CISS350: Data Structures and Advanced Algorithms
Assignment a05

Name: jdoe5@cougars.ccis.edu Score:

Open `main.tex` and enter answers (look for `answercode`, `answerbox`, `answerlong`). Turn the page for detailed instructions. To rebuild and view pdf, in bash shell execute `make`. To build a gzip-tar file, in bash shell execute `make s` and you'll get `submit.tar.gz`.

Objectives

1. Implement comparison–based sorting algorithm
2. Collecting runtime data

Submission

Here's how you organize your submission:

```
...
  a
  |
  +-- a05
      |
      +-- main.tex (and other LaTeX files)
      |
      +-- a05q01
      |   |
      |   +-- main.cpp
      |
      +-- a05q02
          |
          +-- main.cpp
```

When you're done, go to `ciss350/a/` create `a05.tar.gz` (or `submit.tar.gz`) that holds the `a05` directory as usual.

Comamnd-line arguments

Compile this program say the executable is `main.exe`:

```
#include <iostream>

int main(int argc, char * argv[])
{
    for (int i = 0; i < argc; ++i)
    {
        std::cout << i << " ... " << argv[i] << '\n';
    }
    return 0;
}
```

Now in your shell run the program this way:

```
[student@localhost cpp-cmd-line-args]$ g++ main.cpp -o main.exe
[student@localhost cpp-cmd-line-args]$ ./main.exe abc def hello
0 ... ./main.exe
1 ... abc
2 ... def
3 ... hello
```

So you see the following: when you issue a command in your bash shell to run the program `./main.exe` and a sequence of strings, i.e., `abc`, `def`, `hello`, The strings `"./main.exe"`, `"abc"`, `"def"`, and `"hello"` are stored in the array of C-string `argv` (the "argument vector" where in this case vector means array). Furthermore the number of such strings (an integer) is stored in the integer variable `argc` (the "argument count"). Note that the first C-string, `argv[0]`, is the program path.

By the way, the type of `argv` is an array of pointers. If you think about that carefully it means that the following works too:

```
#include <iostream>

int main(int argc, char ** argv)
{
    ...
}
```

Converting string to integer

There are many, many, many ways to compute an integer from a numeric string. Here's one. Run this program:

```
#include <iostream>
#include <sstream>

int main()
{
    char s[100] = "42";
    std::istringstream cin(s);
    int i;
    cin >> i;
    std::cout << i << ' ' << i + 1 << '\n';
    return 0;
}
```

Here's the output:

```
[student@localhost questions] g++ test.cpp; ./a.out
42 43
```

You should write a function for the above because you'll be using it.

Use the web to find out more about the input string stream class.

Note: You're a CS student. You should still be able to write a string-to-integer function on your own in 15 minutes "from scratch" without using library code like the above.

Functor

Here's a quick review of functor. See CISS245 for details,

A functor is an object with the function call operator `operator()`. Here's an example:

```
#include <iostream>

class F
{
public:
    int operator()(int x) const
    {
        return x * x;
    }
};

int main()
{
    F f;
    std::cout << f(2) << '\n';
    return 0;
}
```

The above functor class has a `operator()` that accepts one argument.

You can have any number of arguments. Here's one with two parameters.

```
#include <iostream>

class Less
{
public:
    bool operator()(int x, int y) const
    {
        return (x < y);
    }
};

int main()
{
    Less less;
    std::cout << less(2, 3) << '\n';
    return 0;
}
```

The `Less` is sometimes called a comparator class while `less` is sometimes called a comparator.

Of course you can make the above into a template:

```
#include <iostream>

template < typename T >
class Less
{
public:
    bool operator()(const T & x, const T & y) const
    {
        return (x < y);
    }
};

int main()
{
    Less< int > less;
    std::cout << less(2, 3) << '\n';
    return 0;
}
```

Common interface

All the sorting functions in this section have a common interface. If `x` is an array or a `std::vector< int >` object, then

```
bubblesort(&x[2], &x[6]);
```

will perform bubblesort on the values of `x[2..5]`. The sorting is done in ascending order. You can also execute

```
bubblesort(&x[2], &x[6], less);
```

where `less` is a comparator functor that specifies the order of the sorting. Specifically, `less(a, b)` is true is either `a` is the same as `b` or `a` should appear before `b` in the array that is sorted. For both prototypes above, there's an extra boolean parameter that specifies if the function should display the progress of the sorting process. This boolean parameter has a default value of `false`. So calling

```
bubblesort(&x[2], &x[6], true);
```

will result in displaying the bubblesorting process. For instance you might see this:

```
{1, 9, 4, 2}
{1, 4, 2, 9}
{1, 2, 4, 9}
{1, 2, 4, 9}
```

Here's an example on sorting two integers where the sorting uses a comparator functor for determining the order of sorting. Study this carefully. Modification to the bubblesort using a functor for comparison will be easy using this example as a guide.

```
// file: sort2.cpp

#include <iostream>

template < typename T >
class Less
{
public:
    bool operator()(const T & x, const T & y) const
    {
        return (x < y);
    }
};

template < typename T >
class Greater
{
public:
    bool operator()(const T & x, const T & y) const
```

```
    {
        return (x > y);
    }
};

template < typename T >
void sort2(T * p, T * q, bool verbose=false) // sort in ascending
{
    if (verbose)
    {
        std::cout << *p << ' ' << *q << '\n';
    }
    if (*q < *p)
    {
        T t = *p;
        *p = *q;
        *q = t;
    }
    if (verbose)
    {
        std::cout << *p << ' ' << *q << '\n';
    }
}

template < typename T, typename C >
void sort2(T * p, T * q, const C & less, bool verbose=false) // sort based on less
{
    if (verbose)
    {
        std::cout << "verbose on ... " << *p << ' ' << *q << '\n';
    }
    if (less(*q, *p))
    {
        T t = *p;
        *p = *q;
        *q = t;
    }
    if (verbose)
    {
        std::cout << "verbose on ... " << *p << ' ' << *q << '\n';
    }
}

int main()
{
    int x[] = {1, 0};
    std::cout << '\n';
    sort2(&x[0], &x[1]);
    std::cout << x[0] << ' ' << x[1] << '\n';

    int y[] = {1, 0};
```

```
    std::cout << '\n';
    sort2(&y[0], &y[1], Less< int >());
    std::cout << x[0] << ' ' << x[1] << '\n';

    int z[] = {0, 1};
    std::cout << '\n';
    sort2(&z[0], &z[1], Greater< int >(), true);

    return 0;
}
```

I already mentioned in CISS245, when writing a template (class or function or struct), it's usually a good idea to start off with a non-template version.

C++ lambda expressions

The above section is enough. This section is optional, but you should probably study it.

In the previous section, the ordering of the sorting process is determined by a comparator functor. The functor is an object. So you need to write a class for any comparator functor. There's another way to achieve the same thing, but with something called a lambda expression. (Details about functional programming, lambda functions, etc. is covered in CISS445.)

```
// file: sort2.cpp

#include <iostream>

template < typename T >
void sort2(T * p, T * q, bool verbose=false) // sort in ascending
{
    if (verbose)
    {
        std::cout << *p << ' ' << *q << '\n';
    }
    if (*q < *p)
    {
        T t = *p;
        *p = *q;
        *q = t;
    }
    if (verbose)
    {
        std::cout << *p << ' ' << *q << '\n';
    }
}

template < typename T, typename C >
void sort2(T * p, T * q, const C & less, bool verbose=false) // sort based on less
{
    if (verbose)
    {
        std::cout << "verbose on ... " << *p << ' ' << *q << '\n';
    }
    if (less(*q, *p))
    {
        T t = *p;
        *p = *q;
        *q = t;
    }
    if (verbose)
    {
```

```
        std::cout << "verbose on ... " << *p << ' ' << *q << '\n';
    }
}

int main()
{
    int x[] = {1, 0};
    std::cout << '\n';
    sort2(&x[0], &x[1]);
    std::cout << x[0] << ' ' << x[1] << '\n';

    int y[] = {1, 0};
    std::cout << '\n';
    sort2(&y[0], &y[1], [](int x, int y){ return x < y; });
    std::cout << x[0] << ' ' << x[1] << '\n';

    int z[] = {0, 1};
    std::cout << '\n';
    sort2(&z[0], &z[1], [](int x, int y){ return x > y; }, true);

    return 0;
}
```

The

```
[](int x, int y){ return x < y; }
```

and the

```
[](int x, int y){ return x > y; }
```

are C++ lambda expressions. You can think of them as anonymous functions, i.e., functions without names.

Compare the above with the code from the previous section and you should be able to figure out what the lambda expressions achieve.

Skeleton for all code

For each question, the following skeleton must be used (every implementation accepts a vector parameter and two other parameters – refer to questions for details):

```
// File: skeleton.cpp

#include <iostream>
#include <cstdlib>
#include <vector>

// Your implementation of the sorting algorithm

int main(int argc, char * argv[])
{
    //------------------------------------------------------------------------
    // Seed the random generator. argv[1] is a numeric string for the seed of
    // the random generator.
    // Convert argv[1] to integer value and assign to seed.
    //------------------------------------------------------------------------
    int seed = 0;
    // Assign to seed the integer value from the numeric string argv[1]
    srand(seed);

    //------------------------------------------------------------------------
    // Create a vector x of n random values in the range 0..m.
    // n is the integer from the numeric string argv[2]
    // m is the integer from the numeric string argv[3]
    //------------------------------------------------------------------------
    int n = 0;
    // Assign to n the integer value from the numeric string argv[2]
    std::vector< int > x(n, 0); // int vector of size n, all zeros

    int m = 0;
    // Assign to m the integer value from the numeric string argv[3]

    //------------------------------------------------------------------------
    // k is the integer from the numeric string argv[4]
    //------------------------------------------------------------------------
    int k = 0;
    // Assign to k the integer value from the numeric string argv[4]

    //------------------------------------------------------------------------
    // Call your sorting algorithm to sort x a total of k times and in
    // ascending order and verbose=true.
    //------------------------------------------------------------------------
    double total_time = 0.0;
    for (int i = 0; i < k; ++i)
    {
        // Randomize vector x
```

```
        for (int i = 0; i < n; ++i)
        {
            x[i] = rand() % m;
        }

        // start timer
        bubblesort(&x[0], &x[n], true);
        // end timer
        // total_time += time for this sort
    }

    //-----------------------------------------------------------------------
    // Print the average runtime.
    // p is the integer from the numeric string argv[5]
    // If p is 1, print the average runtime.
    // If p is 0, the average runtime is not printed.
    //-----------------------------------------------------------------------
    int p = 0;
    // Assign to p the integer value from the numeric string argv[5]
    // If p is 1, print the average runtime.

    return 0;
}
```

For instance for Q1, if you run

```
./a.out 42 1000000 100 5 1
```

then you are seeding the random generator with 42, sorting a vector of 1000000 integer values randomly chosen in the range 0..99 a total of 5 times and printing the average runtime of sorting algorithm. The average time in this case is the total time for the 5 runs divided by 5.

Q1. [Bubblesort]

Implement the standard bubblesort algorithm in C++. The following are the prototypes:

```
template < typename T >
void bubblesort(T * start, T * end, bool verbose=false);

template < typename T, typename C >
void bubblesort(T * start, T * end, const C & less, bool verbose=false);
```

The `verbose` parameter, when `true`, will display the values at address `start` up to but not including `end`, and also at the *end* of each pass with the values separated by `", "` and a newline is printed after printing each pass. Actually I will only test your code for the first prototype, where sorting is done in the default ascending order. So the second prototype is optional.

Here's an example execution:

```
./a.out 42 4 10 1 0
{1, 9, 4, 2}
{1, 4, 2, 9}
{1, 2, 4, 9}
{1, 2, 4, 9}
```

The first output `{1, 9, 4, 2}` is the randomize vector with 4 values (unsorted), the second output is the vector at the end of the first pass, the third output is the vector at the end of the second pass, and the last output is the vector at the end of the third pass.

Make sure your output matches the format *exactly* or your implementations will be considered incorrect.

I recommend the following steps:

1. Write a bubblesort that accepts pointers to two integers.
2. Add verbose printing.
3. Add functor for specifying sort order.
4. Convert to function template.

Q2. [Bubblesort variant 1]

Modify the bubblesort so that the algorithm returns once no swap is performed during a pass. Call the function `bubblesort2`. The prototypes are similar to `bubblesort`:

```
template < typename T >
void bubblesort2(T * start, T * end, bool verbose=false);

template < typename T, typename C >
void bubblesort2(T * start, T * end, const C & less, bool verbose=false);
```

The following is a sample execution:

```
./a.out 42 4 10 1 0
{1, 9, 4, 2}
{1, 4, 2, 9}
{1, 2, 4, 9}
```

Notice that the above went through *two* passes and stops.

Q3. [Selection sort]

Implement the standard selection sort algorithm in C++. The following is the prototype:

```
template < typename T >
void selectionsort(T * start, T * end, bool verbose=false);

template < typename T, typename C >
void selectionsort(T * start, T * end, const C & less, bool verbose=false);
```

The following is a sample execution:

```
./a.out 42 4 10 1 0
{1, 9, 4, 2}
{1, 9, 4, 2}
{1, 2, 4, 9}
{1, 2, 4, 9}
```

Q4. [Insertion sort]

Implement the standard insertion sort algorithm in C++. The following is the prototype:

```
template < typename T >
void insertionsort(T * start, T * end, bool verbose=false);

template < typename T, typename C >
void insertionsort(T * start, T * end, const C & less, bool verbose=false);
```

The following is a sample execution:

```
./a.out 42 4 10 1 0
{1, 9, 4, 2}
{1, 9, 4, 2}
{1, 4, 9, 2}
{1, 2, 4, 9}
```

Pretty-print Recursion

For the verbose output, it's particularly important to know how to display things in a nice format to help with debugging. Because of the nature of function calling for recursive functions, if you simply print everything straight to the stdout, things will be extremely hard to read. So let me show you a method that I use for verbose printing for debugging recursive functions. First here's the standard Fibonacci sequence. Run it:

```
#include <iostream>

int f(int n)
{
    if (n < 2)
    {
        return 1;
    }
    else
    {
        return f(n - 1) + f(n - 2);
    }
}

int main()
{
    std::cout << "VALUE:" << f(3) << '\n';
    return 0;
}
```

It's simple enough that you won't get it wrong. But suppose for debugging you want to show when you first enter the function and the output from each function. Run this:

```
#include <iostream>

int f(int n)
{
    std::cout << "f(" << n << ") enter ...\n";
    if (n < 2)
    {
        int ret = 1;
        std::cout << "f(" << n << ") = " << ret << '\n';
        return ret;
    }
    else
    {
        int ret = f(n - 1) + f(n - 2);
        std::cout << "f(" << n << ") = " << ret << '\n';
```

```
        return ret;
    }
}

int main()
{
    std::cout << "VALUE:" << f(3) << '\n';
    return 0;
}
```

You'll see that it's rather confusing even for such a small recursive function. Finally run this:

```
// File: pretty-print-recursion.cpp

#include <iostream>
#include <iomanip>

int f(int n)
{
    static int spaces = -4;
    spaces += 4;
    std::cout << std::setw(spaces) << "";
    std::cout << "f(" << n << ") enter ...\n";
    if (n < 2)
    {
        int ret = 1;
        std::cout << std::setw(spaces) << "";
        std::cout << "f(" << n << ") = " << ret << '\n';
        spaces -= 4;
        return ret;
    }
    else
    {
        int ret = f(n - 1) + f(n - 2);
        std::cout << std::setw(spaces) << "";
        std::cout << "f(" << n << ") = " << ret << '\n';
        spaces -= 4;
        return ret;
    }
}

int main()
{
    std::cout << "VALUE:" << f(3) << '\n';
    return 0;
}
```

Here's the output

```
f(3) enter ...
    f(2) enter ...
        f(1) enter ...
        f(1) = 1
        f(0) enter ...
        f(0) = 1
    f(2) = 2
    f(1) enter ...
    f(1) = 1
f(3) = 3
VALUE:3
```

Study the output and you'll see the point. Make sure you study the above code – it's very important because I don't see this method of debug-printing for recursion ever mentioned in beginner's C++ books. Note that I've used a static `int` which you should review if you don't recall the point of a static variable. You will find the above method extremely useful in the future whenever you have to debug recursion.

Q5. [Mergesort]

Implement the mergesort algorithm in C++. The following is the prototype:

```
template < typename T >
void mergesort(T * start, T * end, bool verbose=false);

template < typename T, typename C >
void mergesort(T * start, T * end, const C & less, bool verbose=false);
```

You should probably have the following functions:

```
template < typename T >
void mergesort_(T * start, T * end, std::vector< T > & t,
                bool verbose=false);

template < typename T, typename C >
void mergesort_(T * start, T * end, std::vector< T > & t,
                const C & less, bool verbose=false);

template < typename T >
void merge(T * start0, T * end0,
           T * start1, T * end1,
           std::vector< T > & t);
```

Note that in the `mergesort_` function, there's a parameter `t`. This is a reference to a vector that is created in `mergesort`. The vector that `t` references is used for the merging process (Recall: I said in class that you do not need to create a temporary array for every merging process – one can be shared among the recursive function calls of `mergesort_`. The `mergesort` function looks like this:

```
template < typename T >
void mergesort(T * start, T * end, bool verbose=false)
{
    int n = (end - start) / 2;
    std::vector< T > t(n);
    mergesort_(start, end, t, verbose);
}
```

The second `mergesort` with the comparator is similar.

Now for the verbose printing of your mergesort. Suppose your array/vector is `{3, 1, 5, 6, 4, 2}`, then your output should look like this:

```
mergesort({3, 1, 5, 6, 4, 2})
split({3, 1, 5, 6, 4, 2}) = {3, 1, 5}, {6, 4, 2}
    mergesort({3, 1, 5})
    split({3, 1, 5}) = {3}, {1, 5}
        mergesort({3})
        return {3}
        mergesort({1, 5})
        split({1, 5}) = {1}, {5}
            mergesort({1})
            return {1}
            mergesort({5})
            return {5}
        merge({1}, {5}) = {1, 5}
        return {1, 5}
     merge({3}, {1, 5}) = {1, 3, 5}
     return {1, 3, 5}
     mergesort({6, 4, 2})
     split({6, 4, 2}) = {6}, {4, 2}
         mergesort({6})
         return {6}
         mergesort({4, 2})
         split({4, 2}) = {4}, {2}
             mergesort({4})
             return {4}
             mergesort({2})
             return {2}
         merge({4}, {2}) = {2, 4}
         return {2, 4}
     merge({6}, {2, 4}) = {2, 4, 6}
     return {2, 4, 6}
merge({1, 3, 5}, {2, 4, 6}) = {1, 2, 3, 4, 5, 6}
return {1, 2, 3, 4, 5, 6}
```

(Note that the above mergesort function does not return a value. Rather it "returns by reference", i.e., the reference variable is used to hold the sorted vector. The `return` in the printout meant "return via the reference variable".)

See section on Pretty-print Recursion.

Pivot Selection for Quicksort

For this assignment, you must use median of three for your pivot selection.

[ASIDE: Once you're done with this assignment, you can think about writing a quicksort that allows you to pass a pivot selection function into your quicksort function.]

Q6. [Quicksort]

Implement the quicksort algorithm in C++, using the algorithm I mentioned in class (the inplace version). The following is the prototype:

```
template < typename T >
void quicksort(T * start, T * end, bool verbose=false);

template < typename T, typename C >
void quicksort(T * start, T * end, const C & less, bool verbose=false);
```

You must use the median of three method to pick the pivot and the partitioning method I mentioned in class where the pivot is first placed in front (the method I called pLRT).

The base case should include array size $n \leq 3$. This is because since we are using median of 3. The recursive case should have $n \geq 3$. But if $n = 3$ in the recursive since, the 3 values would already be sorted by the median of 3. If $n = 3$ is included in the base case, then the function can return right away after sorting 3 values. The sorting done in the base case should not use loops. Just unroll the loop.

Here's the expected verbose printing

```
quicksort({3, 6, 17, 15, 13, 15, 6, 12, 9, 1})
pivot = 3
left = {1}
right = {17, 15, 13, 6, 6, 12, 9, 15}
    quicksort({1})
    return {1}
    quicksort({17, 15, 13, 6, 6, 12, 9, 15})
    pivot = 15
    left = {9, 15, 13, 6, 6, 12}
    right = {17}
        quicksort({9, 15, 13, 6, 6, 12})
        pivot = 9
        left = {6, 6}
        right = {15, 13, 12}
            quicksort({6, 6})
            return {6, 6}
            quicksort({15, 13, 12})
            return {12, 13, 15}
        combine({6, 6}, 9, {12, 13, 15}) = {6, 6, 9, 12, 13, 15}
        return {6, 6, 9, 12, 13, 15}
        quicksort({17})
        return {17}
    combine({6, 6, 9, 12, 13, 15}, 15, {17}) = {6, 6, 9, 12, 13, 15, 15, 17}
    return {6, 6, 9, 12, 13, 15, 15, 17}
combine({1}, 3, {6, 6, 9, 12, 13, 15, 15, 17}) = {1, 3, 6, 6, 9, 12, 13, 15, 15, 17}
return {1, 3, 6, 6, 9, 12, 13, 15, 15, 17}
```

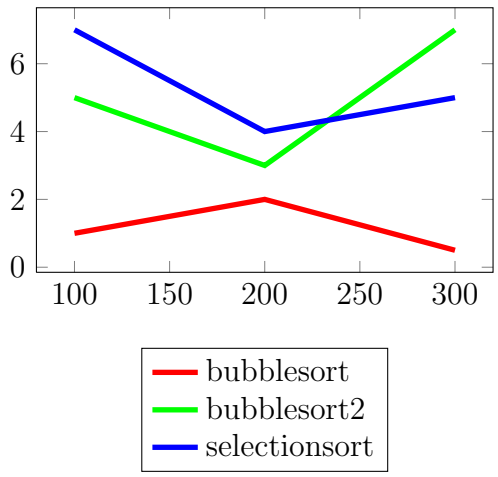
Q7. Collect actual runtime data for each algorithm you have implemented in this assignment (bubblesort, bubblesort2, insertionsort, selectionsort, mergesort, quicksort) and create a graph. See the section Plotting Runtimes on drawing line graphs in LaTeX.

For each algorithm you have implemented, collect data for vector of size 2000, 4000, 6000, 8000, ... and for each size, you should use an average of about 5 cases. Make sure you have enough data points to see the trend of your graph. Of course the size of the array for sorting depends on your laptop. Faster laptops will need larger array sizes, otherwise all the times are too small to see the trend. The values used in your vector should be integers – see the skeleton code. Make sure you have a legend for the plot.

Of course when collecting runtime data, you should set `verbose` to `false` so that output is minimal. This will ensure that the time collected is due to the sorting process and not due to output.

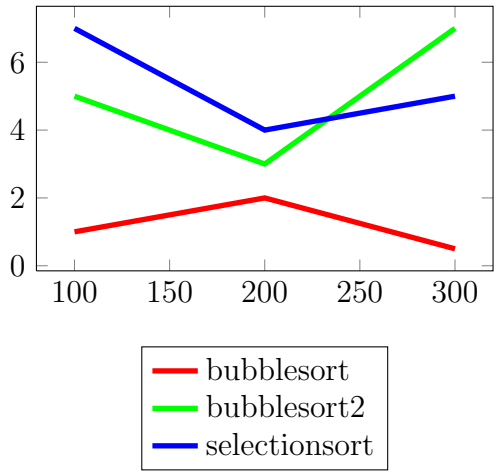
SOLUTION:

[Open q07.tex and modify the graph below with your own data.]



Plotting Runtimes

Let me show you how to create a graph in a pdf using a library that I built. Here's a graph:



The above graph is built by this code:

```
\begin{python}
from latextool_basic import *
plot = FunctionPlot(width="3in", height="2in")

plot.add(((100, 1),
          (200, 2),
          (300, 0.5)),
         line_width='2', color='red', legend='bubblesort')

plot.add(((100, 5),
          (200, 3),
          (300, 7)),
         line_width='2', color='green', legend='bubblesort2')

plot.add(((100, 7),
          (200, 4),
          (300, 5)),
         line_width='2', color='blue', legend='selectionsort')

print(plot)
\end{python}
```

In `q07.tex`, modify the graph with your own data. It should be clear where to put your data. Look at the `plot.add`. `plot.add` adds a graph:

```
plot.add(((100, 1),
```

```
         (200, 2),
         (300, 0.5)),
        line_width='2', color='red', legend='Bubble sort')
```

The `(100, 1)`, etc are the points on this graph. Be careful you don't lose any parentheses or commas, or add any extras. Then run `make` and a file `main.pdf` will be generated – your graph should appear in the pdf.

If you run `make` and have problems, you must talk to someone in CS hangout or in CCCS Discord right away – this means that something is wrong with your virtual machine.

The above graph covers an area of 3-by-2 inches. You should probably use a larger graphing area. You should increase the width and height of the graph. Also, here are some options for color: red, green, blue, cyan, yellow, black, pink, brown, teal.

ASIDE 1: Comparator and `std::sort` and `std::stable_sort`

C++ STL provides `std::sort` and `std::stable_sort`. Check my chapter on comparison-based sorting algorithms for details. Both functions accept comparator functors.

```
// file: cppsort.cpp

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

std::ostream & operator<<(std::ostream & cout, const std::vector< int > & v)
{
    cout << '{';
    std::string delim = "";
    for (auto & e: v)
    {
        cout << delim << e;
        delim = ", ";
    }
    cout << '}';
    return cout;
}

int main()
{
    std::vector< int > v {1, 3, 5, 6, 4, 2};
    std::sort(v.begin(), v.end());
    std::cout << v << '\n';

    v = {1, 3, 5, 6, 4, 2};
    std::sort(v.begin(), v.end(), [](int x, int y){ return x < y; });
    std::cout << v << '\n';

    v = {1, 3, 5, 6, 4, 2};
    std::sort(v.begin(), v.end(), [](int x, int y){ return x > y; });
    std::cout << v << '\n';

    return 0;
}
```

Usage of `std::stable_sort` is similar.

ASIDE 2: Pointers and iterators

DIY: You should also add versions of your sorting functions that accept iterators to `std::vector`. See `cppsort.cpp` example from the previous section.