San José State University
Department of Computer Engineering

CMPE 180-92
Data Structures and Algorithms in C++

Fall 2017
Instructor: Ron Mak

Assignment #12

200 points

Assigned: Friday, November 10
Due: Thursday, November 16 at 5:30 PM
URL: http://codecheck.it/files/1711102047dyq11bzs78qubfstchgrnjw4
Canvas: Assignment 12. Sorting algorithms
Points: 200

Sorting algorithms

This assignment will give you practice coding several important sorting algorithms, and you will be able to compare their performances while sorting data of various sizes.

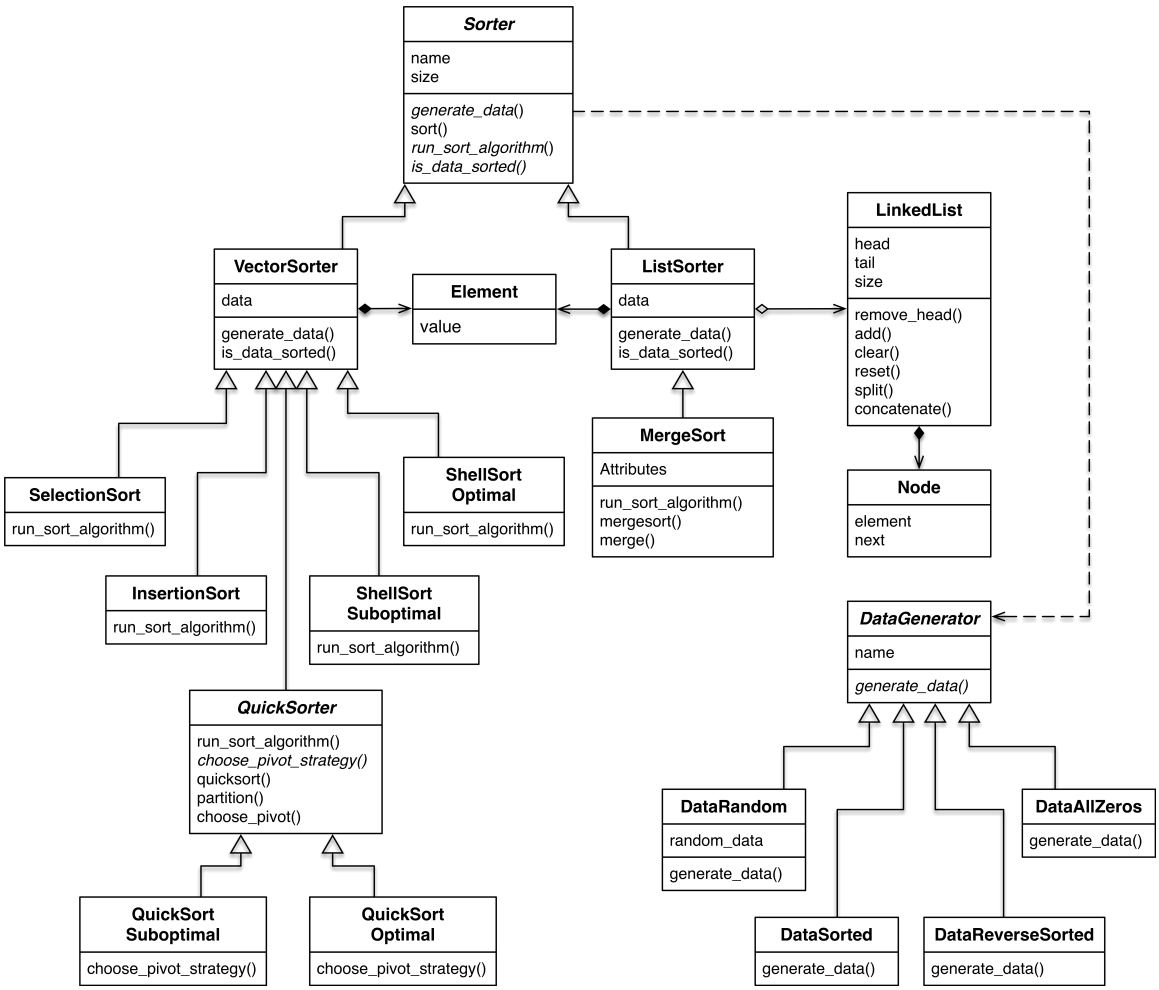
You will sort data elements in vectors with the **selection sort**, **insertion short**, **Shellsort**, and **quicksort** algorithms, and sort data elements in a linked list with the **mergesort** algorithm. There will be two versions of Shellsort, a "suboptimal" version that uses the halving technique for the diminishing increment, and an "optimal" version that uses a formula suggested by famous computer scientist Don Knuth. There will also be two versions of quicksort, a "suboptimal" version that uses a bad pivoting strategy, and an "optimal" version that uses a good pivoting strategy.

You are provided the code for the selection sort algorithm as an example, and you will code all versions of the other algorithms. You are also provided much of the support code.

Class hierarchy

The UML (Unified Modeling Language) class diagram on the following page shows the class hierarchy. The code you will write are in classes **`InsertionSort`**, **`ShellSortSuboptimal`**, **`ShellSortOptimal`**, **`QuickSorter`**, **`QuickSortSuboptimal`** **`QuickSortOptimal`**, **`LinkedList`**, and **`MergeSort`**. Complete class **`SelectionSort`** is provided as an example.

These sorting algorithms are all described in Chapter 10 of the Malik textbook. You can also find many sorting tutorials on the Web.



Class `Element`

Your program will sort **`Element`** data, both in vectors and in linked lists. Each element has a long value. Your program must also keep track of how many times each copy constructor and destructor is called.

Abstract class `Sorter`

Class **`Sorter`** is the base class of all the sorting algorithms. Its member function **`sort()`** calls member function **`run_sort_algorithm()`** which is defined in the sorting subclasses that you will write. Function **`run_sort_algorithm()`** is abstract, and therefore also the class itself is abstract.

Vector sorting classes

The sorting classes **`SelectionSort`**, **`InsertionSort`**, **`ShellSortSuboptimal`**, and **`ShellSortOptimal`** each defines the member function **`run_sort_algorithm()`.** This member function is where you code each sorting algorithm.

For class **`ShellSortSuboptimal`**, use the halving technique for the diminishing increment. The value of the interval h for the first pass should be half the data size. For each subsequent pass, half the increment, until the increment is just 1.

For class **`ShellSortOptimal`**, use Knuth's formula $3i$ +1 for i = 0, 1, 2, 3, ... in reverse for the diminishing increment. For example: ..., 121, 40, 13, 4, 1.

Abstract class `QuickSorter`

Abstract class **`QuickSorter`** does most of the work of the recursive quicksort algorithm. Its member function **`choose_pivot()`** calls abstract member function **`choose_pivot_strategy()`.** The latter is defined by the two subclasses, **`QuickSortSuboptimal`** and **`QuickSortOptimal`**.

In subclass **`QuickSortSuboptimal`**, member function **`choose_pivot_strategy()`** should always return the leftmost value of the subrange as the "bad" pivot value to use to partition the subrange.

In subclass **`QuickSortOptimal`**, member function **`choose_pivot_strategy()`** should always return the "median of three" value of the subrange as the "good" pivot value. Look at the values at the left and right ends of the subrange and the value in the middle, and choose the value that is between the other two.

Subclass `MergeSort`

Unlike the other sorting subclasses, subclass **`MergeSort`** sorts a singly linked list. Given a list to sort, it splits the list into two sublists. It recursively sorts the two sublists, and then it merges the two sublists back together. Merging involves repeatedly adding the head node of either sublist back to the main list. Which sublist donates its head depends on which head node has the smaller value. When one sublist is exhausted, concatenate the remaining nodes of the other sublist to the end of the main list.

When done properly, mergesort does not require any copying of data values. It does all of its work by relinking the nodes to move them from one list to another.

Class `LinkedList`

Class **`LinkedList`** manages a singly linked list. Member function **`split()`** splits the list into two sublists of the same size, plus or minus one. Member function **`concatenate()`** appends another list to the end of the list.

Class `DataGenerator`

Abstract class **`DataGenerator`** is the base class of subclasses **`DataRandom`**, **`DataSorted`**, **`DataReverseSorted`**, and **`DataAllZeros`**. Each subclass's member function **`generate_data()`** generates a vector of data that is random, already sorted, sorted in reverse, and all zeros, respectively.

The `main()` in `SortTests.cpp`

The main program tests each sorting algorithm for data sizes 10, 100, 1000, and 10,000. It tests each algorithm against data that is random, already sorted, sorted in reverse, and all zeros. It outputs a table similar to the one below.

Member functions to complete

Complete the following member functions for this assignment:

- `InsertionSort::run_sort_algorithm()`
- `ShellSortOptimal::run_sort_algorithm()`
- `ShellSortSuboptimal::run_sort_algorithm()`
- `QuickSorter::quicksort()`
- `QuickSorter::partition()`
- `QuickSortOptimal::choose_pivot_strategy()`
- `QuickSortSuboptimal::choose_pivot_strategy()`
- `MergeSort::mergesort()`
- `MergeSort::merge()`
- `LinkedList::split()`
- `LinkedList::concatenate()`

Comparing the algorithms

To compare the performances of the sorting algorithms, keep track of these statistics for each algorithm at each data size:

- The total number of copy constructor calls for the data elements being sorted.
- The total number of destructor calls of the data elements.
- The total number of times a data element is moved. Count one move whenever an element moves from one part of the vector or linked list to another. Whenever two elements are swapped, that counts as two moves.
- The total number of compares of two data elements. Count one compare whenever a data element is compared against another element.
- The amount of time (in milliseconds) required to do the sort.

Sample output

The following pages show sample output. Your output may not be exactly as shown, but your move and compare counts should be close.

```
===============
Unsorted random
===============

N = 10

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort          16          16          14          45           0
           Insertion sort           9           9          21          26           0
     Shellsort suboptimal          22          22          13          27           0
        Shellsort optimal          15          15          21          28           0
     Quicksort suboptimal          23          23          46          44           0
        Quicksort optimal          28          28          56          74           0
                Mergesort           0           0          49          22           0

N = 100

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort         192         192         186       4,950           0
           Insertion sort          99          99       2,587       2,591           0
     Shellsort suboptimal         503         503         692         907           0
        Shellsort optimal         342         342         525         646           0
     Quicksort suboptimal         311         311         622         795           0
        Quicksort optimal         361         361         722       1,069           0
                Mergesort           0           0         837         540           0

N = 1,000

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort       1,993       1,993       1,988     499,500           4
           Insertion sort         999         999     259,430     259,429           4
     Shellsort suboptimal       8,006       8,006      11,545      15,087           0
        Shellsort optimal       5,457       5,457      12,180      13,772           0
     Quicksort suboptimal       3,806       3,806       7,612      13,245           0
        Quicksort optimal       4,390       4,390       8,780      14,965           0
                Mergesort           0           0      11,727       8,730           0

N = 10,000

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort      19,986      19,986      19,974  49,995,000         354
           Insertion sort       9,999       9,999  24,880,971  24,880,973         406
     Shellsort suboptimal     120,005     120,005     207,988     265,518           8
        Shellsort optimal      75,243      75,243     195,946     218,189           7
     Quicksort suboptimal      45,288      45,288      90,576     175,721           3
        Quicksort optimal      51,220      51,220     102,440     192,455           3
                Mergesort           0           0     150,482     120,485           4
```

```
==============
Already sorted
==============

N = 10

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort           9           9           0          45           0
           Insertion sort           9           9           0           9           0
     Shellsort suboptimal          22          22           0          22           0
        Shellsort optimal          15          15           0          15           0
     Quicksort suboptimal          20          20          40          65           0
        Quicksort optimal          20          20          40          69           0
                Mergesort           0           0          42          15           0

N = 100

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort          99          99           0       4,950           0
           Insertion sort          99          99           0          99           0
     Shellsort suboptimal         503         503           0         503           0
        Shellsort optimal         342         342           0         342           0
     Quicksort suboptimal         200         200         400       5,150           0
        Quicksort optimal         200         200         400         980           0
                Mergesort           0           0         613         316           0

N = 1,000

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort         999         999           0     499,500           3
           Insertion sort         999         999           0         999           0
     Shellsort suboptimal       8,006       8,006           0       8,006           0
        Shellsort optimal       5,457       5,457           0       5,457           0
     Quicksort suboptimal       2,000       2,000       4,000     501,500           3
        Quicksort optimal       2,000       2,000       4,000      12,987           0
                Mergesort           0           0       7,929       4,932           0

N = 10,000

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort       9,999       9,999           0  49,995,000         338
           Insertion sort       9,999       9,999           0       9,999           0
     Shellsort suboptimal     120,005     120,005           0     120,005           2
        Shellsort optimal      75,243      75,243           0      75,243           1
     Quicksort suboptimal      20,000      20,000      40,000  50,015,000         341
        Quicksort optimal      20,000      20,000      40,000     163,631           2
                Mergesort           0           0      94,605      64,608           2
```

```
==============
Reverse sorted
==============

N = 10

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort          14          14          10          45           0
           Insertion sort           9           9          54          45           0
     Shellsort suboptimal          22          22          24          27           0
        Shellsort optimal          15          15          24          21           0
     Quicksort suboptimal          20          20          40          65           0
        Quicksort optimal          27          27          54          69           0
                Mergesort           0           0          46          19           0

N = 100

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort         149         149         100       4,950           0
           Insertion sort          99          99       5,049       4,950           0
     Shellsort suboptimal         503         503         516         668           0
        Shellsort optimal         342         342         420         500           0
     Quicksort suboptimal         200         200         400       5,150           0
        Quicksort optimal         252         252         504         980           0
                Mergesort           0           0         653         356           0

N = 1,000

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort       1,499       1,499       1,000     499,500           4
           Insertion sort         999         999     500,499     499,500           7
     Shellsort suboptimal       8,006       8,006       9,072      11,716           0
        Shellsort optimal       5,457       5,457       6,855       8,550           0
     Quicksort suboptimal       2,000       2,000       4,000     501,500           3
        Quicksort optimal       2,502       2,502       5,004      12,987           0
                Mergesort           0           0       8,041       5,044           0

N = 10,000

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort      14,999      14,999      10,000  49,995,000         468
           Insertion sort       9,999       9,999  50,004,999  49,995,000         758
     Shellsort suboptimal     120,005     120,005     124,592     172,578           3
        Shellsort optimal      75,243      75,243      93,666     120,190           2
     Quicksort suboptimal      20,000      20,000      40,000  50,015,000         340
        Quicksort optimal      25,002      25,002      50,004     163,631           2
                Mergesort           0           0      99,005      69,008           2
```

```
==========
All zeroes
==========

N = 10

               ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
          Selection sort           9           9           0          45           0
          Insertion sort           9           9           0           9           0
     Shellsort suboptimal         22          22           0          22           0
        Shellsort optimal         15          15           0          15           0
     Quicksort suboptimal         27          27          54          34           0
        Quicksort optimal         27          27          54          64           0
                Mergesort          0           0          42          15           0

N = 100

               ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
          Selection sort          99          99           0       4,950           0
          Insertion sort          99          99           0          99           0
     Shellsort suboptimal        503         503           0         503           0
        Shellsort optimal        342         342           0         342           0
     Quicksort suboptimal        419         419         838         638           0
        Quicksort optimal        419         419         838         938           0
                Mergesort          0           0         613         316           0

N = 1,000

               ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
          Selection sort         999         999           0     499,500           3
          Insertion sort         999         999           0         999           0
     Shellsort suboptimal      8,006       8,006           0       8,006           0
        Shellsort optimal      5,457       5,457           0       5,457           0
     Quicksort suboptimal      5,938       5,938      11,876       9,876           0
        Quicksort optimal      5,938       5,938      11,876      12,876           0
                Mergesort          0           0       7,929       4,932           0

N = 10,000

               ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
          Selection sort       9,999       9,999           0  49,995,000         355
          Insertion sort       9,999       9,999           0       9,999           0
     Shellsort suboptimal    120,005     120,005           0     120,005           2
        Shellsort optimal     75,243      75,243           0      75,243           1
     Quicksort suboptimal     74,613      74,613     149,226     129,226           2
        Quicksort optimal     74,613      74,613     149,226     159,226           3
                Mergesort          0           0      94,605      64,608           2

Done! 3 seconds.
```

Using code from books and the Web

Many books and Web articles will contain code for these sorting algorithms. If you use code from these sources, **you must cite your sources** (book or URL) in your program comments. Otherwise you can be caught by the software plagiarism checker.

Of course, you should understand what the code is doing, not simply copy it.

Copying from another student's program is still strictly forbidden.

What to submit

If you time out in CodeCheck, then run with only 10, 100, and 1000 data elements. Include 10,000 data elements outside of CodeCheck and copy that output into a text file.

Submit the signed zip file from CodeCheck into **Canvas: Assignment 12. Sorting algorithms**. Also submit the text file containing the output from larger numbers of data elements.

Due to use of random numbers in this assignment, CodeCheck will not compare your output.

Rubrics

Criteria	Maximum points
Output (counts should be close to the sample output) • Insertion sort • Shellsort suboptimal • Shellsort optimal • Quicksort suboptimal • Quicksort optimal • Mergesort	**60** • 10 • 10 • 10 • 10 • 10 • 10
Algorithm code • `Element` • `InsertionSort::run_sort_algorithm()` • `ShellSortSuboptimal::run_sort_algorithm()` • `ShellSortOptimal::run_sort_algorithm()` • `QuickSorter::run_sort_algorithm()` • `QuickSorter::quicksort()` • `QuickSorter::partition()` • `QuickSortSuboptimal::choose_pivot_strategy()` • `QuickSortOptimal::choose_pivot_strategy()` • `Node` • `LinkedList::split()` • `LinkedList::concatenate()` • `MergeSort::mergesort()` • `MergeSort::merge()`	**140** • 10 • 10 • 10 • 10 • 10 • 10 • 10 • 10 • 10 • 10 • 10 • 10 • 10 • 10

Extra credit (up to 50 points)

Choose an appropriate set of values N for the sizes of the data to sort. Create line graphs (Excel graphs are fine) that plot the numbers of moves and compares and the elapsed times of the algorithms. Create as many graphs as you need to show the differences in growth rates.

Academic integrity

You may study together and discuss the assignments, but what you turn in must be your individual work. Assignment submissions will be checked for plagiarism using Moss (http://theory.stanford.edu/~aiken/moss/). **Copying another student's program or sharing your program is a violation of academic integrity.** Moss is not fooled by renaming variables, reformatting source code, or re-ordering functions.

Violators of academic integrity will suffer severe sanctions, including academic probation. Students who are on academic probation are not eligible for work as instructional assistants in the university or for internships at local companies.