CSCE 221 Spring 2021

Programming Assignment 8

Sorting

Approved Includes

```
<functional>
<iostream>
<vector>
<utility>
"heap.h"
"sorts.h"
```

Code Coverage

You must submit a test suite for each task that, when run, covers at least 90% of your code. You should, at a minimum, invoke every function at least once. Best practice is to also check the actual behavior against the expected behavior, e.g. verify that the result is correct. You should be able to do this automatically, i.e. write a program that checks the actual behavior against the expected behavior.

Your test suite should include ALL tests that you wrote and used, including tests you used for debugging. You should have MANY tests.

Starter Code

```
sorts.cpp
sorts.h
sorts_tests.cpp
Makefile
```

Files to Submit

```
heap.h (you created this in PA 7)
sorts.h
sorts_tests.cpp
```

Task Overview

Implement each of these 7 sorting algorithms.

Requirements for all Algorithms

Files

`sorts.h` - contains the template definitions
`sorts_tests.cpp` - contains the test cases and test driver (main)

Ascending Order, Pass by Reference, and Output

All algorithms should modify the input container (`std::vector`) to be sorted in ascending order. Print to standard output (i.e. `std::cout`) the contents of the container before the sort begins (the value of the input) and again after each pass. Some sorts also print auxiliary container values, too.

Note: You could make the algorithms more generic by including a parameter for a comparator. *You could also parameterize the container type. These are not required for this assignment, but are skills that I expect you to have developed by now.*

Insertion Sort

```
template <class Comparable>
void insertion_sort(std::vector<Comparable>&)
```

Sorts the given container using insertion sort. See Chapter 7.2 in the textbook.

Example

Input

```
{81,94,11,96,12,35,17,95,28,58,41,75,15}
```

Output

`[81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15]` ← this is the initial container value
`[81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15]` ← this is after the 1st pass, putting 94 in the correct place.
`[11, 81, 94, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15]` ← this is after the second pass, putting 11 in the correct place.
`[11, 81, 94, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15]`
`[11, 12, 81, 94, 96, 35, 17, 95, 28, 58, 41, 75, 15]`
`[11, 12, 35, 81, 94, 96, 17, 95, 28, 58, 41, 75, 15]`
`[11, 12, 17, 35, 81, 94, 96, 95, 28, 58, 41, 75, 15]`
`[11, 12, 17, 35, 81, 94, 95, 96, 28, 58, 41, 75, 15]`
`[11, 12, 17, 28, 35, 81, 94, 95, 96, 58, 41, 75, 15]`
`[11, 12, 17, 28, 35, 58, 81, 94, 95, 96, 41, 75, 15]`
`[11, 12, 17, 28, 35, 41, 58, 81, 94, 95, 96, 75, 15]`
`[11, 12, 17, 28, 35, 41, 58, 75, 81, 94, 95, 96, 15]`
`[11, 12, 15, 17, 28, 35, 41, 58, 75, 81, 94, 95, 96]`

Shell Sort

```
template <class Comparable>
void shell_sort(std::vector<Comparable>&)
```

Sorts the given container using Shell sort with Hibbard's increment sequence. See Chapter 7.4 in the textbook.

Example

Input

```
{81,94,11,96,12,35,17,95,28,58,41,75,15}
```

Output

```
[81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15] ← the input values
[81, 28, 11, 41, 12, 15, 17, 95, 94, 58, 96, 75, 35] ← after the first pass, h3 sorted, for h3 = 7
[17, 12, 11, 35, 28, 15, 41, 95, 75, 58, 96, 94, 81] ← after the second pass, h2 sorted, for h2 = 3
[11, 12, 15, 17, 28, 35, 41, 58, 75, 81, 94, 95, 96]
```

Heap Sort

```
template <class Comparable>
void heap_sort(std::vector<Comparable>&)
```

Sorts the given container using heap sort. See Chapter 7.5 in the textbook.

Print the initial heap, then print the heap followed by the container after each remove-and-append step.

Since the order of the heap matters for the I/O test, you should use the following tie-break when both kids are equal in the heap and you need to swap with one of them: **swap with the left child**.

Example

Input

```
{81,94,11,96,12,35,17,95,28,58,41,75,15}
```

Output

`[81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15]` ← the initial input
`[0, 11, 12, 15, 28, 41, 35, 17, 95, 96, 58, 94, 75, 81]` ← the initial values on the heap (auxiliary data structure)
`[0, 12, 28, 15, 81, 41, 35, 17, 95, 96, 58, 94, 75]` ← the heap after removing the first value
`[11]` ← the container after inserting the first value removed from the heap
`[0, 15, 28, 17, 81, 41, 35, 75, 95, 96, 58, 94]` ← the heap after removing the second value from the heap and appending it to the container.
`[11, 12]` ← the container after removing the second value from the heap and appending it to the container.

```
[0, 17, 28, 35, 81, 41, 94, 75, 95, 96, 58]
[11, 12, 15]
[0, 28, 41, 35, 81, 58, 94, 75, 95, 96]
[11, 12, 15, 17]
[0, 35, 41, 75, 81, 58, 94, 96, 95]
[11, 12, 15, 17, 28]
[0, 41, 58, 75, 81, 95, 94, 96]
[11, 12, 15, 17, 28, 35]
[0, 58, 81, 75, 96, 95, 94]
[11, 12, 15, 17, 28, 35, 41]
[0, 75, 81, 94, 96, 95]
[11, 12, 15, 17, 28, 35, 41, 58]
[0, 81, 95, 94, 96]
```

```
[11, 12, 15, 17, 28, 35, 41, 58, 75]
[0, 94, 95, 96]
[11, 12, 15, 17, 28, 35, 41, 58, 75, 81]
[0, 95, 96]
[11, 12, 15, 17, 28, 35, 41, 58, 75, 81, 94]
[0, 96]
[11, 12, 15, 17, 28, 35, 41, 58, 75, 81, 94, 95]
[0]
[11, 12, 15, 17, 28, 35, 41, 58, 75, 81, 94, 95, 96]
```

Merge Sort

```
template <class Comparable>
void merge_sort(std::vector<Comparable>&)
```

Sorts the given container using merge sort. See Chapter 7.6 in the textbook.

When dividing an odd number of elements into halves, the left half should get the extra element, e.g. for 5 elements, the left half would have 3 elements and the right half would have 2 elements.

Print the container at the end of the algorithm (after merging).

Example

Input

```
{81,94,11,96,12,35,17,95,28,58,41,75,15}
```

Output

```
[81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15] ← initial container
[81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15] ← container after sorting [81] and [94] and merging into [81, 94]
[81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15] ← container after sorting [11] and [96] and merging into [11, 96]
[11, 81, 94, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15] ← container after sorting [81, 94] and [11, 96] and merging into [11, 81, 94, 96]
[11, 81, 94, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15]
[11, 81, 94, 96, 12, 17, 35, 95, 28, 58, 41, 75, 15]
[11, 12, 17, 35, 81, 94, 96, 95, 28, 58, 41, 75, 15]
[11, 12, 17, 35, 81, 94, 96, 28, 95, 58, 41, 75, 15]
[11, 12, 17, 35, 81, 94, 96, 28, 58, 95, 41, 75, 15]
[11, 12, 17, 35, 81, 94, 96, 28, 58, 95, 41, 75, 15]
[11, 12, 17, 35, 81, 94, 96, 28, 58, 95, 15, 41, 75]
[11, 12, 17, 35, 81, 94, 96, 15, 28, 41, 58, 75, 95]
[11, 12, 15, 17, 28, 35, 41, 58, 75, 81, 94, 95, 96]
```

Quick Sort

```
template <class Comparable>
void quick_sort(std::vector<Comparable>&)
```

Sorts the given container using quick sort. See Chapter 7.7 in the textbook.

Use the median of three between the first, middle (the right-middle when the number of elements is even), and last elements to choose the pivot value. Only swap pivot (median) and end when finding the pivot and preparing to partition (**don't** do what the book does, let partition do the rest of the moving).

Assuming a < b, here is how to break ties:

```
    value of        index of
first middle last |  pivot
------------------+----------
  a      a     a   |  middle
  a      a     b   |  first
  a      b     a   |  first
  b      a     a   |  middle
```

When you have a 3-way tie, pick the middle. When you have a 2-way tie, pick the lesser indexed value (first or middle).

When partitioning, do the swaps on elements equal to the pivot value to get more balanced subcontainers (i.e. <= goes to left, and >= goes to right).

When the number of elements to sort is 10 or fewer, delegate the remainder of the sort to insertion sort.

Print the container at the beginning of the recursive quicksort algorithm (and also at the beginning and after each pass of insertion sort).

Example

Input

```
{81,94,11,96,12,35,17,95,28,58,41,75,15}
```

Output

`[81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15]` ← initial container
`[15, 12, 11, 17, 94, 35, 81, 95, 28, 58, 41, 75, 96]` ← the pivot value is 17. this is the container after the partition step (beginning of recursive quicksort left step), which is then the initial container sent to insertion sort (to sort [15, 12, 11])
`[15, 12, 11, 17, 94, 35, 81, 95, 28, 58, 41, 75, 96]` ← initial container in insertion sort

`[12, 15, 11, 17, 94, 35, 81, 95, 28, 58, 41, 75, 96]`
`[11, 12, 15, 17, 94, 35, 81, 95, 28, 58, 41, 75, 96]` ← end of insertion sort on left half
`[11, 12, 15, 17, 94, 35, 81, 95, 28, 58, 41, 75, 96]` ← initial container in recursive quicksort right step
`[11, 12, 15, 17, 94, 35, 81, 95, 28, 58, 41, 75, 96]` ← beginning of insertion sort on right half (since right half has 9 elements: [94, ..., 96])
`[11, 12, 15, 17, 35, 94, 81, 95, 28, 58, 41, 75, 96]`
`[11, 12, 15, 17, 35, 81, 94, 95, 28, 58, 41, 75, 96]`
`[11, 12, 15, 17, 35, 81, 94, 95, 28, 58, 41, 75, 96]`
`[11, 12, 15, 17, 28, 35, 81, 94, 95, 58, 41, 75, 96]`
`[11, 12, 15, 17, 28, 35, 58, 81, 94, 95, 41, 75, 96]`
`[11, 12, 15, 17, 28, 35, 41, 58, 81, 94, 95, 75, 96]`
`[11, 12, 15, 17, 28, 35, 41, 58, 75, 81, 94, 95, 96]`
`[11, 12, 15, 17, 28, 35, 41, 58, 75, 81, 94, 95, 96]`

Input

`{30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1}`

Output

`[30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]` ← initial array
`[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]` ← after partition around 15 (beginning of quicksort left: [2, …, 1])
`[1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 3, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]` ← after partition around 2 (beginning of quicksort left: [1])
`[1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 3, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]` ← beginning of quicksort right [4, …, 3]
`[1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 5, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]` ← after partition around 4 (beginning of quicksort left: [3])
`[1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 5, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]` ← beginning of quicksort right: [6, …, 5]

```
[1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 5, 15, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]
```
← beginning of insertion sort of [6, …, 5]

```
[1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 5, 15, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]
[1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 5, 15, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]
[1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 5, 15, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]
[1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 5, 15, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]
[1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 5, 15, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]
[1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 5, 15, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]
[1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 5, 15, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]
[1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 5, 15, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 16]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 18]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 18]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 20]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 20]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 22]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 22]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 22]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 22]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 22]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 22]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 22]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 22]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 22]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 22]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
```

Bucket Sort

```
void bucket_sort(std::vector<unsigned>&)
```

Sorts the given container using bucket sort. See Chapter 7.11 in the textbook.

Note that this bucket sort can only be used to sort non-negative integers.

Print the counting array after sorting into buckets. Then, print the container after dumping each bucket into the container (print once per unique value).

Example

Input

```
{8,1,9,4,1,1,9,6,1,2,3,5,1,7,9,5,2,8,5,8,4,1,7,5,1,5}
```

Output

```
[8, 1, 9, 4, 1, 1, 9, 6, 1, 2, 3, 5, 1, 7, 9, 5, 2, 8, 5, 8, 4, 1, 7, 5, 1, 5] ← the initial container
[0, 7, 2, 1, 2, 5, 1, 2, 3, 3] ← the counting array (buckets) after the first step.
[1, 1, 1, 1, 1, 1, 1] ← the container after inserting the 1s
[1, 1, 1, 1, 1, 1, 1, 2, 2] ← the container after inserting the 2s.
[1, 1, 1, 1, 1, 1, 1, 2, 2, 3]
[1, 1, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4]
[1, 1, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4, 5, 5, 5, 5, 5]
[1, 1, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4, 5, 5, 5, 5, 5, 6]
[1, 1, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4, 5, 5, 5, 5, 5, 6, 7, 7]
[1, 1, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4, 5, 5, 5, 5, 5, 6, 7, 7, 8, 8, 8]
[1, 1, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4, 5, 5, 5, 5, 5, 6, 7, 7, 8, 8, 8, 9, 9, 9]
```

Radix Sort

```
template <class Comparable>
void radix_sort(std::vector<Comparable>&)

void radix_sort(std::vector<std::string>& array)
```

Sorts the given container using radix sort. See Chapter 7.11 in the textbook.

For integers, use 10 buckets (base 10) and sort from least to highest significance place (last digit to first).

For strings, use 128 buckets (the “non-extended” ASCII table) and sort lexicographically (dictionary order), right padding with null characters, from back to front (last character to first).

Print the bucket contents and the reconstituted container after each pass (each place or index).

Examples

Input

```
{64, 8, 216, 512, 27, 729, 0, 1, 343, 125}
```

Output

`[64, 8, 216, 512, 27, 729, 0, 1, 343, 125]` ← initial container
`[0] [1] [512] [343] [64] [125] [216] [27] [8] [729]` ← buckets after sorting by 1s place
`[0, 1, 512, 343, 64, 125, 216, 27, 8, 729]` ← container after dumping buckets back in.
`[0, 1, 8] [512, 216] [125, 27, 729] [] [343] [] [64] [] [] []` ← buckets after sorting by 10s place
`[0, 1, 8, 512, 216, 125, 27, 729, 343, 64]` ← container after dumping buckets back in.
`[0, 1, 8, 27, 64] [125] [216] [343] [] [512] [] [729] [] []`
`[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]`

Input

```
{"64","8","216","512","27","729","0","1","343","125"}
```

Output

`["64", "8", "216", "512", "27", "729", "0", "1", "343", "125"]` ← initial container

["64", "8", "27", "0", "1"] [] [] [] [] [] [] [] [] [] [] [] [] [] []
[] []
[] [] [] [] [] [] [] [] [] [] [] [] ["512"] ["343"] [] ["125"] ["216"]
[] [] ["729"] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []
[] []
[] []
[] [] [] [] [] ← buckets after sorting by character in index 2 (null for strings of length less than 3).
["64", "8", "27", "0", "1", "512", "343", "125", "216", "729"] ← container after dumping buckets back in
["8", "0", "1"] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []
[] []
[] [] [] [] [] [] [] ["512", "216"] ["125", "729"] [] ["64", "343"] []
[] ["27"] []
[] []
[] []
[] [] [] [] [] [] ← buckets after sorting by character in index 1 (null for strings of length less than 2)
["8", "0", "1", "512", "216", "125", "729", "64", "343", "27"] ← container after dumping buckets back in.
[] []
[] []
[] [] ["0"] ["1", "125"] ["216", "27"] ["343"] [] ["512"] ["64"]
["729"] ["8"] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []
[] []
[] []
[] [] [] [] [] []
["0", "1", "125", "216", "27", "343", "512", "64", "729", "8"] ← Note how lexicographic order differs from numeric order: "512" < "64"