**Sequences**

A **sequential (the ability to access or search for information only by traversing in order)** search is the easiest of the searching algorithms to implement. However, it is not the most efficient method. It involves traversing through an array from beginning to end (linearly) while comparing values with each element. Normally, once an element is found, the searching stops. In the worst case scenario, a sequential search has to go all the way through the array; this can be time-consuming for extremely large arrays.

You can compare a sequential search to going through your biology notebook to see if you have a page on single cell organisms. In the best case, you find the page soon after you start looking (say after four or five pages) and you stop. In the worst case, you flip through all 200 pages of your notebook to find that you did not even have a page on single cell organisms.

If an array is sorted, you can put a few twists on a standard sequential sort to shorten the search time. By noting that the element you are looking for must be in a certain part of the array, and once you find an element that indicates you have passed that part of the array, you can stop, as the element you want to find cannot be in the array. For example, if you have your biology notebook organized and know that the single cell page can only be in the first 40 pages of your notebook, if you reach page 41 and have not found the page you want to find, you can stop searching.

### Part 1

Sequential searching is very simple, you traverse through every element of an array from start to finish until you find what you are looking for or determine it is not present.

The process of looking through a data structure for a particular element or elements is called *searching*.

If we search through an array by traversing its elements one by one in the order in which they occur in the array, then we are said to be conducting a *sequential search*. (You might also see it referred to as a *linear search*.) In the following code, for example, a sequential search is made for an Item with a myN value of 3:

  public static void main( String[] args )   
  {   
    Item[] array =    
      {    
        new Item( 1 ),    
        new Item( 5 ),    
        new Item( 3 ),    
        new Item( 4 ),    
        new Item( 7 ),    
        new Item( 10 ),   
      };   
  
    Item.displayArray( array );   
  
    boolean found = false;   
    int i = 0;   
    while ( !found && i < array.length )   
    {   
      if ( array[ i ].getN() == 3 )   
      {   
        found = true;   
        System.out.println( "Found 3 at index " + i );   
      }   
  
      i++;   
    }   
  }

Typically, a sequential search stops once the sought-for element is found. The use of while together with a boolean whose value changes once the element is found can be very effective. An alternative approach is to use a for-loop that is terminated by a return statement once the item is found. This approach is used in the following code:

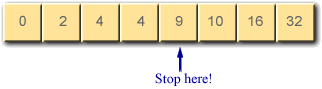
  public static int sequentialSearch( Item[] array, int target )   
  {   
    for ( int i = 0 ; i < array.length ; i++ )   
    {       
      if ( array[ i ].getN() == target )   
        return i;   
    }   
  
    return -1; // not found   
  }   
  
  public static void main( String[] args )   
  {   
    Item[] array =    
      {    
        new Item( 1 ),    
        new Item( 0 ),    
        new Item( 4 ),    
        new Item( 5 ),    
        new Item( 3 ),    
        new Item( 0 ),   
      };   
  
    Item.displayArray( array );   
  
    int i = sequentialSearch( array, 3 );   
    System.out.println( "Found 3 at index " + i );   
  }

A disadvantage of a sequential search is that it may have to traverse the entire array. This is not a problem in the context of small arrays such as the ones on the previous page, but it would be extremely time-consuming when searching, say, an array containing information on the tens of thousands of bank accounts at a single bank.

The sequential search algorithm is particularly inefficient if the target does not appear in the array, since we must look at each and every element of the array before we can conclude that the target is not there. For example, to discover that the integer 5 is not contained in the array shown below, we must search the entire array:

Sequential search, unordered

However, if it is known that the elements of the array are sorted into some kind of order, then the seriousness of this problem is often reduced. In the next figure, the integers in the above array have been sorted into increasing order. If we are searching this (ordered) array for the integer 5 and we reach an integer that is greater than 5, then we can conclude that there is no 5 in the array. We may therefore stop the search before completing the traversal of the array. In this case, we reach the integer 9 and immediately stop the search, knowing that 5 is not to be found in this array.



Implement a new class method, findItem, that takes two arguments, Item[] array and int target, where it is known that the Items in the input array are sorted into ascending order according to the values of their myN instance variables. The method should conduct a sequential search for an Item having a myN instance variable with value target. If such an Item is present in the input array, findItem should return the Item; if not, it should return a null reference. The method should take advantage of the fact that array is sorted to improve efficiency in the case when there is no Item in array with a myN instance variable having the desired value.

public class 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 );   
  }   
  
}   
  public static Item findItem( Item[] array, int target )  
  {  
    int i = 0;  
    while ( i < array.length && array[ i ].getN() <= target )  
    {  
      if ( array[ i ].getN() == target )  
        return array[ i ];  
  
      i++;  
    }  
  
    return null;  
  }  
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    Item[] array =   
      {    
         new Item( 0 ), new Item( 3 ),    
         new Item( 3 ), new Item( 5 ),    
         new Item( 5 ), new Item( 5 ),    
         new Item( 7 ), new Item( 10 )    
      };   
  
    System.out.println( "Found " + Item.findItem( array,  ) );   
  }   
}

### Part 2

Even though a sequential search is straightforward, it is important to master the details of this algorithm. The Lecture Notes and demonstration programs will allow you to quickly begin applying sequential searching techniques.