Review

- A Review of our Toolkit
- The Object class
  - Pro: A variable of type Object can hold a value of any other type
  - Con: Processing does not know what in the Object variable
- Type Casting

```
float f = 12.0;
int i = (int)f;
Object o = new PImage(100, 100);
PImage p = (PImage)o;
```

- Built-in Collection Classes
  - ArrayList
    - Items are accessed by a consecutive integer.
  - HashMap
    - Items are accessed by an Object key.
  - Both hold Object types. May require type-casting.

Signature Polymorphism

poly = many, *morph* = form

- It is possible to define multiple functions with the <u>same name</u>, but <u>different signatures</u>.
  - A *function signature* is defined as
    - The function name, and
    - The order of variable types passed to the function

- Consider the built-in color() function …

```
color(gray)
color(gray, alpha)
color(value1, value2, value3)
color(value1, value2, value3, alpha)
…
```

Signature Polymorphism

```
void draw() { }

void mousePressed() {
  int i;
  i = 10;
  i = increment(i, 2);
  //i = increment(i);
  println(i);
}

// increment a variable
int increment(int j, int delta) {
  j = j + delta;
  return j;
}

int increment(int k) {
  k = increment(k, 1);
  return k;
}
```

In this case it is said that the increment function is ***overloaded***

Algorithm

- A well-defined set of instructions for solving a particular kind of problem.

- Algorithms exist for systematically solving many types of problems
  - Sorting
  - Searching
  - …

Euclid's algorithm for greatest common divisor

- Problem:
  - Find the greatest common divisor of two numbers A and B
- GCD Algorithm
  1. While B is not zero, repeat the following:
     - If A > B, then A=A-B
     - Otherwise, B=B-A
  2. A is the GCD

```
int A = 40902;
int B = 24140;

print("GCD of " + A + " and " + B + " is ");

while (B != 0) {
  if (A > B) {
    A = A - B;
  } else {
    B = B - A;
  }
}

println(A);
```

Sorting

- Selection Sort
  - Scan a list top to bottom and find the value that should come first.
  - Swap that item with the top position.
  - Repeat scan starting at next lowest item in the list.
  - Works best when swapping is expensive.

Selection Sort

```
// Selection Sort Example
ArrayList list = new ArrayList();
int start = 0;

void setup() {
  size(500, 500);

  // Fill the ArrayList
  list.add("Purin");
  list.add("Landry");
  list.add("Chococat");
  list.add("Pekkle");
  list.add("Cinnamoroll");

  noLoop(); // Draw once
  drawList(list);
}

void draw() { }

// Perform one pass of selection sort
void mousePressed() {
  selectOnce(list, start);
  if (start < list.size()-1) start++;
  //selectionSort(list);
}

// Perform a complete Selection Sort
void selectionSort(ArrayList al) {
  for (int i=0; i<al.size(); i++) {
    selectOnce(al, i);
  }
}
```

```
// Perform once pass of Selection Sort.
void selectOnce(ArrayList al, int i) {

  String bestVal = (String)al.get(i);
  int bestIdx = i;

  for (int j=i+1; j<al.size(); j++) {
    String s1 = (String)al.get(j);
    if (s1.compareTo(bestVal) < 1) {
      bestVal = (String)al.get(j);
      bestIdx = j;
    }
  }

  // Swap best with top position
  al.set(bestIdx, (String)al.get(i));
  al.set(i, bestVal);

  drawList(al); // Redraw list
  delay(1000);
}

// Draw the ArrayList to the sketch
void drawList(ArrayList al) {
  background(0);
  fill(255);
  textSize(20);

  int y=100;
  for (int i=0; i<al.size(); i++) {
    String s = (String)al.get(i);
    text(s, 100, y);
    y=y+50;
  }
  redraw();
}
```

Sorting

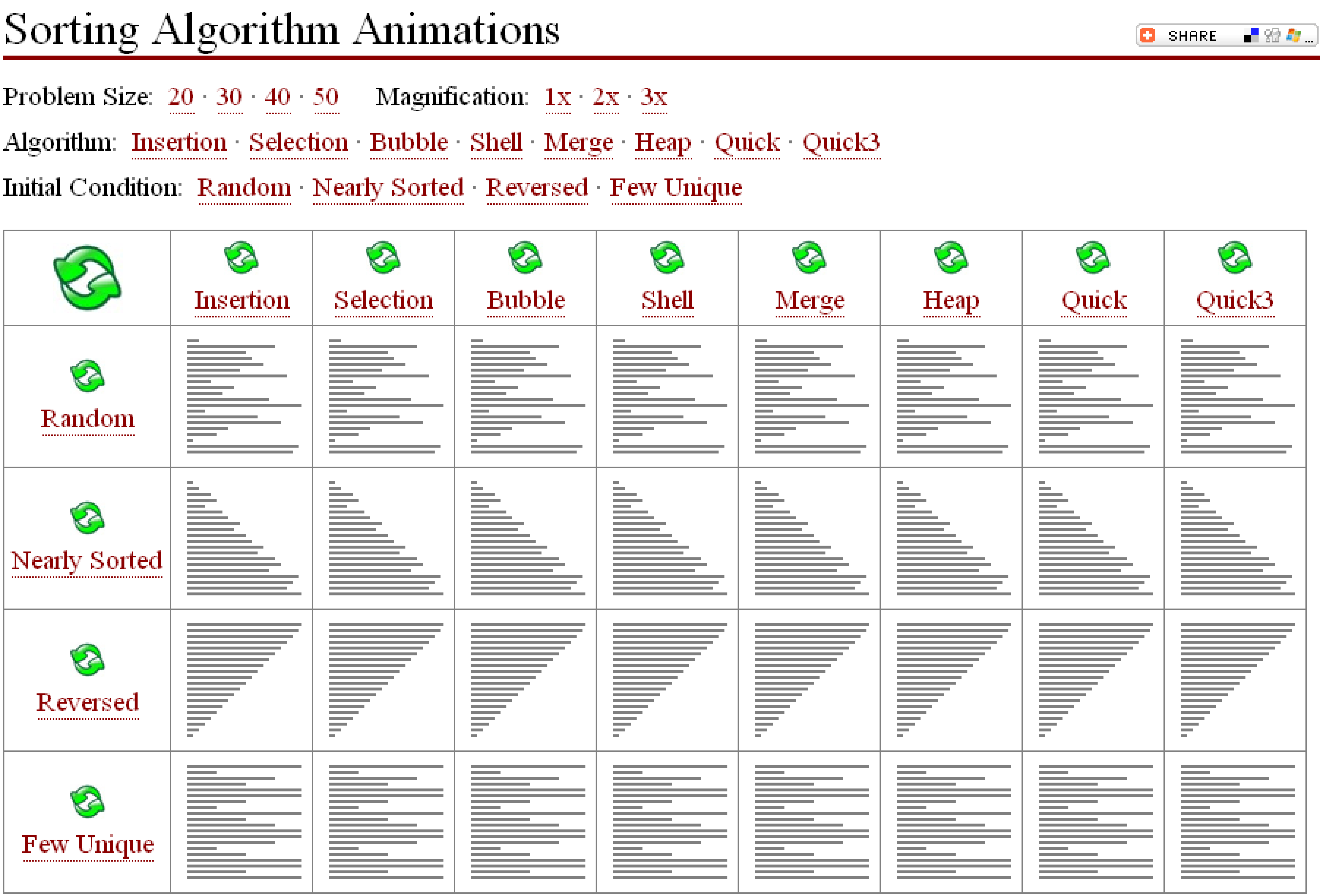
- Bubblesort
  - Scan through a list from bottom to top.
  - Compare successive adjacent pairs of items.
  - If two items are out of order, swap them.
  - After a complete scan, the first item is in place (bubbles to top). Skip that item on subsequent scans.
  - Repeat scan until no changes are made (completely ordered).
  - Works best when there are few items out of order.

Bubble-sort with Hungarian ("Csángó") folk dance
http://www.youtube.com/watch?v=lyZQPjUT5B4

Bubble Sort

```
// Bubblesort Example
ArrayList list = new ArrayList();

void setup() {
  size(500, 500);

  // Fill the ArrayList
  list.add("Purin");
  list.add("Landry");
  list.add("Chococat");
  list.add("Pekkle");
  list.add("Cinnamoroll");

  // Draw once
  noLoop();
  drawList(list);
}

void draw() { }

// On mousePressed, bubble once
void mousePressed() {
  bubbleOnce(list);
  //bubbleSort(list);
}

// Perform a complete Bubblesort
void bubbleSort(ArrayList al) {
  while ( true ) {
    if (bubbleOnce(al) == false) break;
  }
}
```

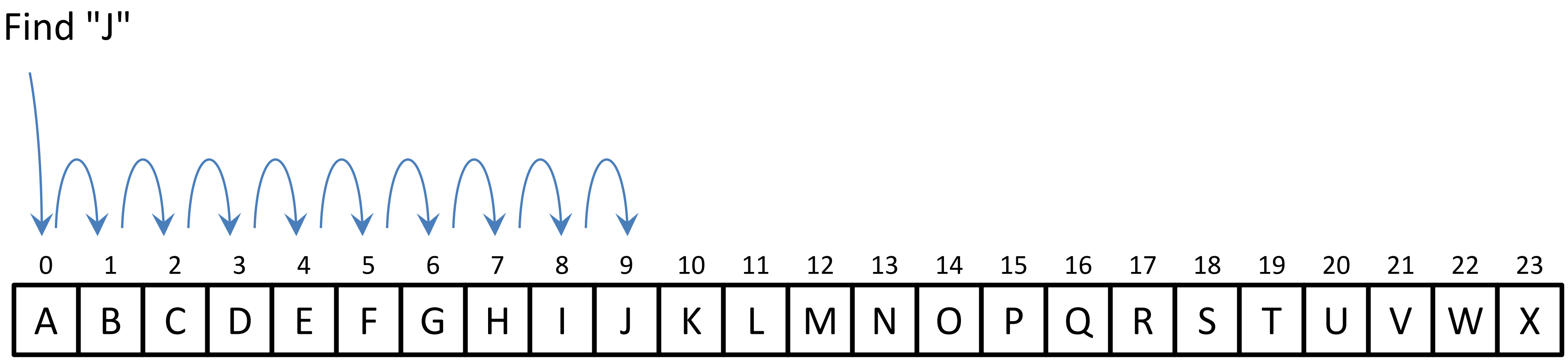
```
// Perform once pass of Bubblesort.
// Return true if any changes.
boolean bubbleOnce(ArrayList al) {
  boolean changed = false;

  // Loop over all pairs
  for (int i=0; i<al.size()-1; i++) {
    String s1 = (String)al.get(i);
    String s2 = (String)al.get(i+1);

    // Swap if pair is not in order
    if (s1.compareTo(s2) > 0) {
      list.set(i, s2);
      list.set(i+1, s1);
      changed = true;

      drawList(al); // Redraw list if changed
      delay(1000);
    }
  }
  return changed;
}

// Draw the ArrayList to the sketch
void drawList(ArrayList al) {
  background(0);
  fill(255);
  textSize(20);

  int y=100;
  for (int i=0; i<al.size(); i++) {
    String s = (String)al.get(i);
    text(s, 100, y);
    y=y+50;
  }
  redraw();
}
```



http://www.sorting-algorithms.com/

Exhaustive (Linear) Search

– Systematically enumerate all possible values and compare to value being sought.
– For an array, iterate from the beginning to the end, and test each item in the array.



Exhaustive (Linear) Search

```
// Search for a matching String val in the array vals.
// If found, return index. If not found, return -1.

int eSearch(String val, String[] vals) {

  // Loop over all items in the array

  for (int i=0; i<vals.length; i++) {

    // Compare items
    int rslt = val.compareTo( vals[i] );

    if ( rslt == 0 ) {              // Found it
      return i;                     // Return index
    }
  }

  return -1;     // If we get this far, val was not found.
}
```

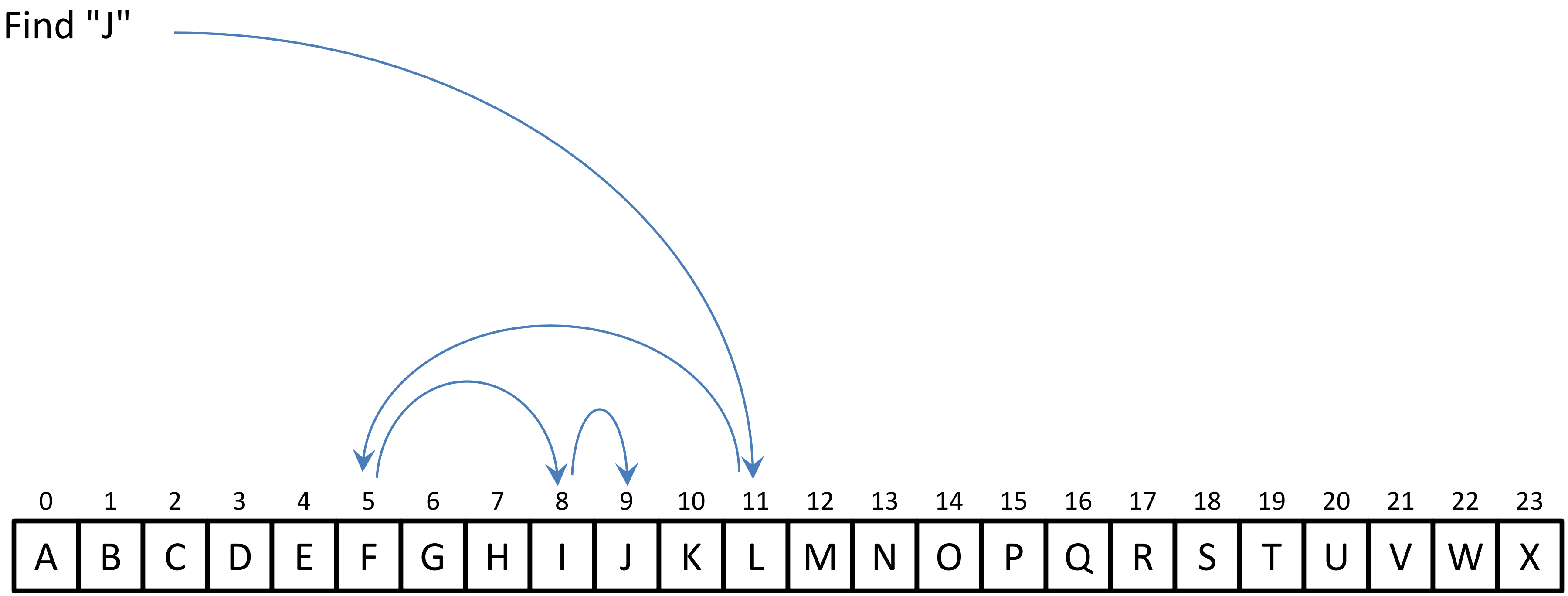
Binary Search

- Quickly find an item (val) in a sorted list.

- Procedure:
  1. Init **min** and **max** variables to lowest and highest index
  2. Repeat while **min** ≤ **max**
     a. Compare item at the **middle** index with that being sought (**val**)
     b. If **item** at **middle** equals **val**, return **middle**
     c. If **val** comes before **middle,** then reset **max** to **middle-1**
     d. If **val** comes after **middle**, reset **min** to **middle+1**
  3. If **min** > **max**, **val** not found

The most efficient way to play "guess the number" …

Binary Search

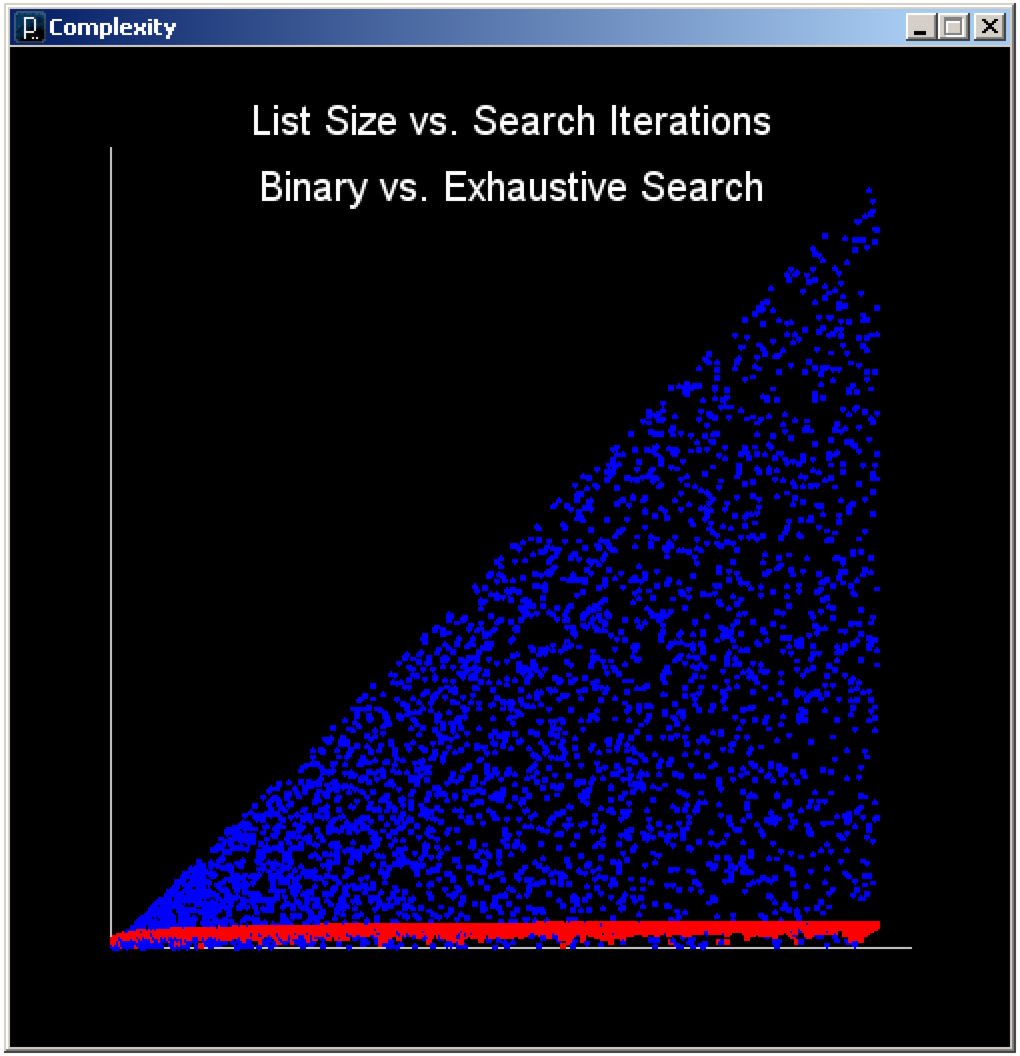


```
// Search for a matching val String in the String array vals
// If found, return index. If not found, return -1
// Use binary search.

int bSearch(String val, String[] vals) {
  int min = 0;
  int max = vals.length-1;
  int mid;
  int rslt;

  while (min <= max) {
    mid = int( (max + min)/2 );          // Compute next index

    rslt = val.compareTo( vals[mid] );  // Compare values

    if ( rslt == 0 ) {                  // Found it
      return mid;                       // Return index
    } else if ( rslt < 0 ) {            // val is before vals[mid]
      max = mid - 1;                    // Reset max to item before mid
    } else {                            // val is after vals[mid]
      min = mid + 1;                    // Reset min to item after mid
    }
  }

  // If we get this far, val was not found.
  return -1;
}
```

An Experiment - Exhaustive vs. Binary Search

- For names (Strings) in arrays of increasing size…
  - Select 10 names at random from the list
  - Search for each name using Binary and Exhaustive Search
  - Count the number of iterations it takes to find each name
  - Plot number of iterations for each against list size

- Start with an array of 3830+ names (Strings)



Wow! That's fast!

Worst Case Running Time

- **Exhaustive Search**

  N items in a list

  Worst case: Number of iterations = N

  (we must look at every item)

- **Binary Search**

  After 1st iteration, N/2 items remain ($N/2^1$)

  After 2nd iteration, N/4 items remain ($N/2^2$)

  After 3rd iteration, N/8 items remain ($N/2^3$)

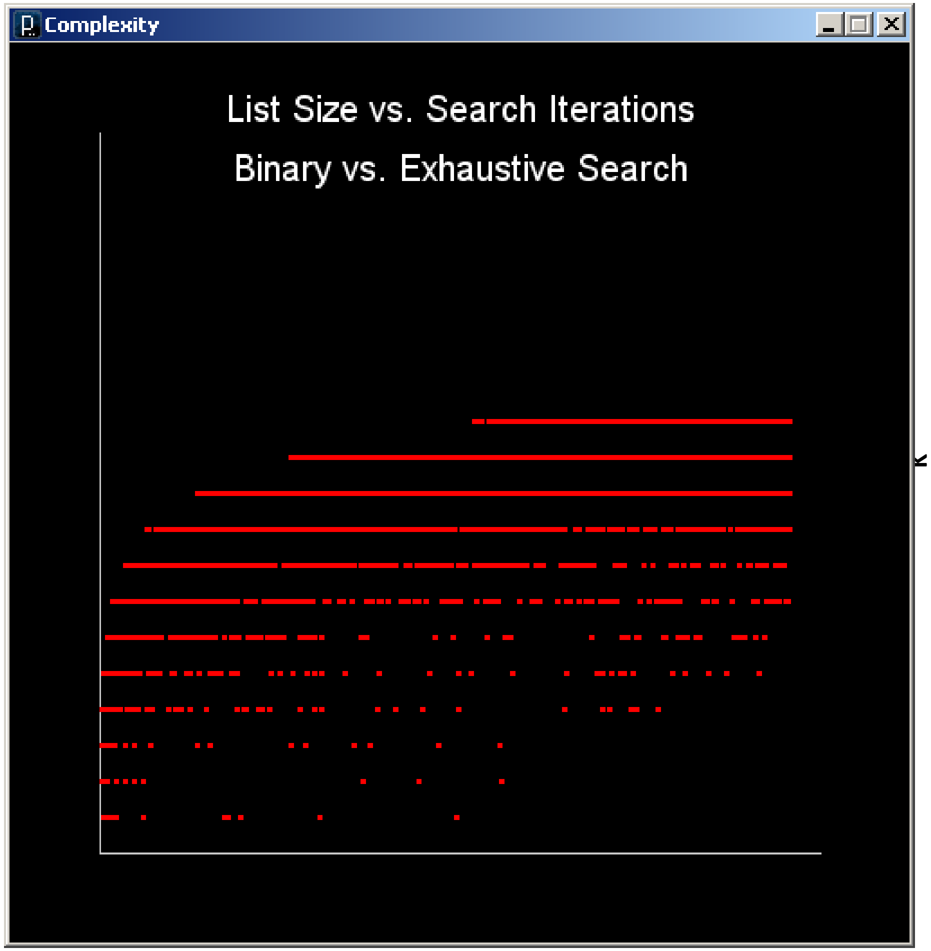
  …

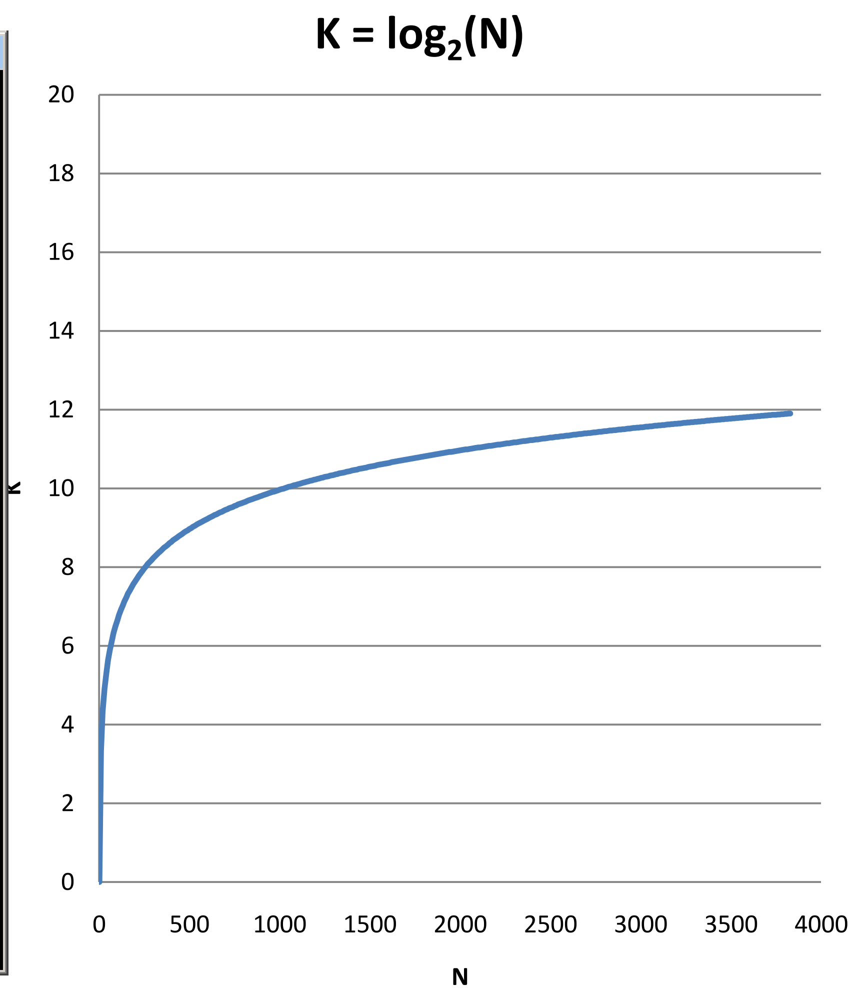
  Search stops when items to search ($N/2^K$) $\rightarrow$ 1

  i.e. $N = 2^K$, $\log_2(N) = K$

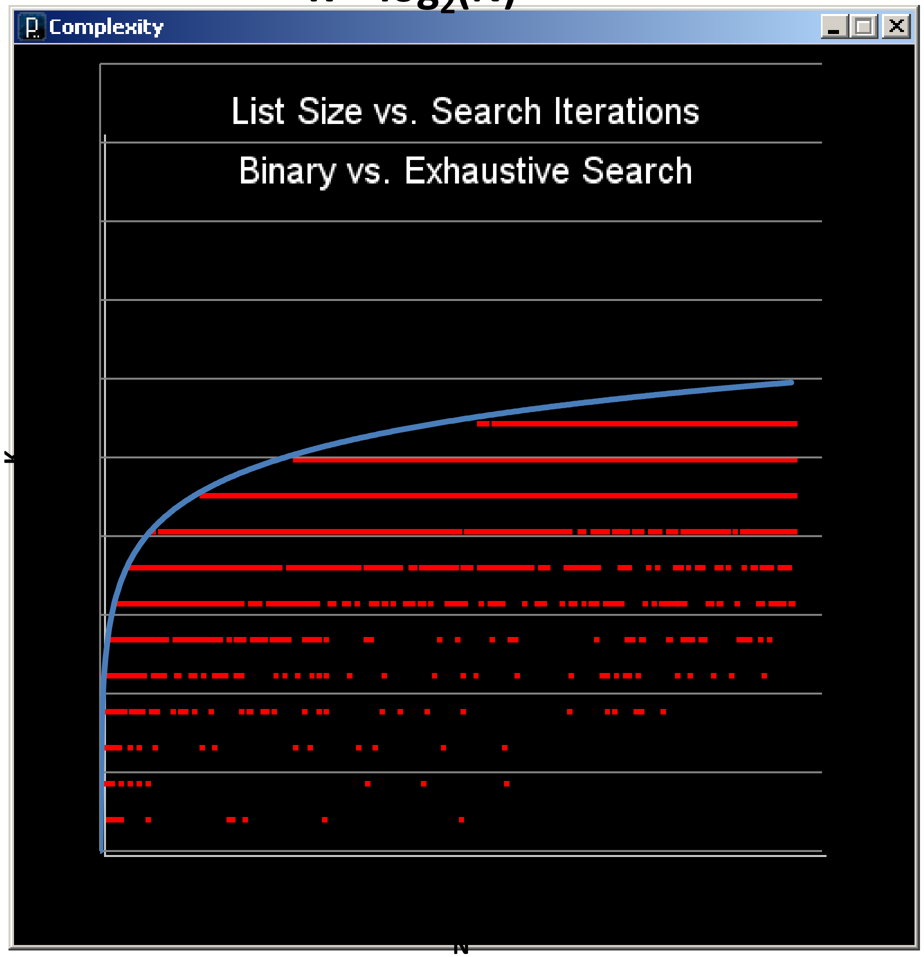
  Worst case: Number of iterations is $\log_2(N)$

It is said that Binary Search is a logarithmic algorithm and executes in O(logN) time.


Complexity
List Size vs. Search Iterations
Binary vs. Exhaustive Search


K = log2(N)
20
18
16
14
12
10
8
6
4
2
0
K
0
500
1000
1500
2000
2500
3000
3500
4000
N

K = $\log_2(N)$



Complexity
List Size vs. Search Iterations
Binary vs. Exhaustive Search
K
N