

**Reading:**

This week will finish Chapter 4 and cover some aspects of stacks and queues from Chapters 5 and 6. Begin reading Chapter 8, sections 1–4. Be sure to keep straight the difference between a general tree and a binary tree, and only then think about a binary search tree—heaps are different too. The final project will use general trees plus heaps out of section 8.5 (to be covered next week).

This problem set is mostly based on Chapter 4. **It is to be done in hardcopy with C++ iterator syntax.** The point is to make your code clear enough to read analytically. There are stated limits on the number of lines. First, please *study* the following example, which sets the pattern for all of the problems.

```

/** Strings x and y differ by exactly one char delete or one char change.
    This is called having "edit distance 1".
 */
bool ed1(const string& x, const string& y) {
    int xleft = 0; int yleft = 0;
    int xright = x.length() - 1; int yright = y.length() - 1;
    //INV: Any and all chars to left of xleft and yleft match each other,
    //ditto chars to right of xright and yright. [In logic, these statements
    //hold by default even though initially there are no such chars.]

    while (xleft <= xright && yleft <= yright && x[xleft] == y[yleft]) {
        xleft++; yleft++;
    }
    //Control here means all chars have matched on LHS up to xleft,yleft.
    //If one has gone off the string, accept iff other's length is 1 more.
    if (xleft > xright) {
        return (xleft == yright); //INCLUDES case x == y, return false.
    }
    if (yleft > yright) {
        return (yleft == xright);
    }
    //Control here means we have a mismatched character

    while (xright >= xleft && yright >= yleft && x[xright] == y[yright]) {
        xright--; yright--;
    }
    return (xright <= xleft && yright <= yleft);
}

/** Iterator Version */
bool ed1a(const string& x, const string& y) {
    string::const_iterator xleft = x.begin();
    string::const_iterator xright = --(x.end());
    string::const_iterator yleft = y.begin();
    string::const_iterator yright = --(y.end());
    //INV: x[0..xleft-1] == y[0..yleft-1] && x[xright+1..] == y[yright+1..]

```

```

while (xleft <= xright && yleft <= yright && (*xleft) == (*yleft)) {
    xleft++; yleft++;
}
//Control here means all chars have matched on LHS up to xleft,yleft.
//If one has gone off the string, accept iff other's length is 1 more.
if (xleft > xright) {
    return (xleft == yright); //INCLUDES case x == y, return false.
}
if (yleft > yright) {
    return (yleft == xright);
}
//Control here means we have a mismatched character

while (xright >= xleft && yright >= yleft && (*xright) == (*yright)) {
    xright--; yright--;
}
return (xright <= xleft && yright <= yleft);
}

```

**Important Note:** Even though the iterator *movements* can all be done with a BidirectionalIterator, the *ordering comparisons* are allowed only with a RandomAccessIterator. **Even more important note**, these names are official in C++ documentation but they are **not** names of classes—which is why they are not in **teletype** font. At the very end of the course we may see how C++ wants “generic code” with these categories handled via templates to look.

(1) Translate the following code into iterator notation. One important point is that the `string` class has a constructor that takes two iterators denoting a substring of another string as arguments. (18 pts.)

```

/** True if switching 2 adjacent unequal letters in x leaves y.
 */
bool transpose(const string& x, const string& y) {
    if (x.length() != y.length()) { return false; }
    // else
    int xlenm1 = x.length() - 1;
    for (int i = 0; i < xlenm1; i++) {
        if (x[i] != y[i]) { //mismatch, so moment-of-truth is now
            if (x[i] == y[i+1] && x[i+1] == y[i] && x[i] != x[i+1]
                && x.substr(i+2) == y.substr(i+2)) {
                return true;
            } else {
                return false;
            }
        }
    }
} //control here means x == y
return false;
}

```

(2) Write code to find the index  $j$  into an array `cont` that maximizes the value `f(cont[j])` for some given numerical function `f`. If there are multiple such  $j$ , return the least one. **Then** translate

your code into iterator notation, returning an iterator pointing to that element. What is the least category of iterator needed for this task? (12 + 9 = 21 pts.)

[Note that this is an abstraction of the Assignment 6 task of finding the greatest number of words in a phrase, except now you are asked to identify a phrase that gives it. For 9 pts. extra credit, write this code as a “higher-order function” taking `f` as a parameter.]

(3) Write iterator code to merge two containers `cont1` and `cont2`, creating a third container `cont3`. Your REQ is that both `cont1` and `cont2` are sorted according to a `lessThan` method in non-decreasing order, and this must be an ENS for `cont3`. A particular case of what this means is that for any iterator `it` into `cont1` that is in a valid position (i.e., pointing to a real element not a dummy node, which entails that `cont1` is nonempty), the following is true:

```
(*cont.begin()).lessThan(*it) || (*(cont.begin())) == (*it)
```

Here we are assuming that the client type `I` (same as `ItemType` in the text) has a value-equality operator that together with `lessThan` obeys trichotomy. Note also that we are allowing the container to have multiple copies of items that compare equal (later we will say that their *keys* compare equal), though by the sortedness requirement, all such elements must be consecutive within any of the sorted containers. Again say what is the least category you need: `ForwardIterator`? `BidirectionalIterator`? `RandomAccessIterator`? Your code should have the header

```
void merge(const Container& cont1, const Container& cont2, Container& cont3)
```

and can assume that the call-by-reference parameter `cont3` has methods like `push_back` that you need. (21 points)

(4) Write iterator code to perform binary search on a nonempty container `cont` sorted in non-decreasing order (like the last problem). Use iterators `right` and `left`, and obey the following header and INV:

```
/**Return iterator itr to first element such that !((*itr).lessThan(item)), end() if none.
*/
```

```
iterator binsearch(const I& item) { ... }
```

```
    //INV: *left is <= item is < *right (or right==end())
```

Again, do you need a random-access iterator? (24 pts., for 84 total on the set)