2. You could use *function templates*, another form of polymorphism. The term *generic programming* is also sometimes applied to C++ function templates.

   Function templates allow you to write code with variables of a generic type rather than a specific type such as `int`. You then instantiate specific instances of the code by giving specific types.

3. You could attach the sort function to the particular type of data being sorted. We will see this approach in a later project when we introduce C++ classes.

where to start

You are to use the function prototypes in sort.hpp. These may not be modified. You may find the sample test driver main.cpp and the functions in sort_utils.cpp helpful. A sample makefile is given in Makefile. If you want to compile and link by hand rather than use `make`, the incantation

```
g++ -o snort -g -std=c++11 -Wall -pedantic foo.cpp bar.cpp
```

would compile and link the code in `foo.cpp` and `bar.cpp` and create a binary executable named `snort` (or possibly `snort.exe` on a Windows machine).

function templates

Here is insertion sort written as a template function, also using C++ vectors, which are templated:

```
template <typename T>
void insertion_sort (vector<T> &a) {
  for (unsigned long int i = 1; i < a.size(); i++) {
    for (unsigned long int j = i; (j > 0) && (a[j] < a[j-1]); j--) {
      std::swap(a[j], a[j-1]);
    }
  }
}
```

This is a template for a generic type T. This code will work for any type T for which the statement

```
  a[j] < a[j-1]
```

makes sense. This includes integers, floating point numbers, and C++ strings, but also C++ objects for which we have defined a < operator.

This is code is just a template, however. To instantiate a particular type, we need to specify the type T. If we want to apply insertion sort to a vector of double precision numbers, for instance, we can bring the appropriate function into being by adding

```
template void insertion_sort<double>(vector<double>&);
```

at the end of `sort_utils.cpp`. (This is how it works for `g++`. Instantiation of template functions is a complicated matter, and depends on the compiler being used. For more information on this topic, see the g++ documentation.)

What you are to do

public API

You should implement recursive mergesort and iterative mergesort with the following signatures:

- template <typename T> void recursive_mergesort (vector<T> &a)
- template <typename T> void iterative_mergesort (vector<T> &a)

In each of these calls, `a` refers to the vector being sorted.

Function prototypes are given in sort.hpp. You should include this header file in your code using a `#include` statement.

under the hood

For the auxiliary storage in

- template <typename T> void recursive_mergesort (vector<T> &a),
- template <typename T> void iterative_mergesort (vector<T> &a),

you should use an appropriate vector type. You should allocate a auxiliary space only once per user call to either of these functions. (If you allocate auxiliary space at every level of the recursion in recursive mergesort you will incur a tremendous amount of unnecessary overhead that will make your code run slowly for large vectors.)

what to turn in, and how

Submit the code in the Project01 Google Drive folder. Place your code in a file named `sort.cpp`, but `sort.cpp` must NOT include a `main()`. If I have a test program named `main.cpp` that calls both mergesort routines, then your `sort.cpp` must be such that

```
g++ -o snort -g -std=c++11 -Wall -pedantic main.cpp sort.cpp
```

will successfully build an executable.

grading

No credit will be given for

- code that does not compile and link,
- code that opens any files,
- code that includes an active `main()`.

Here are other problems that will have a negative effect on your grade.

- Your program crashes during testing.
- Code that appears to work but nevertheless has memory access errors.
- Code that is incorrect.
- Memory leaks.
- Code that is unnecessarily inefficient.
- Code that spews output to stdout or stderr. Be sure to disable all debugging i/o!!

tips

- Take a look at the code in the text. It will give you pretty good guidance on the details of what you need to do.
- You will need at least on additional function besides those specified in the API (i.e., a merge routine) to implement the algorithms.
- Be sure to test your code on arrays that are large enough to show the runtime (e.g., $n \geq 1,000,000$). A reasonable figure to sort $10,000,000$ ints on one of the CS lab machines would be
  - less than 2 seconds, if you enable optimization when you compile (the `-O` option);
  - less than 9 seconds, if you do not turn enable optimization.

  If your code takes a lot longer than this, you may have a bug.
- Be sure to check your code with valgrind! It will detect memory access errors that otherwise might be invisible, and make debugging a LOT easier.
- Be sure that not only is the array sorted upon return, but that it's the same array. A common error is to return values that don't correspond to the original inputs. (I.e., if we return an array of all zeros, it's sorted, but it's probably not the data we wanted to sort.)