San José State University
Department of Computer Engineering

CMPE 180-92

Data Structures and Algorithms in C++

Fall 2016
Instructor: Ron Mak

Assignment #12

170 points
With up to 115 points extra credit

Assigned: Monday, November 21
Due: Friday, December 2 at 11:59 PM
URL: http://codecheck.it/codecheck/files/16112406537pbyc2ouvrt86esy1xvl2284g
Canvas: Assignment 12. Sorting algorithms
Points: 170

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.

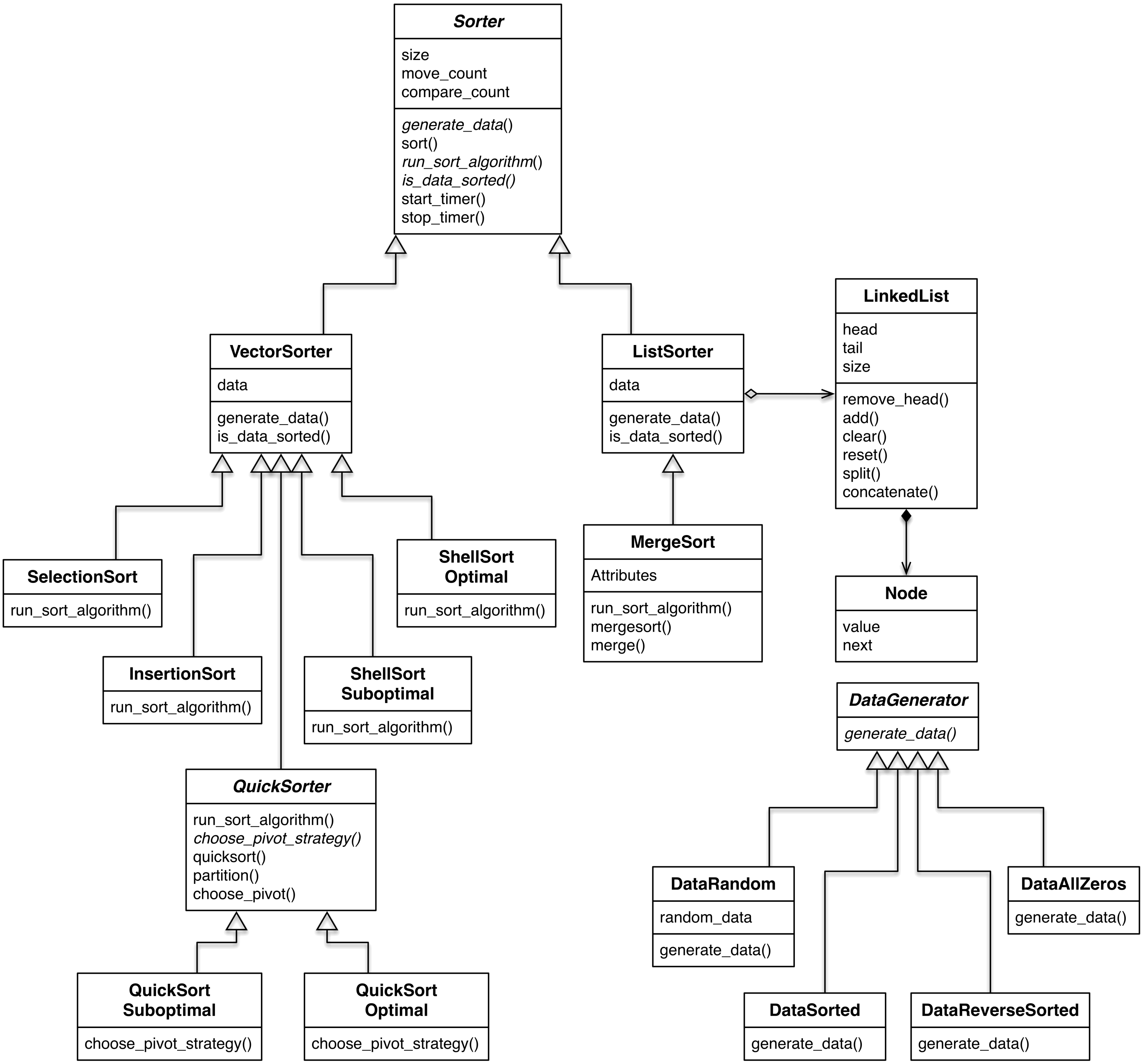
You will sort integer data in vectors with the **selection sort**, **insertion short**, **Shellsort**, and **quicksort** algorithms, and sort integer data in a linked list with the **mergesort** algorithm. There will be two versions of Shellsort, a "suboptimal" version that uses the halving technique 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.



Abstract root class `Sorter`

Class **`Sorter`** is the root class of all the sorting algorithms. Its member function **`sort()`** calls member function **`run_sort_algorithm()`** which is defined in the sorting subclasses and which you will write. Function **`run_sort_algorithm()`** and therefore the class itself are 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 $i = 0, 1, 2, 3, \ldots$ 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 moving nodes from one list to another by relinking them.

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 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 100, 1000, 10,000, and 100,000. 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. If you have a slow computer, you can stop at 10,000.

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 their performances, keep track of how many moves and compares each sorting algorithms makes during a sort. Count one move whenever a data value moves from one part of the vector or linked list to another. Whenever two values are swapped, that counts as two moves. Count one compare whenever a data value is compared against another value (such as another value in the vector or linked list). Also time each sort to the millisecond.

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 = 100

              ALGORITHM          MOVES       COMPARES   MILLISECONDS
         Selection sort            190          4,950              0
         Insertion sort          2,588          2,586              0
   Shellsort suboptimal            599            811              0
      Shellsort optimal            557            675              0
   Quicksort suboptimal            590            831              0
      Quicksort optimal            734          1,017              0
              Mergesort            836            539              0

N = 1,000

              ALGORITHM          MOVES       COMPARES   MILLISECONDS
         Selection sort          1,964        499,500              0
         Insertion sort        250,797        250,796              0
   Shellsort suboptimal         10,767         14,428              0
      Shellsort optimal         12,379         14,052              0
   Quicksort suboptimal          7,584         12,545              0
      Quicksort optimal          8,828         13,847              0
              Mergesort         11,694          8,697              0

N = 10,000

              ALGORITHM          MOVES       COMPARES   MILLISECONDS
         Selection sort         19,978     49,995,000             71
         Insertion sort     24,892,581     24,892,577             57
   Shellsort suboptimal        208,786        266,654              1
      Shellsort optimal        216,984        238,269              0
   Quicksort suboptimal         91,440        171,522              0
      Quicksort optimal        103,382        186,968              0
              Mergesort        150,359        120,362              1

N = 100,000

              ALGORITHM          MOVES       COMPARES   MILLISECONDS
         Selection sort        199,968  4,999,950,000          6,802
         Insertion sort  2,491,031,949  2,491,031,949          5,950
   Shellsort suboptimal      3,783,086      4,462,793             16
      Shellsort optimal      3,607,677      3,870,708             14
   Quicksort suboptimal      1,065,244      2,213,091             10
      Quicksort optimal      1,195,190      2,229,376             11
              Mergesort      1,836,282      1,536,285             25
```

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

N = 100

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

N = 1,000

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

N = 10,000

             ALGORITHM          MOVES       COMPARES   MILLISECONDS
        Selection sort              0     49,995,000             72
        Insertion sort              0          9,999              0
  Shellsort suboptimal              0        120,005              0
     Shellsort optimal              0         75,243              0
  Quicksort suboptimal         40,000     50,015,000             48
     Quicksort optimal         40,000        163,631              0
             Mergesort         94,605         64,608              0

N = 100,000

             ALGORITHM          MOVES       COMPARES   MILLISECONDS
        Selection sort              0  4,999,950,000          6,726
        Insertion sort              0         99,999              0
  Shellsort suboptimal              0      1,500,006              3
     Shellsort optimal              0        967,146              1
  Quicksort suboptimal        400,000  5,000,150,000          4,878
     Quicksort optimal        400,000      1,968,946              2
             Mergesort      1,115,021        815,024             10
```

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

N = 100

             ALGORITHM          MOVES       COMPARES   MILLISECONDS
        Selection sort            100          4,950              0
        Insertion sort          5,049          4,950              0
   Shellsort suboptimal           516            668              0
     Shellsort optimal            420            500              0
   Quicksort suboptimal           400          5,150              0
     Quicksort optimal            504            980              0
             Mergesort            653            356              0

N = 1,000

             ALGORITHM          MOVES       COMPARES   MILLISECONDS
        Selection sort          1,000        499,500              0
        Insertion sort        500,499        499,500              1
   Shellsort suboptimal         9,072         11,716              0
     Shellsort optimal          6,855          8,550              0
   Quicksort suboptimal         4,000        501,500              0
     Quicksort optimal          5,004         12,987              0
             Mergesort          8,041          5,044              0

N = 10,000

             ALGORITHM          MOVES       COMPARES   MILLISECONDS
        Selection sort         10,000     49,995,000             65
        Insertion sort     50,004,999     49,995,000            120
   Shellsort suboptimal       124,592        172,578              0
     Shellsort optimal         93,666        120,190              0
   Quicksort suboptimal        40,000     50,015,000             52
     Quicksort optimal         50,004        163,631              0
             Mergesort         99,005         69,008              0

N = 100,000

             ALGORITHM          MOVES       COMPARES   MILLISECONDS
        Selection sort        100,000  4,999,950,000          6,557
        Insertion sort  5,000,049,999  4,999,950,000         11,960
   Shellsort suboptimal     1,655,808      2,244,585             10
     Shellsort optimal      1,091,006      1,533,494              5
   Quicksort suboptimal       400,000  5,000,150,000          4,854
     Quicksort optimal        500,004      1,968,946              3
             Mergesort      1,153,901        853,904              9
```

```
=========
All zeros
=========

N = 100

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

N = 1,000

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

N = 10,000

               ALGORITHM        MOVES     COMPARES  MILLISECONDS
          Selection sort            0   49,995,000            65
          Insertion sort            0        9,999             0
    Shellsort suboptimal            0      120,005             0
       Shellsort optimal            0       75,243             0
    Quicksort suboptimal      149,226      129,226             0
       Quicksort optimal      149,226      159,226             0
               Mergesort       94,605       64,608             0

N = 100,000

               ALGORITHM        MOVES     COMPARES  MILLISECONDS
          Selection sort            0 4,999,950,000         6,496
          Insertion sort            0       99,999             0
    Shellsort suboptimal            0    1,500,006             2
       Shellsort optimal            0      967,146             1
    Quicksort suboptimal    1,830,060    1,630,060             4
       Quicksort optimal    1,830,060    1,930,060             5
               Mergesort    1,115,021      815,024            10


Done! 55 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.

Extra credit #1 (15 points)

Implement the pivot strategy used by Malik, which is to simply pick the middle element of the subrange. Include this version of quicksort with the other tests.

How does Malik's strategy compare with the "suboptimal" and the "optimal" quicksort versions? Under what data conditions does choosing the right pivot strategy really matter, and why?

Extra credit #2 (100 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. Excel graphs are fine, and create as many as you need to show the differences in growth rates.

Explain the growth rate differences in the numbers of moves and compares and in the elapsed times among the sorting algorithms.

What to submit

Run with only 100 and 1000 data elements in CodeCheck. Otherwise, you risk having timeouts. Run with larger numbers of 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 against a master.

Also submit any files for extra credit work. It's best to submit all your files at the same time. If you submit files at different times, include a message to let the graders know.

Rubrics

Criteria	Maximum points
Output statistics (move and compare 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 • `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()`	**110** • 10 • 10 • 10 • 10 • 10 • 10 • 10 • 10 • 10 • 10 • 10
Extra credit #1 • Implement Malik's pivot strategy • Answers to pivot strategy questions	**15** • 10 • 5
Extra credit #2 • Line graphs • Explanations of the growth rate differences	**100** • 50 • 50