CS 316 Data Structures – Fall 2019
Project 4 Due by Monday, November 18th at the **beginning** of class. 15 points.

Project Learning Outcomes

You will be modifying sorting algorithms and analyzing the runtimes of these algorithms via empirical analysis. By changing the base case of the recursive quick sort algorithm you are modifying, you will gain a better knowledge how recursive algorithms work and the algorithm itself. By analyzing the runtime, you will appreciate the importance of reducing the runtime of algorithms for large input.

Project Description

Prompt the user for an input filename and the number of elements N to sort. The input file will contain a relatively small number of integers. Each line of the input file will contain one integer. You will read this input into a vector. In addition to running your code against the vector initialized with the input file, you will be generating vectors to sort containing N elements where N will eventually be quite large. The vectors of size N that you will be initializing will contain random integers, integers in ascending order and integers in descending order.

Prove your code works on a small dataset. Before each sorting algorithm invoked, you will print the elements in this vector you populated from the user's input file and after sorting you will print the elements in this vector. Note, you will NOT be printing the vectors for large N on which you will be empirically analyzing. Using the small number of integers from the input file is a good way to test your code and verify your code works for a small set of numbers. Printing the vector for large N will take way too long.

Empirical analysis on large datasets. Additionally, you will run each algorithm for varying datasets capturing the runtime for each algorithm and each dataset where the number of elements N in the dataset is the size given by the user. That is, the prompt you give the user for the number of elements to read will be the size of the vector you are sorting. For each sorting algorithm, you will create vectors of size N containing integers in a **random** order, in an **ascending** order and in a **descending** order and run the algorithms against each of these vectors. Be careful not to use the same sorted vector for each algorithm. You will provide the actual runtime and your explanation of the runtime analysis for each algorithm and dataset (a table that you must populate and include in your report is shown later in this description). NOTE: There may be an algorithm that will not complete in an acceptable time.

Example dialog:

```
Please enter the filename: sorting.dat
Enter the number of integers to sort: 10000

vector before heap sort:  8 2 3 6 1 5 9 7
vector after heap sort: 1 2 3 5 6 7 8 9

vector before merge sort:  8 2 3 6 1 5 9 7
vector after merge sort: 1 2 3 5 6 7 8 9

vector before quick sort (no cutoff):  8 2 3 6 1 5 9 7
vector after quick sort (no cutoff): 1 2 3 5 6 7 8 9

vector before insertion sort:  8 2 3 6 1 5 9 7
vector after insertion sort: 1 2 3 5 6 7 8 9

runtime of algorithms for N = 10000 items
heap sort random: dd.ddd
heap sort ascending:  dd.ddd
heap sort descending: dd.ddd
```

```
merge sort random: dd.ddd
merge sort ascending:  dd.ddd
merge sort descending: dd.ddd

quick sort (no cutoff) random: dd.ddd
quick sort (no cutoff) ascending:  dd.ddd
quick sort (no cutoff) descending: dd.ddd

insertion sort random: dd.ddd
insertion sort ascending:  dd.ddd
insertion sort descending: dd.ddd
```

Sorting Algorithms and Project Files to Implement

You must adhere to the naming conventions for the code files and place these all of these files in the Brightspace for `Project3`.

1. `insertsort.h` for the insertion sort algorithm is implemented as a function template. A simple implementation of this code is in the book, use this code.
2. `heapsort.h` for the heap sort algorithm is implemented as a function template. An implementation of this function is in the book, use this code.
3. `mergesort.h` for the merge sort algorithm is implemented as a function template. An implementation of this function is in the book, use this code.
4. `quicksort.h` is implemented as a function template. An implementation of this function is in the book, but you must MODIFY this code. The code in the book cuts off to the insertion sort algorithm when the subarray contains less than ten elements. Your code will not cutoff to run another sorting algorithm. You will need to determine the base case(s) and do a simple comparison sort once the subarray reaches the size of these base case(s). For example, for a base case of two elements, if the left element is greater than the right element, swap the two elements. This algorithm invokes the `median3()` function to select the pivot, consider what needs to be done with this function for the base case(s).
5. `sorting.cpp` will contain your `main()` function that invokes the sorting algorithms. You will need to create and populate the integer vectors used when invoking the sorting algorithms. Use caution, after invoking a sorting algorithm, the vector that is returned should be sorted so you may not be able to use this sorted the vector to test the various sorting functions (randomized or descending).
    a. You will prompt the user for a filename to use for the initial integer data to sort; this will be a small set of integers. Here is an example of the data in the input file `sorting.dat` available here: http://www.cs.uakron.edu/~dforeback/classes/ds/Projects/sorting.dat. These integers should be used to populate the vector per the order received in the input file.
    ```
    8
    2
    3
    6
    1
    5
    9
    7
    ```
    b. For the input data obtained in the input file, and only this input data, you should give the following output. For runs of the sorting algorithms using data that you generate for large input size N, you will not display the vector—it takes too long to display this to the screen.
    ```
    vector before insertion sort:  8 2 3 6 1 5 9 7
    vector after insertion sort: 1 2 3 5 6 7 8 9
    ```

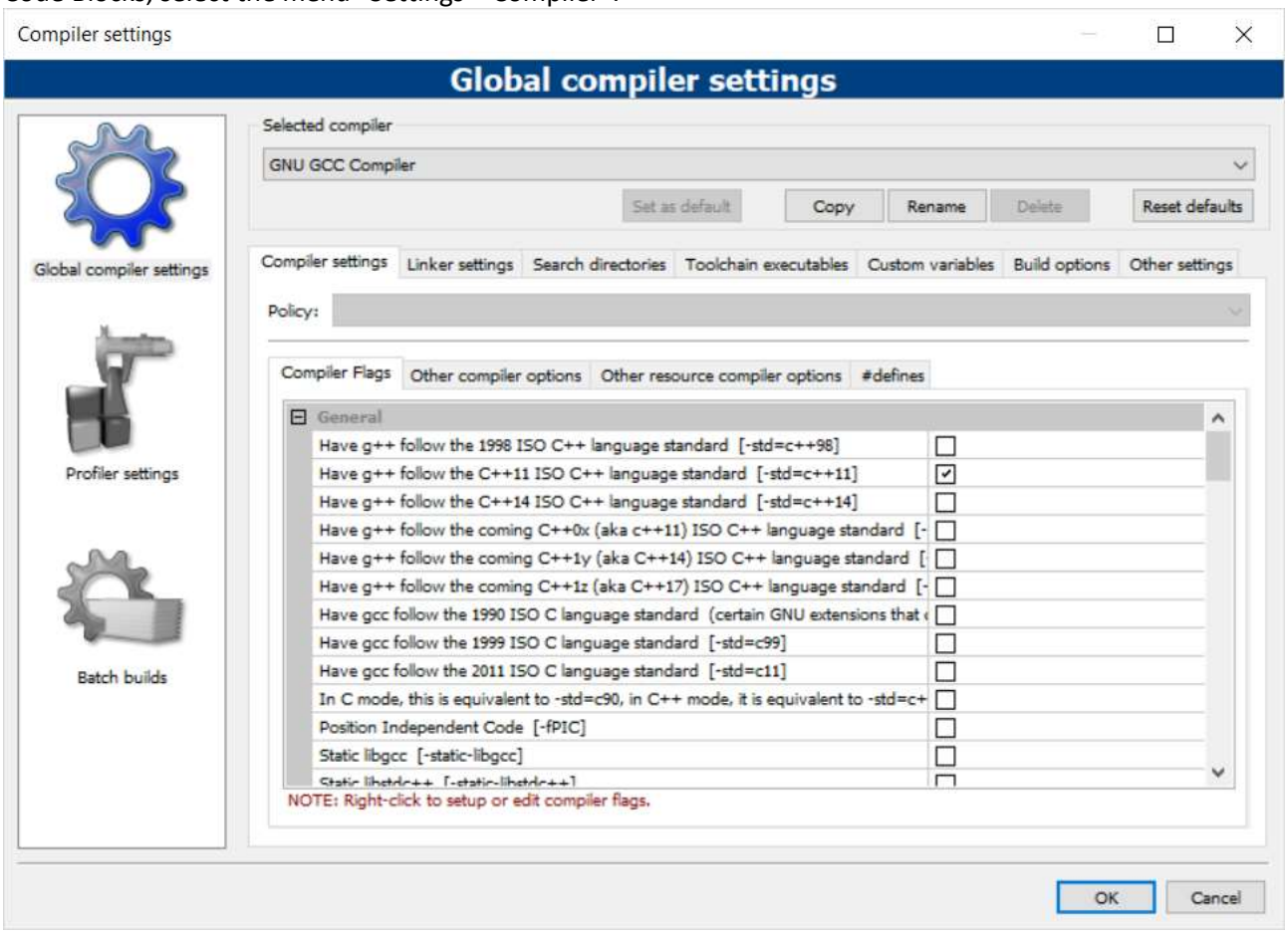
```
vector before heap sort:  8 2 3 6 1 5 9 7
vector after heap sort: 1 2 3 5 6 7 8 9

vector before merge sort:  8 2 3 6 1 5 9 7
vector after merge sort: 1 2 3 5 6 7 8 9

vector before quick sort (no cutoff):  8 2 3 6 1 5 9 7
vector after quick sort (no cutoff): 1 2 3 5 6 7 8 9
```

c. You will be analyzing runtime via clock ticks divided by the number of clock ticks per second. Here is some code that can be used to determine the runtime (this code differs from Project3):

```
#include <ctime>
clock_t start, stop;
start = clock ();
…
stop = clock ();
cout << "runtime: "<< static_cast<double>(end-start)/CLOCKS_PER_SEC;
```

Coding Requirements

Your code must run on the lab machines within the Code Blocks IDE. The compiler flag "Have g++ follow the C++11 ISO C++ language standard [-std=c++11] per the dialog box below. To launch this dialog box, within Code Blocks, select the menu "Settings > Compiler".



Project 4 Due by Monday, November 18th at the **beginning** of class. 15 points.

Your code should check for appropriate errors. For example, if the input file does not exist, inform the user the file does not exist and exit; do not have your program crash.

You must adhere to the project specifications and the prototypes for each function. All files should be named as defined in this specifications please. If you do not code the quicksort algorithm without a cut off you will lose 50% of the points; please do this coding.

You must upload your files to SVN in a folder named Project4.

Project Submission and Deliverables

Your HARD COPY report is due at the BEGINNING of class on the due date. NO ELECTRONIC COPIES OF YOUR REPORT ARE ACCEPTED. You must bring the report to class. Again, do not email the report to me or the TA, it will not be accepted. Submit your code files and report to SVN in the Project 4 folder.

Your report should be named Project4.pdf and must include:

1. Your name
2. This should be written as a report. That is, when answering questions, do not write something like "5 xyz occurred." Explain what is occurring and potentially why. Somebody reading a report should not have to go back to the specifications to determine what you are writing about. Points will be deducted if you do not write this as a report.
3. A brief summary of the project and its implementation.
4. A manifest of all files submitted and a description of each file; this should include a list of all code files and your report. The files to submit are:
    - `heapsort.h`
    - `mergesort.h`
    - `quicksort.h` (your code file modified with your name at the top)
    - `insertsort.h`
    - `sorting.cpp` (your code file that invokes each function with your name at the top)
    - `Project3.pdf` (your report with your name at the top). Again, you must submit a hard copy of this report to me at the beginning of class.
5. Answers to the following questions, again, include a description of the question and its answer.
    a. Include the table below with your runtime for each algorithm and number of elements and the Big-Oh runtime as well. This Excel sheet is available for you to populate and include in your report http://www.cs.uakron.edu/~dforeback/classes/ds/Projects/runtimeTbl.xlsx. A suggestion for including this completed table in your report is to populate it in Excel then copy the cells pasting it as a picture into your Word document.

number of integers N	runtime									theoretical Big-Oh runtime		
	randomized integers			presorted in increasing order			presorted in decreasing order			random order	increasing order	decrease order
	10,000	100,000	1,000,000	10,000	100,000	1,000,000	10,000	100,000	1,000,000			
heap sort												
merge sort												
quick sort (no cutoff)												
insertion sort												

b. For each sorting algorithm, explain the difference in runtimes for the different types of data. That is, why do you think the randomized, ascending, descending data yields different/similar runtimes for the merge sort algorithm; is there an inherent reason with the way the code works? Answer this question for each algorithm; i.e. for heap sort, merge sort, quick sort and insertion sort. Explain please.
c. Explain the difference in runtimes between the merge sort algorithm and quick sort. Why do you think there are differences/similarities?
d. Explain the differences/similarities in runtimes between the heap sort and quick sort algorithms. Why do you think there are differences/similarities?

Hopefully you will not be including this in your report. But, to potentially receive partial credit for non-working code that compiles, provide your descriptive analysis of what is occurring in the non-working code and why you believe this is occurring. Again, this is to "potentially" receive some amount of credit.

At this stage, your code should compile. Please test your code on the lab machines to verify it compiles and runs as you expect.

Grading

- Working Code (3 points) that compiles and runs on the departmental lab machines in the Code Blocks IDE
- Report (12 points) – you must answer all questions and supply entries to the runtime table. To do this, your code must run. Thus, in reality, although working code is only listed as 3 points, it is actually worth more due to the requirement that the code must properly run for each sorting algorithm to supply the required report information.
- All work must be submitted to SVN before the beginning of class and a hard copy of your report should be turned in at the beginning of class on the due date. Late assignments not accepted. If you do not submit your hardcopy of the report to me at the beginning of class, you lose 15 points.

If your code does not compile, a score of zero is earned.

To potentially receive partial credit for non-working code that compiles, include in your report a descriptive analysis of what is occurring in the non-working code and why you believe this is occurring. Again, this is to "potentially" receive some amount of credit.