Simple Searching

Many computer applications involve searching for particular data that has been previously stored.

Problem:

Find the position of an element, `xelem`, in an array of integers:

```
int  alist[size];
```

- Cannot assume that `xelem` is in the list.
- If `xelem` does not exist in `alist[]` then return -1 (an impossible location).

Linear Searching

Simply scan the list until `xelem` is found or the end of the list is reached.

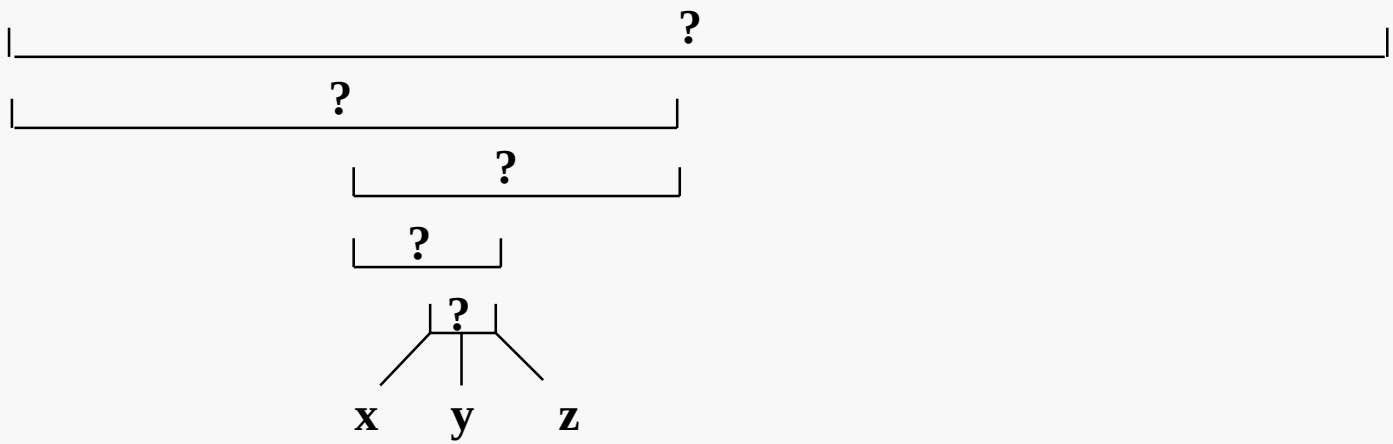
```
const int MISSING = -1 ;


int LinearSearch(const int alist[], int  xelem, int  size)
{
    int scan = 0;        // begin search at first cell

    while  ( ( scan < size )  && ( alist[scan] != xelem ) )
        scan++ ;

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

Binary Search

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

```
alist[0]  <=  alist[1]  <=  ...  <=  alist[size-1]
```

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



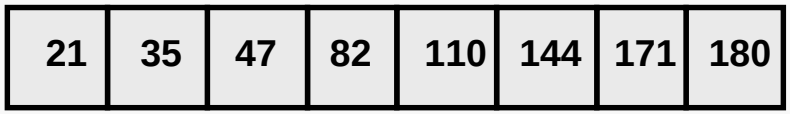
Binary Search Code

```
const int MISSING = -1 ;

int BinSearch(const int alist[] , int xelem, int lo, int hi )
{
    int mid;

    while ( lo <=  hi ) {

        mid =  ( lo + hi ) / 2;           // locate middle

        if ( alist[mid] == xelem )        // check for target
            return ( mid );
        else if ( xelem < alist[mid] )
            hi = mid - 1;                 // look in lower half
        else
            lo = mid + 1;                 // look in upper half
    }

    return ( MISSING );       // xelem not found
}
```

```
   assert ( (lo > 0) && (lo < MAX_ARRAY_SIZE)  &&
            (hi > 0) && (hi < MAX_ARRAY_SIZE)) ;
// no need to assert( lo <= hi ) ;
```

Binary Search Trace

Suppose the array contained the following values:



Find the position of 110.

lo	hi	mid
0	7	

the call

bin_search (a, 110, 0, 7)

returns ->

Find the position of 38.

lo	hi	mid
0	7	

the call

bin_search (a, 38, 0, 7)

returns ->

Cost of Searching

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 `xelem`.

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

$N = 1024 \quad \log_2 N = 10$
$N = 1024^2 \quad \log_2 N = 20$

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

Sorting

Many computer applications involve sorting items into some specified order.

To sort a group of items, the following relationships must be clearly defined over the items to be sorted:

a	<	b
a	>	b
a	=	b

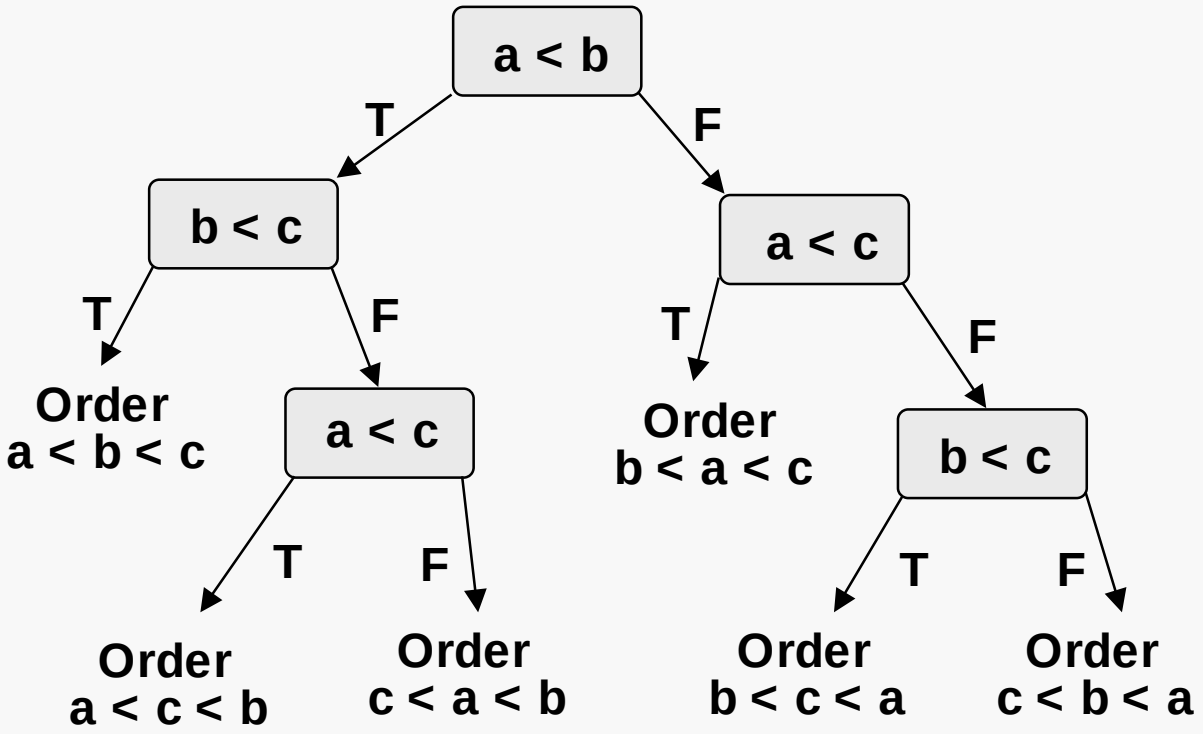
One goal of sorting is to minimize the number of comparisons necessary to sort the list of items.

Ascending Order: low ... high , first ... last

Descending Order: high ... low , last ... first

Decision Trees

To sort three items (a, b, c), how many comparisons must be made?



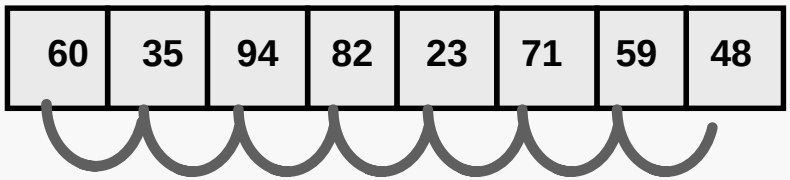
The decision tree above assumes that the three items (a, b, c), are unique.

Bubble Sort Algorithm

1. Initialize the size of the list to be sorted to be the actual size of the list.

2. Loop through the list until no element needs to be exchanged with another to reach its correct position.

    2.1 Loop (i) from 0 to size of the list to be sorted -1.
        2.1.1 Compare the i^{th} & i^{th} + 1 elements in the unsorted list.
        2.1.2 Swap the i^{th} & i^{th} + 1 elements if not in order (ascending or descending as desired).



Adjacent array locations are compared and swapped until the array is sorted

Bubble Sort Code

```
void BubbleSort(int aList[] , const int Size)  {

   void Swap( int& elem1, int& elem2);

   for (int Stop = Size - 1; Stop > 0; Stop--) {

     for (int Check = 0; Check < Stop; Check++) {

           if (aList[Check] > aList[Check + 1])
              Swap(aList[Check] , aList[Check + 1]);
     }
   }
}

void Swap(int& elem1, int& elem2) {
    int temp = elem1;
    elem1    = elem2;
    elem2    = temp;
}
```

This can be improved by modifying the outer loop to exit if the list is already sorted.

Bubble Sort Trace

0	1	2	3	4	5	<- indices
						Original array
						First Pass
						First Pass
						First Pass
						Second Pass
						Second Pass
						Third Pass
						Third Pass
						Fourth Pass
						Fifth Pass

Characteristics:

- Compares adjacent elements.
- Simple, but not very efficient.

Selection Sort Algorithm

1. Loop (i) from 0 to the (number of elements to be sorted - 2)

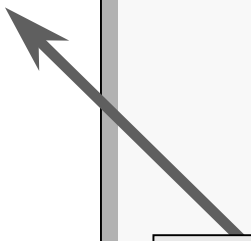
    1.1 Assume the smallest remaining item is at the i^{th} position, call this location smallest.

    1.2 Loop (j) through the remainder of the list to be sorted (i+1 .. size-1).

        1.2.1 Compare the j^{th} & smallest elements in the unsorted list.

        1.2.2 If the j^{th} element is < the smallest element then reset the location of the smallest to the j^{th} location.

    1.3 Move the smallest element to the head of the unsorted list, (i.e. swap the ith and smallest elements).



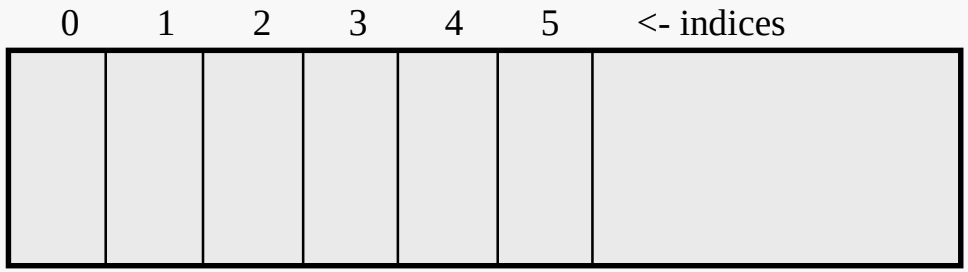
After sorting all but 1 element the remaining element must be in its correct position.

Straight Selection Sort

```
void SelectionSort(int aList[] , const int Size)  {

   int Begin, SmallSoFar, Check;

   void Swap(int& elem1, int& elem2);                // see previous slide


   for (Begin = 0; Begin < Size - 1; Begin++) {

      SmallSoFar = Begin;                                  // set head of tail

      for (Check = Begin + 1; Check < Size; Check++) { // scan current tail

            if (aList[Check] < aList[SmallSoFar])
                SmallSoFar = Check;
      }

      Swap(aList[Begin], aList[SmallSoFar]);
   }
}
```

Assumes the entire list from 0 .. Size-1 is to be sorted. Parameters (first , last) would be required to allow a subset of the list to be sorted.

Selection Sort Trace



Characteristics:

- Every pass places one more element in its final sorted position.
- Performs approximately the same number of comparisons as bubble sort.
- Performs many fewer swaps than bubble sort.

Cost of Sorting

Suppose the array to be sorted contains N elements. The cost of sorting may be measured by the number of array elements that must be compared to each other, or by the number of times two array elements must be swapped.

Using those measures (approximately):

	Comparisons		Swaps	
	Worst	Average	Worst	Average
Bubble Sort	N^2	N^2	N^2	N^2
Selection Sort	N^2	N^2	N-1	N-1

In some cases, comparisons are more expensive than swaps; in other cases, swaps may be more expensive than comparisons. In any case, there is no cost advantage to using Bubble Sort.

There are other, more complex, sorting algorithms whose costs are on the order of $\log_2 N$ — those are introduced in CS 2574.