UNBC Laboratory Assignment
Computer Science 200—Fall 2005

Robust Sorting

Purpose:

To implement a high-quality $\Theta(n \log n)$ [unstable] sorting algorithm.

Due Date:

This assignment is due **2006-11-17** at the beginning of class.

Tasks to Complete:

⇒ Implement an $O(n)$ random shuffle algorithm that completely randomly shuffles an array. (See the Exercises at the end of Chapter 2 in *Weiss*). Shuffle a four element array one million times to test for complete randomness.

⇒ Find the average time to shuffle 100 000 and 1 000 000 element arrays in order to verify $O(n)$-time.

⇒ Implement a three parameter version of heap sort with interface

```
template<class Iterator, class Predicate>
void heap_sort(Iterator begin, Iterator end, Predicate less) ;
```

that sorts the data with indices in the range [`begin`, `end`) according to the strict weak order `less`.

⇒ Implement `quick_sort` in a manner similar to *Weiss*. Add at least one more parameter to be passed in the recursive version of the algorithm. Use the extra parameter to track the depth of the recursion, and switch to heap sort if the depth is greater than 40.

⇒ Implement `insertion_sort`. Think about whether or not you want to use the sentinel element trick.

⇒ Using the above pieces, construct a `robust_sort` algorithm that uses at most $O(\log n)$ extra storage, and $O(n \log n)$ running time, but that is as fast as `quick_sort` most of the time.

⇒ Test your sorting algorithms for *correctness*. That is, write code that verifies that your sorting algorithm produces (a) a sorted array, and (b) an array that is a permutation of the original.

⇒ Time your sorting procedures for various different sized data collections. Think about whether you are trying to estimate worst case time or average case time.

⇒ You *must* show that your actual running times are consistent with the Θ-behavior for the sorting algorithm.

Helpful Hints:

If you don't like thinking about random access iterators abstractly, you can always begin an algorithm involving them with

```
int n = end - begin ;
Iterator data = begin ;
```

then write `data[i]` as you would in sorting an array.

(If you wish to be technically more perfect `#include <iterator>` and

```
typedef typename std::iterator_traits<Iterator>::difference_type dist_t;
```

and replace `int n = end - begin` with `dist_t n = end - begin`. This will work even on machines where the difference between pointers is a `long int`.)

Writing a two argument sort routine

To write a two-argument interface so that you can sort a `std::vector<double>` array `data` with `sort(data.begin(), data.end())`, you can do the following:

```
template<class Iterator, class Element>
inline void sort_helper(Iterator begin, Iterator end, Element ignored)
{
   sort(begin, end, less<Element>()) ;
   return ;
}
template<class Iterator>
void sort(Iterator begin, Iterator end)
{
    if (begin!=end)
        sort_helper(begin, end, *begin) ;
    return;
}
```

A less more mysterious, but less clunky solution is to `#include <iterator>` and `<functional>` and write

```
template<class Iterator>
void sort(Iterator begin, Iterator end)
{
    typedef typename
        std::iterator_traits<Iterator>::value_type value_t ;
    sort(begin, end, std::less<value_t>) ;
    return;
}
```