Asymptotic Analysis Project

COT 4400, Spring 2016

February 25, 2016

1 Overview

This project asks you to implement four sorting algorithms in C++ or Java. You will then run your algorithm data of different size and type, and analyze these results to see how their run times relate to their theoretical asymptotic analysis.

2 Algorithms

Implement the following sorting algorithms in the file `Sorting.hpp` or `Sorting.java`. For full credit, each function must:

- Implement the algorithms as described Section 5.1, using the given function signatures.
- Be generic (i.e., a function that takes as input an array of any type of item). You may assume that the item type overloaded the comparison operators (i.e., you may use the $<$ and $>$ operators on the array of items).
- Compile with no errors. If your code fails to compile on the C4 Linux Lab machines, you will receive major penalties on other sections of this project.
- Correctly sort the given input data.
- Be efficient (i.e., implementations that take too long to solve sample problems will be assessed a penalty)

- Be readable and easy to understand. You should include comments to explain when needed, but you should not include excessive comments that makes the code difficult to read.
- (*C++ only*) Have no memory leaks.
- (*Java only*) All of your functions should be defined as static member functions of the class Sorting.

Your final submission *should not* include a `main` function. However, you may implement additional helper functions as needed. Helper functions must be located in the same file.

You will be provided with source files that include a `main` function, as well as several helper functions that you should use. You should not include either of these files in your code submission, though you may invoke their functions (and `#include` them).

2.1 Algorithm pseudocode

Pseudocode descriptions for the four algorithms appear at the end of this document. Note, array indices are given with base of 0 to reduce confusion in implementation. Also, due to differences between the languages, two method signatures and pseudocode algorithms are provided for QuickSort and MergeSort. The pseudocode implementations QuickSortJ and MergeSortJ correspond to the Java method signatures and function implementations.

Your implementation of QuickSort (in either case) will use the median-of-three strategy for selecting the *pivot* element. Other strategies for pivot selection, such as choosing the first, the last, or a random element, have different advantages and disadvantages, though we will not compare the various pivot strategies in this project.

2.2 C++ method signatures

```
template <class T> void selectionsort(T* data, int size)
template <class T> void insertionsort(T* data, int size)
template <class T> void mergesort(T* data, int size, T* temp)
template <class T> void quicksort(T* data, int size)
```

2.3 Java method signatures

```
public static <T extends Comparable<T>> void selectionsort(T[] data)
public static <T extends Comparable<T>> void insertionsort(T[] data)
public static <T extends Comparable<T>> void mergesort(T[] data,
    int left, int right, T[] temp)
public static <T extends Comparable<T>> void quicksort(T[] data,
    int left, int right)
```

2.4 Argument descriptions

- `data`: the list of elements to sort; must be comparable
- `size`: the number of elements in items
- `left`: the index of the first element in the array (0 initially)
- `right`: the index after the last element in the array (n initially)

3 Project report

Your project report should be divided into two parts, Results and Analysis. For the Results section, you will need to prepare a data file describing the performance of your algorithms, and you will need to prepare four tables describing their complexity, as well as a response to these results, in the Analysis portion.

3.1 Results

Run each of the four sorting algorithms on constant, sorted, and random constant arrays of sizes 1,000–100,000 (according to the table below), and record the timing results. Timing results should be reported to the nearest millisecond. You should use the provided helper functions to construct the input arrays. Runs that take more than 15 minutes (900 seconds) can be terminated early; you should record such entries as "too long" or "TL".

n
1,000
1,500
3,000
5,500
10,000
15,000
30,000
55,000
100,000

Note, you may need to ensure that you have sufficient stack space before testing QuickSort to ensure that you do not run out ("stack overflow"). If you are using C++, you can run `ulimit -s unlimited` in Linux before executing the algorithm to increase the available stack space. For Java, you can pass the `-Xss[size]` argument to define a new stack size (e.g., `-Xss1024m` gives 1024 MB of stack space).

You should enter your results into a comma-separated value (CSV) file. The CSV file should contain 13 columns, with 10 rows. Your first column should list the data sizes for your experiment in rows 2–10, while the first row should label the 12 different experiments in columns 2–13. Your column labels should include the algorithm name (SelectionSort, InsertionSort, MergeSort, or Quicksort) and input type (Sorted, Random, or Constant). You may abbreviate these labels as S, I, M, Q, and S, R, C. (For example, MS represents your MergeSort result on a sorted array.) An example table appears below. You may use Excel (or any other software) to prepare your data.

	SC	SS	SR	IC	IS	IR	MC	MS	MR	QC	QS	QR
1000												
1500												
3000												
5500												
10000												
15000												
30000												
55000												
100000												

3.2 Analysis

In this section, you will estimate the complexity of the four algorithms by taking a ratio between the largest and smallest runs for which you were able

to get timing data, excluding runs that were recorded as 0 ms and runs that were too long (> 15 minutes). You should compare your ratio with the ratios for linear, $n \lg n$, and quadratic functions and label each experiment according to the behavior it most accurately represents. Function values for n, $n \lg n$, and n^2 are provided in the table below:

n	Linear	$n \lg n$	Quadratic
1,000	1,000	9,966	1,000,000
1,500	1,500	15,826	2,250,000
3,000	3,000	34,652	9,000,000
5,500	5,500	68,339	30,250,000
10,000	10,000	132,877	100,000,000
15,000	15,000	208,090	225,000,000
30,000	30,000	446,180	900,000,000
55,000	55,000	866,093	3,025,000,000
100,000	100,000	1,660,964	10,000,000,000

For example, if your smallest nonzero run was $n = 3000$ and your largest run less than 15 minutes was $n = 30000$, the linear, $n \lg n$, and quadratic ratios would be $30000/3000 = 10$, $446180/34652 \approx 12.88$, and $900000000/9000000 = 100$, respectively. You should label the algorithm based on which of these three ratios is most similar to your own timing ratio.

For each of the 4 algorithms, you should create a small chart that includes the min nonzero value of n ($t_{\min}$), the largest value of n that took less than 15 minutes ($t_{\max}$), the timing ratio ($f(t_{\max})/f(t_{\min})$), and the estimated behavior of this function (linear, $n \lg n$, or quadratic), for all three input types (constant, sorted, or quadratic). An example chart for QuickSort appears below:

QuickSort				
	$t_{\max}$	$t_{\min}$	Ratio	Behavior
Constant	100000	10000	10.1	Linear
Sorted	100000	1500	101.85	$n \lg n$
Random	30000	1000	975	Quadratic

You should then write a summary of (1) how your results compare to the theoretical analysis for the four algorithms (below), and (2) why your results make sense or are surprising. You should spend more time explaining your results when they are unusual or unexpected.

	Best-case complexity	Average-case complexity	Worst-case complexity
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
InsertionSort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
MergeSort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
QuickSort	$\Omega(n \lg n)$	$\Theta(n \lg n)$	$O(n^2)$

4 Submission

For this project, you should submit a zip archive containing (1) your code (in Sorting.hpp or Sorting.java) containing the four sorting functions, (2) a CSV file containing your results (described in Section 3.1), and (3) your tables and analysis (described in Section 3.2), in PDF format.

Note: This is an individual project. You are not allowed to submit work that has been pulled from the Internet, nor work that has been done by your peers. Your submitted materials will be analyzed for plagiarism.

5 Grading

Algorithm implementations (4)	10 points, each
Data file containing results	20 points
Complexity tables (4)	6 points, each
Analysis	16 points

Requirements for each portion of the grade are described in Sections 2, 3.1, and 3.2.

5.1 Pseudocode

Input: *data*: the items to sort (must be comparable)
Input: n: the number of elements in *data* (*data.length* for Java)
Output: permutation of items such that $data[0] \leq \ldots \leq data[n-1]$

```
1 Algorithm: SelectionSort
2 for i = 0 to n do
3 |   Let m be the location of the min value in the array data[i..n];
4 |   Swap data[i] and data[m];
5 end
6 return data;
```

Input: *data*: the data to sort (must be comparable)
Input: n: the number of elements in *data* (*data.length* for Java)
Output: permutation of data such that $data[0] \leq \ldots \leq data[n-1]$

```
1 Algorithm: InsertionSort
2 for i = 1 to n − 1 do
3 |   Let j = location of predecessor of data[i] in data[0..i − 1], or -1 if
  |   data[i] is min;
4 |   Shift data[j + 1..i − 1] to the right one space;
5 |   data[j + 1] = ins;
6 end
```

```
   Input: data: the data to sort (must be comparable)
   Input: n: the number of elements in data
   Input: temp: temporary array of size n for use during MergeSort
   Output: a permutation of data such that data[0] ≤ ... ≤ data[n − 1]
 1 Algorithm: MergeSort
 2 if n > 1 then
 3 |   mid =floor((n + 1)/2);
 4 |   left = data[0..mid − 1];
 5 |   right = data[mid..n − 1];
 6 |   Call MergeSort(left, mid, temp[0..mid − 1]);
 7 |   Call MergeSort(right, n − mid, temp[mid..n − 1]);
 8 |   ℓ = r = s = 0;
 9 |   while ℓ < mid and r < n − mid do
10 |   |   if left[ℓ] < right[r] then
11 |   |   |   temp[s] = left[ℓ];
12 |   |   |   ℓ = ℓ + 1;
13 |   |   else
14 |   |   |   temp[s] = right[r];
15 |   |   |   r = r + 1;
16 |   |   end
17 |   |   s = s + 1;
18 |   end
19 |   Copy left[ℓ..mid − 1] to temp[s..s + mid − ℓ − 1];
20 |   s = s + mid − ℓ;
21 |   Copy right[r..n − mid − 1] to temp[s..s + n − mid − 1 − r];
22 |   Copy temp[0..n − 1] to data[0..n − 1];
23 end
```

```
   Input: data: the data to sort (must be comparable)
   Input: left: the first element of data to sort
   Input: right: the element after the last element of data to sort
   Input: temp: temporary array of size n for use during MergeSort
   Output: a permutation of data such that
                data[left] ≤ ... ≤ data[right − 1]
 1 Algorithm: MergeSortJ
 2 if right − left > 1 then
 3 |   mid =floor((left + right)/2);
 4 |   Call MergeSortJ(data, left, mid, temp);
 5 |   Call MergeSortJ(data, mid, right, temp);
 6 |   ℓ = left;
 7 |   r = mid;
 8 |   s = 0;
 9 |   while ℓ < mid and r < right do
10 |   |   if data[ℓ] < data[r] then
11 |   |   |   temp[s] = data[ℓ];
12 |   |   |   ℓ = ℓ + 1;
13 |   |   else
14 |   |   |   temp[s] = data[r];
15 |   |   |   r = r + 1;
16 |   |   end
17 |   |   s = s + 1;
18 |   end
19 |   Copy data[ℓ..mid − 1] to temp[s..s + mid − ℓ − 1];
20 |   s = s + mid − ℓ;
21 |   Copy data[r..right − 1] to temp[s..s + right − r];
22 |   Copy temp[0..right − left − 1] to data[left..right − 1];
23 end
```

Input: *data*: the data to sort (must be comparable)
Input: n: the number of elements in data
Output: permutation of data such that $data[1] \leq \ldots \leq data[n]$

```
1  Algorithm: QuickSort
2  if n ≤ 1 then
3  |  return data;
4  mid =floor((n + 1)/2);
5  Let pivot be such that data[pivot] is the median of data[0], data[mid],
   and data[n − 1];
6  Swap data[pivot] and data[0];
7  left = 0;
8  right = n − 1;
9  repeat
10 |  while left < right and data[left] ≤ data[0] do
11 |  |  left = left + 1;
12 |  end
13 |  while left < right and data[right] > data[0] do
14 |  |  right = right − 1;
15 |  end
16 |  Swap data[left] and data[right];
17 until left ≥ right;
18 if data[left] > data[0] then
19 |  left = left − 1;
20 end
21 Swap data[0] and data[left];
22 Call QuickSort on data[0..left − 1];
23 Call QuickSort on data[left + 1..n − 1];
```

```
   Input: data: the data to sort (must be comparable)
   Input: left: the first element in data to sort
   Input: right: the index after the last element of data to sort
   Output: permutation of data such that
                data[left] ≤ ... ≤ data[right − 1]
 1 Algorithm: QuickSortJ
 2 if right − left ≤ 1 then
 3 |   return data;
 4 mid =floor((left + right)/2);
 5 Let pivot be such that data[pivot] is the median of data[left],
   data[mid], and data[right − 1];
 6 Swap data[pivot] and data[left];
 7 ℓ = left;
 8 r = right − 1;
 9 repeat
10 |   while ℓ < r and data[ℓ] ≤ data[left] do
11 |   |   ℓ = ℓ + 1;
12 |   end
13 |   while ℓ < r and data[r] > data[left] do
14 |   |   r = r − 1;
15 |   end
16 |   Swap data[ℓ] and data[r];
17 until ℓ ≥ r;
18 if data[ℓ] > data[left] then
19 |   ℓ = ℓ − 1;
20 end
21 Swap data[left] and data[ℓ];
22 Call QuickSortJ(data, left, ℓ);
23 Call QuickSortJ(data, ℓ + 1, right);
```