This document is officially released and open for comment in the discussion forum.

Project 2: Sort Algorithm Optimization

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, SelectionSort, HeapSort, MergeSort, and QuickSort, re-using code as much as possible from the course library. The implementations should cover both default order and order by passed predicate object.
- Optimize MergeSort and QuickSort to achieve better performance, given a performance criterion.
- Discuss the capabilities and use constraints for each of these generic algorithms, including assumptions on assumed iterator type, worst and average case runtimes.
- Discuss the techniques used for improving performance and the amount of improvement obtained by them.

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.

The sorts to be developed and tested are selection sort, insertion sort, heap sort (in several variations), merge sort (both top-down and bottom-up), quick sort (in several variations).

Develope optimized versions of MergeSort and QuickSort, also as generic algorithms.

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

Deliverables: Two files:

```
gsort.h          # contains the generic algorithm implementations of comparison sorts
report.txt       # report on findings during optimization of algorithms
```

Procedural Requirements

1. The official development, testing, and assessment environment is `g++47 -std=c++11 -Wall -Wextra` on the `linprog` machines. Code should compile without error or warning.

2. 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.

3. Develop and fully test optimized versions of MergeSort and QuickSort, testing with the same procedures used for the initial tests.

4. Place all of the generic sort algorithms in the file `gsort.h`.

5. Your test data files should have descriptive names so that their content can be inferred from the name.

6. Turn in `gsort.h` and `report.txt` using the script `LIB/proj2/proj2submit.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. A listing of these data files should appear as an appendix to your report.*

Divide-and-Conquer Algorithms

MergeSort [top-down version] is a prototypical divide-and-conquer algorithm, dividing a problem (in this case, a sorting problem) into two problems that are 1/2 as big and constructing a solution from solutions of the sub-problems. The two sub-problems are solved by recursive calls.

This divide-and-conquer technique is very powerful, resulting in both simple implementing code and a natural way to reason about the algorithm with mathematical induction.

The various recursive calls form a binary tree. For example, consider the MergeSort algorithm operating on the array

```
    [50 30 70 95 10 50 20 60 30 40 50 70 60 90 80 20]
```

The first two recursive calls are `MergeSort(0,8)` and `MergeSort(9,15)`. Each of these in turn makes two recursive calls to the algorithm on smaller arrays, and so on, until the array size is one, the base case of the recursion. The call tree looks like this (showing the arrays, but omitting the function name):

```
                            [50 30 70 95 10 50 20 60 30 40 50 70 60 90 80 20]

                       [50 30 70 95 10 50 20 60]                [30 40 50 70 60 90 80 20]

                 [50 30 70 95]          [10 50 20 60]         [30 40 50 70]          [60 90 80 20]

              [50 30]     [70 95]    [10 50]     [20 60]    [30 40]     [50 70]    [60 90]     [80 20]

             [50]  [30]   [70]  [95]  [10]  [50]   [20]  [60]  [30]  [40]   [50]  [70]  [60]  [90]   [80]  [20]
```

The dynamic trace of the algorithm running on the input array follows a left-to-right DFS path in the call tree (actually a postorder traversal of the tree). The call stack contains the path from root to current location, exactly like the stack used to implement a post-order binary tree iterator. For example, when the recursive call `MergeSort(4,6)` is made, the call stack contains the path shown in red.

Using the binary tree model of the recursive calls, as above, we can see that there are many calls made near the bottom of the tree where the array sizes are small. (In fact, 1/2 of the calls are made at the leaves of the tree!) If we can stop the recursion process several layers up from the leaf layer, we can eliminate a huge number of these calls.

An optimization that is useful whenever there is a non-recursive solution to the problem that runs quickly on very small input is to terminate the recursion at small input size and apply the non-recursive alternative instead, thus eliminating many recursive calls on small input sizes. For recursive divide-and-conquer sort algorithms such as MergeSort and QuickSort, using this idea to cut off the recursion and apply InsertionSort on small size input can measurably improve performance. InsertionSort is a good choice because it is simple and also is efficient on ranges that are completely or partially sorted. On the other hand, InsertionSort has worst-case runtime $\Omega(n^2)$, so we certainly do not want to run it on large ranges. The "cutoff" size is typically optimal somewhere between 4 and 16.

Improving Performance of MergeSort

In addition to applying the "cutoff" to InsertSort for small range sizes, MergeSort can be improved by (1) eliminating the merge operation when the two ranges are already in order, and (2) handling memory more efficiently.

Re (1): Notice that when the two recursive calls to sort subranges return, if the largest element in the left subrange is no larger than the smallest element in the right subrange, the entire range is already sorted without a call to the merge operation. This condition is easily detected by comparing the two elements in question, and the resulting non-call to the merge operation saves on the order of end - beg comparisons.

Re (2): It is difficult (and costly) to eliminate the need for extra space - required as the temporary destination/target for the merge operation. But as stated in its pure form, this destination memory is stack-allocated whenever the call to the merge operation is made, making the subsequent copy of data back into the original range unavoidable. Both the stack allocation and the subsequent data copy can be eliminated, resulting in a more efficient and faster implementation of the algorithm.

The stack allocation of data memory is eliminated by declaring space statically in the MergeSort body directly, and using that space as a target for the data merge operation. The subsequent data copy is eliminated by applying the next algorithm call to the data space and merging back into the original range. Thus the calls to merge alternate targets - on odd calls merging from the original range into the data store and on even calls merging from the data store back to the original range. At the end of the algorithm, if the total number of merges is odd, then a one-time data copy is made to get the sorted data back into its original location. A change in the organization of the code will be necessary.

This last idea to avoid most of the data copy calls is a tricky optimization to implement, and extra credit will be awarded for successfully accomplishing this improvement to MergeSort.

Improving Performance of QuickSort

In addition to applying the "cutoff" to InsertSort for small range sizes, QuickSort can be improved in two ways.

The first is to avoid the $\Omega(n^2)$ worst case runtime by eliminating the case of pre-sorted input: either randomize the input range in advance, or use a random selector for the pivot index during the run of the algorithm. The second improvement is really a change of the basic algorithm to "3-way QuickSort" (but keeping the concept).

Three-way QuickSort

As we know, pre-sorted data, where there is no actual change required for the data, produces the worst case runtime for QuickSort. A related situation occurs with data that contains many duplicate entries: the vanilla QuickSort algorithm will sometimes require one recursive call for each one of the repeated values. An elegant improvement for the algorithm to handle the case of highly redundant data uses the idea of 3-way partitioning, wherein all of the elements that are equal to the partition value are moved to a contiguous portion of the range, after which the entire portion may be omitted from further consideration:

```
Before partitioning: [50 30 70 95 10 50 20 60 30 40 50 70 60 90 80 20]
                      ^^
                      parition element

 After partitioning: [30 10 20 30 20 30 40 20 50 50 50 70 95 70 90 80]
                                              ^^^^^^^^
                                              partition value range [8,11)

 3 ranges:           [30 10 20 30 20 30 40 20 50 50 50 70 95 70 90 80]

 Only the two red ranges need to be sorted:
 Recursive calls:    QuickSort (0,8); QuickSort (11,15);
```

At this point, because the result is a range of indices, it is more convenient to subsume the partitioning code into the main algorithm. For an array, the code would look like this, with the partitioning code embodied in the while loop:

```
void g_quick_sort_3w (T* beg, T* end)
{
  ...
  T* low = beg;
  T* hih = end;
  T v = *beg;
  T* i = beg;
  while (i != hih)
  {
    if (*i < v) Swap(*low++, *i++);
    else if (*i > v) Swap(*i, *--hih);
    else ++i;
  }
  g_quick_sort_3w(beg, low);
  g_quick_sort_3w(hih, end);
}
```

(This very sleek code by Robert Sedgewick goes back to the discovery of 3-way QuickSort.) The while loop takes one of 3 actions for each i:

- if (*i < v) : swap *low and *i; increment both low and i
- if (*i > v) : decrement hih; swap *hih and *i
- if (*i == v) : increment i

Note that in all three cases, the distance between i and hih is made smaller, so the loop terminates after n = `end - beg` iterations. On termination of the while loop, the range [low,hih) consists entirely of elements equal to v, all elements with index < low have value less than v, and all elements with index >= hih have value greater than v. Thus the range [low,hih) is constant and we need only sort the range [beg,low) to its left and the range [hih,end) to its right.

Code Requirements and Specifications

1. The following is a list of the generic algorithms that should be implemented in `gsort.h`:

```
g_selection_sort    # version in class notes
```

```
g_insertion_sort    # already implemented in gsort_stub.h
g_merge_sort        # the pure top-down version from the class notes
g_merge_sort_bu     # the bottom-up version from the class notes
g_merge_sort_opt    # the top-down version with "cutoff" and conditional calls to merge
g_quick_sort        # the pure version from the class notes and Cormen
g_quick_sort_opt    # the same as above, with "cutoff" for small ranges
g_quick_sort_3w     # 3-way QuickSort
g_quick_sort_3w_opt # 3-way QuickSort, with "cutoff" for small ranges
```

Note that the following are implemented and distributed elsewhere in the course library:

```
List::Sort          # tcpp/list.h, list.cpp
alt::g_heap_sort    # tcpp/gheap_advanced.h
fsu::g_heap_sort    # tcpp/gheap_advanced.h
cormen::g_heap_sort # tcpp/gheap_cormen.h
```

2. The sort algorithm file `gsort.h` is expected to operate using the supplied test harnesses: `fgsort.cpp` and `sortspy.cpp`. Note that this means, among other things, that all generic sorts in `gsort.h` should work with ordinary arrays as well as iterators of the appropriate category.
3. All the sorts, including optimized versions, should be implemented as generic algorithms with template parameters that are iterator types.
4. Each 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. All the sorts should be in `namespace fsu`.
6. Some of the sorts will require specializations (for both the default and predicate versions) to handle the case of arrays and pointers, for which `I::ValueType` is not defined. (See `g_insertion_sort`.)
7. 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`).
8. Note that we are *not* implementing the random initialization optimization for any of the versions of `g_quick_sort`. The effect of that optimization should be mentioned in the report, but its effect is well understood from theory, and using it tends to obscure drawing conclusions from the other advances.

Report Guidelines

1. Your report should address the main points you are investigating, which should include:
   i. Specific optimizations made
   ii. How any optional constants, such as the "cutoff" size, were chosen, and why. (This is a good place for some experimentation.)
   iii. What gains are made by the optimizations - preferable broken down by data characteristics (such as "random", "almost sorted", "few unique keys").
2. Your report should be structured something like the following outline. You are free to change the titles, organization, and section numbering scheme. This is an informal report, which is why you are submitting a text file (as opposed to a pdf document).
   1. **Title: Improving Performance of MergeSort and QuickSort**
   2. **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]
   3. **Introduction**
      [A narrative overview "story" of what the paper is about: what, why, how, and conclusions, including how the paper is organized]
   4. **Background**
      [what knowledge underpins the paper, such as theory and the known runtime and runspace characteristics of the (pure, un-optmized) sorts, with references]
   5. **Data Analysis Process or Procedure**
      [details 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 and detailed documentation on any software used]

6. **Analysis Results**
   [State and possibly elaborate your conclusions]
7. **Recommendations**
   [Give a summary of your recommendations for best practice use of the various generic sorts.]
8. **Appendix 1**
   Give tables of collected data [time in particular] used to back up your conclusions.
9. **Appendix 2**
   Give detailed descriptions of all input files, including how they were built, there size, and constraints on content. (Do not put the actual input files in the report.)

Hints

- `g_heap_sort` is already done and distributed in `LIB/tcpp/`. There are three slightly different heap sort algorithms implemented: `alt::g_heap_sort`, which is discussed in the lecture notes; `fsu::g_heap_sort`, which uses a faster O(*n*) "make heap" algorithm; and `cormen::g_heap_sort`, which is the version discussed in the Cormen text. At some point before midterm exams, you should understand the distinctions among the three.

- Similarly, `g_insertion_sort` is fully implemented in `gsort_stub.h`. The prototypes for the default and predicate versions should be useful as models for the other generic comparison sorts. Note this is an example where a specialization is required for arrays because of the need for `ValueType` in the generic versions.

- TAKE NOTES! Use either an engineers lab book or (thoughtfully named) text files 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.

- Several code files that support data collection are provided:

  ```
  sortspy.cpp         # computes comp_count and timing data for generic comparison sorts
  ranuint.cpp         # generates file of randum unsigned integers
  ```

  A good data file nomenclature uses a base file name, such as "ran" or "dupes" that you can set for a series of data files. The suffix is the count of items in the file.

- Your file naming system should reflect the nature of the contents of the data files. For example, "ran" could be the base name for a series of data files with unconstrained random data. Then "ran.1 ran.10, ran.100, ran.1000, ..., ran.1000000" would be a series of data files of sizes 1, 10, 100, 1000, ... , 1000000..

- Be sure that your timing data collection plan uses file naming conventions that are compatible with your data file names. You can use a simple listing of file names to indicated the data you use in your investigations.