Computer Science

PUMPKIN PIE TEA

Friday, November 16th

4:30pm ~ 6:00pm

Intro Lab Park 231

Review

- Inheritance
  - A new behavior can be added easily to all child classes by adding once to a common base class
  - A common behavior of all child classes can be modified easily by making changes to a base class
  - Entirely new child classes can be created quickly by defining only how it differs wrt its base class
- Shape -> Rectangle, Ellipse, Triangle hierarchy
- Subtype Polymorphism and Duck Typing

```
# Rectangle Class - After
class Rectangle(Shape):
    def __init__(self, pts):
        Shape.__init__(self, pts)
```

Algorithm

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

- Researched, implemented, studied and documented, in order to solve many kinds of problems, using the most effective methods…
  - Sorting
  - Searching
  - …

Euclid's algorithm for greatest common divisor
(subtraction based version)

- 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 $\leftarrow$ A-B
     - Otherwise, B $\leftarrow$ B-A
  2. A is the GCD

Euclid's algorithm for greatest common divisor
(subtraction based version)

- 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

```
# gcd.py
A = 40902
B = 24140
print("GCD of " + str(A) + " and " + str(B) + " is:")

while B != 0:
    if A > B:
        A = A - B
    else:
        B = B - A

print(A)
```

gcd.py

Sorting

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

Selection Sort

```
# On mousePressed, perform one pass of selection sort
def mousePressed(o, e):
    global start
    selectOnce(items, start)
    if start < len(items)-1:
        start = start + 1

…

# Fill a list
items = []
items.append("Purin")
items.append("Landry")
items.append("Chococat")
items.append("Pekkle")
items.append("Cinnamoroll")

start = 0                          # Track start of unsorted
drawList()                         # Draw once to get started
onMousePressed += mousePressed # Perform one sort step
```

selection.py

Selection Sort

```python
# Perform once pass of Selection Sort.
def selectOnce(al, i):

    # Init to first element
    bestVal = al[i]
    bestIdx = i

    # Start looping at item after current top
    j = i + 1
    while j < len(al):
        # Find best value
        if al[j] < bestVal:
            bestVal = al[j]
            bestIdx = j
        j = j + 1

    # Swap best with top position
    al[bestIdx] = al[i]
    al[i] = bestVal

    # Redraw items
    drawList()
```

selection.py

Selection Sort

```
# Perform a complete Selection Sort
def selectionSort(al):
    i = 0
    while i < len(items):
        selectOnce(al, i)
        i += 1
```

selection.py

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

```
# Perform one pass of Bubblesort.
def bubbleOnce(al):
    changed = False

    # Loop over all pairs
    i = 0
    while i < len(al)-1:
        s1 = items[i]
        s2 = items[i+1]

        # Swap if pair is not in order
        if s1 > s2:
          items[i] = s2
          items[i+1] = s1
          changed = True

        i += 1

    # Redraw list if changed
    drawList()

    # Return True if list changed
    return changed
```

bubble.py

Bubble Sort

```python
# On mousePressed, bubble once
def mousePressed(o, e):
    bubbleOnce(items)

# Perform a complete Bubblesort
def bubbleSort(al):
    while True:
        if bubbleOnce(al) == False:
            break
```

bubble.py

Sorting Algorithm Animations

SHARE

Problem Size: 20 · 30 · 40 · 50 Magnification: 1x · 2x · 3x

Algorithm: Insertion · Selection · Bubble · Shell · Merge · Heap · Quick · Quick3

Initial Condition: Random · Nearly Sorted · Reversed · Few Unique

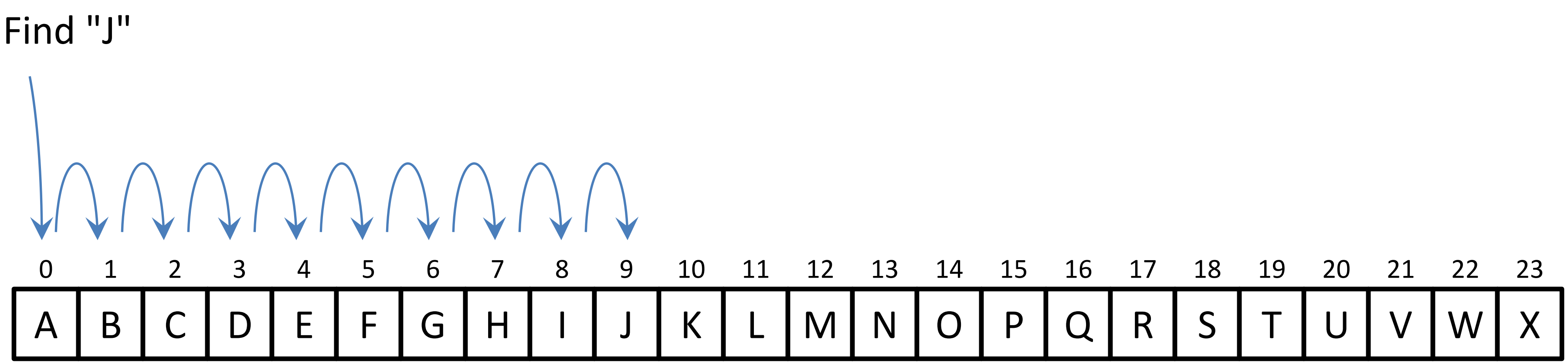
	Insertion	Selection	Bubble	Shell	Merge	Heap	Quick	Quick3
Random								
Nearly Sorted								
Reversed								
Few Unique								

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

Exhaustive (Linear) Search

– Systematically enumerate all possible values and compare to value being sought.

– For a list, iterate from the beginning to the end, and test each item in the list.



Exhaustive (Linear) Search

```
# Search for a matching String val in the array vals.
# If found, return index. If not found, return None.

def eSearch(val, items):

    # Loop over all items in the list
    i = 0
    while i < len(items):
        # Compare items
        if val == items[i]:
            return i
        i += 1

    # If we get this far, val was not found.
    return None
```

linear.py

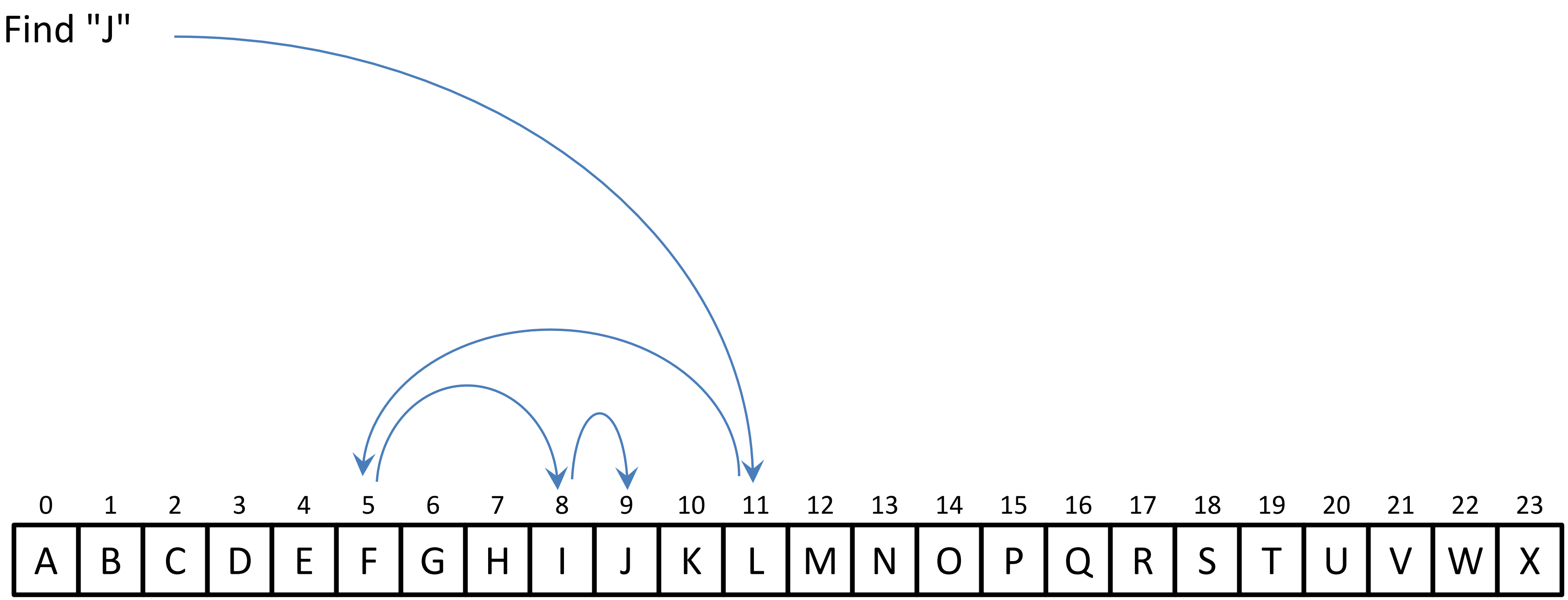
Binary Search

- Quickly find an item (val) in a <u>sorted</u> 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



Binary Search

```
# binary.py
# Binary search

# Search for a matching val in items
# If found, return index. If not found, return None
# Use binary search.
def bSearch(val, items):
    mid, min, count = 0, 0, 0
    max = len(items)-1

    while min <= max:
        count += 1                                  # Track iterations
        mid = int((max + min) / 2.0)                # Compute next index
        print("[" + str(min) + ", " + str(max) + "] --> " + str(mid))

        if val == items[mid]:                       # Found it
            print(str(val) + " found at index " + str(mid)
                       + " (" + str(count) + " iterations)")
            return mid                              # Return index
        elif val < items[mid]:                      # val is before items[mid]
            max = mid - 1                           # Reset max to item before mid
        else:                                       # val is after items[mid]
            min = mid + 1                           # Reset min to item after mid

    # If we get this far, val was not found.
    print(str(val) + " not found in " + str(count) + " iterations")
    return None
```

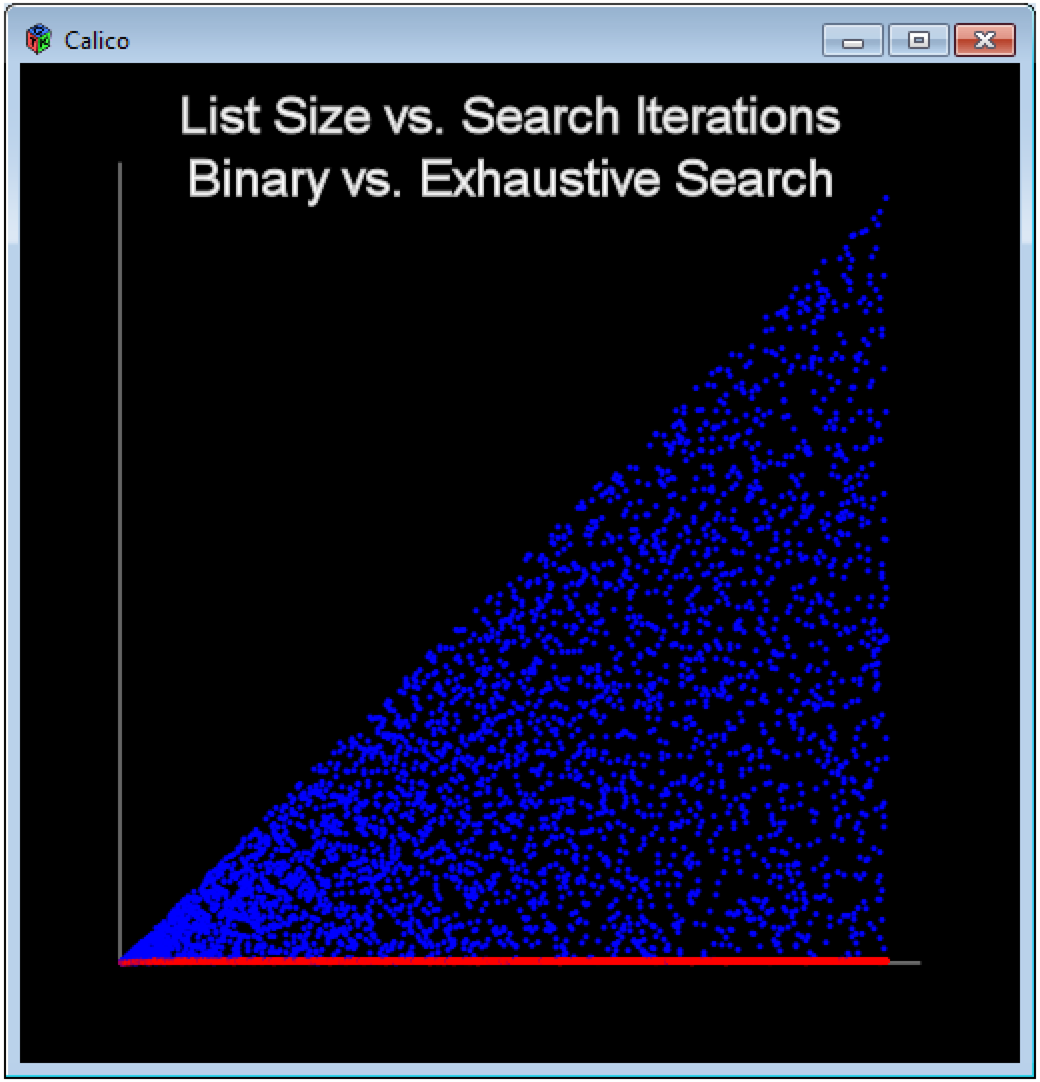
Binary Search

```
# Fill list with letters
letters = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M",
           "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y"]

# Search for a letter
bSearch("A", letters)

# [0, 23] --> 11
# [0, 10] --> 5
# [0, 4] --> 2
# [0, 1] --> 0
# A found at index 0 (4 iterations)


# Search for a letter
bSearch("Z", letters)

# [0, 24] --> 12
# [13, 24] --> 18
# [19, 24] --> 21
# [22, 24] --> 23
# [24, 24] --> 24
# Z not found in 5 iterations
```

An Experiment - Exhaustive vs. Binary Search

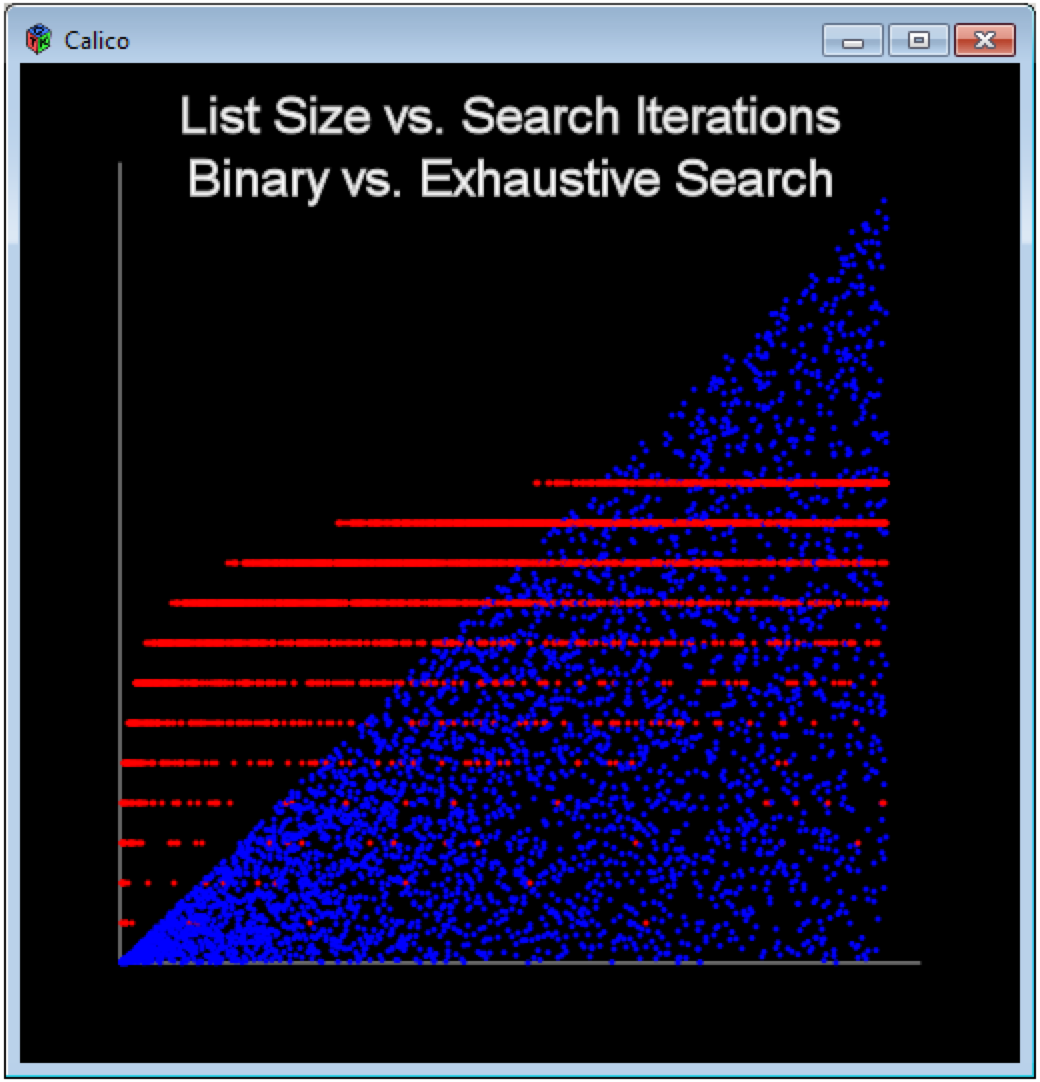
- For names in arrays of increasing size…
  - Select 10 names at random from the list
  - Search for each name using Binary and Exhaustive Search
  - Count the number of iterations it takes to find each name
  - Plot number of iterations for each against list size

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



Wow! That's fast!

worstCase.py



Binary magnified 200 times

worstCase.py

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 ($N/2^3$)

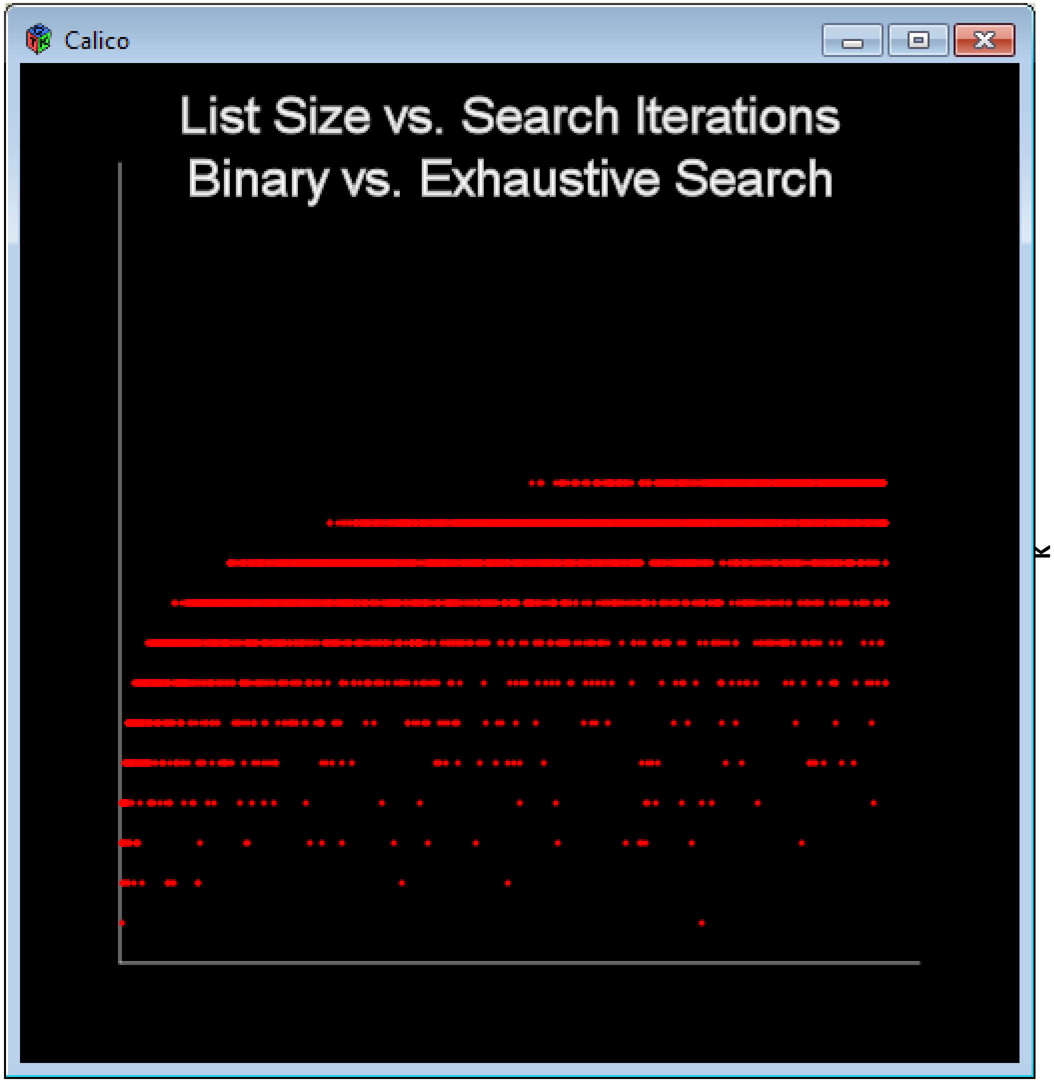
…

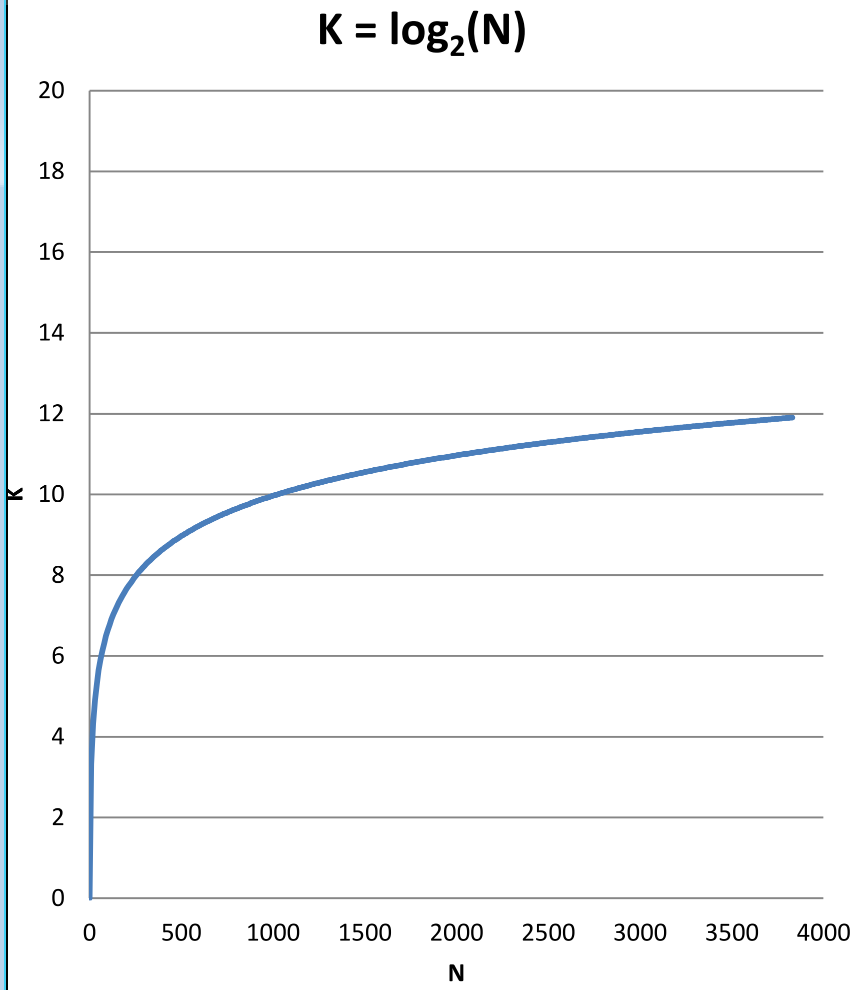
Search stops when items to search ($N/2^K$) $\rightarrow$ 1

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

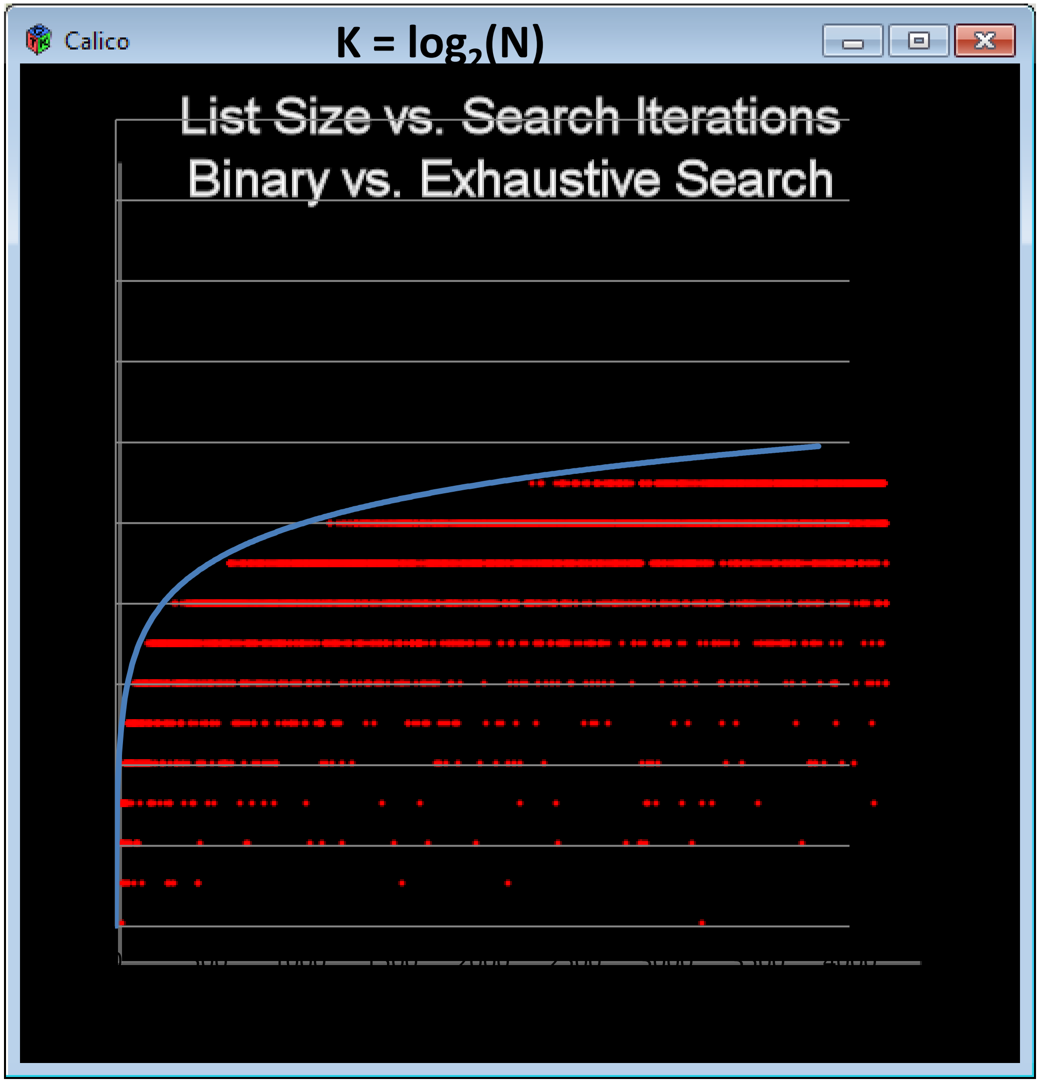
Worst case: Number of iterations is $\log_2(N)$

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





worstCase.py



Theory agrees with practice.

worstCase.py