**Binary**

The **binary (the base 2 number system, which only uses the digits 0 and 1)** search algorithm works by repeatedly checking the middle element and then rejecting one-half of the remaining array elements until only one element is left. In order for this algorithm to be effective, the array has to be sorted.

At each step, the item you want to find is compared with the middle element of the remaining segment of the array. If it matches, one instance of the item has been found. If not, then we have a choice to make.

We can either look in the first half or the second half of the array, but not both. In comparing the element we want to find to the middle element, we will know if it is equal to, less than, or greater than the middle element. If it is less, then we reject the section of the array to the right of the middle and look in the left section; otherwise, we reject the left section and look in the right section. We keep splitting the array in half until we either find the match or run out of elements to search.

### Part 1

The binary search should remind you of the recursive principle of divide and conquer. It can only be applied to information that is already sorted, but since you have mastered sort algorithms, there should be no problem. The code for the binary search will take some close analysis on your part, so use the eIMACS labs wisely.

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: 67**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 17 | 26 | 41 | 67 | 67 | 67 | 79 | 97 |  |
| low |  |  |  |  |  |  |  |  | high |

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**   |  |  | | --- | --- | |  |  | |  |  | | **Output** |

**Notes:** The highlighted code will create an array variable of data type int to which it will assign {2, 2, 3, 4, 5, 5, 7, 9, 10, 12, 12, 16}.

 Target: 3

**Single Stepper**

1. 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 = 3;  
     
   while ( high - low > 1 )  
   {  
     probe = ( high + low ) / 2;  
     if ( array[ probe ] > target )  
       high = probe;  
     else  
       low = probe;  
   }

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Watch**   |  |  | | --- | --- | |  |  | |  |  | | **Output** |

1. **Notes:** The highlighted code will create an array variable of data type int to which it will assign {2, 2, 3, 4, 5, 5, 7, 9, 10, 12, 12, 16}.



You may have noticed that, in both parts of the above exercise, when the algorithm concludes, the elements at low and high are adjacent and the element at low is the target. The next exercise explores cases when the target is nowhere to be found in the array.

#### Exercise 175

1. 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, single step through the code, carefully watching the changing values of the indexes stored in low, high and probe. After each code fragment completes execution, make a note of the values of the variable low and (when such an element exists) the element array[ low ].
   1. Target: 11

**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 = 11;  
  
while ( high - low > 1 )  
{  
  probe = ( high + low ) / 2;  
  if ( array[ probe ] > target )  
    high = probe;  
  else  
    low = probe;  
}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Watch**   |  |  | | --- | --- | |  |  | |  |  | | **Output** |

**Notes:** The highlighted code will create an array variable of data type int to which it will assign {2, 2, 3, 4, 5, 5, 7, 9, 10, 12, 12, 16}.

 Target: –6

**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 = -6;  
  
while ( high - low > 1 )  
{  
  probe = ( high + low ) / 2;  
  if ( array[ probe ] > target )  
    high = probe;  
  else  
    low = probe;  
}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Watch**   |  |  | | --- | --- | |  |  | |  |  | | **Output** |

**Notes:** The highlighted code will create an array variable of data type int to which it will assign {2, 2, 3, 4, 5, 5, 7, 9, 10, 12, 12, 16}.

 Target: 20

**Single Stepper**

1. 

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 = 20;  
  
while ( high - low > 1 )  
{  
  probe = ( high + low ) / 2;  
  if ( array[ probe ] > target )  
    high = probe;  
  else  
    low = probe;  
}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Watch**   |  |  | | --- | --- | |  |  | |  |  | | **Output** |

**Notes:** The highlighted code will create an array variable of data type int to which it will assign {2, 2, 3, 4, 5, 5, 7, 9, 10, 12, 12, 16}.

 In view of these observations and those in the previous exercise, describe how low may be used to discover whether the search is successful and, if so, where the target is located in the array.

1. When execution of the code fragment concludes, the asked-for values are as follows:
   1. low: 8; array[ low ]: 10 (which is less than the target)
   2. low: -1; array has no element with an index of -1
   3. low: 11; array[ low ]: 16 (which is less than the target)
2. If the value of low is non-negative and the element of array at index low is the target, then the search has been successful and low is the index of an occurrence of the target in array. Otherwise, the search has been unsuccessful and the target is not to be found in array.

Implement and test a static method binarySearch that takes as its arguments int[] array (presumed to be an ordered array of ints) and int target, and conducts a binary search of array for target, returning the index of an occurrence of target in array (if there is one) or -1 (if not).

  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;  
  }

On this page, we stray outside the requirements of the Advanced Placement exam in that the exercise involves an interface and asks you to define consistent compareTo and equals methods. However, performing a search with the help of a compareTo method is an activity that often occurs when dealing with an array or an ArrayList of Strings, and such a search continues to be an examinable topic.

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=E2346022675&cid=162149). To illustrate, let us modify the Item class so that it implements that interface.

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 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 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 ) ) );   
  }   
}

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;  
  }

By simply changing the signature of this method to

binarySearch( String[] array, String target)

and leaving everything else unchanged, a method is produce that implements a binary search on an array of Strings.

#### Exercise 178

Suppose that array is an array of ints (arranged in increasing order) and target is an int. Consider the follow two code segments, implementing a sequential search (on the left) and a binary search (on the right):

|  |  |
| --- | --- |
| int i = 0;  boolean found = false;   while ( !found &&            i < array.length )  {    if ( array[ i ] == target )    {      found = true;      System.out.println(          "Found at " + i        );    }     i++;  } | int low = -1,      high = array.length,      probe;   while ( high - low > 1 )  {    probe = ( high + low ) / 2;    if ( array[ probe ] > target )      high = probe;    else      low = probe;  }   if ( low != -1 &&          array[ low ] == target )    System.out.println(        "Found at " + low      ); |

Compare the operations of these two search algorithms by answering the following questions. (You will probably find that it helps to use pencil and paper as you single-step carefully through the code with a view to counting the number of iterations.)

1. If array contains 10 elements and there is just one occurrence (at index 0) of target as an element of array, how many iterations does
   1. the sequential search
   2. the binary search

require in order to find target?

1. If array contains 10 elements and there is just one occurrence (at index 9) of target as an element of array, how many iterations does
   1. the sequential search
   2. the binary search

require in order to find target?

1. If array contains 10 elements and target is not one of them, how many iterations does it take
   1. the sequential search
   2. the binary search

to discover this fact?

1. (i) 1 (ii) 3
2. (i) 10 (ii) 4
3. (i) 10 (ii) at most 4 (It depends how big target is in relation to the elements of array. If all the elements are less than target, then 4 iterations will be required; if all the elements are greater than target then 3 iterations will be required. Otherwise 4 iterations will be required.)

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 2n–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 log2k (denoted symbolically by "⌈log2k⌉").

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 | ⌈log2k⌉ |
| 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.

#### Exercise 179

To calculate the number of iterations of a binary search, take the number of elements in the array and find the smallest power of 2 greater than or equal to it. The number of iterations is equal to the exponent of that power. For example, consider an array of 1,000 elements. The smallest power of 2 greater than or equal to 1,000 is 1,024. Since 1024 = 210, we conclude that a binary search of a 1,000-element array will take at most 10 iterations.

1. At most how many iterations will be needed for
   1. a sequential search
   2. a binary search

of a 60-element array?

1. At most how many iterations will be needed for
   1. a sequential search
   2. a binary search

of a 250-element array?

1. At most how many iterations will be needed for
   1. a sequential search
   2. a binary search

of a 500-element array?

1. (i) 60 (ii) 6 (26 = 64)
2. (i) 250 (ii) 8 (28 = 256)
3. (i) 500 (ii) 9 (29 = 512)

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. 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.

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, low, probe );  
    else   
      low = binarySearch( array, target, probe, high );  
  }  
  
  if ( low == -1 || array[ low ] != target )  
    return -1;  
  
  return low;  
}  
}

### Part 2

Binary searching is very efficient and is the algorithm of choice when the data structure is very large. However, it can be tricky to implement, so carefully study the Lecture Notes and the demo programs provided.