CS14 Fall 2016 Lab 3
Implementing Cubic Sort: An $O(N\sqrt[3]{N})$ comparison-based sort

October 26, 2016

DUE: Submit by iLearn by 1:00am Wednesday, November 9, 2016
LATE SUBMISSION (50% penalty): 1:00am Saturday, November 12, 2016

1 Introduction

This assignment is designed to give you practice using templates and the vector class from the STL. Your task is to create vectors of data elements by reading test cases from a file, sort each of those vectors using the *Cubic sort algorithm* described below, and then output the sorted version of each test case to another file.

2 Converting each test case a vector

Represent the sequence of data elements by the STL implementation for vector, without any changes. Test your program when the data data type for each element is a simple `int` as well as the more complex user-defined structure `class Fraction`, which is defined here.

`http://www.cs.ucr.edu/~mart/CS14/frac.cpp`

Your driver program should accept *one* input filename as a command-line argument and generate *one* similarly-named output file with the original filename extension (if any) changed to "`.out`". For example, if the input file were `sample_input.txt`, then the output file would be `sample_input.out`.

The input file is a plain ASCII text file, consisting of a variable number of test cases (with two rows of data per test case), terminating with end-of-file. For each test case, the first row contains a single character, where "i" indicates that the next row contains a sequence of integers each separated by a single white space, and "f" indicates that the next row contains a sequence of fractions, where pairs of integers represent the numerator and denominator of each fractional value, and all integers are once again separated by a single space.

For each test case, your driver program must use the Standard Template library to construct a vector of data elements from the input sequence, then pass the list as a reference parameter to your sort function, and finally print out the sequence after sorting. Similar to the input format, the output file should contain **three rows per test case**:

1. the first indicating the problem type (i.e., "i" or "f"), and
2. the second row gives the sequence of values from the list which has been sorted using your Square Sort function
3. the third row gives the sequence of values from the list which have been sorted using your Cubic Sort function.

Make sure both functions start from the original unsorted list of data elements, using the order given in the input file. **Do not use the sorted output vector generated by one algorthm as the input vector for the other algorithm!**

Include the output method given with the `Fraction` class in your program so the fractional values will print with "/" between the numerator and denominator to make it easier to see that the results have actually been sorted correctly.

3 Square Sort Algorithm

Square Sort is a two-step algorithm for sorting a vector of N elements, represented by rectangle `A` in Figure 3. Each step uses a simple $O(N^2)$ sorting algorithm to carry out part of task, but by cleverly combining the two algorithms we obtain a much-faster overall efficiency of $O(N\sqrt{N})$. For example, when $N = 1024 \equiv 2^{10}$, the running time for a simple $O(N^2)$ sorting algorithm would be 2^{20} (i.e., about 1 million), while Square Sort's running time of 2^{15} (i.e., about 32,000) is not much worse than a fast $O(N \log N)$ algorithm whose running time would be $10 \times 2^{10} \approx 10,000$.

3.1 Step 0: Partition the vector into $\sqrt{N}$ blocks of size $\sqrt{N}$

Before the first sorting step you must partition the vector (represented by rectangle `A` in Figure 1) into a sequence of non-overlapping "blocks" each containing about $\sqrt{N}$ elements (represented by the rectangles `B`, `C` and `D` in the same Figure). Since N may not a perfect square, I suggest you use `b=int(`$\sqrt{N}$`)` for the number of blocks and `s=int(N/b)` for the size of each block (except the last one, whose size must be $\geq$ `s` but $<$ `2s`). With a little more effort you can choose the block boundaries so that the size of every block is either `s` or `s+1`.

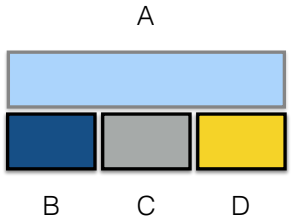

Figure 1: The vector `A` is split into three blocks `B`, `C` and `D`. For best performance, we want the total number of blocks, `b`, to match the size of each block, `s`.

Store the index of the first element of each block in another vector, called `left` say, with `b+1 elements` whose entries are chosen so that element `A[j]` from the data vector belongs to block `i` if and only if

`left`[`i`] $\leq$ `j` $<$ `left`[`i` + 1].

3.2 Step 1: Create a Reverse Insertion Sort Function and apply it to each block

Insertion Sort is a well-known $O(N^2)$ sorting algorithm, for which the code is given in both the textbook and my lecture slides. However, in this application we will find it more convenient to *reverse the direction of operation*, as shown by the four steps in the transformation of one block in Figure 2.

The initial situation is shown in Figure 2(a), where almost the entire block is unsorted (indicated here by dark shading), except for the last element (indicated by red up-arrow) which is a trivial group of one sorted elements (indicated by light shading and horizontal black arrow below). The task in the first iteration of Insertion Sort's outer loop is to expand the current group of already-sorted elements by one by inserting the *second-last element* (indicated by blue down-arrow) into the correct position relative to the "sorted" group (which in this case is really just the last element). In later iterations (Figures 2(b)–(c)), the size of the sorted group keeps expanding towards the left, thus reducing the remaining number of unsorted elements until its last element (indicated by blue down-arrow) "falls off" the left side of the now completely-sorted block (Figure 2(d)).

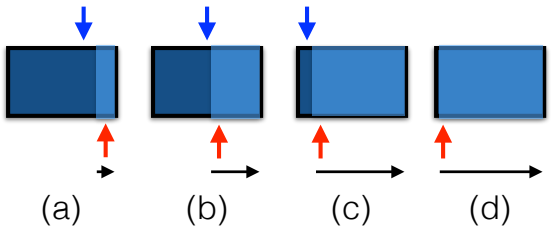

Figure 2: Four snapshots showing the progress of *Reverse Insertion Sort* applied to one block from vector `A`. Notice how the sorted group (light colored shading, with horizontal arrow below) expands from right to left.

3.2.1 Inner loop as a block repair function

When Insertion Sort's inner loop is executed for the kth time, it combines the last $k+1$ elements from the block, i.e., the last unsorted element (blue down-arrow) followed by the group of k already-sorted elements (light shading above horizontal arrow), to form a completely-sorted group of $k+1$ elements. We will need this exact code for *block repair* during Square Sort Part 2, so let's package it as a function called `insertRight` with three parameters:

- the passed-by-reference vector `A`, which contains the current block;
- the position `p` of an element from `A`, which is the single element from the current block that must be added to the already-sorted part of the block on its right during this iteration; and
- the position `l`, which is the last element of the current block, i.e., the already-sorted part of this block consists of those elements from `A` in positions `p+1` through `l`.

Note that you could represent the parameters `p` and `l` as either integer subscripts or iterators, but your code must use iterators to get full credit for the assignment.

As usual, `insertRight` creates a "hole" by moving the "new" element `A[p]` to a temporary location, then left-shifts each element from the sorted list that is smaller than the "new" element — thus moving the "hole" towards the right until it reaches the correct location for storing the "new" element.

3.3 Step 2: Apply Selection Sort to the individually-sorted blocks

During Square Sort Step 1, each block gets individually sorted. Thus, the task for Square Sort Step 2 is to efficiently combine this sequence of individually-sorted blocks to form a single globally-sorted vector.

We will be using the well-known $O(N^2)$ algorithm called Selection Sort in Step 2. Although the code for Selection Sort is not shown in the textbook, the algorithm is very easy to explain using vector `A` (upper wide block) from Figure 3 to illustrate its operation.

In iteration `p` for Selection Sort, $0 \leq$ `p` $<$ `size(A)`, we call `A[p]` the *target element*, and mark it by the blue down-arrow in Figure 3. The purpose of iteration `p` is to find the corresponding *donor element* `A[q]` which has the minimum value among all elements between element `p` and the end of the vector. If `p` $\neq$ `q`, then we swap `A[p]` and `A[q]`, so that at the end of this iteration `A[0]` through `A[p]` contain the `p+1` smallest values from vector `A` in sorted order (indicated by the medium-blue shading and black horizontal arrow above the block `A`).

Clearly the standard Selection Sort algorithm is an $O(N^2)$ algorithm, because each element `p` from the vector `A` will serve as the target element for one iteration, and every element to the right of `p` must be examined to find the donor for this iteration.

When Selection Sort is applied to Square Sort Step 2, it still requires N iterations of the outer loop because every element `p` from vector `A` needs to serve as the target. However, because every individual block of $\sqrt{N}$ elements is *already sorted*, we only need to check the *smallest remaining element of each block* (as indicated by the red up-arrows for the elements `x`, `y` and `z`) to find the corresponding donor element. This change reduces the number of iterations for inner loop from $O(N)$ to $O(\sqrt{N})$.

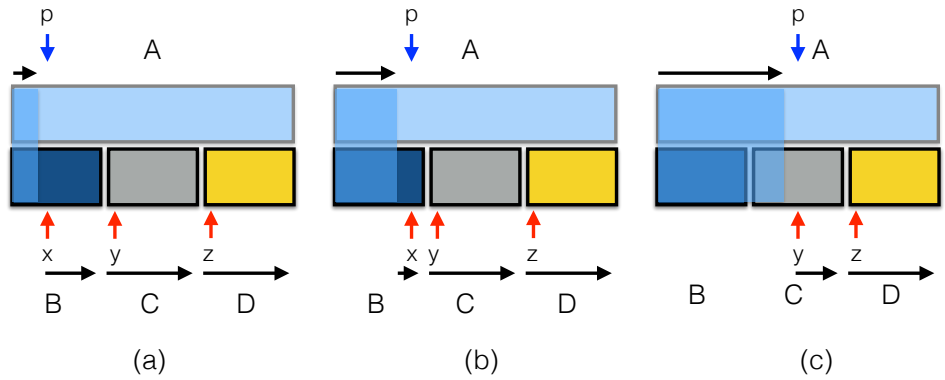

Figure 3: Three snapshots showing the progress of *Selection Sort* applied to the vector `A`. In standard Selection Sort, we search all of the upper block to the right of the target element `p` (indicated by blue down-arrow) to find the next donor element. However, because Square Sort splits `A` into three separately-sorted blocks `B`, `C` and `D`, we only need to examine the smallest remaining element from each block (i.e., `x`, `y` or `z`, indicated by red up-arrows) to find the donor element.

Now suppose the inner loop has now chosen the donor element to be element `y` from block `C`, instead of element `x` from block `B` which sits in the exact location of current target `p`. We now swap the values of elements `A[x]` and `A[y]` in order to move the smallest remaining element to target position `p`, which is an $O(1)$ operation. Clearly, this swap must *decrease* the value of `A[x]` or we would never have exchanged the two elements! Thus, we can guarantee that block `B` is still sorted and ready for the next iteration of Selection Sort. However, the same swap must have *increased* the value of element `A[y]`, so we don't know whether donor block `C` is still sorted because the relation between `A[y]` and the rest of the block is currently unknown. Fortunately, every block including `C` was completely sorted at the start of this iteration of Selection Sort, so the rest of the elements from donor block `C` must still be sorted. Thus, we can repair the sorting of donor block `C` by making a single call to the `insertRight` function defined above at a cost of $O(\sqrt{N})$.

We have now established that both tasks from the inner loop for Selection Sort, i.e., finding the donor element and later repair of the sorting of the donor block, are $O(\sqrt{N})$ operations. Thus the complexity of both Square Sort Step 2 and the entire Square Sort algorithm must be $O(N\sqrt{N})$.

3.3.1 Implementation details for Square Sort Step 2

Each iteration of Selection Sort's outer loop advances the target position `p` one step forward through the vector `A`. But note that vector `A` is partitioned into blocks, and those blocks can classified into three categories:

1. Zero or more *completed blocks*, which have been "completely absorbed" into the final sorted region, i.e., block `B` in Figure 3(c). Completed blocks no longer have an independent existence, since every element already contains its final value, so the completed blocks will be ignored during all future iterations of the inner-loop search for the smallest-remaining donor element.

2. A single *active block*, which contains the target element for the current iteration (indicated by blue down-arrow) as its smallest-remaining element, i.e., block `B` in Figures 3(a)–(b) and block `C` in Figure 3(c). Thus, the position of the smallest-remaining element in the active block (indicated by red up-arrow) moves one step to the right at each iteration of Selection Sort's the outer loop until the block has been completely absorbed into the final sorted region.

3. One or more *donor blocks*, which are now still completely outside the final vector, i.e, both blocks `C` amd `D` in Figures 3(a)–(b) and only block `D` in Figure 3(c). (Note that we can skip the case of zero donor blocks because the entire vector must be sorted as soon as the last block becomes active.) The *position* of the smallest-remaining element in each donor bloc until the block has been completely absorbed into the final sorted region. cannot change from its initial value at the start of step 2. However, the *value stored* in this position may increase every time this doner block participates in a swap.

Therefore, you will find it easier to split up Selection Sort's outer loop into two nested loops. The primary loop chooses which block will be active, while the secondary loop chooses the next target element from the active block. Since Selection Sort's inner loop will be inside the secondary loop, it is easy to determine the range of donor blocks (i.e., from one beyond the active block to the end).

4 Cubic Sort Algorithm

Cubic Sort is a multi-step algorithm for sorting a vector of N elements. Unlike Square Sort, howewever, Cubic Sort decomposes the vector into two-level hierarchy of regular blocks within superblocks as shown in Figure 4.

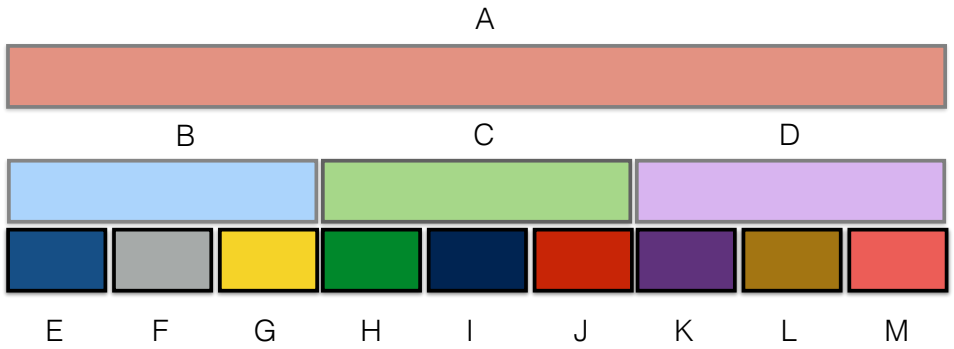

Figure 4: Vector `A` is first split into the sequence of three *superblocks* `B` – `D`. Then each superblock is further split into a sequence of three regular blocks (e.g., `B` is split into `E` – `G`).

4.1 Step 0: Partition the vector into $N/\sqrt[3]{N}$ blocks grouped into $\sqrt[3]{N}$ superblocks

Follow the same general logic as Section 3.1 to determine the target values for block size, number of blocks per superblock, and total number of superblocks. You should try to keep all of these values as close as possible to size $\sqrt[3]{N}$. HINT: You should modify the definition of the index vector `left` to become a *vector-of-vectors*, where we can say that element `A[j]` from the data vector:

- belongs to superblock `k` if

  $$\texttt{left[k][0]} \leq \texttt{j} < \texttt{left[k+1][0]}$$

- belongs to the `i`th block within superblock `k` if

  $$\texttt{left[k][i]} \leq \texttt{j} < \texttt{left[k][i+1]}$$

To simplify the formulas shown in the remainder of this document, I will assume that $N \equiv s^3$ is a perfect cube. In this case, I am assuming that Cubic Sort partitions the initial vector `A` into a total of s^2 regular blocks, which are in turn grouped into s superblocks. However, **your program must handle the general case where N is *not* a perfect cube** by suitably adjusting the sizes of various blocks and superblocks.

4.2 Step 1: Sort each block, then apply one exchange step per superblock

Apply the Reverse Insertion Sort function you created in section 3.2 separately to each of the bottom-level regular blocks. Since this requires running s^2 copies of an $O(N^2)$ sorting algorithm, each working on a block of size s, the total cost of Step 1 will be $O(s^4) \equiv O(N\sqrt[3]{N})$.

We now extract the code from a *single iteration of Selection Sort*, as described in Section 3.3, and use it to create a function called `updateTarget`, which can be applied to any individual superblock. More specifically, using superblock `D` (which includes blocks `K`, `L` and `M`) as an example, `updateTarget` must:

- Identify the target element `s` as the *location* of the smallest remaining element `u` from the active block `K` for its superblock `D`, at cost $O(1)$.
- Examine the first elements from each block (i.e., `u`, `v` and `w`) within its superblock `D`, at a cost of $O(s)$, to determine which of those elements is the smallest, say element `w` which makes `M` the donor block.
- swap `A[u]` and `A[w]` to move this superblock's smallest value to the target location, at cost $O(1)$.
- call the `insertRight` function to restore the sorting of donor block `M`, at a further cost of $O(s)$.

Clearly the total cost of executing `updateTarget` on a single superblock must be $O(s)$. Thus, since we have a total of s superblocks, this additional preprocessing of the superblocks adds a total cost of $O(s^2)$ to Cubic Sort Step 1, which is insignificant compared to the $O(s^4)$ cost of sorting every individual block.

4.3 Step 2: Apply Hierarchical Selection Sort to the preprocessed superblocks

At the end of Cubic Sort Step 1, the current situation will match the structure shown in Figure 5, and we must execute a sequence of iterations of a Hierarchical Selection Sort, similar to the ones described in section 3.3 for Square Sort.

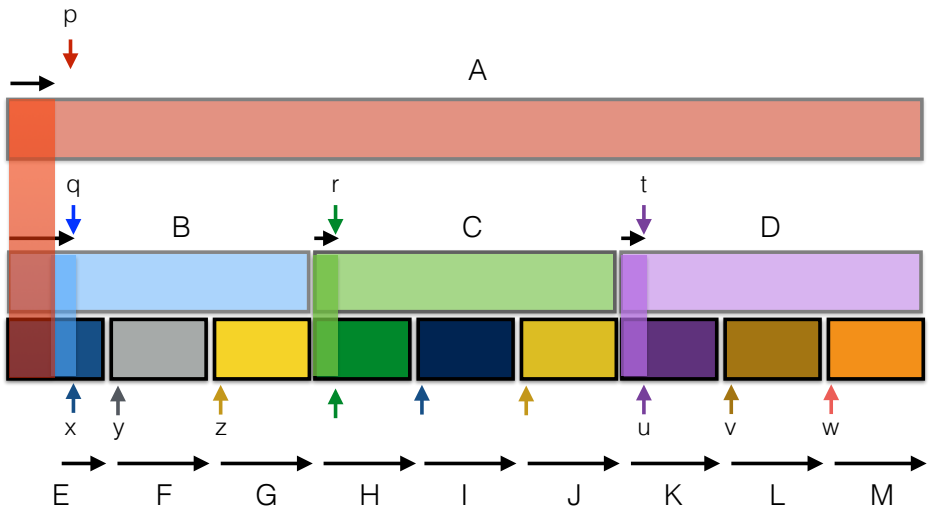

Figure 5: Applying Hierarchical Selection Sort to the problem setup shown in Figure 4. We assume Cubic Sort Step 1 has already been completed, so every block `E` – `M` belonging to any of the three superblocks `B` – `D` has been individually sorted (as indicated by black horizontal arrows below), and the first element from the first block of each superblock (indicated by down-arrows above) contains the globally-smallest value within that superblock. Black horizontal arrow above tall rectangular region at left marks the initial part of vector `A` that is already completely sorted.

For iteration `p`, $0 \leq$ `p` $<$ `size(A)`, we call `A[p]` the *target element*, and mark it by a red down-arrow in Figure 5. We assume that the elements `A[0]` through `A[p-1]` (if any) already contain the globally-smallest values from vector `A` in sorted order. (These elements are indicated by a vertical band of red shading topped by a black horizontal arrow.)

The purpose of iteration `p` is to find the *donor element*, which is the globally smallest-remaining element from vector `A` between element `p` and the end, and swap the donor element into position `A[p]`. Because of the preprocessing of superblocks carried out during Step 1 (and additional steps below to preserve this property for later iterations), we can guarantee that the donor element must be the *smallest remaining element from one of the superbocks* (i.e., one of the three elements `q`, `r` or `t` indicated by colored down-arrows in Figure 5). Thus, finding the donor element has a cost of $O(s)$.

For this discussion, let's assume the donor element is `t` from superblock `D`. Thus, as soon as we can swap `A[p]` and `A[t]`, we know that the range of globally-sorted elements now includes `A[0]` through `A[p]`,

as required for this iteration. However, we are not yet finished with the current iteration because several cleanup steps are required to restore the structure of the data to match Figure 5 in preparation for the next iteration. More specifically:

1. Swapping the values of `A[p]` and `A[t]` had the unwanted effect of replacng the smallest element in block `K` by a larger value. Thus, we must apply the `insertRight` function to restore the sorting of block `K`, at a cost of $O(s)$.

2. Once we execute step 1, we know that all the blocks belonging to donor superblock `D` are once-again properly sorted, be we don't know whether element `t` contains the smallest element in the superblock until we apply `updateTarget` to this superblock, at a further cost of $O(s)$.

3. Now consider the block `E` whose first element `x` was in the same position as the target element `p` for this iteration. Since this element is being absorbed into the range of globally-sorted elements at the end of the current iteration, both the top-level index `p` and block level index `x` need to shift one step towards the right.

   (a) *If block* `E` *still exists after the loss of its first element*, then the right-shifted index `x` must point to its smallest remaining value because we have assumed that all blocks are always maintained in sorted order. However, we cannot assume that the previously second-smallest element of block `E` is the the globally-smallest element in the entire superblock `B`.

   (b) Conversely, *if element* `x` *was the last element remaning in block* `E`, then we have no idea which of the remaing blocks `F` or `G` from superblock `B` contains its gobally-smallest element.

   Thus, we must apply `updateTarget` to target superblock `B` to make sure its globally-smallest value is located in the first element of its first block, at a cost of $O(s)$.

In summary, the outer loop of Hierarchical Selection Sort is executed once for every element in vector `A`, so the total number of iterations is s^3 (or N). Within each iteration of the outer loop we must complete a fixed number of tasks, each of which has cost $O(s)$, so the total cost of executing Cubic Sort Step 2 must be $O(s^4)$. Since we already showed that the total cost of execution for Cubic Sort Step 1 is also $O(s^4)$, the cost of executing the entire Cubic Sort algorithm must be $O(s^4) \equiv O(N\sqrt[3]{N})$.

4.4 Does Cubic Sort really need to be that complicated?

At this point, you are probably confused by all the tricky details in Cubic Sort and wondering whether all the drama is really necessary to reach the desired complexity of $O(N\sqrt[3]{N}$. Unfortunately, the answer is yes: the "easy" way to use Square Sort to sort the three-level hierarchy shown in Figure 4 is actually worse than using ordinary Square Sort!

Suppose we applied ordinary Square Sort separately to each of the s superblocks shown in Figure 4. Since each superblock contains s^2 elements, and Square Sort is an $O(N\sqrt{N})$ algorithm, the complexity of this step will be s copies multiplied by $O(s^3)$ cost per step, or $O(s^4)$, which looks good so far, but.....

We have now created a problem setup that looks remarkably similar to Figure 3, except that the ratio of sizes is wrong: instead of b blocks of size b (where $b = \sqrt{N}$), we have s blocks of size s^2 (where $s = \sqrt[3]{N}$). As a result, Square Sort Step 2 still makes $N \equiv s^3$ iterations of the outer loop, and the cost of finding the donor at each iteration is only $O(s)$. However, this time the cost of executing `insertRight` to restore the storing of the donor block after a swap is $O(s^2)$ because the fully-sorted superblocks are so large. Thus the total cost of Square Sort Step 2 applied to the superblocks must be $O(s^5) \equiv O\left(N(\sqrt[3]{N})^2\right)$, which is considerably larger than ordinary Square Sort's value of $O(N\sqrt{N})$.

To put these differences in complexity into context, let us revisit to our numerical example from the beginning of section 3, where we assume $N = 1024 \equiv 2^{10} \approx 10^3$. Recall that the running time for a fast $O(N \log N)$ algorithm would be $10 \times 2^{10} \approx 10,000$, while running time for Square Sort would be 2^{15} (i.e., about 32,000). Since $\sqrt[3]{1024} \approx \sqrt[3]{1000} \equiv 10$, the running time for Cubic Sort would be very close to an $O(N \log N)$ algorithm, whereas the "lazy" two-level Square Sort algorithm described in this section with complexity $O\left(N(\sqrt[3]{N})^2\right)$ would have a running time of about 100,000.

Let me caution you, however, that this one numerical example does not prove that Cubic Sort is almost as good as a fast $O(N \log N)$ sorting algorithm! To see this, consider what happens to the running times if we increase the problem size to $N = 2^{20} \approx 10^6$, where we find $\log N = 20$, $\sqrt{N} = 2^{10} \approx 10^3$, and $\sqrt[3]{N} \approx 10^2$.

5 What to turn in

Turn in one program that implements both the Square Sort and Cubic Sort algorithms, and carries out the series ot test cases as described above in Section 2.