It turns out that if it is known that an array is sorted then there is a much better algorithm for finding a target value. The algorithm is called a *binary search*.

The algorithm works by repeatedly rejecting one-half of the remaining array elements until only one element is left. At each step we compare the target with an element from the middle of the remaining segment of the array, and we reject either the bottom or the top half of that segment according as the element is less than or more than the target.

To encode this algorithm, we create variables low, high, and probe. As the algorithm runs, there is at any given moment a collection of array elements that have not yet been rejected. The variables low and high store the integers between which lie the indexes of the array elements that are still under consideration. If the target is present in the array, its index will at all times be greater than or equal to low and less than or equal to high. To begin with, all the array elements are still in play, so we store -1 in low and in high we place the length of the array (that is, one more than the highest index). Next we set probe to the value of (low + high) / 2. (Remember: this is integer arithmetic; so we have in mind the quotient when the sum of low and high is divided by 2.) We compare the element at probe with the target, and depending on the result we discard either the elements whose indexes are greater than probe or those with indexes that are less than probe.

The algorithm is presented in the simulation shown below. Click the **Start** button to generate a sorted array of random integers and a randomly selected target. Then repeatedly click the **Step** button. Run the simulation several times until you are able follow what is happening and feel comfortable with the way the algorithm works. In particular, try to figure out in what way the situation when Instruction #1 says to stop reveals the solution to the search.

1. **If high - low <= 1 then stop.**
2. Calculate (high + low) / 2 and store the result in probe.
3. If the element at probe is greater than target, set high = probe and go back to step 1.
4. Set low = probe and go back to step 1.

Size: 

**Target: 89**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 9 | 25 | 49 | 63 | 73 | 74 | 89 |  |
|  |  |  |  |  |  |  |  | low  probe | high |

Number of comparisons: 4

When executed, each of the following code fragments performs a binary search for a target int in an ordered array of ints. In each case, study the code and predict exactly how many iterations of the while loop occur. Then single step through the code, carefully watching the changing values of the indexes stored in low, high, and probe. For each code segment, record the number of iterations of the loop, and make a note of the final values of the variable low and the element array[ low ].

1. Target: 10

**Single Stepper**

int[] array = { 2, 2, 3, 4, 5, 5, 7, 9, 10, 12, 12, 16 };  
  
int high = array.length;  
int low = -1;  
int probe;  
int target = 10;  
  
while ( high - low > 1 )  
{  
  probe = ( high + low ) / 2;  
  if ( array[ probe ] > target )  
    high = probe;  
  else  
    low = probe;  
}

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Watch**   |  |  | | --- | --- | | array | {2, 2, 3, 4, 5, 5, 7, 9, 10, 12, 12, 16} | | high | 9 | | low | 8 | | probe | 9 | | target | 10 | | **Output** |

**Notes:** The program has finished.

  public static int binarySearch( int[] array, int target )  
  {  
    int high = array.length;  
    int low = -1;  
    int probe;  
  
    while ( high - low > 1 )  
    {  
      probe = ( high + low ) / 2;  
      if ( array[ probe ] > target )  
        high = probe;  
      else  
        low = probe;  
    }  
    if ( low >= 0 && array[ low ] == target )  
      return low;  
    else  
      return -1;  
  }

So far, all our examples of binary searching have involved integers, which of course have a built-in numerical ordering. The technique is applicable, however, in any context that involves objects on which there is a natural ordering. In particular, it is applicable in situations involving objects that implement the [Comparable<T> interface](https://www.eimacs.com/eimacs/mainpage?epid=E1971627750&cid=162149). To illustrate, let us modify the Item class so that it implements that interface.

**Exercise 177**

In the code area below,

1. complete the compareTo and equals methods of the Item class, making sure that they are both based solely on a comparison of the values of the myN instance variable.
2. complete and test the definition of the class method binarySearch.

public class Item implements Comparable<Item>   
{   
  private int myN;   
  
  public Item( int n )   
  {   
    myN = n;   
  }

public int compareTo( Item i )

{

    return myN - i.getN();

}

public boolean equals( Object o )

{

    return ( o instanceof Item ) &&

              ( compareTo( (Item)o ) == 0 );

}

  public String toString()   
  {   
    return "Item: " + myN;   
  }   
  
  public int getN()    
  {   
    return myN;   
  }   
  
  public static Item[] makeItemArray( int len )   
  {   
    Item[] a = new Item[ len ];   
    int i;   
    for ( i = 0 ; i < len ; i++ )   
      a[ i ] = new Item( i );   
    return a;   
  }   
     
  public static void displayArray( Item[] array )   
  {   
    for ( Item item : array )   
      System.out.println( item );   
  }   
  
  // Precondition: Item[] array is an array of Item objects.   
  // This method returns the index in the array at which there is    
  // an Item with the same value of myN as the target, or -1   
  // if no such item can be found. The method implements    
  // a binary search.

public static int binarySearch( Item[] array,

                                   Item target )

   {

     int high = array.length;

     int low = -1;

     int probe;

     while ( high - low > 1 )

     {

       probe = ( high + low ) / 2;

       if ( target.compareTo( array[ probe ] ) < 0 )

         high = probe;

       else

         low = probe;

     }

     if ( low >= 0 && target.equals( array[ low ] ) )

       return low;

     else

       return -1;

   }

}   
  
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    Item a = new Item( 1 );    
    Item b = new Item( 21 );   
    Item c = new Item( 1 );   
       
    // should be a negative integer   
    System.out.println( a.compareTo( b ) );   
       
    // should be 0   
    System.out.println( a.compareTo( c ) );   
       
    // should be a positive integer   
    System.out.println( b.compareTo( a ) );   
  
    Item[] array = { new Item( 2 ), new Item( 4 ),   
       new Item( 6 ), new Item( 8 ),   
       new Item( 10 ), new Item( 12 ) };   
  
    // should be 1   
    System.out.println( Item.binarySearch( array, new Item( 4 ) ) );   
  
    // should be -1   
    System.out.println( Item.binarySearch( array, new Item( 5 ) ) );   
  }   
}

-20   
0   
20   
1   
-1

### Exercise 177

Suitable definitions are as follows:

  public int compareTo( Item i )  
  {  
    return myN - i.getN();  
  }

  public boolean equals( Object o )  
  {  
    return compareTo( (Item)o ) == 0;  
  }

Strictly speaking, by overriding the equals method in this way we are violating the general contract for equals, which is required never to throw an exception. The usual way to avoid this uses the instanceof operator (which is not part of the Advanced Placement Java subset):

  public boolean equals( Object o )  
  {  
    return ( o instanceof Item ) &&  
             ( compareTo( (Item)o ) == 0 );  
  }

  public static int binarySearch( Item[] array,  
                                  Item target )  
  {  
    int high = array.length;  
    int low = -1;  
    int probe;  
  
    while ( high - low > 1 )  
    {  
      probe = ( high + low ) / 2;  
      if ( target.compareTo( array[ probe ] ) < 0 )  
        high = probe;  
      else  
        low = probe;  
    }  
    if ( low >= 0 && target.equals( array[ low ] ) )  
      return low;  
    else  
      return -1;  
  }

In general, given an array of length *k*, a sequential search will make at most *k* iterations. (This happens either when the array does not contain the target or when the target is the last element of the array.) It can be proved that a binary search of an array of 2*n*–1 elements will take at most *n* iterations. This is equivalent to saying that a binary search of an array of length *k* will take at most *t* iterations, where *t* is the smallest integer greater than or equal to log2*k* (denoted symbolically by "⌈log2*k*⌉").

For an array of just ten elements, this makes virtually no difference. However, for extremely large arrays, the difference can be very significant, as the following table shows:

|  |  |  |
| --- | --- | --- |
| **Array size** | **Sequential** | **Binary** |
| *k* | *k* | ⌈log2*k*⌉ |
| 4 | 4 | 2 |
| 8 | 8 | 3 |
| 10 | 10 | 4 |
| 16 | 16 | 4 |
| 32 | 32 | 5 |
| 100 | 100 | 7 |
| 1,000 | 1,000 | 10 |
| 10,000 | 10,000 | 14 |
| 100,000 | 100,000 | 17 |
| 1,000,000 | 1,000,000 | 20 |
| 100,000,000 | 100,000,000 | 27 |

For the sake of argument, suppose that each iteration in a sequential search takes one-thousandth of a second, that is, one millisecond or 0.001 seconds. And let's suppose that one iteration in a binary search also takes 0.001 seconds. The above table tells us that a sequential search of 1,000,000 elements could take as long as 1,000 seconds or about 16 minutes. A binary search of the same array would take at most 20 milliseconds, that is, one-fifth of a second.

Of course, the sequential search will often take less time than the worse case scenarios listed in the table. In fact, for a given array size, the algorithm we have been using for binary searching takes the same number of iterations no matter where the target is located. So if the target is located close to the start of the array then a sequential search will often beat a binary search. But even if we assume that on average the target is found about half-way through the array, a binary search is still by far the best algorithm for large arrays.

Top of Form

### Exercise 180

1. With this algorithm it is no longer the case that, for a given size of array, the binary search always takes the same number of iterations. Because this algorithm checks whether or not array[ middle ] is the target, it is possible for the algorithm to terminate before the collection of possible array elements has been reduced to a single element. In fact, we could be lucky and hit the target on the very first iteration!

In addition, a very minor increase in efficiency results from the fact that, on each iteration of the while-loop, the middle element is removed from consideration, thereby narrowing the field of possible elements somewhat more rapidly than the previous algorithm does.

1. With the previous algorithm, each iteration of the while-loop involves only one comparison. With this algorithm, each iteration that does not result in termination of the algorithm requires two comparisons. The potential exists, therefore, for this algorithm to require twice as many comparisons as the previous algorithm. Since comparison of integers occurs almost — but not quite — instantaneously, the resulting decrease in efficiency would be barely detectable for all but the most enormous arrays. But if this algorithm were applied to arrays of Strings, say, comparisons of which are much more time-consuming, then possibly doubling the number of comparisons could really slow down the operation of the algorithm.

In practice, the choice of a search algorithm involves more parameters than we have examined here. Issues that have an impact on the decision include these:

* how many elements are there likely to be in the arrays being searched?
* are there any patterns in the data? For example, does the target being looked for often appear early in the array?
* how time-consuming is the element-comparison process?

It is often the case that a conclusion cannot be reached until a computer engineer performs a battery of trial runs using each of the candidate algorithms on collections of simulated data.

Bottom of Form

**Exercise 180**

The following code uses a slightly different algorithm for a binary search in an ordered array of integers than the one we have been using so far:

public static int binarySearch( int[] array, int target )   
{   
  int left = 0;   
  int right = array.length - 1;   
  int middle;   
  
  while ( left <= right )   
  {   
    middle = ( left + right ) / 2;   
    if ( array[ middle ] == target )   
      return middle;   
      
    if ( array[ middle ] > target )   
     right = middle - 1;   
   else   
     left = middle + 1;   
  }   
  
  return -1;   
}

1. Describe one or more ways in which this algorithm is preferable to our previous binary search algorithm.
2. Describe one or more ways in which our previous binary search algorithm is preferable to this one.
3. It is also possible to perform a binary search recursively. Complete the following recursive implementation of a binary search:

public class MainClass   
{   
  public static int binarySearch( int[] array, int target, int low, int high )   
  {   
    int probe;   
  
    if ( high - low > 1 )   
    {   
      probe = ( high + low ) / 2;   
      if ( array[ probe ] > target )   
        low = binarySearch( array, target, , );   
      else    
        low = binarySearch( array, target, , );   
    }   
  
    if ( low == -1 || array[ low ] != target )   
      return -1;   
  
    return low;   
  }

}

4