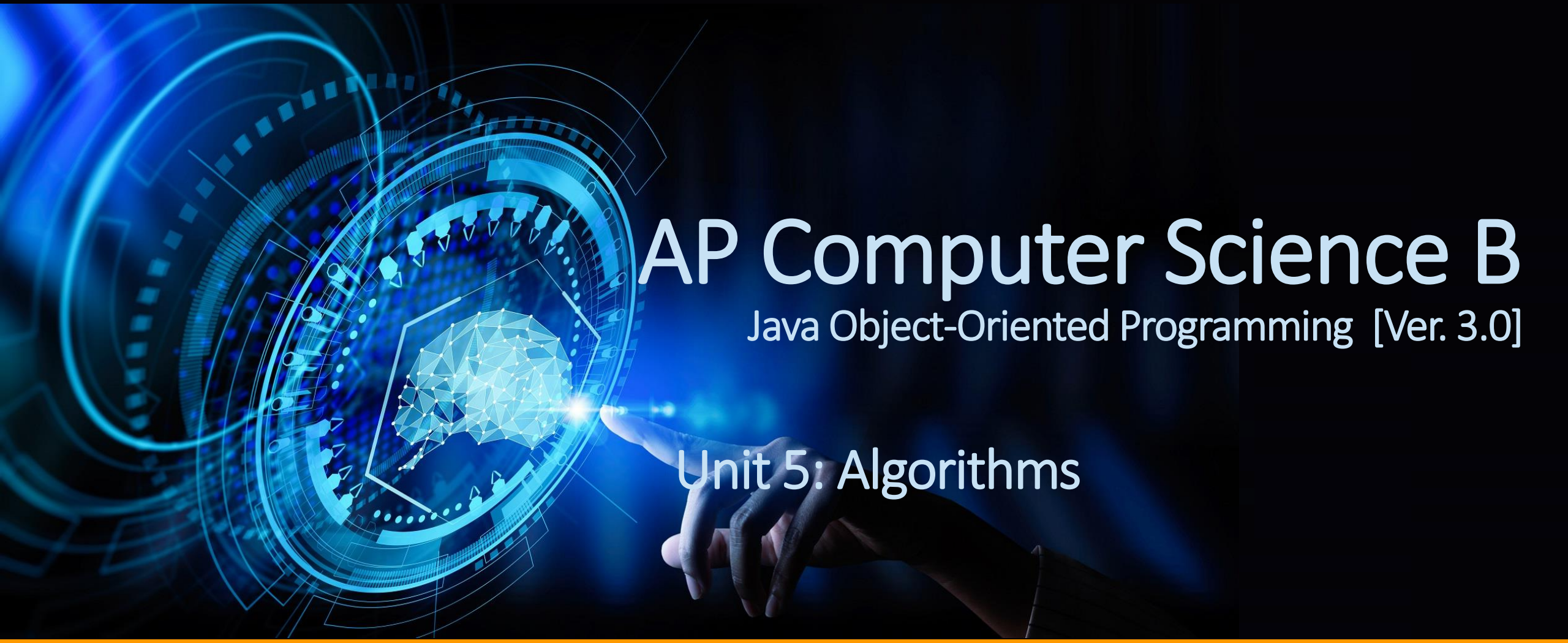


AP Computer Science B

Java Object-Oriented Programming [Ver. 3.0]

Unit 5: Algorithms

CHAPTER 16A: BASIC ALGORITHMS

DR. ERIC CHOU IEEE SENIOR MEMBER

Objectives

- Standard Algorithms in APCSA
- Modulus Arithmetic
- GCD Algorithms
- Prime Number Algorithms
- Searching Algorithms
  - Linear Search
  - Binary Search

Standard Algorithms in APCSA

LECTURE 1

Free Response 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 fore 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)

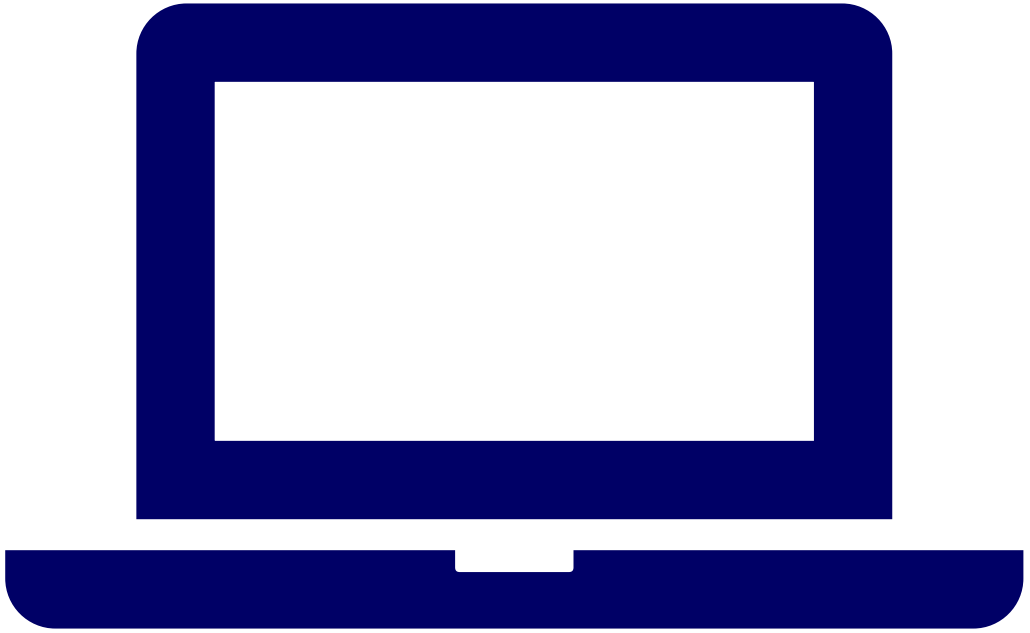
(4) Sum/average (Ch. 7)

(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

Modulus Arithmetic

LECTURE 2

Modular Arithmetic

- Modular arithmetic is a system of arithmetic for integers, which considers the remainder. In modular arithmetic, numbers "wrap around" upon reaching a given fixed quantity (this given quantity is known as the modulus) to leave a remainder.
- Modular arithmetic is often tied to prime numbers, for instance, in Wilson's theorem, Lucas's theorem, and Hensel's lemma, and generally appears in fields like cryptography, computer science, and computer algebra.

Modular Arithmetic

- An intuitive usage of modular arithmetic is with a 12-hour clock. If it is 10:00 now, then in 5 hours the clock will show 3:00 instead of 15:00. 3 is the remainder of 15 with a modulus of 12.

Congruence

- A number ***x* mod *N*** is the equivalent of asking for the remainder of ***x*** when divided by ***N***. Two integers ***a*** and ***b*** are said to be congruent (or in the same equivalence class) modulo ***N*** if they have the same remainder upon division by ***N***. In such a case, we say that $a \equiv b \pmod{N}$.

Modular Arithmetic as Remainders

- The easiest way to understand modular arithmetic is to think of it as finding the remainder of a number upon division by another number. For example, since both 15 and -9 leave the same remainder 3 when divided by 12, we say that

  $15 \equiv -9 \pmod{12}$.

- This allows us to have a simple way of doing modular arithmetic: first perform the usual arithmetic, and then find the remainder. For example, to find 123+321(mod11)123+321(mod11), we can take

  $123+321=444$

- and divide it by 11, which gives us

  $123+321 \equiv 4 \pmod{11}$.

Congruence

- For a positive integer ***n***, the integers ***a*** and ***b*** are congruent **mod *n*** if their remainders when divided by ***n*** are the same.

  52≡24(mod7)

- As we can see above, 52 and 24 are congruent (mod 7) because 52(mod7)=3 and 24(mod7)=3.

Note that = is different from ≡.

Addition

Properties of addition in modular arithmetic:

1. If $a + b = c$, then $a \pmod N + b \pmod N \equiv c \pmod N$.
2. If $a \equiv b \pmod N$, then $a + k \equiv b + k \pmod N$ for any integer k.
3. If $a \equiv b \pmod N$ and $c \equiv d \pmod N$, then $a + c \equiv b + d \pmod N$.
4. If $a \equiv b \pmod N$, then $-a \equiv -b \pmod N$.

Example

- It is currently 7:00 PM. What time (in AM or PM) will it be in 1000 hours?

Time "repeats" every 24 hours, so we work modulo 24. Since

$$\mathbf{1000 \equiv 16 + (24 \times 41) \equiv 16 \pmod{24},}$$

the time in 1000 hours is equivalent to the time in 16 hours. Therefore, it will be 11:00 AM in 1000 hours.

Multiplication

Properties of multiplication in modular arithmetic:

1. If $a \cdot b = c$, then $a \pmod{N} \cdot b \pmod{N} \equiv c \pmod{N}$.
2. If $a \equiv b \pmod{N}$, then $ka \equiv kb \pmod{N}$ for any integer k.
3. If $a \equiv b \pmod{N}$ and $c \equiv d \pmod{N}$, then $ac \equiv bd \pmod{N}$.

Example

- What is $(8\times16)(\mathrm{mod}7)$?
- Since $8\equiv1(\mathrm{mod}7)$ and $16\equiv2(\mathrm{mod}7)$, we have $(8\times16)\equiv(1\times2)\equiv2(\mathrm{mod}7)$.

Exponentiation

Property of Exponentiation in Modular Arithmetic:

If $a \equiv b \pmod{N}$, then $a^k \equiv b^k \pmod{N}$ for any positive integer k.

Modulo Processing

LECTURE 3

Wrap-around

- There are many cases that your program may need to design some counter that can wrap-around.
- E.g. Hour keeper that wrap-around at 12.
- E.g. Chinese Zodiac also wrap-around at 12.

Basic Wrap-around

- Counter Design for a counter that count from 0 to 5 then wrap-around back to 0.
- [0, 1, 2, 3, 4, 5]
- COUNT = 6

Basic Wrap-around

```
2  public class WrapAround
3  {
4      public static int countTo6(int x){
5          x += 1;
6          if (x >=6) return 0;
7          return x;
8      }
9      public static void main(String[] args){
10         System.out.print("\f");
11         int c = 0;
12         for (int i=0; i<20; i++){
13             System.out.print(c+"  ");
14             if (i%10==9) System.out.println();
15             c = countTo6(c);
16          }
17     }
18 }
```

```
0  1  2  3  4  5  0  1  2  3
4  5  0  1  2  3  4  5  0  1
```

Formula for Wrap-around Counter

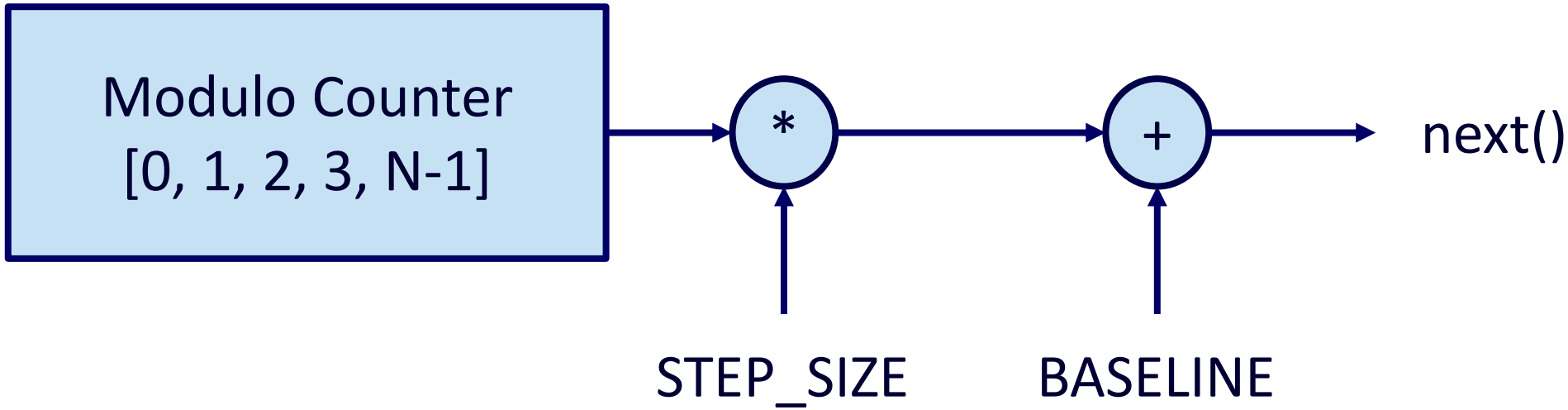
- Wrap-around counter states = [4, 8, 12, 16, 20]
- COUNT = 5; STEP_SIZE = 4; BASELINE = 4.

Formula for Wrap-around Counter

```
public class WrapAround2
{
    public static int next(int x, int COUNT){
        x += 1;
        if (x >=COUNT) return 0;
        return x;
    }
    public static void main(String[] args){
        System.out.print("\f");
        int COUNT = 5;
        int STEP = 4;
        int BASE = 4;
        int c = 0;
        for (int i=0; i<20; i++){
            System.out.print(c*STEP+BASE+"  ");
            if (i%10==9) System.out.println();
            c = next(c, COUNT);
         }
    }
}
```

```
4  8  12  16  20  4  8  12  16  20
4  8  12  16  20  4  8  12  16  20
```

Wrap-Around Counter



Formula for Wrap-around Counter

```
2  public class WrapAround3
3  {
4      public static int next(int x, int COUNT){
5          return ++x % COUNT;
6      }
7      public static void main(String[] args){
8          System.out.print("\f");
9          int COUNT = 5;
10         int STEP = 4;
11         int BASE = 4;
12         int c = 0;
13         for (int i=0; i<20; i++){
14             System.out.print(c*STEP+BASE+"  ");
15             if (i%10==9) System.out.println();
16             c = next(c, COUNT);
17          }
18     }
19 }
```

```
4  8  12  16  20  4  8  12  16  20
4  8  12  16  20  4  8  12  16  20
```

WrapAround Counter Pairs

```
public static int next(int x, int COUNT){
    return ++x % COUNT;
}
public static int prev(int x, int COUNT){
    return (--x+COUNT) % COUNT;
}
```

```
public static void main(String[] args){
    System.out.print("\f");
    int COUNT = 5;
    int STEP = 4;
    int BASE = 4;
    int c = 0;
    for (int i=0; i<10; i++){
        System.out.print(c*STEP+BASE+"  ");
        if (i%10==9) System.out.println();
        c = next(c, COUNT);
     }
    System.out.println();
    c=0;
    for (int i=0; i<10; i++){
        System.out.print(c*STEP+BASE+"  ");
        if (i%10==9) System.out.println();
        c = prev(c, COUNT);
     }
}
```

```
4  8  12  16  20  4  8  12  16  20

4  20  16  12  8  4  20  16  12  8
```

WrapCounter

```
public class WrapCounter
{
    int state = 0;
    int COUNT = 1;
    WrapCounter(int COUNT, int initial_state){
      this.COUNT = COUNT;
      state = initial_state;
    }
    public int next(){
        state = ++state % COUNT;
        return state;
    }
    public int prev(){
        state = (--state+COUNT) % COUNT;
        return state;
    }
}
```

```java
public class WrapAround5
{
    public static void main(String[] args){
        System.out.print("\f");
        int COUNT = 5;
        int STEP = 4;
        int BASE = 4;
        WrapCounter wc = new WrapCounter(COUNT, 0);
        int c=0;
        for (int i=0; i<10; i++){
            System.out.print(c*STEP+BASE+"  ");
            if (i%10==9) System.out.println();
            c = wc.next();
         }
        System.out.println();
        c=0;
        for (int i=0; i<10; i++){
            System.out.print(c*STEP+BASE+"  ");
            if (i%10==9) System.out.println();
            c = wc.prev();
         }
    }
}
```

```
4  8  12  16  20  4  8  12  16  20

4  20  16  12  8  4  20  16  12  8
```

Prime Number Algorithms

LECTURE 5

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

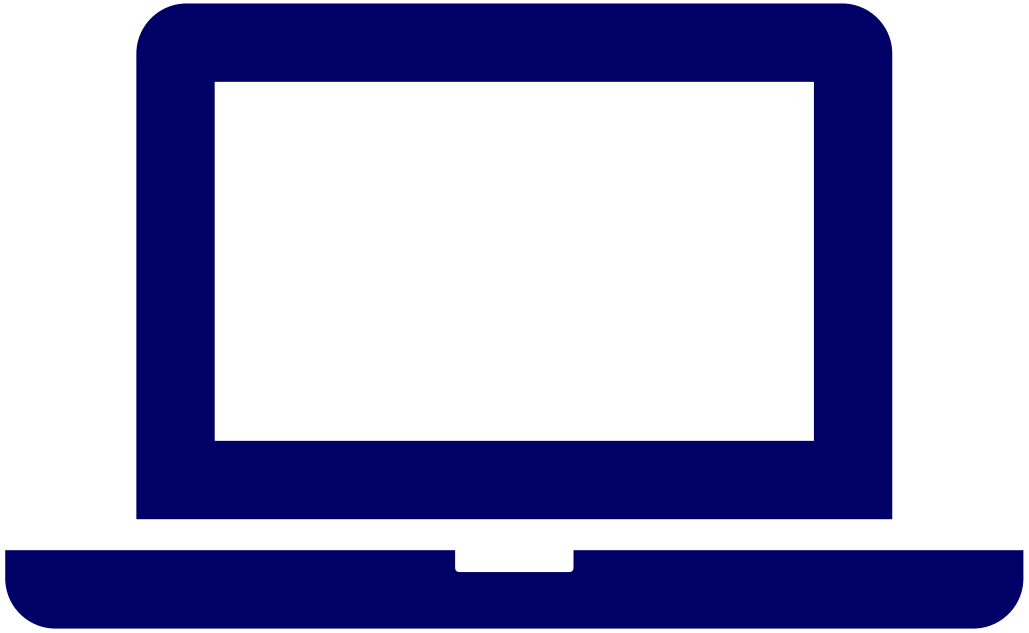
Prime Number check of different efficiency:

(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.

GCD Algorithms

LECTURE 4

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) ≤ gcd(b, r). The reverse is also true because every divisor of b and r also divides a.

Example

Example

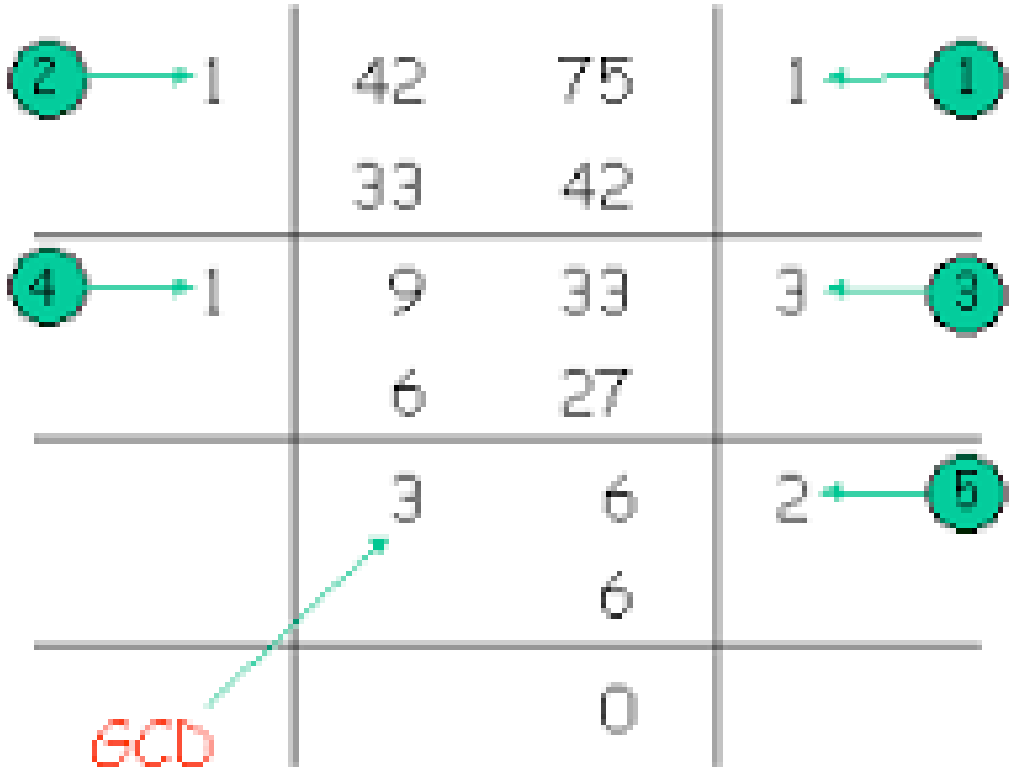
Let a = 2322, b = 654.

$2322 = 654 \cdot 3 + 360$	$\gcd(2322, 654) = \gcd(654, 360)$
$654 = 360 \cdot 1 + 294$	$\gcd(654, 360) = \gcd(360, 294)$
$360 = 294 \cdot 1 + 66$	$\gcd(360, 294) = \gcd(294, 66)$
$294 = 66 \cdot 4 + 30$	$\gcd(294, 66) = \gcd(66, 30)$
$66 = 30 \cdot 2 + 6$	$\gcd(66, 30) = \gcd(30, 6)$
$30 = 6 \cdot 5$	$\gcd(30, 6) = 6$

Therefore, gcd(2322,654) = 6.

Euclidean Algorithm for GCD (GCF)

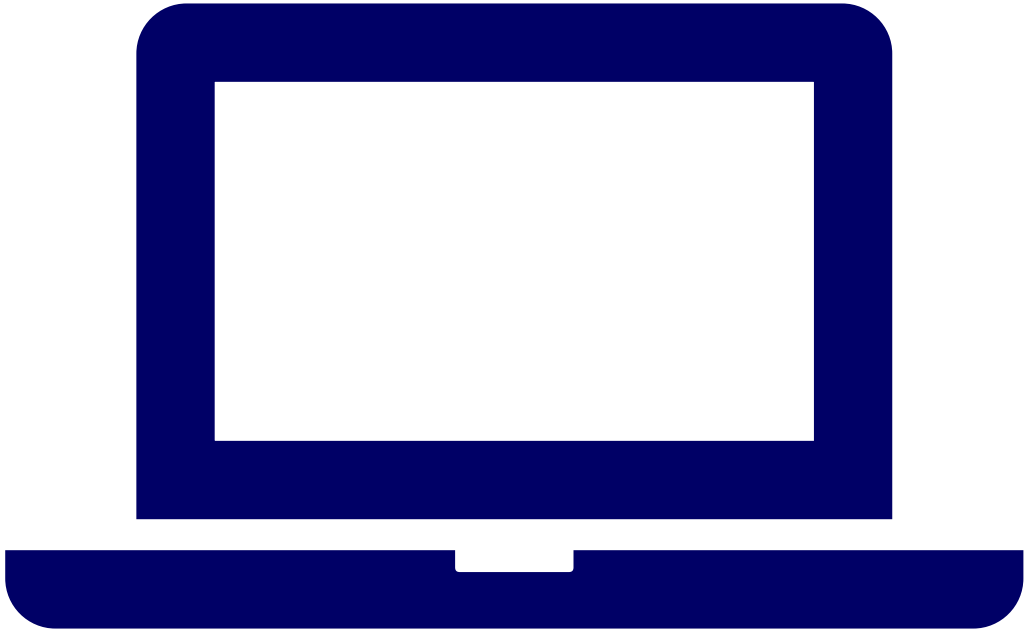
x = 42
y = 75



Euclid's Algorithm for GCD/GCF

```java
// 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.

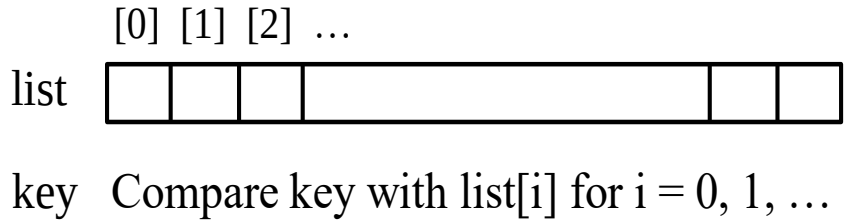
Searching Algorithms

LECTURE 6

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**.

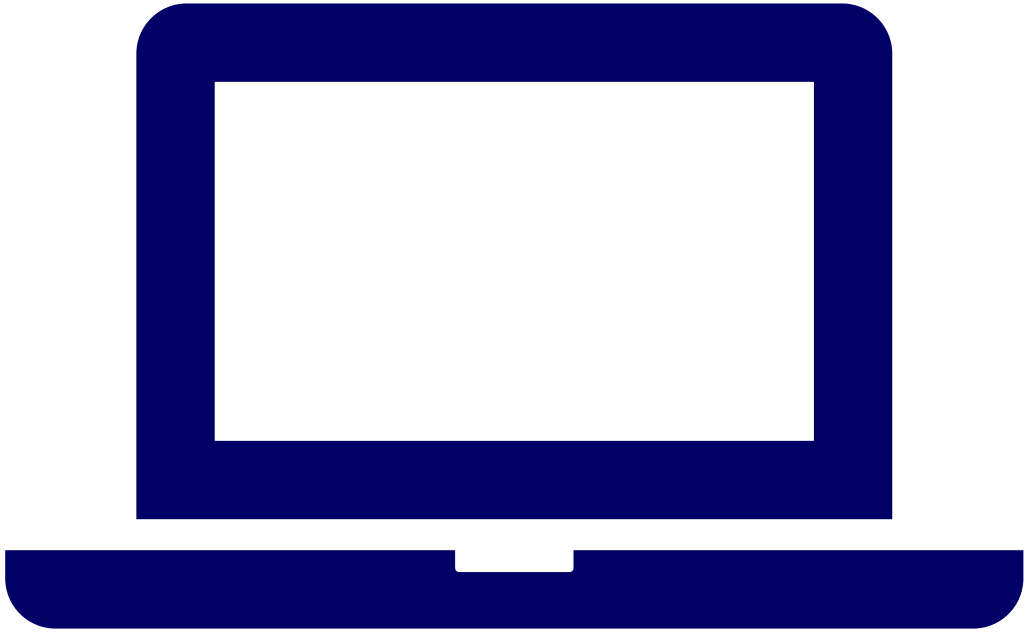
Linear Search Animation

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 (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

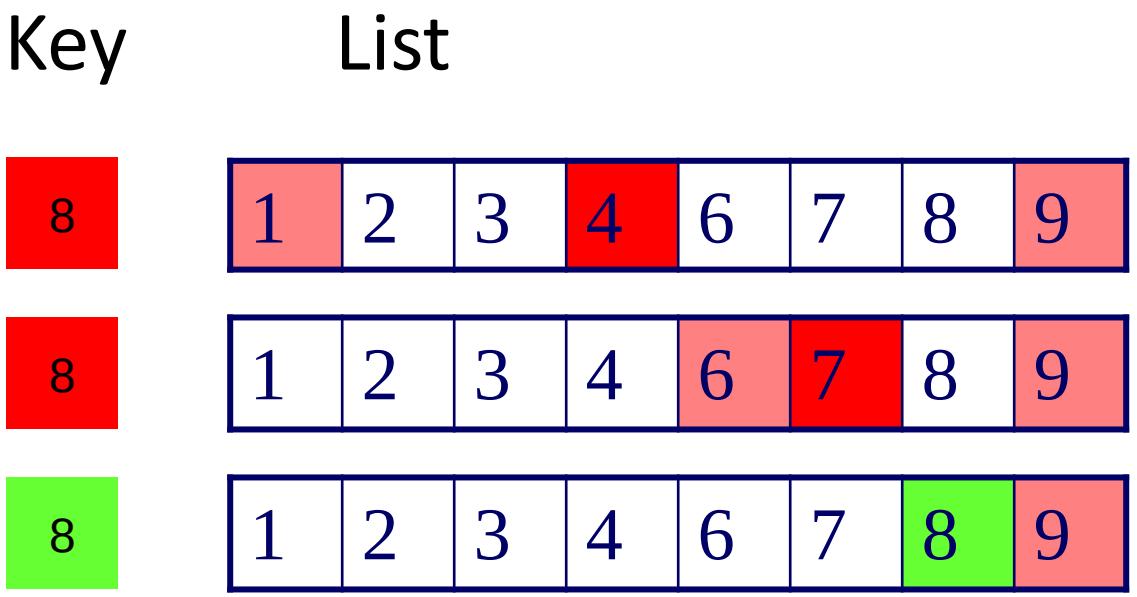
- The binary search first compares the key with the element in the middle of the array.

Binary Search, cont.

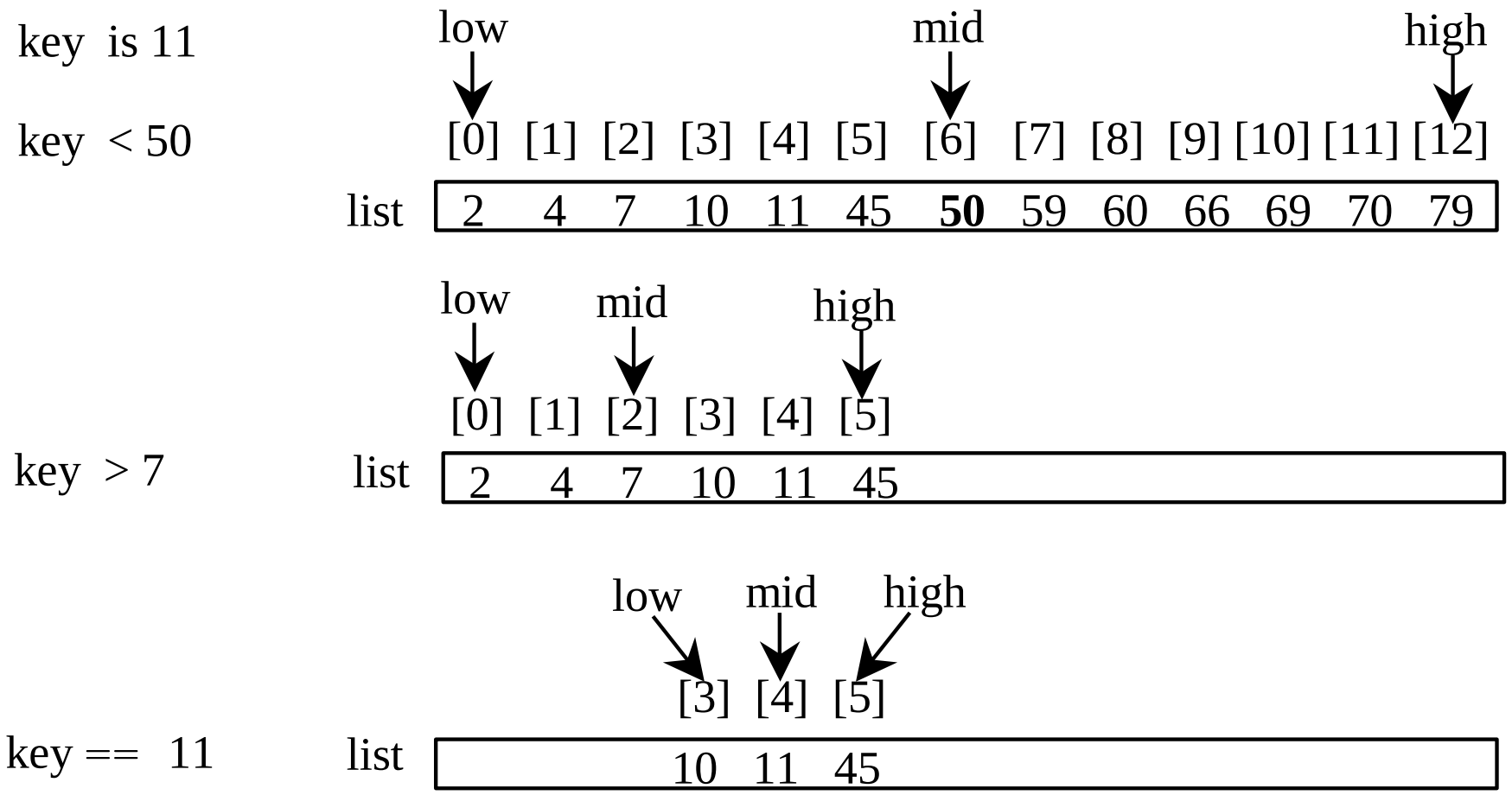
Consider the following three cases:

- 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

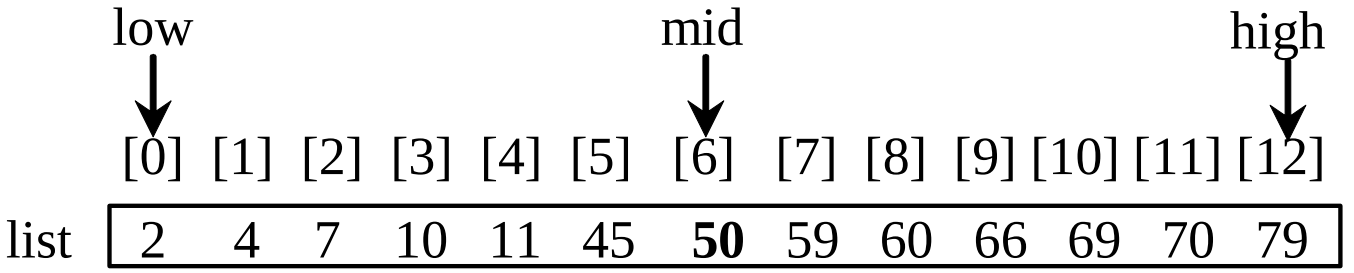


Binary Search, cont.

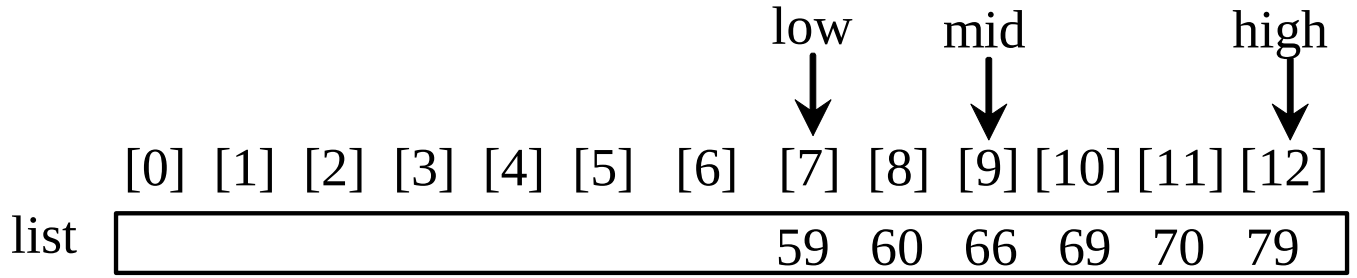


key is 54

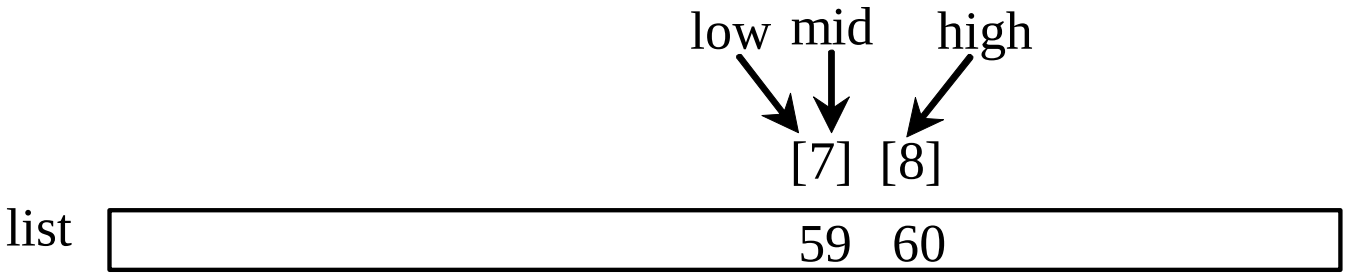
key > 50

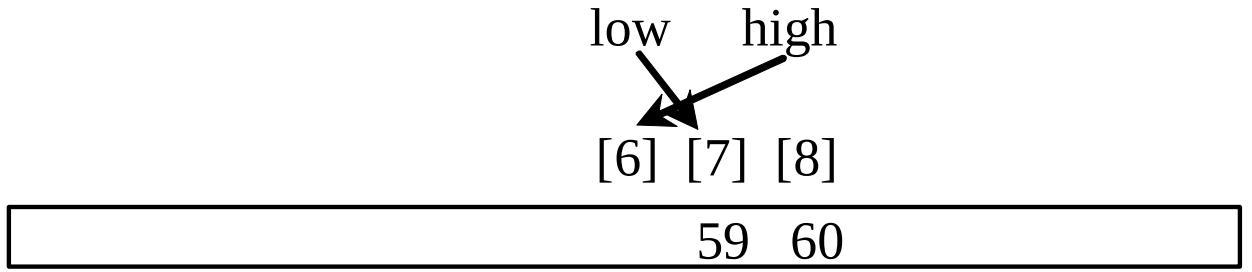


key < 66



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;
}
```

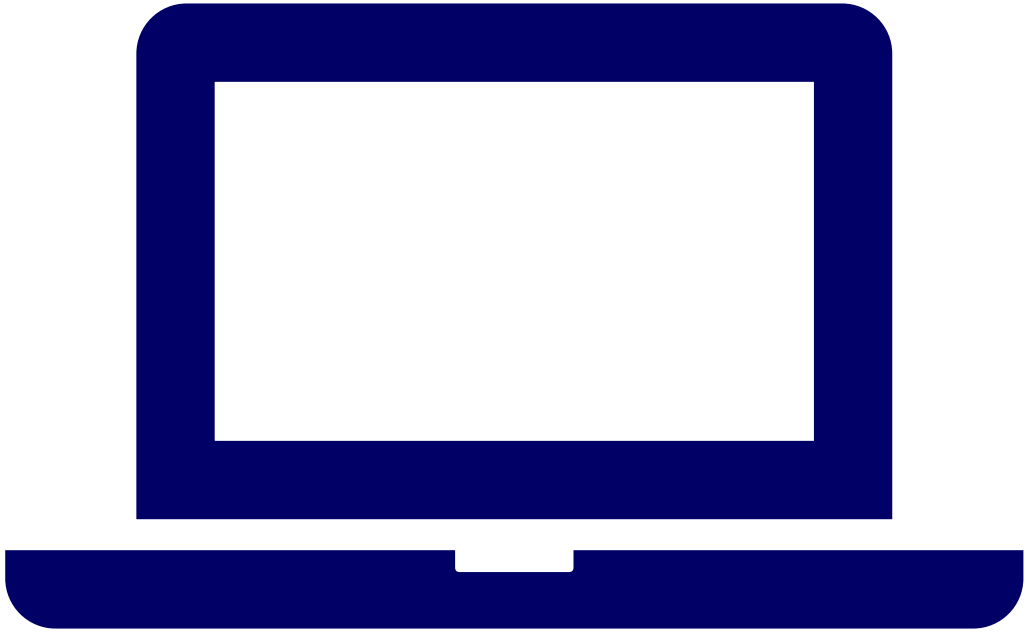
Logarithm: Analyzing Binary Search

- 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=logn. Since binary search eliminates half of the input after two comparisons,

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

Logarithmic Time

- Ignoring constants and smaller terms, the complexity of the binary search algorithm is *O(logn)*. An algorithm with the time complexity is called a *logarithmic algorithm O(logn)*.
- 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