Linear search

```java
/**
  A class for executing linear searches in an array.
*/
public class LinearSearcher
{
  /**
     Finds a value in an array, using the linear search
     algorithm.
     @param a the array to search
      @param value the value to find
      @return the index at which the value occurs, or -1
      if it does not occur in the array
    */
    public static int search(int[] a, int value)
    {
      for (int i = 0; i < a.length; i++)
      {
         if (a[i] == value) { return i; }
      }
      return -1;
    }
}
```

```java
import java.util.Arrays;
import java.util.Scanner;

/**
   This program demonstrates the linear search algorithm.
*/
public class LinearSearchDemo
{
   public static void main(String[] args)
   {
      int[] a = ArrayUtil.randomIntArray(20, 100);
      System.out.println(Arrays.toString(a));
      Scanner in = new Scanner(System.in);

      boolean done = false;
      while (!done)
      {
         System.out.print("Enter number to search for, -1 to quit: ");
         int n = in.nextInt();
         if (n == -1)
         {
            done = true;
         }
         else
         {
            int pos = LinearSearcher.search(a, n);
            System.out.println("Found in position " + pos);
         }
      }
   }
}
```

Continued

Suppose you need to look through 1,000,000 records to find a telephone number. How many records do you expect to search before finding the number?

Answer: On average, you'd make 500,000 comparisons.

- Average 500,000 is n/2 when n=1,000,000.
- Worst case is "n"
- **Oh notation: O(n)**

- There are a fixed number of actions in each visit independent of n.
- A loop with n iterations has $O(n)$ running time if each step consists of a fixed number of actions.

How to calculate a double loop

What is the big-Oh running time of the following algorithm to check whether an array has a duplicate value?

```
for (int i = 0; i < a.length; i++)
{
   for (j = i + 1; j < a.length; j++)
   {
      if (a[i] == a[j]) { return true; }
   }
}
return false;
```

Answer: It is an $O(n^2)$ algorithm—the number of visits follows a triangle pattern.

Binary search

- The array must be a sorted array
- Use linear search if the array is not sorted

```java
/**
  A class for executing binary searches in an array.
*/
public class BinarySearcher
{
  /**
     Finds a value in a range of a sorted array, using the binary
     search algorithm.
     @param a the array in which to search
     @param low the low index of the range
     @param high the high index of the range
     @param value the value to find
     @return the index at which the value occurs, or -1
     if it does not occur in the array
   */

  public static int search(int[] a, int low, int high, int value)
  {
    if (low <= high)
    {
      int mid = (low + high) / 2;

      if (a[mid] == value)
      {
        return mid;
      }
      else if (a[mid] < value )
      {
        return search(a, mid + 1, high, value);
      }
      else
      {
        return search(a, low, mid - 1, value);
      }
    }
    else
    {
      return -1;
    }
  }
}
```

Binary Search

- Count the number of visits to search a sorted array of size n
  - We visit one element (the middle element) then search either the left or right subarray
  - Thus: $T(n) = T(n/2) + 1$
- If n is $n / 2$, then $T(n / 2) = T(n / 4) + 1$
- Substituting into the original equation: $T(n) = T(n / 4) + 2$
- This generalizes to: $T(n) = T(n / 2^k) + k$

- Assume n is a power of 2, $n = 2^m$ where $m = \log_2(n)$
- Then: $T(n) = 1 + \log_2(n)$
- A binary search locates a value in a sorted array in $O(\log(n))$ steps.

Binary Search

- Should we sort an array before searching?
  - Linear search - O(n)
  - Binary search - O(n log(n))
- If you search the array only once
  - Linear search is more efficient
- If you will make many searches
  - Worthwhile to sort and use binary search

Table 1 Common Big-Oh Growth Rates

Big-Oh Expression	Name
$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(n)$	Linear
$O(n \log(n))$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Growth Rate	Name	Code e.g.	description
1	Constant	a+=1;	statement (one line of code)

log(n)	Logarithmic	`while (n>1){`
		`  n=n/2;`
		`}`
			Divide in half (binary search)

n	Linear	`for(c=0; c<n; c++){`
		`  a+=1;`
		`}`
			Loop

n*log(n)	Linearithmic	Mergesort, Quicksort, ...	Effective sorting algorithms

n^2	Quadratic	`for(c=0; c<n; c++){`
		`  for(i=0; i<n; i++){`
		`    a+=1;`
		`  }`
		`}`
			Double loop

n^3	Cubic	`for(c=0; c<n; c++){`
		`  for(i=0; i<n; i++){`
		`    for(x=0; x<n; x++){`
		`      a+=1;`
		`    }`
		`  }`
		`}`
			Triple loop

2^n	Exponential	Trying to braeak a password generating all possible combinations	Exhaustive search

This table is on http://adrianmejia.com/

How to measure algorithm more precisely

- We create a class of "StopWatch"
- Then use it to measure the processing time of a method

```
StopWatch timer = new StopWatch();
timer.start();
// call a method here, to measure performance
...
timer.stop();
System.out.println("Elapsed time: "
            + timer.getElapsedTime() + " milliseconds");
```

Here the class in the text book (chapter 14: 14.2)

```
/**
   A stopwatch accumulates time when it is running. You can
   repeatedly start and stop the stopwatch. You can use a
   stopwatch to measure the running time of a program.
*/
public class StopWatch
{
   private long elapsedTime;
   private long startTime;
    private boolean isRunning;

    /**
       Constructs a stopwatch that is in the stopped state
       and has no time accumulated.
    */
    public StopWatch()
    {
       reset();
    }

   /**
      Starts the stopwatch. Time starts accumulating now.
   */
   public void start()
   {
      if (isRunning) { return; }
      isRunning = true;
      startTime = System.currentTimeMillis();
   }

   /**
      Stops the stopwatch. Time stops accumulating and is
      is added to the elapsed time.
   */
   public void stop()
   {
      if (!isRunning) { return; }
      isRunning = false;
      long endTime = System.currentTimeMillis();
      elapsedTime = elapsedTime + endTime - startTime;
   }
```

```
/**
   Returns the total elapsed time.
   @return the total elapsed time
*/
public long getElapsedTime()
{
   if (isRunning)
   {
      long endTime = System.currentTimeMillis();
      return elapsedTime + endTime - startTime;
   }
   else
   {
      return elapsedTime;
   }
}

  /**
     Stops the watch and resets the elapsed time to 0.
  */
  public void reset()
  {
     elapsedTime = 0;
     isRunning  = false;
  }
}
```