San José State University
Department of Computer Engineering

# CMPE 180A
# Data Structures and Algorithms in C++

Fall 2020
Instructor: Ron Mak

## Assignment #12

120 points

**Assigned:** Tuesday, November 10
**Due:** Tuesday, November 17 at 5:30 PM
**URL:** http://codecheck.it/files/20042106449v8h269ox2cydek52zp0kv5rm
**Canvas:** Assignment 12: Sorting algorithms

### 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.

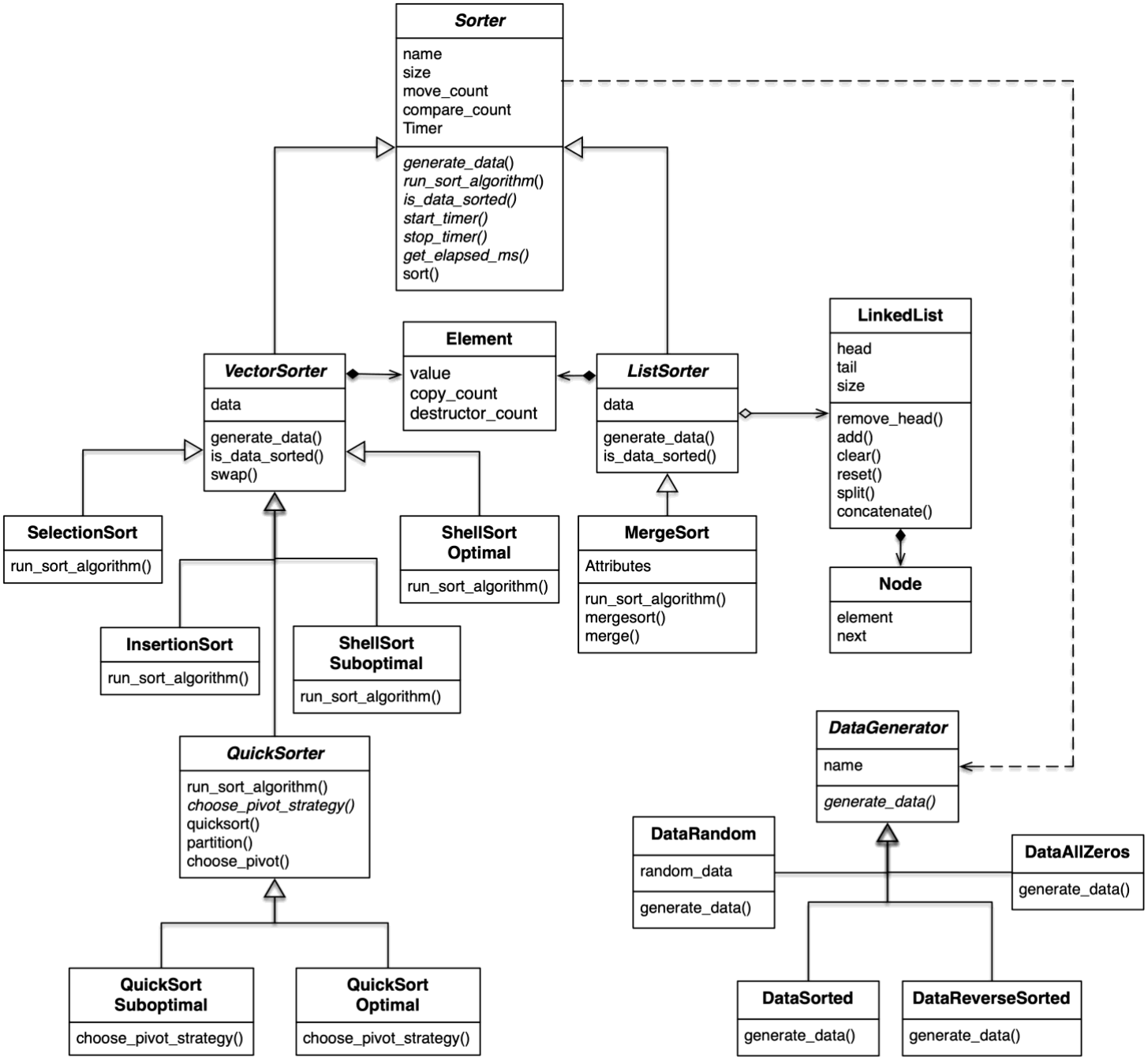
### Class hierarchy

The UML (Unified Modeling Language) class diagram on the following page shows the class hierarchy. You are provided the complete code for class **`SelectionSort`** as an example, and you will code all versions of the other algorithms. You are also provided much of the support code.

The code you will write are in these classes:

- **`InsertionSort`**
- **`ShellSortSuboptimal`**
- **`ShellSortOptimal`**
- **`QuickSorter`**
- **`QuickSortSuboptimal`**
- **`QuickSortOptimal`**
- **`LinkedList`**
- **`MergeSort`**

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




**Abstract class `Sorter`**

Class **`Sorter`** is the base class of all the sorting algorithms. Its member function **`sort()`** calls abstract member function **`run_sort_algorithm()`** which is defined in the sorting subclasses that you will write. It contains the sort timers.

**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 `VectorSorter`**
This is a subclass of **`Sorter`** and the base class for the vector sorting subclasses.

**The vector sorting classes**
The sorting classes **`SelectionSort`**, **`InsertionSort`**, **`ShellSortSuboptimal`**, and **`ShellSortOptimal`** are subclasses of **`VectorSorter`**. 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`**
This a subclass of **`VectorSorter`** and the base class for the quicksort subclasses. It does most of the work of the recursive quicksort algorithm. Its member function **`choose_pivot()`** calls abstract member function **`choose_pivot_strategy()`** which is defined by the two subclasses, **`QuickSortSuboptimal`** and **`QuickSortOptimal`**.

**Class `QuickSortSuboptimal`**
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.

**Class `QuickSortOptimal`**
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 then choose the value that is between the other two.

**Class ListSorter**
This is a subclass of **`Sorter`** and the base class for the **`MergeSort`** subclass. It has a pointer to a **`LinkedList`** of **`Node`** objects.

**Class LinkedList**
Class **`LinkedList`** manages a singly linked list of **`Node`** objects. 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 `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.

### The data generation classes

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 class `SortTests`

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.

### 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 elapsed time (in milliseconds) required to do the sort.

Collect these statistics only during sorting.

### Sample output

The following pages show sample output. Your statistics 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          17          17          16          45           0
           Insertion sort           9           9          18          19           0
     Shellsort suboptimal          22          22          11          26           0
        Shellsort optimal          15          15          12          18           0
     Quicksort suboptimal          24          24          48          40           0
        Quicksort optimal          26          26          52          69           0
                Mergesort           0           0          47          20           0

N = 100

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort         196         196         194       4,950           0
           Insertion sort          99          99       2,615       2,616           0
     Shellsort suboptimal         503         503         570         802           0
        Shellsort optimal         342         342         580         685           0
     Quicksort suboptimal         314         314         628         865           0
        Quicksort optimal         365         365         730       1,028           0
                Mergesort           0           0         840         543           0

N = 1,000

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort       1,989       1,989       1,980     499,500           2
           Insertion sort         999         999     252,734     252,734           3
     Shellsort suboptimal       8,006       8,006      10,963      14,633           0
        Shellsort optimal       5,457       5,457      11,622      13,313           0
     Quicksort suboptimal       3,833       3,833       7,666      12,070           0
        Quicksort optimal       4,425       4,425       8,850      14,277           0
                Mergesort           0           0      11,706       8,709           0

N = 10,000

                ALGORITHM      COPIES   DESTRUCTS       MOVES    COMPARES   MILLISECS
           Selection sort      19,989      19,989      19,980  49,995,000         212
           Insertion sort       9,999       9,999  24,928,366  24,928,365         239
     Shellsort suboptimal     120,005     120,005     202,538     260,034           4
        Shellsort optimal      75,243      75,243     211,119     232,521           3
     Quicksort suboptimal      45,046      45,046      90,092     179,938           2
        Quicksort optimal      51,918      51,918     103,836     181,102           2
                Mergesort           0           0     150,477     120,480           2
```

```
==============
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           1
          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           2
        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         206
          Insertion sort      9,999       9,999           0       9,999           0
     Shellsort suboptimal   120,005     120,005           0     120,005           1
        Shellsort optimal    75,243      75,243           0      75,243           0
     Quicksort suboptimal    20,000      20,000      40,000  50,015,000         202
        Quicksort optimal    20,000      20,000      40,000     163,631           1
                Mergesort          0           0      94,605      64,608           1
```

```
==============
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           2
          Insertion sort         999         999     500,499     499,500           4
     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           1
        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         291
          Insertion sort       9,999       9,999  50,004,999  49,995,000         490
     Shellsort suboptimal    120,005     120,005     124,592     172,578           2
        Shellsort optimal     75,243      75,243      93,666     120,190           1
     Quicksort suboptimal     20,000      20,000      40,000  50,015,000         196
        Quicksort optimal     25,002      25,002      50,004     163,631           1
                Mergesort          0           0      99,005      69,008           1
```

```
==========
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           1
          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         203
          Insertion sort       9,999       9,999           0       9,999           0
     Shellsort suboptimal    120,005     120,005           0     120,005           1
        Shellsort optimal     75,243      75,243           0      75,243           0
     Quicksort suboptimal     74,613      74,613     149,226     129,226           1
        Quicksort optimal     74,613      74,613     149,226     159,226           2
                Mergesort          0           0      94,605      64,608           1

Done! Total 2,128 ms.
```

**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, and 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 comparable to the sample output)<br>• Unsorted random<br>• Already sorted<br>• Reverse sorted<br>• All zeroes | **40**<br>• 10<br>• 10<br>• 10<br>• 10 |
| **Algorithm classes**<br>• `InsertionSort`<br>• `ShellSortSuboptimal`<br>• `ShellSortOptimal`<br>• `QuickSorter`<br>• `QuickSortSuboptimal`<br>• `QuickSortOptimal`<br>• `LinkedList`<br>• `MergeSort` | **80**<br>• 10<br>• 10<br>• 10<br>• 10<br>• 10<br>• 10<br>• 10<br>• 10 |

**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.