Asymptotic Analysis Project

COT 4400, Fall 2015

October 12, 2015

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 2.4, 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 may use. You should not include either of these files in your code submission, though you may invoke their functions (and `#include` them).

2.1 C++ method signatures

```
void selectionsort(T* data, int size)
void insertionsort(T* data, int size)
void mergesort(T* data, int size, T* temp)
void quicksort(T* data, int size)
```

2.2 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)
```

2.3 Arguments

- `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 of the last element in the array ($n-1$ initially)

2.4 Pseudocode

Pseudocode for the four algorithms appears below. Note, array indices are given with base of 0 to reduce confusion in implementation.

Your implementation of QuickSort will use the median-of-three strategy for selecting the *pivot* element, the value used to partition the array. 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.

```
   Input: $data$: the items to sort (must be comparable)
   Input: $n$: the number of elements in items
   Output: permutation of items such that $data[1] \leq \ldots \leq data[n]$
1  Algorithm: SelectionSort
2  for $i = 0$ to $n-1$ do
3      Let $m$ be the location of the min value in the array $data[i..n-1]$;
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
   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
7  return $data$;
```

```
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 $data[1] \leq \ldots \leq data[n]$
Algorithm: MergeSort
if $n > 1$ then
    $mid$ =floor($(n+1)/2$);
    $left$ =MergeSort($data[0..mid-1], mid, temp[0..mid-1]$);
    $right$ =MergeSort($data[mid..n-1], n-mid, temp[mid..n-1]$);
    $\ell = r = s = 0$;
    while $\ell < mid$ and $r < n - mid$ do
        if $left[\ell] < right[r]$ then
            $temp[s] = left[\ell]$;
            $\ell = \ell + 1$;
        else
            $temp[s] = right[r]$;
            $r = r + 1$;
        end
        $s = s + 1$;
    end
    Copy $left[\ell..mid-1]$ to $temp[s..s+mid-\ell]$;
    $s = s + mid - \ell$;
    Copy $right[r..n-mid-1]$ to $temp[s..s+n-mid-1-r]$;
    Copy $temp[0..n-1]$ to $data[0..n-1]$;
end
return $data$;
```

```
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
        $data[left] \leq \ldots \leq data[right-1]$
Algorithm: MergeSortJ
if $n > 1$ then
    $mid$ =floor($(left + right)/2$);
    Call MergeSortJ($data, left, mid, temp$);
    Call MergeSortJ($data, mid, right, temp$);
    $\ell = left$;
    $r = mid$;
    $s = 0$;
    while $\ell < mid$ and $r < right$ do
        if $data[\ell] < data[r]$ then
            $temp[s] = data[\ell]$;
            $\ell = \ell + 1$;
        else
            $temp[s] = data[r]$;
            $r = r + 1$;
        end
        $s = s + 1$;
    end
    Copy $left[\ell..mid - 1]$ to $temp[s..s + mid - \ell]$;
    $s = s + mid - \ell$;
    Copy $right[r..n - mid - 1]$ to $temp[s..s + n - mid - r]$;
    Copy $temp[0..right - left - 1]$ to $data[left..right - 1]$;
end
return $data$;
```

```
Input: $data$: the data to sort (must be comparable)
Input: $n$: the number of elements in data
Output: permutation of data such that $data[1] \leq \ldots \leq data[n]$
Algorithm: QuickSort
if $n \leq 1$ then
    return $data$;
$mid$ =floor($(n+1)/2$);
Let $pivot$ be such that $data[pivot]$ is the median of $data[0]$, $data[mid]$,
and $data[n-1]$;
Swap $data[pivot]$ and $data[0]$;
$left = 0$;
$right = n - 1$;
repeat
    while $left < right$ and $data[left] \leq data[0]$ do
        $left = left + 1$;
    end
    while $left < right$ and $data[right] > data[0]$ do
        $right = right - 1$;
    end
    Swap $data[left]$ and $data[right]$;
until $left \geq right$;
if $data[left] > data[0]$ then
    $left = left - 1$;
end
Swap $data[0]$ and $data[left]$;
Call QuickSort on $data[0..left-1]$;
Call QuickSort on $data[left+1..n-1]$;
return $data$;
```

```
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
        $data[left] \leq \ldots \leq data[right-1]$
Algorithm: QuickSortJ
if $right - left \leq 1$ then
    return $data$;
$mid$ =floor(($left + right$)/2);
Let $pivot$ be such that $data[pivot]$ is the median of $data[left]$,
$data[mid]$, and $data[right-1]$;
Swap $data[pivot]$ and $data[left]$;
$\ell = left$;
$r = right - 1$;
repeat
    while $\ell < r$ and $data[\ell] \leq data[left]$ do
        $\ell = \ell + 1$;
    end
    while $\ell < r$ and $data[r] > data[left]$ do
        $r = r - 1$;
    end
    Swap $data[\ell]$ and $data[r]$;
until $\ell \geq r$;
if $data[\ell] > data[left]$ then
    $\ell = \ell - 1$;
end
Swap $data[left]$ and $data[\ell]$;
Call QuickSortJ($data, left, \ell$);
Call QuickSortJ($data, \ell + 1, right$);
return $data$;
```

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 16 equations that model the performance of your algorithms, as well as a response to these results, in the Analysis portion.

3.1 Results

Test each of the four sorting algorithms on increasing, decreasing, random, and constant arrays of sizes 10,000–100,000, in multiples of 10,000. 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 17 columns, with 11 rows. Your first column should list the data sizes for your experiment in columns 2–11, while the first row should label the 16 different experiments in columns 2–17. Your column labels should include the algorithm name (SelectionSort, InsertionSort, MergeSort, or Quicksort) and input type (Increasing, Decreasing, Random, or Constant). You may abbreviate these labels as S, I, M, Q, and I, D, R, C. (For example, MC represents your MergeSort result on a constant array.) Each cell in the table should represent your median-of-three result for the given experiment. An example table appears below. You may use Excel (or any other software) to prepare your data.

	SI	SD	SR	SC	II	ID	IR	IC	MI	MD	MR	MC	QI	QD	QR	QC
10000																
20000																
30000																
40000																
50000																
60000																
70000																
80000																
90000																
100000																

3.2 Analysis

For this portion of the report, you will want to use a software package capable of manipulating matrices. You may use MATLAB, which is installed on the C4 Lab machines, or you download R (for free) at: `http://www.r-project.org`.

In this section, you will estimate the complexity of the four algorithms using *ordinary least-squares (OLS) regression*. Specifically, you will estimate the median runtime for each algorithm and array combination as a linear combination of four functions: n^2, $n \lg n$, n, and 1, where n is the input size. If you have forgotten (or never learned) how to do this, please see the appendix for instructions. Your results should be in the following form:

Algorithm:	SelectionSort
Increasing:	$S_1(n) \approx a_1 n^2 + b_1 n \lg n + c_1 n + d_1$
Decreasing:	$S_2(n) \approx a_2 n^2 + b_2 n \lg n + c_2 n + d_2$
Constant:	$S_3(n) \approx a_3 n^2 + b_3 n \lg n + c_3 n + d_3$
Random:	$S_4(n) \approx a_4 n^2 + b_4 n \lg n + c_4 n + d_4$

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	$\Theta(n)$	$O(n^2)$	$O(n^2)$
MergeSort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
QuickSort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$O(n^2)$

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 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	15 points, each
Data file containing results	10 points
Regression equations	10 points
Analysis	20 points

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

Appendix: Least-squares regression

Input: set of k points $P : \{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}$, set of m functions $\mathcal{F} : \{f_1(x), f_2(x), \ldots, f_m(x)\}$

Output: Find a set of coefficients $\hat{\beta} = (\beta_1, \beta_2, \ldots, \beta_m)^\mathsf{T}$ such that $\hat{y} = \beta_1 f_1(x) + \beta_2 f_2(x) + \ldots + \beta_m f_m(x)$ best approximates y. In other words, find $\hat{\beta}$ such that $||y - \hat{y}||^2$ is minimized.

Algorithm

1. Form the observation vector $\vec{y} = (y_1, y_2, \ldots, y_k)^\mathsf{T}$ as a column vector, based on the input.

2. Form the $k \times m$ design matrix X such that the entry in row i and column j of X is $f_j(x_i)$, using the functions and data points given as input. In other words, each function f_j should become a column of X, and each x_i should become a row.

3. Calculate $\hat{\beta} = (X^\mathsf{T} X)^{-1} X^\mathsf{T} \vec{y}$. This value is the orthogonal projection of $\vec{y}$ into the column space of X, so it will minimize the squared error $||\vec{y} - X\hat{\beta}||^2$.

4. The best-fit equation for $\vec{y}$ is $\hat{y} = \beta_1 f_1(x) + \beta_2 f_2(x) + \ldots + \beta_m f_m(x)$, where β_i is the i^{th} entry of $\hat{\beta}$. (The order of the coefficients in $\hat{\beta}$ will match the order of the functions in the design matrix X.)

5. (Optional) Calculate the error term $||\epsilon||^2 = ||X\hat{\beta} - \vec{y}||^2$, where $||\epsilon||^2 = \epsilon \cdot \epsilon$. This value is also known as the R^2 ("R squared") statistic and measures the goodness-of-fit.

6. (Optional) Plot the original data series $(\vec{x}, \vec{y})$ against $(\vec{x}, \hat{y} = X\hat{\beta})$ to examine the fit.