Project 2: Sortmania

Educational Objectives: On successful completion of this assignment, the student should be able to

- Implement a variety of comparison sort algorithms as generic algorithms, including Insertion Sort, Selection Sort, Heap Sort, Merge Sort, and Quick Sort, re-using code as much as possible from the course library. The implementations should cover both default order and order by passed predicate object.
- Discuss the capabilities and use constraints for each of these generic algorithms, including assumptions on assumed iterator type, worst and average case runtimes.
- Implement the Counting Sort algorithm for specified arrays of integers
- Implement the Counting Sort algorithm as a template function taking function object parameter that is used to define the sort value of the input integers, obtaining Bit Sort and Reverse Sort as special cases.
- Collect data and use the method of least squares to find a best fit scalability curve for each sort algorithm (including Counting Sort), based on a form derived from the known asymptotic runtime for the algorithm.
- Perform a comparative qualitative analysis of these algorithms using asymptotic runtime analysis as well as quantitative analysis using data collected from implementations.

Background Knowledge Required: Be sure that you have mastered the material in these chapters before beginning the project: Sequential Containers, Function Classes and Objects, Iterators, Generic Algorithms, Generic Set Algorithms, Heap Algorithms, and Sorting Algorithms

Operational Objectives: Implement various comparison sorts as generic algorithms, with the minimal practical constraints on iterator types. Each generic comparison sort should be provided in two froms: (1) default order and (2) order supplied by a predicate class template parameter. Also implement some numerical sorts as template functions, with the minimal practical constraints on template parameters. Again there should be two versions, one for default order and one for order determined by a function object whose class is passed as a template parameter. All of these sorts should then be evaluated for scalability using CPU timers on the architecture of your choice, fitting a scalability curve to each sort based on its theoretical asymptotic runtime. The results should be collected in a written report.

The sorts to be developed and tested are selection sort, insertion sort, heap sort, merge sort, quick sort, counting sort, and radix sort.

Deliverables: At least six files:

```
gsort.h         # contains the generic algorithm implementations of comparison
sorts
nsort.h         # contains the numerical sorts and class Bit
*.cpp           # source code for your testing
*.dat           # data files used for testing
makefile        # builds all testing executables
sort_report.pdf # your report
```

Scalability Curves

By a *scalability curve* for an algorithm implementation we shall mean an equation whose *form* is determined by known asymptotic properties of the algorithm and whose *coefficients* are determined by a least squares fit to actual timing data for the algorithm as a function of input size. For example, the selection sort algorithm,

implemented as a generic algorithm named `g_selection_sort()`, is known to have asymptotic runtime $\Theta(n^2)$, so the form of the best fit curve is taken to be

$$R = A + B\,n + C\,n^2$$

where R is the predicted run time on input of size n. To obtain the concrete scalability curve, we need to obtain actual timing data for the sort and use that data to find optimal values for the coefficients A, B, and C. Note this curve will depend on the implementation all the way from source code to hardware, so it is important to keep the compiler and testing platform the same in order to compare efficiencies of different sorts using their concrete scalability curves.

The following table shows the code name and forms for scalabality curves of each algorithm to be examined:

Sort	*Code Name*	*Scalability Curve Form*	*A*	*B*	*C*
Insertion Sort	`g_insertion_sort()`	$R = A + B\,n + C\,n^2$			
Selection Sort	`g_selection_sort()`	$R = A + B\,n + C\,n^2$			
Heap Sort	`g_heap_sort()`	$R = A + B\,n + C\,n \log n$			
Merge Sort	`g_merge_sort()`	$R = A + B\,n + C\,n \log n$			
Quick Sort	`g_quick_sort()`	$R = A + B\,n + C\,n \log n$			
Counting Sort	`counting_sort()`	$R = A + B\,n$			
Radix Sort (base 2)	`bit_sort()`	$R = A + B\,n$			

The last three columns of the table are where the coefficients for the form would be given, thus determining the concrete scalability curve.

The method for finding the coefficients A, B, and C is the *method of least squares*. Assume that we have sample runtime data as follows:

Input size:	n_1	n_2	...	n_k
Measured runtime:	t_1	t_2	...	t_k

and the scalability form is given by

$$f(n) = A + B\,n + C\,g(n)$$

Define the *total square error* of the approximation to be the sum of squares of errors at each data point:

$$E = \Sigma\,[t_i - f(n_i)]^2$$

where the sum is taken from $i = 1$ to $i = k$, k being the number of data points. The key observation in the method of least squares is that total square error E is minimized when the gadient of E is zero, that is, where all three partial derivatives $\mathbf{D}_A E$, $\mathbf{D}_B E$, and $\mathbf{D}_C E$ are zero. Calculating these partial derivatives gives:

$$\mathbf{D}_X E = 2\,\Sigma\,[t_i - f(n_i)]\,\mathbf{D}_X f$$

$$= 2\,\Sigma\,[t_i - (A + B\,n_i + C\,g(n_i))]\,\mathbf{D}_X f$$

(where X is A, B, or C). This gives the partial derivatives of E in terms of those of f, which may be calculated to be:

$$\mathbf{D}_A f = 1$$
$$\mathbf{D}_B f = n$$
$$\mathbf{D}_C f = g(n)$$

(because n and $g(n)$ are constant with respect to A, B, and C.) Substituting these into the previous formula and setting the results equal to zero yields the following three equations:

$$A\,\Sigma\,1 + B\,\Sigma\,n_i + C\,\Sigma\,g(n_i) = \Sigma\,t_i$$
$$A\,\Sigma\,n_i + B\,\Sigma\,n_i^2 + C\,\Sigma\,n_i\,g(n_i) = \Sigma\,t_i\,n_i$$
$$A\,\Sigma\,g(n_i) + B\,\Sigma\,n_i\,g(n_i) + C\,\Sigma\,(g(n_i))^2 = \Sigma\,t_i\,g(n_i)$$

Rearranging and using $\Sigma\,1 = k$ yields:

$$k\,A + [\Sigma\,n_i]\,B + [\Sigma\,g(n_i)]\,C = \Sigma\,t_i$$

$$[\Sigma\,n_i]\,A + [\Sigma\,n_i^2]\,B + [\Sigma\,n_i\,g(n_i)]\,C = \Sigma\,t_i\,n_i$$

$$[\Sigma\,g(n_i)]\,A + [\Sigma\,n_i\,g(n_i)]\,B + [\Sigma\,(g(n_i))^2]\,C = \Sigma\,t_i\,g(n_i)$$

These are three linear equations in the unknowns A, B, and C. With even a small amount of luck, they have a unique solution, and thus optimal values of A, B, and C are determined. (Here is a link to a more detailed derivation in the quadratic case $g(n) = n^2$.)

Note that all of the coefficients in these equations may be calculated from the original data table and knowledge of the function $g(n)$, in a spreadsheet or in a simple stand-alone program. The solution to the system of equations itself is probably easiest to find by hand by row-reducing the 3x4 matrix of coefficients to upper triangular form and then back-substitution.

Procedural Requirements

I. Part I: Coding

    I. Develop and fully test all of the sort algorithms listed under requirements below. Make certain

that your testing includes "boundary" cases, such as empty ranges, ranges that have the same element at each location, and ranges that are in correct or reverse order before sorting. Place all of the generic sort algorithms in the file `gsort.h` and all of the numerical sort algorithms in the file `nsort.h`. Your test programs should have filename suffix `.cpp` and your test data files should have suffix `.dat`.

II. Turn in `gsort.h` and `nsort.h` using the script `LIB/proj21submit.sh`.

Warning: *Submit scripts do not work on the* `program` *and* `linprog` *servers. Use* `shell.cs.fsu.edu` *to submit projects. If you do not receive the second confirmation with the contents of your project, there has been a malfunction.*

II. Part II: Analysis

III. Use a CPU timing system to obtain appropriate timing data for each sort. Input sizes should range from small to substantially large. Any timing programs used should have suffix `.cpp`. Again all data files should have suffix `.dat`, including output data. Make certain that any file containing items relevant to (or mentioned in) your report has suffix `.cpp` or `.dat` and is preserved for project submission.

IV. Use the method of least squares (sometimes called regression) to calculate the coefficients of a scalability curve for each sort.

V. Write a report on your results. This report should include (1) a description of your process and methods, including how the scalability coefficients were calculated; (2) summary data in tables, along with plots of timing data overlaid with concrete scalability curves, and (3) a general discussion of the findings and their implications for various sort algorithms. Illustrate all major conclusions and recommendations graphically where appropriate, for example with a single figure comparing concrete scalability curves superimposed in the same graphical display. Your report should be in PDF format and reside in the file `sort_report.pdf` format.

VI. Turn in `gsort.h`, `nsort.h`, `*.cpp`, `*.dat`, and `sort_report.pdf` using the script `LIB/proj22submit.sh`.

Warning: *Submit scripts do not work on the* `program` *and* `linprog` *servers. Use* `shell.cs.fsu.edu` *to submit projects. If you do not receive the second confirmation with the contents of your project, there has been a malfunction.*

VII. Also submit your report `sort_report.pdf` to the Blackboard course web site.

Code Requirements and Specifications

1. The two sort algorithm files are expected to operate using the supplied test harnesses: `fgsort.cpp` (tests `gsort.h`) and `fnsort.cpp` (tests `nsort.h`). Note that this means, among other things, that:

    i. All generic sorts in `gsort.h` should work with ordinary arrays as well as iterators of the appropriate category
    ii. Both classic (4-argument) `counting_sort` and the 5-argument version should work
    iii. `bit_sort` should work with the class `Bit` defined in `nsort.h`

2. Issuing the single command "`make`" should build executables for all of your test functions.

3. The comparison sorts should be implemented as generic algorithms with template parameters that are iterator types.

4. Each comparison sort should have two versions, one that uses default order (operator `<` on `I::ValueType`) and one that uses a predicate object whose type is an extra template parameter.

5. Re-use as many components as possible, especially existing generic algorithms such as `g_copy` (in `genalg.h`), `g_set_merge` (in `gset.h`), and the generic heap algorithms (in `gheap.h`).

6. Two versions of `counting_sort` should be implemented: the classic 4-parameter version, plus one that takes a function object as an argument. Here is a prototype for the 5-parameter version:

```
template < class F >
void counting_sort(const int * A, int * B, size_t n, size_t k, F f)
// Pre:  A,B are arrays of type unsigned int
//       A,B are defined in the range [0,n)
//       f is defined for all elements of A and has values in the range
[0,k)
// Post: A is unchanged
//       B is a stable f-sorted permutation of A
```

   Test and submit both versions of `counting_sort`.

7. Also test and submit a specific instantiation of radix sort called `bit_sort`, implemented using a call to `counting_sort` with an object of type `Bit`:

```
class Bit
{
public:
  size_t operator () (unsigned long n)
  {
    return (0 != (mask_ & n));
  }
  Bit() : mask_(0x00000001) {}
  void SetBit(unsigned char i) { mask_ = (0x00000001 << i); }
private:
  unsigned long mask_;
};
```

   Test and submit `bit_sort` (in file `nsort.h`).

Analysis and Report Requirements

1. Be sure that you use and document good investigation habits by keeping careful records of your analysis and data collection activities.

2. Before beginning any data collection, think through which versions of sorts you are going to test. These should ideally be versions that are most comparable across all of the sorts and for which the "container overhead" is as low as practical. Usually, this would mean using the array cases for all sorts. If however you do not have all sorts working for the case of arrays, you will have to devise a plan and defend the choices made.

3. You also need to plan what kinds and sizes of data sets you will use. It is generally advisable to create these in advance of collecting runtime data and to use the same data sets for all of the sorts, to reduce the affects of randomness in the comparisons. On the other hand, the data sets themselves should "look random" so as not to hit any particular weakness or strenght of any particular algorithm. For example: if a data set has size 100,000 but consists of integers in the range 0 .. 1000 then there will be lots of repeats in the data set, which could be bad for quicksort.

   Generally, it is best to use unsigned integer data for the data sets, so that they can be consumed by all of the sorts, including the numerical sorts.

4. If you use a multi-user machine to collect data, there will be the possibility that your timings are exaggerated by periods when your process is idled by the OS. One way to compensate for this is to do several runs and use the lowest time among all of the runs in your analysis.

5. Use the framework of (pseudo) random object generators in `LIB/cpp/xran.*` to generate test data. Be sure to keep your test data generating program and turn it in.

6. Use the CPU timing framework in `LIB/cpp/timer.*` to collect timing data. Again, your timing programs should be kept and turned in.

7. Choose units for time as appropriate, for example milisecond `ms`. Whatever units are chosen, you will need to deal with large differences of elapsed time, ranging over several orders of magnitude.

8. Your report should be structured something like the following outline. You are free to change the titles, organization, and section numbering scheme.

   i. **Abstract or Executive Summary**
   [brief, logical, concise overview of what the report is about and what its results and conclusions/recommendations are; this is for the Admiral or online library]
   ii. **Introduction**
   [A narrative overview "story" of what the paper is about: what, why, how, and conclusions, including how the paper is organized]
   iii. **Background**
   [what knowledge underpins the paper, such as theory, in this case the known asymptotic runtimes of the sorts, with references, and the statistical procedure to be used, with references]
   iv. **Analysis Process or Procedure**
   [datails on what data decisions are made and why; how input data is created; how timing data is collected; and how analysis is accomplished, including references for any software packages that are used]
   v. **Analysis Results**
   [give results in both tabular and graphical form]
   vi. **Conclusions**
   [use results, including comparative tabular and/or graphical displays, to back up your conclusions]
9. Reading your report should make it clear how to use the test functions and how data was collected from them.

Hints

- Heapsort is already done and distributed in `LIB/tcpp/gheap.h`. The prototypes for the two versions of heapsort should be useful as models for the other generic comparison sorts. *Don't re-*

invent this wheel.

- You will need specializations for some generic sort algorithms (`g_insertion_sort` and `g_merge_sort`) so that they work with arrays (raw pointers), because the generic versions use the iterator feature `ValueType` that pointers do not have.

- An example of using the timer and random object generators (used to demonstrate the relative speed of different implementations of Queue) is in `LIB/examples/dsraces/`.

- TAKE NOTES! Use either an engineers lab book or a text file to keep careful notes on what you do and what the results are. Date your entries. This will be of immense assistance when you are preparing your report. In real life, these could be whipped out when that argumentative know-it-all starts to question the validity of your report.

- The National Institute for Standards and Technology (NIST) has a good reference for statistics: NIST Engineering Statistics Handbook.