

In many applications it is necessary to search a list of data elements for one (or more) that matches some specific criterion.

For example:

- find the largest integer in a list of test scores
- find the location of the string "Fred" in a list of names
- find all `Trip` variables whose destination is "Jackson, WY"

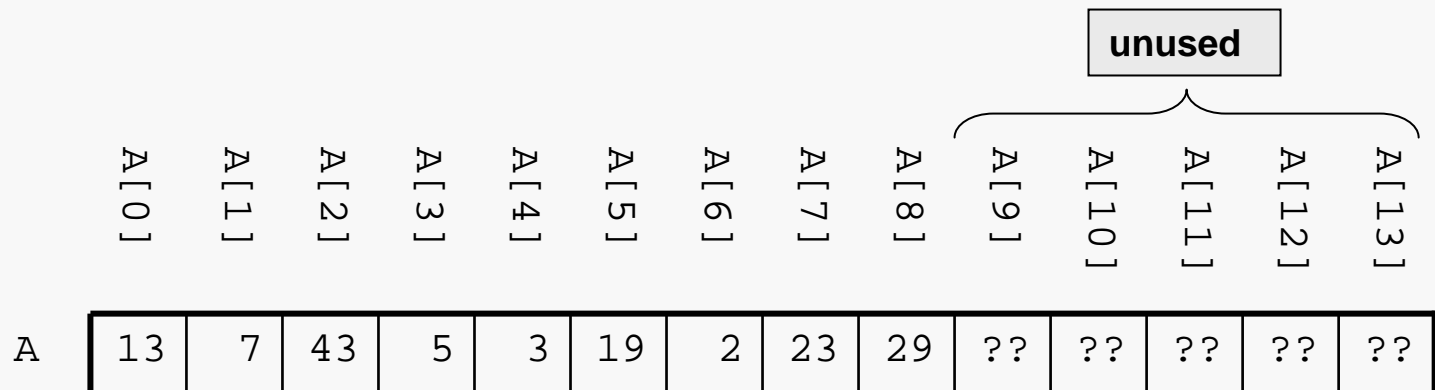
There are several basic issues when searching:

- how do we recognize and keep track of matches to the search criteria?
- how do we terminate the search?
- what do we report? (value, location, yes/no, . . .)
- how do we recognize that there is no matching data element?
- what do we do if no matching data element is found?

Here we will only consider the problem of searching among the elements of an array.

When searching an array of simple data elements, we may resolve these questions easily:

- We recognize matches by comparing with the equality operator.
- We terminate the search if we find an element that equals the search target, or if we have rejected the last used cell in the array.
- We may report a match by providing the index or a copy of the element.
- We recognize that there is no matching data element if we reach the end of the used cells in the array without finding a match.
- If no matching element is found, we return an impossible index (e.g., -1) or a dummy value that cannot logically be a valid data element.



The simplest technique for searching a list is to just "walk" the list, examining each element in turn until you either find a match or reject the last used cell:

```
const int MISSING = -1 ;

int LinearSearch(const int List[], int Target, int Size) {

    int Scan = 0;           // begin search at first cell

    // Continue searching while there are more entries
    // to consider and the target value has not been
    // found:
    while ( ( Scan < Size ) && (List[Scan] != Target) )
        Scan++;

    if ( Scan < Size )      // Target was found
        return( Scan );
    else
        return ( MISSING ); // Target was not found
}
```

Consider implementing an application that will store data for a list of trips and perform lookup operations on that list (e.g., report the distance or time for a trip given a specific origin and destination).

Clearly we may use an array of `Trip` variables, as defined earlier, to store the data.

Assume that the application will use two input files, one containing the trip data as we've seen before, and a second input file that will contain the lookup commands the program is supposed to process, using the general syntax:

```
<command string><tab><tab-separated command parameters>
```

For example:

mileage	Knoxville, TN	Nashville, TN
time	Tucumcari, NM	Albuquerque, NM
neighbors	Birmingham, AL	
. . .		

In this case, the array elements are complex structures and the search function must access the appropriate member(s) to make the comparison:

```
int reportMileage(const Trip& toFind, const Trip dB[], int numTrips) {  
  
    int Idx;  
    for (Idx = 0; Idx < numTrips; Idx++) {  
  
        if ( (toFind.Origin      == dB[Idx].Origin      &&  
            toFind.Destination == dB[Idx].Destination)  
            ||  
            (toFind.Origin      == dB[Idx].Destination &&  
            toFind.Destination == dB[Idx].Origin) ) {  
  
            return dB[Idx].Miles;  
        }  
    }  
    return MISSING;  
}
```

The designer has determined that the direction of the trip does not matter, which complicates the Boolean test for a match somewhat. How would the implementation change if direction did matter?

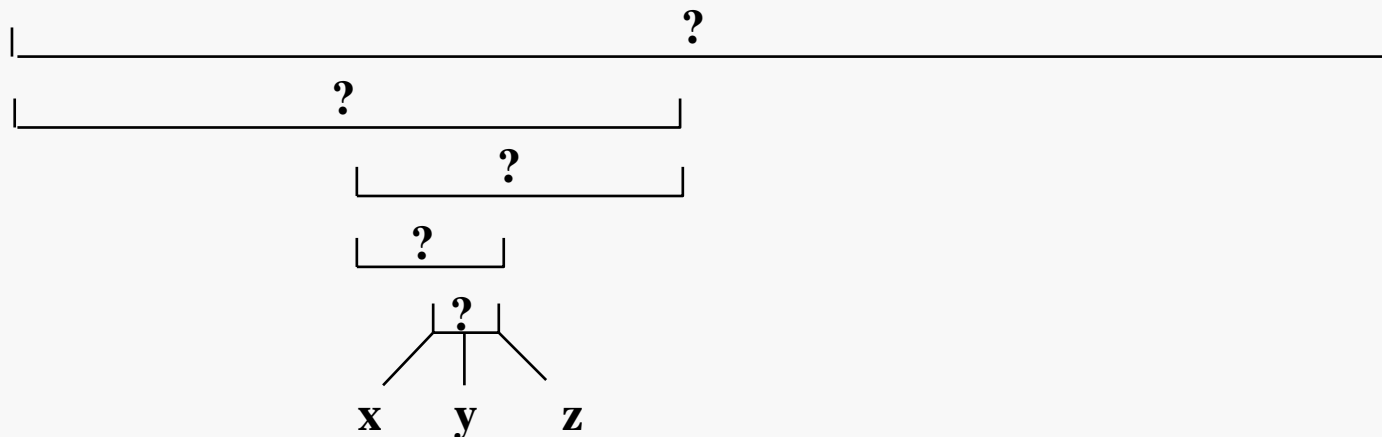
Note the use of `const` in the parameter list. Is this good design? Why or why not?

Linear search will always work, but there is an alternative that is, on average, much faster. The difficulty is that binary search may only be applied under the following:

Assumption: the list is sorted in ascending (or descending) order:

$$\text{List}[0] \leq \text{List}[1] \leq \dots \leq \text{List}[\text{Size}-1]$$

Binary search: Examine the middle element. If the target element is not found, determine to which side it falls. Divide that section in half and examine the middle element, etc, etc ...



An implementation of binary search is more complex logically than a linear search, but the performance gain is impressive:

```
const int MISSING = -1;

int BinSearch(const int List[] , int Target, int Lo, int Hi) {
    int Mid;

    while ( Lo <= Hi ) {

        Mid = ( Lo + Hi ) / 2;           // find "middle" index

        if ( List[Mid] == Target )      // check for target
            return ( Mid );
        else if ( Target < List[Mid] )
            Hi = Mid - 1;               // look in lower half
        else
            Lo = Mid + 1;               // look in upper half
    }
    return ( MISSING );                // Target not found
}
```

Note this implementation allows the caller to search a specified portion of `List[ ]`.

Suppose an array contains the values:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	21	35	41	43	51	82	85	86	93	??	??	??	??	??

Consider searching A[ ] for the value 86.

What would the call look like?

Trace the execution:

Lo	0	5	7	
Mid	4	6	7	
Hi	8			

Consider searching A[ ] for the value 50.

What would the call look like?

Trace the execution:

Lo	0		2	3
Mid	4	1	2	3
Hi	8	3		

Suppose the array to be searched contains  $N$  elements. The cost of searching may be measured by the number of array elements that must be compared to Target.

Using that measure:

	Best Case	Worst Case	Average Case
Linear Search	1	$N$	$N/2$
Binary Search	1	$\log_2 N$	$\log_2 N$

To get an idea of how much cheaper binary search is, note that

$$\begin{array}{ll}
 N = 1024 & \log_2 N = 10 \\
 N = 1024^2 & \log_2 N = 20
 \end{array}$$

Of course, binary search is only feasible if the array is sorted . . .

## Unequal Access Probabilities

- Implemented when a small subset of the list elements are accessed more frequently than other elements.

## Static Probabilities

- When the contents of the list are static the most frequently accessed elements are stored at the beginning of the list.
- Assumes that access probabilities are also static

## Dynamic Probabilities

- For nonstatic lists or lists with dynamic probability element accesses, a dynamic element ordering scheme is required:
  - Sequential Swap Scheme
    - † Move each element accessed to the start of the list if it is not within some **threshold** units of the head of the list.
  - Bubble Scheme
    - † Swap each element accessed with the preceding element to allow elements to “bubble” to the head of the list.
  - Access Count Scheme
    - † Maintain a counter for each element that is incremented anytime an element is accessed.
    - † Maintain a sorted list ordered on the access counts.

## ■ Setup:

- Store the desired element at the end of the array:

```
const int MISSING = -1;

int SeqSearch4 (Item A[], Item K, int size) {
    int i;

    A[size] = K;
    for ( i = 0; !(K == A[i]); i++ )
        ;

    if ( i < size )
        return ( i );
    else
        return ( MISSING );
}
```

- Requires storage at the end of the array to always be available.
- Ensures that the loop will terminate.
- Array parameter must be passed by reference to allow the sentinel insertion.

## Variation of Binary Searching

- Attempts to more accurately predict where the item may fall within the list. Similar to looking up telephone numbers
- Standard Binary Search Midpoint Computation:

$$\text{Midpoint} = (L+R) / 2;$$

- General Binary Search Midpoint Computation:

$$\text{Midpoint} = L + 1/2 * (R - L);$$

- Interpolation replaces the 1/2 (in the above formula) with an estimate of where the desired element is located in the range, based on the available values (be careful of int arithmetic):

```
Interp = L + // base loc +
        ((K - A[L]) / // % of distance K is from
         (A[R] - A[L])) * // A[L] to A[R] *
        (R - L); // length of search space
```

- Example: Assume 30K recs of SSNs in the range from 0 ... 600 00 0000
- Searching for 222 22 2222 yields an initial estimate of:

```
Interpolation = 0 + ((222222222 - 0) /
                    (600000000 - 0)) *
                (30000 - 0);
                = 11111
```

- Worst Case Order approximately =  $O(\log \log N)$
- Can be assumed to be a constant of about 5 since  $\cong (\lg \lg 10^9)$
- Assumes the search values are evenly distributed over the search range, ( ! True for SSNs)
- Inefficient for searching small number of elements