CSC112 Spring 2011
Fundamentals of Computer Science
Searching Unordered and Ordered Lists

Unordered Lists and Linear Search

If the items on a list are in no particular order, and you want to search the list for a given target, then **linear search** is the only reasonable option. In a linear search, the items on the list are compared sequentially with the target until either 1) the target is found, or 2) we run out of items.

Pseudo-code for this type of search is given below

```
// Input:   a -- an array of data.
//          n -- the number of data items in the array.
//          target -- the target of the search.
//
// Output:  if the target was found, the search returns the position
//          in the array where target was found.  Otherwise, returns -1
//

int linear_search( a, n, target )
{
   k = 0 ;           //  Loop index
   found = false ;   // We have not yet found the target.
   while ( ( k < n ) and (not found) ) {
      found = (a[k] == target) ;
      if ( not found ) k = k + 1 ;
   }
   if (found) return k ;
   else return -1      ;
}
```

On average, the time taken by a linear search is proportional to the number of items on the list.

Ordered Lists and Binary Search

If the items on a list are ordered (e.g. alphabetical, or in numerical order) and you want to search the list for a given target, then linear search works, but **binary search** is much more efficient. The basic idea for binary search is very simple: compare the target to an item in the middle of the ordered list. If, by luck, the target matches the middle item, then the search is complete. If the target comes before the middle item, then narrow your search to the first half of the list. If the target comes after the middle item, then narrow your search to the second half of the list. Repeat.

This method of searching is very efficient. For a list of size n, the time needed to complete the search is proportional to $\log_2(n)$. For example, a list with a thousand items can be searched in just 10 comparisons; a list size one million can be searched in 20 comparisons; a list size one billion can be searched in 30 comparisons.

While the idea is simple, it is fairly easy to get the code wrong. One pitfall is that an infinite loop may be created when the search target is not present on the list. The key to getting this algorithm correct is to understand and to write our code consistent with a **loop invariant**.

A **loop invariant** is a logical statement that is true before the loop is entered for the first time, and remains true on every iteration. A loop invariant can be verified, and can be used to prove the correctness of a search algorithm. For our purposes, we start with two variables **left** and **right**, such that the remaining search region is strictly between the left position and the right position.

Here, our loop invariant is given by:

$$\textbf{left} < \textbf{position of search target} < \textbf{right} \tag{1}$$

Notice the use of strict inequalities, i.e, "$<$", not "$\leq$".

Pseudo-code for binary search based on this loop invariant is given below.

```
// Input:   a -- an array of data.
//          n -- the number of data items in the array.
//          target -- the target of the search.
//
// Output:  if the target was found, the search returns the position
//          in the array where target was found.  Otherwise, returns -1
//

int binary_search( a, n, target )
{
   left = -1 ;       //  Left boundary
   right = n ;       //  Right boundary
   while ( ( right - left ) > 1 )  {  //  Search region is not empty
      mid = ( left + right ) / 2 ;    //  Find the mid-point.
      if ( a[mid] == target) return mid ;  // FOUND IT !!
      if ( target < a[mid] ) right = mid ;
      else left = mid ;
   }
   return -1      ;
}
```

Getting Binary Search Wrong

You might think that binary search should start with the boundaries 0, and $n-1$. In this approach, the region to be searched includes the endpoints. This approach can lead us to a logical error if we are not mindful of the new loop invariant in use. For example, the following pseudo-code enters an infinite loop when the target is not in the array. Can you explain why ?

```
// Input:   a -- an array of data.
//          n -- the number of data items in the array.
//          target -- the target of the search.
//
// Output:  if the target was found, the search returns the position
//          in the array where target was found.  Otherwise, returns -1
//

int binary_search_wrong( a, n, target )
{
   left = 0 ;       //  Left boundary
   right = n-1 ;       //  Right boundary
   while ( ( right - left ) >= 1 )  {  //  Search region is not empty
      mid = ( left + right ) / 2 ;    //  Find the mid-point.
      if ( a[mid] == target) return mid ;  // FOUND IT !!
      if ( target < a[mid] ) right = mid ;
      else left = mid ;
   }
   return -1      ;
}
```

Getting Binary Search Right (Again) The flaw in the code above results from not using a loop invariant consistently. It is possible to use the following alternative loop invariant, but it must applied consistently throughout the algorithm.

$$\textbf{left} \leq \textbf{position of search target} \leq \textbf{right} \tag{2}$$

Here, the position of the target is kept between the boundaries **left** and **right**, but the boundaries themselves are included as possible positions for the target. We now consider pseudo-code based on this loop invariant

```
// Input:   a -- an array of data.
//          n -- the number of data items in the array.
//          target -- the target of the search.
//
// Output:  if the target was found, the search returns the position
//          in the array where target was found.  Otherwise, returns -1
//
```

```
int binary_search_version_2( a, n, target )
{
   left = 0 ;       //  Left boundary
   right = n-1 ;       //  Right boundary
   while ( ( right - left ) >= 0 )  {  //  Search region is not empty
      mid = ( left + right ) / 2 ;    //  Find the mid-point.
      if ( a[mid] == target) return mid ;  // FOUND IT !!
      if ( target < a[mid] ) right = mid - 1 ;
      else left = mid + 1 ;
   }
   return -1      ;
}
```

Notice the update of **left** and **right**. Once we have verified that the condition `(a[mid] == target)` does not hold, we move the boundaries one position past the point **mid**. Notice this maintains the loop invariant, but ensures that the boundaries **left** and **right** do not stagnate (i.e., the boundaries move at every iteration).

There are trade-offs between the time taken by various operations (insert, delete, search, sort) and whether the list is ordered or not. The following table summarizes these trade-offs.

	List Type	
Operation	**Unordered**	**Ordered**
Search	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$
Insert	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Delete	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(1)$

Big "O" Notation

The performance (speed) of algorithms is often measured asymptotically. I.e., the important question is: **How does the time taken by the algorithm increase as the size of the problem becomes large ?**

The answer to this question is often given as an upper bound on the worst-case behavior of an algorithm. When we say an algorithm is $\mathcal{O}(f(n))$ we mean that in the worst case, the time taken by that algorithm is no more than a constant multiple of $f(n)$.

The "constant multiple" part of big "O" notation is to allow us to abstract away issues such as processor speed, bus speed, memory hierarchy, cache size, and memory speed. We seek a simple function that describes the nature of the relationship between problem size, and time taken to complete a computation.

For example, the selection sort is an $\mathcal{O}(n^2)$ algorithm.