CS 207 Paper Exercises 1 // Due W 2/15 in class or via email to kohler@seas Solutions by Loren McGinnis

(1) Design an interface to binary search that takes only a comparator, rather than a comparator and an item to find. Write out a doxygen specification comment for the search. Pay special attention to the comparator's signature.

```
/** Binary search of @a a using comparator @a c
 * @pre for all indices 0<=i<j<n,
 *     @a c[@a a[i]] == 0 iff @a c[@a a[j]] >= 0
 *     @a c[@a a[i]] > 0 iff @a c[@a a[j]] > 0
 * i.e. there is a total ordering of @a a w.r.t @a c
 * @param a input array of length @a n
 * @param c comparator fuction
 *
 * @return index i into @a a where @a c(@a a[i])==0, or -1 if no such i exists
 * Complexity: O(log n) */
template <typename T, typename COMP> int binary_search(T * a, int n, int (*c) (T));
```

(2) Why might the C++ standard library have implemented the version with item and comparator, rather than the version with only a comparator?

A new comparator would have to be implemented for every value that we wanted to search for in the array, since the value to be searched for would be baked into the comparator's implementation.

(3) Here's an incorrect comparator.

```
template <typename T> bool bad_comparator(T a, T b) {
    return true;
}
```

It is incorrect because it returns true for both "`bad_comparator(a, b)`" and "`bad_comparator(b, a)`", and thus does not totally order objects of type T.

What will happen if you pass `bad_comparator` to a binary search function?

The binary search function will always return -1 (assuming -1 means "the item was not found"), because according to the comparator, all items are both greater than and less than the search value.

(4) What will happen if you pass `bad_comparator` to the following `quick_sort` function?

partition will always return last, since it puts all elements in [first, last) for which comparator returns true before all elements for which it returns false, and returns the first element for which it returns false (none of them), meaning the first recursive call in the else statement is identical to the current call. It will recurse infinitely.

```
template <typename T, typename COMP> struct bound_comparator {
    T a_; COMP comparator_;
    bound_comparator(T a, COMP comparator)
        : a_(a), comparator_(comparator) {
    }
    bool operator()(T x) {
        return comparator_(x, a_);
    }
};

template <typename T, typename COMP>
void quick_sort(T* first, T* last, COMP comparator) {
    if (first != last) {
        bound_comparator<T, COMP> partitioner(*first, comparator);
        T* midpoint = std::partition(first, last, partitioner);
        if (midpoint == first)   // *first was a minimum element; assume it hasn't moved
            quick_sort(midpoint + 1, last, comparator);
```

```
        else {
            quick_sort(first, midpoint, comparator);
            quick_sort(midpoint, last, comparator);
        }
    }
}
```

(5) Write an assertion that, if inserted into `quick_sort`'s implementation, would turn the bad behavior you found in (4) into an assertion failure.

At the beginning of the `else{}` statement:

```
assert(midpoint != last);
```

(6) Write the minimal precondition for `quick_sort` so that, if `quick_sort`'s arguments obey the precondition, the assertion you added in (5) will never fail.

By "minimal," we mean that your precondition should only prevent the assertion failure.

```
@pre ∃ some pointer first≤i<last s.t. !comparator(*i,*first)
```

(7) Write a full precondition for `quick_sort` so that, if `quick_sort`'s arguments obey the precondition, the output range `[first, last)` will be sorted according to `comparator`. This sort requirement is expressed precisely by this postcondition:

```
// Post: ∀i, j with first ≤ i ≤ j < last, !comparator(*j, *i).
```

```
@pre  ∀i, j, k with first ≤ i, j, k < last,
    comparator(*i, *j) implies !comparator(*j, *i)
    comparator(*i, *j) and comparator(*j, *k) implies comparator(*i, *k)
```