A common error made by beginning users of the Standard Library containers and algorithms is to write code whose run-time complexity (big-O) is a flaming disaster. These facilities were invented by algorithm and data structure fanatics who took big-O really seriously; their goal was to make it possible for you to use "best of breed" data structures and algorithms very easily. They certainly did not intend to help you write awful code easily!

In fact, in most cases, the Standard Library tries to keep you out of trouble by making it inconvenient for you to use it in a really inefficient way. But thanks to the great generality and power of the C++ templates used in the Standard Library, there are loopholes that allow you to write hopelessly inefficient code as easily as very efficient code. One such loophole is that you can easily write Standard Library code that does a binary search on a linked list, which is so ridiculously inefficient that saying "binary search on a linked list" is actually a geeky programming joke! Linked-lists are supposed to be searched linearly! The purpose of this document is to explain why the Standard Library makes telling this joke so easy to do, and demonstrate with some run-time comparisons why it is so bad. There is actually a situation in which it conceivably makes sense, but it is really unlikely to be needed.

The Standard Library allows you to apply the binary_search and lower_bound algorithms to any sorted sequence container, including std::list, and it will produce a correct result. The following works for any sequence container:

```cpp
binary_search(container.begin(), container.end(), probe)
```

However, if you look at any normal code for binary search (e.g. as in Kernighan and Ritchie), it is written to use array subscripting. Array subscripting, a so-called random-access mechanism, runs in constant time, regardless of the size of the array or value of the subscript. In contrast, a linked list has the property that the only way you can find a particular node is to start at one end of the list and follow the links from one node to the next, checking them one at a time. Unlike with array subscripting, there is no way to compute the location of a list node directly from its numerical position in the list - it could be anywhere in memory. So how is it that you can do a binary_search on a std::list?

**How binary_search is implemented.** Below is a somewhat simplified copy of the Metrowerks Standard Library version of lower_bound; the binary_search algorithm just calls lower_bound and checks the result (other implementations might differ, but only in details).

```cpp
template <class ForwardIterator, class T>
ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last, const T& value)
{
    typedef typename iterator_traits<ForwardIterator>::difference_type difference_type;
    difference_type len = distance(first, last);
    while (len > 0)
    {
        ForwardIterator i = first;
        difference_type len2 = len / 2;
        advance(i, len2);
        if (*i < value)
        {
            first = ++i;
            len -= len2 + 1;
        }
        else
            len = len2;
    }
    return first;
}
```
First, see how this algorithm is written in terms of iterators, so that it can apply to any sequence container that supports
the standard iterator interface. The two input iterators are `first` and `last`, marking the beginning and end of the range to be
searched. Among the iterator types, input iterators can be iterators pointing into any type of container.

The basic binary search algorithm involves calculating the midpoint of a range of values, and then checking the value at
that midpoint. The code does this by calling `std::distance`, which returns the numerical distance between the first and last
iterators. This distance is divided by two, and then `std::advance` is called to move the first iterator forward by that amount
to get to the midpoint of the range. The `distance` and `advance` functions are also function templates that are defined so that
they work with iterators into any type of container. Template magic is used to specialize them for different iterator types. For
random-access iterators (which behave like pointers or subscripts, supplied by `std::vector` and `std::deque`), the definition
of distance that is used is:

```cpp
template <class RandomAccessIterator>
inline
typename iterator_traits<RandomAccessIterator>::difference_type
_distance(RandomAccessIterator first, RandomAccessIterator last, random_access_iterator_tag)
{
    return last - first;
}
```

The subtraction operator is defined for these iterators because the internal pointers can simply be subtracted to get the
distance directly via pointer arithmetic. This is exactly what subtracting the indices would do in the array form of the binary
search algorithm.

However, for the more general input iterators, which can only move forward or back by one step at a time, the definition
used to implement distance is:

```cpp
template <class InputIterator>
inline
typename iterator_traits<InputIterator>::difference_type
_distance(InputIterator first, InputIterator last, input_iterator_tag)
{
    typename iterator_traits<InputIterator>::difference_type result = 0;
    for (; first != last; ++first)
        ++result;
    return result;
}
```

In other words, compute the distance between two general input iterators by incrementing the first until it equals the last,
and count how many increments are required.

The `advance` function has a similar pair of specializations. Advancing a random-access iterator simply adds the number
of steps to the iterator, corresponding directly to pointer arithmetic, but advancing an input iterator requires incrementing the
iterator the supplied number of times.

When these functions are applied to a built-in array or a `std::vector` container, the `distance` and `advance` functions
will compile down to simple subtraction and addition of the subscript values/pointers, taking almost no time. But if applied to
a list, whose iterators support only moving forward or back by one step at a time, then the binary search will require using the
distance function to repeatedly count the nodes between the ends of the narrowing range and then `advance` with increment
and count again to position at the midpoint. Surely all this link-following will add up to a substantial amount of time!

The Standard in fact states that `lower_bound` and `binary_search` will run in $O(\log n)$ time when applied to a con-
tainer with random access iterators. When applied to a container that lacks random-access iterators, like `std::list`, the
Standard states that the search will be logarithmic with the number of comparisons, but linear with the number of nodes vis-
ted. So whether it runs faster than a linear search depends on how much time it takes to do the comparisons compared to
counting the nodes over and over again.
Let’s find out what happens. Sometimes you need benchmarks to see how theory works out in practice. I defined two classes of objects which contain an ID value used in operator< and operator==, and with constructors that give each object a unique value. One class, Cheap, uses a single integer for the ID, so comparisons that should be very fast. The other, Expensive, uses an ID **std::string** containing 31 characters, the first 28 of which are identical. The comparison is done with **std::string**'s < and = operators, which will have to test the first 28 characters before finding the different ones.

My benchmark code filled containers with 10, 50, 100, 500, or 1000 objects, in order, and then used **std::find** or **binary_search** looking for each one of the objects, so each possible position in the container was searched for. This search sequence was repeated more for shorter containers so that the final results show the total time for 50,000 searches distributed evenly over all the positions in the container, giving reasonably stable average run times. I compared run times for the eight combinations of **std::vector** vs. **std::list** containers, linear search with **std::find** vs. **std::binary_search**, and Cheap vs. Expensive objects.

The results are shown in the graphs below. First, the Expensive object searches required about ten times longer than the corresponding Cheap object searches, so the difference in the two classes is significant. The clear winner is **binary_search** on the **vector**, whose logarithmic search time is so fast that it looks almost flat when plotted on this scale for both Cheap and Expensive objects. There is hardly any difference between searching 10 objects and searching 1000! Clearly, this combination is the best solution, and it should be used unless there is some compelling reason to use something else. Using a list container or linear search looks very poor by comparison.
Let's see what these results tell us about binary search on a linked list, starting with Expensive objects; here the search would be logarithmic with the number of comparisons, which explains why it is second-fastest for Expensive objects. Notice how it is still looks linear - this is what big-O tells us; the time is dominated by the fastest-increasing time component of the algorithm, which would be time required to count through the list to find the middle element, which is linear with the list length. In contrast, the linear searches for Expensive objects are horribly slow for both containers. Those Expensive comparisons really add up when you have to do so many of them!

The Cheap objects show some different effects because the comparison is so fast that the other time costs of the data structure and algorithm become visible. The binary search of list does pretty poorly here because we have to do a lot more linear node traversals to do the counting compared to simply doing a linear search of the list; the logarithmic number of comparisons isn't a visible advantage since the comparisons are so fast. In turn, the linear search of a list ran somewhat faster than the linear search of a vector; apparently, the constant-time multiplication and addition arithmetic for computing the address of each cell is slower than merely following a pointer to the next node. This is all very interesting, but again, notice how the binary search of a vector is the clear winner with Cheap objects as it was with Expensive objects; the time is really fast, and is essentially the same in the range from 10 to 1000 objects in the container!

Conclusion: Is there ever a reason to do a binary search of a linked list? Take a look again at the Expensive graph above. These results suggest that if (1) the objects are very expensive to compare, and (2) for some reason you have to use a linked-list container, then yes, binary search of a linked-list would be a good choice. Of course, your mileage will vary depending on the cost of comparing your objects and the lengths of your lists and the distributions of your searches.

When would this combination of circumstances happen? It seems unlikely. Possibly if objects had to be added or removed from the container frequently, and the objects were very expensive to copy (as well as compare), a linked-list might be competitive with a vector. But in those cases, one of the associative containers (set, map, or their unordered equivalents) might be a better choice. Running some benchmarks could help you decide. The other EECS 381 handout, Fill’er Up: Winners and Losers for Filling an Ordered Container, demonstrates this for another case.

Advice: Don’t unleash the awesome power of the Standard Library to do a binary search of a linked-list unless you know that it is actually the best choice to solve your problem.

---

1 The slightly higher middle value is due to "noise" in the measurement - which is more visible at such small time values.