

AP Computer Science B

Java Object-Oriented Programming [Ver. 2.0]

Unit 5: Algorithm Study

WEEK 11: CHAPTER 16 ALGORITHMS (PART 1: BASIC ALGORITHMS)

DR. ERIC CHOU

IEEE SENIOR MEMBER



Objectives

- Standard Algorithms Review
- Euclid's Algorithm
- Finding Prime Numbers
- Linear Search
- Binary Search



Standard Algorithms

LECTURE 1



Written Essay in Exams

- (1) No computer allowed.
- (2) Test on the programming ideas.
- (3) Time limited.
- (4) Design-centric. (will not be data processing centric or GUI-centric)



Purpose of this Lecture

A Collection of Algorithms and Patterns for Exams

- This lecture works as a program collection.
- It may grow over the time. (I will keep expanding).
- When a certain big exam comes, you may always come there for the latest version.





Memorize Algorithms and Design Patterns

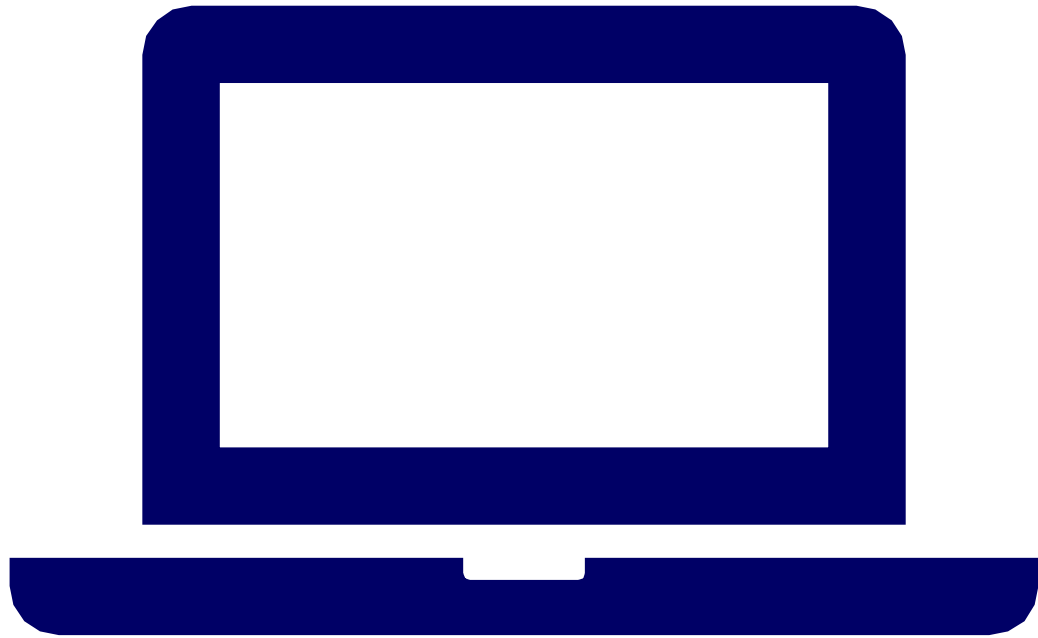
Study a few Java Program patterns:

- (1) Swap (Ch. 7), Rotation (Left, Right), Reverse(String, Array)
- (2) Shuffle (Ch. 7)
- (3) Finding max/min (Ch. 7)
- (4) Sum/average (Ch. 7)
- (5) Sort (Ch. 15)
- (6) 2-D array for shortest distance among a group of points.
(Ch. 8)



Memorize Algorithms and Design Patterns

- (7) 2-D traversal. (Ch.8)
- (8) conversion of 1-D Array to 2-D, 2-D array to 1-D (Ch.8)
- (9) split and merging of arrays (Ch. 15, Ch. 7)
- (10) insertion, deletion, replacement of array and arraylist items.
(Ch. 7, 8)
- (11) copy a block in 2-D array (Ch. 8, Lab 2)
- (12) String matching and operations (Ch. 3, Ch. 6, 7)
- (13) recursive calls. (Ch. 14)



Demonstration Program

STANDARD ARRAY PACKAGE



Euclidean Algorithm (Non-AP Topic)

LECTURE 2



Euclid's Algorithm

Given two numbers not prime to one another, to find their greatest common measure.

- What Euclid called "common measure" is termed nowadays a common factor or a common divisor.
- Euclid VII.2 then offers an algorithm for finding the greatest common divisor (gcd) of two integers. Not surprisingly, the algorithm bears Euclid's name.



Euclidean Algorithm

$a = c * f$; $b = d * f$; $a = b * t + r$; r must be f 's multiple. If f is a common factor of a and b .

- The algorithm is based on the following two observations:
 1. If $b | a$ then $\gcd(a, b) = b$.
 - This is indeed so because no number (b , in particular) may have a divisor greater than the number itself (I am talking here of non-negative integers.)
 2. If $a = bt + r$, for integers t and r , then $\gcd(a, b) = \gcd(b, r)$.
- Indeed, every common divisor of a and b also divides r . Thus $\gcd(a, b)$ divides r . But, of course, $\gcd(a, b) | b$. Therefore, $\gcd(a, b)$ is a common divisor of b and r and hence $\gcd(a, b) \leq \gcd(b, r)$. The reverse is also true because every divisor of b and r also divides a .



Example

Example

Let $a = 2322$, $b = 654$.

$$2322 = 654 \cdot 3 + 360 \quad \gcd(2322, 654) = \gcd(654, 360)$$

$$654 = 360 \cdot 1 + 294 \quad \gcd(654, 360) = \gcd(360, 294)$$

$$360 = 294 \cdot 1 + 66 \quad \gcd(360, 294) = \gcd(294, 66)$$

$$294 = 66 \cdot 4 + 30 \quad \gcd(294, 66) = \gcd(66, 30)$$

$$66 = 30 \cdot 2 + 6 \quad \gcd(66, 30) = \gcd(30, 6)$$

$$30 = 6 \cdot 5 \quad \gcd(30, 6) = 6$$

Therefore, $\gcd(2322, 654) = 6$.



Euclidean Algorithm for GCD (GCF)

$$x = 42$$
$$y = 75$$

2 → 1	42	75	1 ← 1
	33	42	
4 → 1	9	33	3 ← 3
	6	27	
	3	6	2 ← 5
		6	
		0	

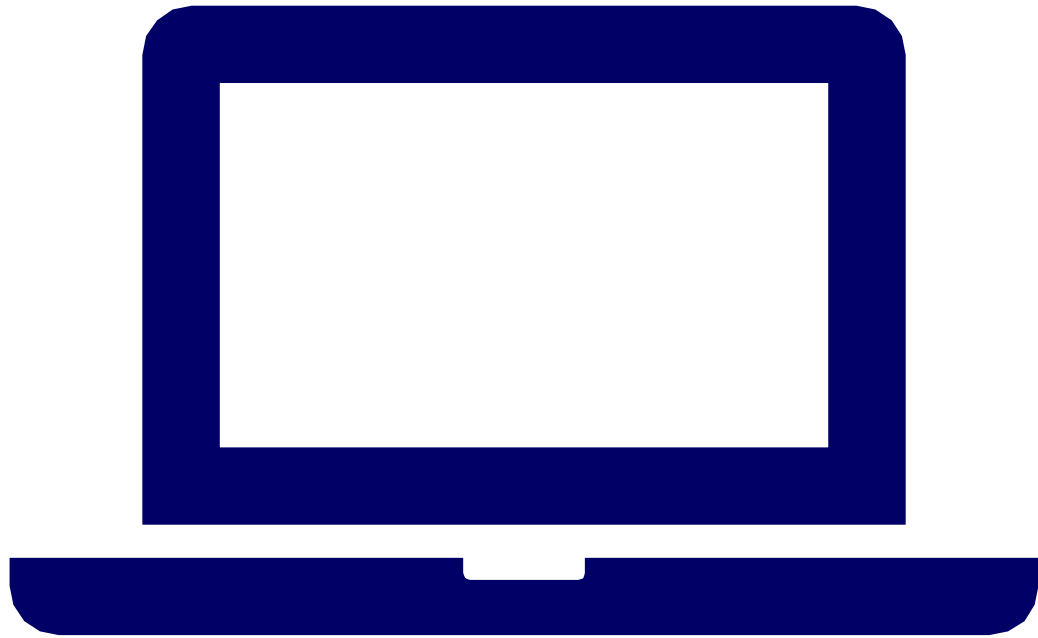
GCD



Euclid's Algorithm for GCD/GCF

```
// recursive implementation
public static int gcd(int p, int q) {
    if (q == 0) return p;
    else return gcd(q, p % q);
}
```

```
// non-recursive implementation
public static int gcd2(int p, int q) {
    while (q != 0) {
        int temp = q;
        q = p % q;
        p = temp;
    }
    return p;
}
```



Demonstration Program

EUCLID.JAVA



Demo Program:

[Euclid.java](#)

Moral of the story: good algorithm may not come from learning algorithms but from inventing them.



Finding Prime Numbers (Non-AP Topic)

LECTURE 3



Prime Number

- A **prime number** (or a prime) is a natural number greater than 1 that has no positive divisors other than 1 and itself.
- A natural number greater than 1 that is not a prime number is called a composite number.

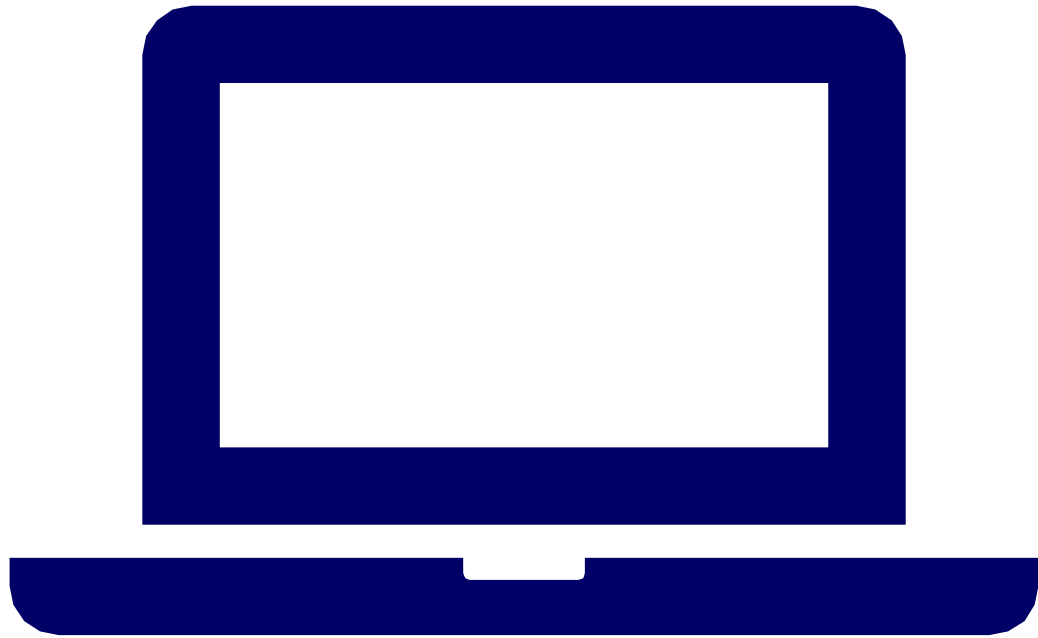


boolean isPrime(int n);

check if the number n is a prime number

Prime Number check of different efficiency:

- (1) A number is prime if all of the number smaller than or equal to it can not divide it.
- (2) A number is prime if all of the number smaller than or equal to half of it can not divide it.
- (3) A number is prime if all of the numbers smaller than or equal to the square root of it can not divide it.
- (4) A number is prime if all of the prime numbers smaller than it can not divide it.



Demonstration Program

PRIMENUMBER.JAVA



Demo Program: PrimeNumber.java

Moral of Story: Don't get stuck at the low performance algorithms. There might be some better ways of doing things. Find the best algorithm before you perform coding.



Linear Search

(Enhanced with Implementation
from Chapter 7)

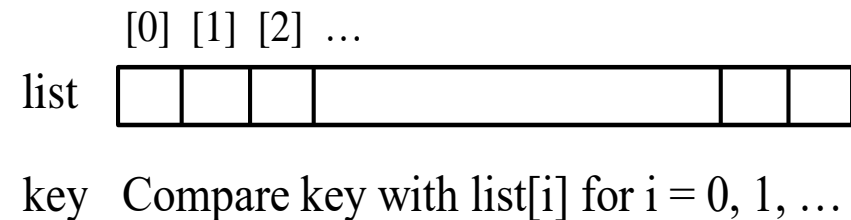
LECTURE 4



Searching Arrays

Searching is the process of looking for a specific element in an array; for example, discovering whether a certain score is included in a list of scores. Searching is a common task in computer programming. There are many algorithms and data structures devoted to searching. In this section, two commonly used approaches are discussed, **linear search** and **binary search**.

```
public class LinearSearch {  
    /** The method for finding a key in the list */  
    public static int linearSearch(int[] list, int key) {  
        for (int i = 0; i < list.length; i++)  
            if (key == list[i])  
                return i;  
        return -1;  
    }  
}
```





Linear Search

The linear search approach compares the key element, **key**, *sequentially* with each element in the array **list**. The method continues to do so until the key matches an element in the list or the list is exhausted without a match being found. If a match is made, the linear search returns the index of the element in the array that matches the key. If no match is found, the search returns **-1**.

Linear Search Animation

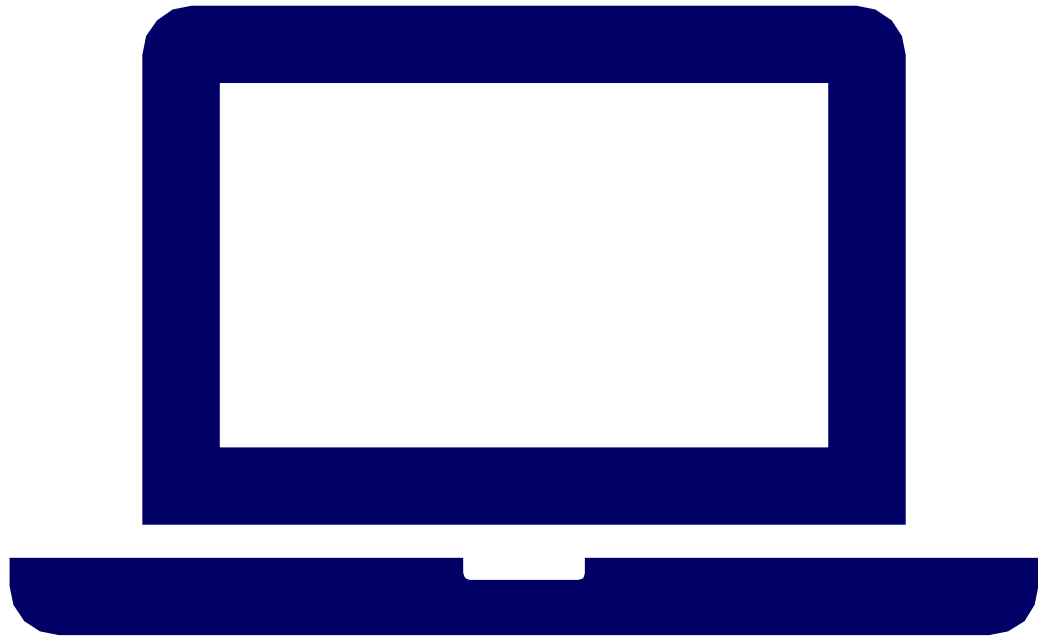
Key	List							
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8

From Idea to Solution

```
/** The method for finding a key in the list */  
public static int linearSearch(int[] list, int key) {  
    for (int i = 0; i < list.length; i++)  
        if (key == list[i])  
            return i;  
    return -1;  
}
```

Trace the method

```
int[] list = {1, 4, 4, 2, 5, -3, 6, 2};  
int i = linearSearch(list, 4); // returns 1  
int j = linearSearch(list, -4); // returns -1  
int k = linearSearch(list, -3); // returns 5
```



Demonstration Program

LINEARSEARCH.JAVA



Binary Search

(Enhanced with Implementation
from Chapter 7)

LECTURE 5



Binary Search (on sorted array)

- For binary search to work, the elements in the array must already be ordered. Without loss of generality, assume that the array is in ascending order.

e.g., 2 4 7 10 11 45 50 59 60 66 69 70 79

- The binary search first compares the key with the element in the middle of the array.



Binary Search, cont.

Consider the following three cases:

- If the **key** is less than the middle element, you only need to search the key in the first half of the array.
- If the **key** is equal to the middle element, the search ends with a match.
- If the **key** is greater than the middle element, you only need to search the key in the second half of the array.



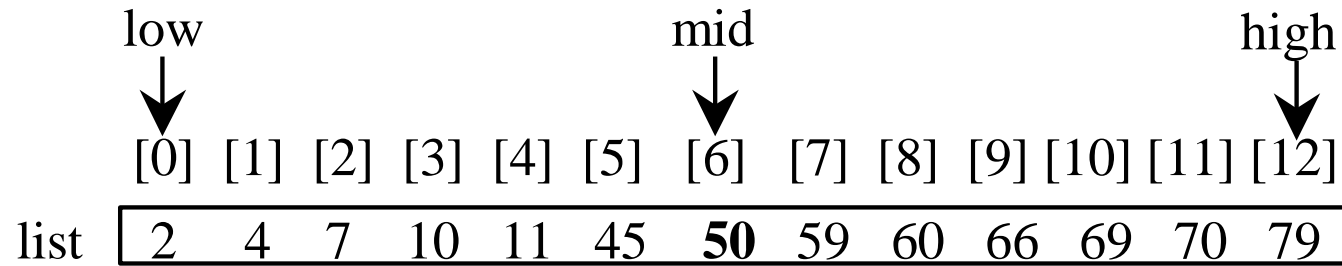
Binary Search

Key	List								
8	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table>	1	2	3	4	6	7	8	9
1	2	3	4	6	7	8	9		
8	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table>	1	2	3	4	6	7	8	9
1	2	3	4	6	7	8	9		
8	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table>	1	2	3	4	6	7	8	9
1	2	3	4	6	7	8	9		

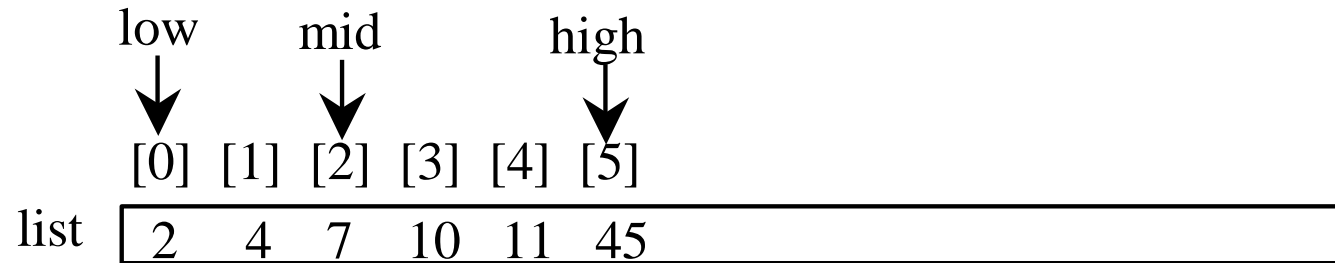
Binary Search, cont.

key is 11

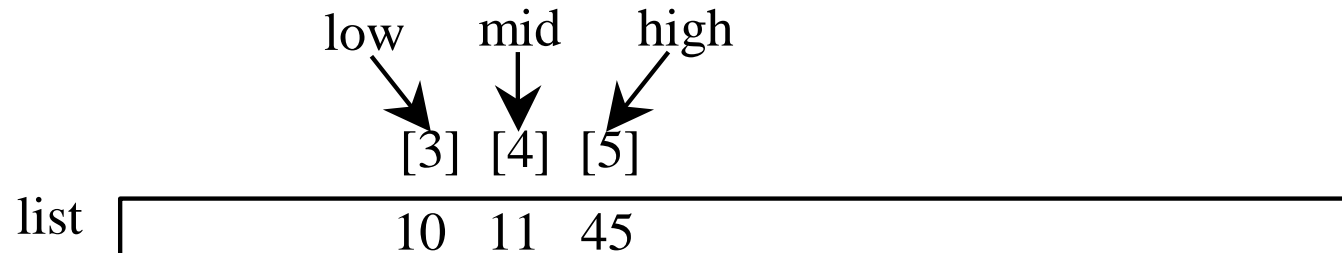
key < 50



key > 7



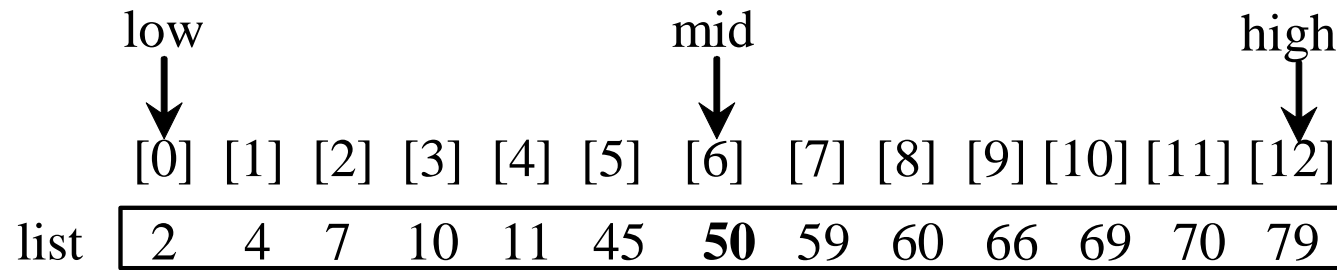
key == 11



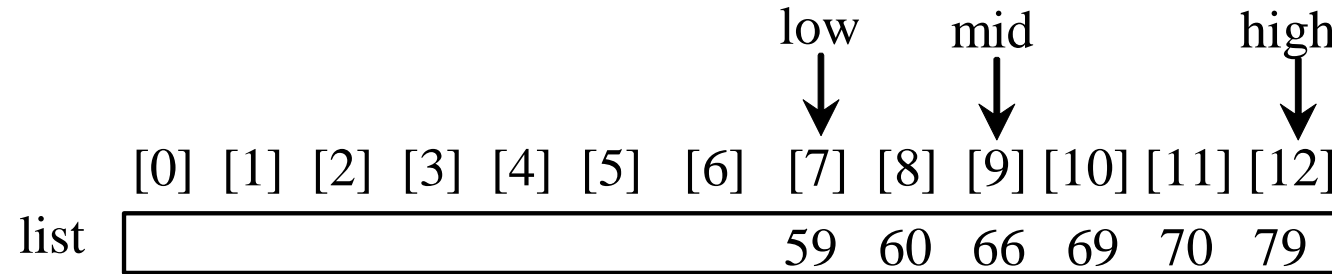


key is 54

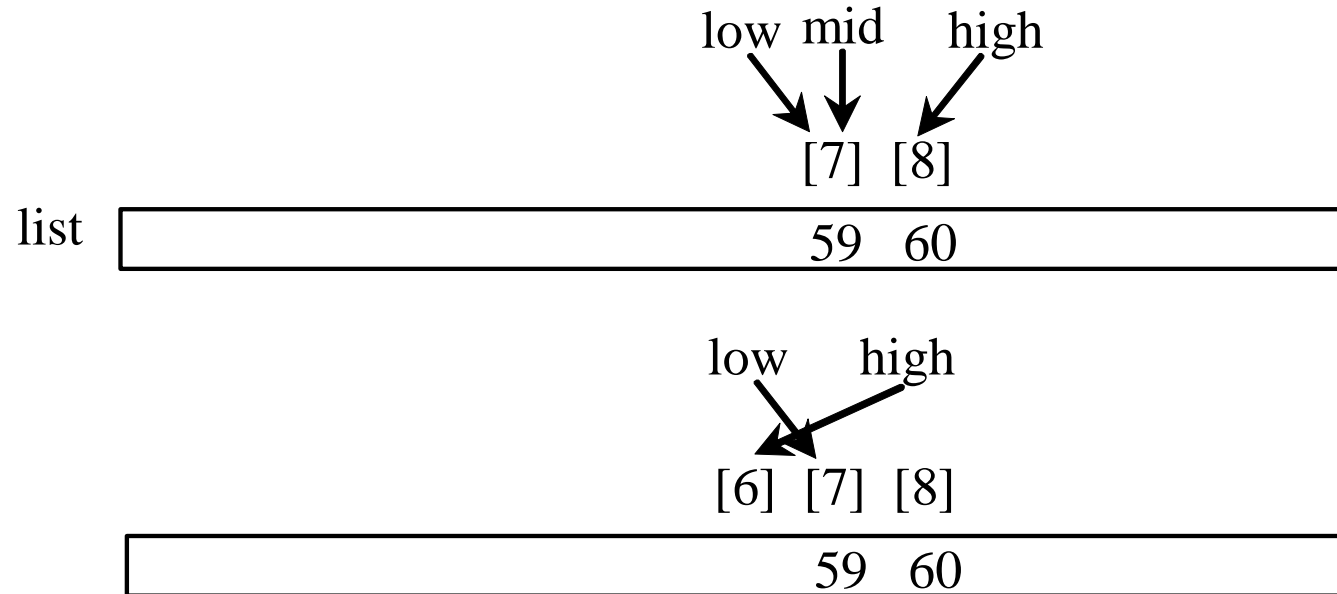
key > 50



key < 66



key < 59





Binary Search, cont.

- The binarySearch method returns the index of the element in the list that matches the search key if it is contained in the list. Otherwise, it returns **`-insertion point - 1`**.
`insertion point = -(return+1)`
- The insertion point is the point at which the key would be inserted into the list.

Exemplary Binary Search Method

```
public static int binarySearch(int[] list, int key) {  
    int low = 0;  
    int high = list.length - 1;  
    while (high >= low) {  
        int mid = (low + high) / 2;  
        if (key < list[mid])  
            high = mid - 1;  
        else if (key == list[mid])  
            return mid;  
        else  
            low = mid + 1;  
    }  
    return -1 - low;  
}
```



Logarithm: Analyzing Binary Search

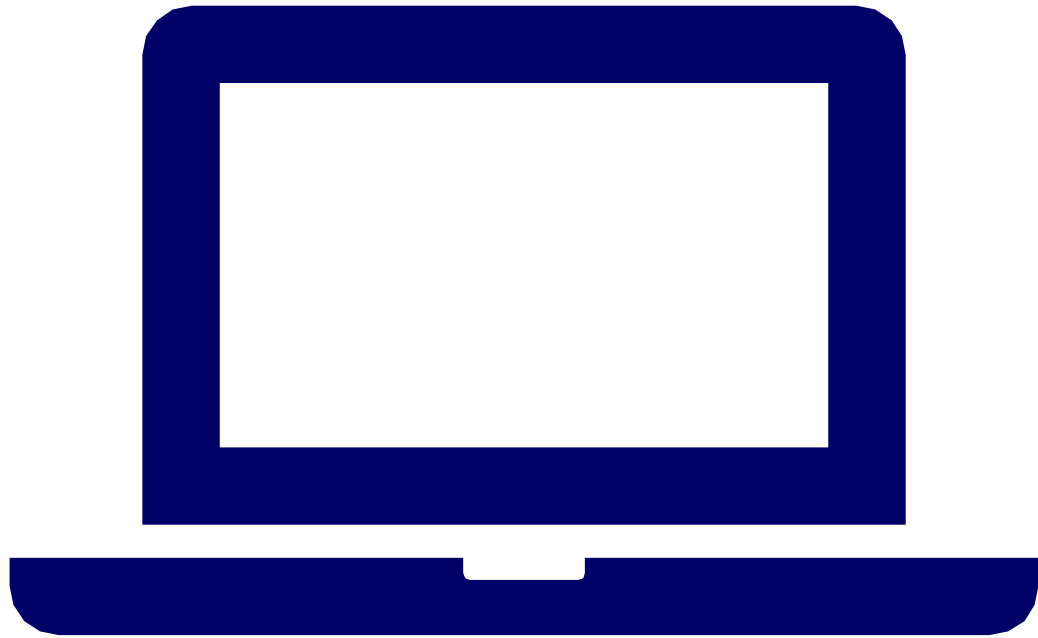
- BinarySearch.java, searches a key in a sorted array. Each iteration in the algorithm contains a fixed number of operations, denoted by c . Let $T(n)$ denote the time complexity for a binary search on a list of n elements. Without loss of generality, assume n is a power of 2 and $k = \log n$. Since binary search eliminates half of the input after two comparisons,

$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{2^2}\right) + c + c = \dots = T\left(\frac{n}{2^k}\right) + ck = T(1) + c \log n = 1 + c \log n$$



Logarithmic Time

- Ignoring constants and smaller terms, the complexity of the binary search algorithm is $O(\log n)$. An algorithm with the time complexity is called a *logarithmic algorithm* $O(\log n)$.
- The base of the log is 2, but the base does not affect a logarithmic growth rate, so it can be omitted. The logarithmic algorithm grows slowly as the problem size increases. If you square the input size, you only double the time for the algorithm.



Demonstration Program

BINARYSEARCH.JAVA