Assignment 3

Procedures

Turn in answers to the exercises below on the UAF Blackboard Learn site, under Assignment 3 for this class.

Your answers should consist of two files: `contigsum.hpp`, from Exercise A, and `inversions.hpp`, from Exercise B. These two files should be attached to your submission.
Send only the above! I do not want things I already have, like the test programs.
I may not look at your homework submission immediately. If you have questions, e-mail me.

Exercises (35 pts total)

Exercise A — Greatest Contiguous Sum

Purpose

In this exercise, you will implement a Divide-and-Conquer algorithm.

Background

Given a sequence of numbers, the greatest contiguous sum (GCS) is the greatest sum that can be made by adding up some contiguous subsequence of the numbers.

For example, consider the following sequence

$6, -8, 4, 3, -1, -2, 5, 8, -9, 4$

The contiguous subsequence with the greatest sum begins with the first 4 and ends at the 8. So the GCS is $4+3+(-1)+(-2)+5+8=17$. We could get a greater sum by adding in 6 and 4, but then the numbers added up would not be contiguous.

It is acceptable to add up no numbers, thus getting a sum of zero. For example, the GCS of the sequence

$-3,-4,-2$

is 0 (zero), gotten by adding up none of the numbers.
We wish to write an efficient algorithm to find the GCS of a given sequence. The brute-force algorithm just finds the sums of all contiguous subsequences: $\Theta(n^3)$ addition operations when given a sequence of *n* items. It is easy to improve this to $\Theta(n^2)$, but using Divide-and-Conquer we can do even better. A naïve Divide-and-Conquer algorithm can easily get $\Theta(n \log n)$ (how?), but we can actually do better than that by calculating even more. (This technique is very similar to “loading the induction hypothesis.”)

Algorithm

Here is an idea for an efficient algorithm to find the GCS using the Divide-and-Conquer strategy. Given a sequence of numbers, our algorithm computes and returns the following four values.

A. The GCS of the sequence.
B. The greatest possible sum of a contiguous subsequence that includes the first value in the sequence, or zero if all such sums are negative.
C. The greatest possible sum of a contiguous subsequence that includes the last value in the sequence, or zero if all such sums are negative.
D. The sum of the entire sequence.

The recursive case of the algorithm proceeds as follows. The sequence is divided into two nearly equal-sized parts: the first half and the second half. One recursive call is made for each of these halves. Then the A, B, C, D values are computed for the sequence based on the A, B, C, D values of each half.

For example, suppose we have found A, B, C, D for both halves; we wish to find A for the sequence as a whole. The contiguous subsequence giving the greatest possible sum is either (1) contained entirely in the first half, (2) contained entirely in the second half, or (3) meets both halves. In case (1) the value is A for the first half. In case (2) the value is A for the second half. In case (3) the value is C for the first half plus B for the second half. So our A value is the maximum of 1st-A, 2nd-A, and (1st-C + 2nd-B).

Here is a specific example. Looking at the first sequence above, we would split into two subsequences of size 5:

$6,-8,4,3,-1$ and $-2,5,8,-9,4$

For the first half, the A, B, C, D values are:

A. $4+3=7$.
B. $6=6$.
C. $4+3+(-1)=6$.
D. $6+(-8)+4+3+(-1)=4$.

For the second half, the A, B, C, D values are:

A. $5+8=13$.
B. $(-2)+5+8=11$.
C. $5+8+(-9)+4=8$.
D. $(-2)+5+8+(-9)+4=6$.

Again, the A value for the sequence as a whole is the maximum of 1st-A, 2nd-A, and (1st-C + 2nd-B). That is, it is $\max\{7,13,6+11\}=17$, which is what we found originally.

The B, C, and D values of the sequence can be computed similarly. For instance, the B value of the whole sequence is either contained entirely in the first half (in which case it is 1st-B) or it stretches to the second half (in which case it is 1st-D + 2nd-B). And of course we need a base case.

Specification

Write a function template `contigSum`, whose code is in header contigsum.hpp; there should be no associated source file. Prototype the function template as follows.

```
template<typename RAIter>
int contigSum(RAIter first, RAIter last);
```

- Function contigSum is given a sequence of values of type int, specified as usual using two random-access iterators. It returns the GCS of the sequence.
- The implementation of function contigSum must be based on the Divide-and-Conquer algorithm outlined above.
- Function contigSum should be able to handle any random-access range of int values. In particular, it should return a correct result (zero) for an empty range.
- Function contigSum should not be unnecessarily inefficient. In particular, the “extra work” of the algorithm—what it does after the two recursive calls—must use only a constant number of integer operations.

Test Program

A test program is available: contigsum_test.cpp. If you compile and run this program (unmodified!) with your code, then it will test whether your code works properly.

Do not turn in the test program.

Exercise B — Counting Inversions

Purpose

In this exercise, you will modify an existing Divide-and-Conquer algorithm to give it additional functionality.

Background

An inversion in a sequence of numbers is a pair of numbers that are in descending order.

For example, the sequence

    1,4,2,3

has 2 inversions: (4,2) and (4,3). The sequence

    2,2,1,1

has 4 inversions, all of the form (2,1). The sequence

    1,2,3,3

has no inversions.

We wish to write an efficient algorithm to count the number of inversions in a given sequence. The brute-force solution compares all pairs of sequence items: $\Theta(n^2)$ comparisons for a sequence of size *n*. Once again, using Divide-and-Conquer we can do better.

Algorithm

We can design an algorithm to count the inversions in a sequence based on any stable sort. Whenever the sort moves an item *x* to a different place in the sequence, we count the number of other items that item *x* skipped over. The total of all such counts is the number of inversions.

For example suppose we are sorting the following sequence (using some mysterious stable sort):

    5,1,2,4,3

We start by moving the 5 to the next-to-last place, thus skipping over 3 items.

    1,2,4,5,3

Next, we move the 3 to just before the 5, skipping over 2 items.

    1,2,3,4,5

And we are done. We skipped over 3 items and then 2 items. And 3+2=5. So the original sequence had 5 inversions; check to see that this is correct.

The stable sort we wish to modify is Merge Sort. This algorithm moves an item to a new place in the sequence in exactly one situation: during the main loop of the Stable Merge operation, when the item chosen comes from the second half. In this case, the number of items skipped over is the number of items remaining in the first half.

Specification

Write a function template inversions, whose code is in header inversions.hpp; there should be no associated source file. Prototype the function template as follows.

```
template<typename RAIter>
size_t inversions(RAIter first, RAIter last);
```

Function inversions is given a sequence of orderable values, specified as usual using two random-access iterators. It sorts the sequence and returns the number of inversions in the given sequence.
The implementation of function inversions must be based on a Merge Sort, modified to count inversions as discussed above. You may start with any implementation of Merge Sort, as long as its license allows and you give the original author credit. Starting from merge_sort.cpp (discussed in class and attached with this assignment on Blackboard), would be fine.
Function inversions must not make use of any global variables.
Function inversions should not be unnecessarily inefficient. In particular, it should do at most $\Theta(n \log n)$ value-type operations for a sequence of size n.

Test Program

A test program is available: inversions_test.cpp. If you compile and run this program (unmodified!) with your code, then it will test whether your code works properly.

Do not turn in the test program.

Coding Standards

The following apply to all programming assignments in this class.

- Code must compile & execute using a standard-conforming compiler.
- If a test program is provided, then code must compile with the test program.

The above requirements are absolute; if your code does not compile, then there is no point in turning it in.

In addition, to receive full credit, submitted code should satisfy the following conditions.

- Code should be neat and readable.
- Code should conform to standard conventions (conventional use of header and source files, where applicable, `const`-correctness, etc.).
- Comments should be included, indicating filename, authorship, and last revision date of each file, as well as the purpose of each file and each module that is larger than a function (e.g., a C++ class).
- All comments in the code should be *correct*.
- Programs *may* be based on code written by someone else. However:
    - Your submissions should be largely your own design, and a significant portion of the code should be your own work.
    - You must give credit in the code to the author of any code that is included (in original or modified form) in your submission.
    - Code written by others may only be used in a way that does not violate applicable laws and licenses.