Search

- Suppose have an array with 100 integers
  - Unordered & unique
  - How many will you have to look at to find a particular integer that is in the set
    - Best case
    - Worst case
    - Average case

Unordered Search

```
public class UnSearch {
    public static void main(String[] args) {
            int reps = Integer.parseInt(args[1]);
            int steps=0;
            int[] arr = new int[Integer.parseInt(args[0])];
            for (int ii=0; ii<arr.length; ii++)
                arr[ii] = (int)(Math.random()*10000);
            for (int kk=0; kk<reps; kk++) {
                        int tgt = arr[(int)(Math.random()*arr.length)];
                        for (int jj=0; jj<arr.length; jj++) {
                                    steps++;
                                    if (arr[jj]==tgt)
                                        break;
                            }
                }
        System.out.println("reps="+reps+"   steps="+steps+"    average="+(steps/reps));
    }
}
```

Improve on search speed?

- As long as the list is unordered this is as good as you can do
- But, suppose that the list is ordered
  - Set hi=length, lo=0
  - mid=hi+lo/2
  - If value at mid == target STOP
  - If value at mid > target set hi=mid+1
  - If value at mid < target set lo=mid-1
  - Return to mid= step
- Each rep removes ½ of the possibilities

Ordereed Search Program

```
public class OrSearch {
    public static void main(String[] args) {
            int reps = Integer.parseInt(args[1]);
            int steps=0;
            int[] arr = new int[Integer.parseInt(args[0])];
            arr[0]=(int)(Math.random()*10);
            for (int ii=1; ii<arr.length; ii++)
                arr[ii] = arr[ii-1]+1+(int)(Math.random()*20);
            for (int kk=0; kk<reps; kk++) {
                    int tgt = arr[(int)(Math.random()*arr.length)];
                    int hi=arr.length-1;      int lo=0;        int mid=hi/2;
                    steps++;
                    while (arr[mid] !=tgt) {
                            steps++;
                            if (arr[mid]>tgt) hi=mid-1;
                            if (arr[mid]<tgt) lo=mid+1;
                            mid=(lo+hi)/2;
                      }}
            System.out.println("reps="+reps+"   steps="+steps+"    average="+(steps/reps));
    }}
```

Comparisons needed for Search

Number of Elements	Comparisons
10	
100	
1,000	
10,000	
100,000	
1,000,000	
10,000,000	
100,000,000	
1,000,000,000	

Are the steps the same??

Efficiency

- How do we measure efficiency?
- Two main ways to determine efficiency:
  - *Empirical (measure time taken by running program)*
  - *Analytical (analyze the running time theoretically)*

Measurement

- Run a program on some input and time it

↑ *Extremely accurate.*
↑ *Easy to do.*

↓ *Not very good predicting value on*
  ↓ *Other inputs.*
  ↓ *Other computers.*
↓ *Can only be done after program is written.*

Analysis

- Examine a program to determine how long it will take
  - Choose a Model of Computation specifying what the basic instructions are and how much each one costs
  - Write a program (or pseudo-code) using these basic instructions
  - Count to figure out running time

Main Model for This Course

- Proposal:
  - ***Basic operations are Java instructions.***
  - ***Cost of each basic operation is 1 step!***
- Is this reasonable? – some Java instructions take much longer
- We only want ballpark numbers
- We want the analysis to hold on many different machines

Running time analysis: Example

```
public static int f(int x) {
  int y = x * x;
  y = y + x;
  return (y * x + 3 * y + 3 * x + 1);
}
```

- How many steps are the above?
- Look at the **`return`** instruction!

Examples with arrays and loops

```
public static void setToOnes (int[] a) {
  for (i = 0; i < a.length; i++) {a[i] = 1;}
}
```

```
public static int search (int[] a, int x) {
  int i = 0;
  while (i < a.length) {
    if (x == a[i]) {return (i);}
    i = i+1;
  return -1;
}
```

Basis of Time Measurement
Size of Input

- The number of elements in an array
- A program might take different times on different inputs of the same size
- Worst-case analysis focuses on inputs on which a program take the longest time

Analyzing Java constructs

```
int i;
for (i = 1; i < 10; i++) {
  System.out.print(i + “ “); }
```

Takes about 10 steps.

Nested for loops

```
int i,j;
for (i = 0; i < 10; i++) {
  for (j = 0; j < 10; j++) {
    System.out.print((i+j) + “ “); }
```

Takes about 10 * 10 = 100 steps.

Tougher Example

```
public static void tsro (int[] a) {
  int i, j;
  for (i=0; i < a.length; i++) {      // Outer loop
    for (j=i; j < a.length; j++) { // Inner loop
      if (a[i] > a[j]) {
        // some code
    }
  }
}
```

- The inner loop is executed
  `a.length – i` times every outer loop

Upper Bounds and Lower Bounds

- Upper bound – worst-case analysis – how long does it take, at most?
- Lower bound – best-case analysis – how long does it take, at least?

WORST → n^2

BEST → $n^2 / 4$

This analysis is a pain... let’s be sloppy

- Recall that we already decided to not count Java’s instructions precisely in our model of computation!
- Rules for precise sloppiness:
  - It is how long a program takes on LARGE inputs that matters
  - Constants do not matter. That is, a program that performs $5n$ instructions is just as good as one that performs n instructions.

Asymptotic notation

- We talk of running time as a fucntion of the input size:

  "a program takes (at most) f(n) ***time"***

- If we have two programs, one takes ***f(n)*** and the other takes ***g(n)***, which one is better?

Big-O Notation

$$f(n) \quad O(F(n))$$

- A way of denoting that fact that as ***n*** gets larger ***f(n)*** eventually becomes proportional to some function ***F(n)***
- The idea is that ***f(n)*** is at least as good as ***F (n)***
- ***F(n)*** is usually some standard function whose complexity is easy to see.

While Loops

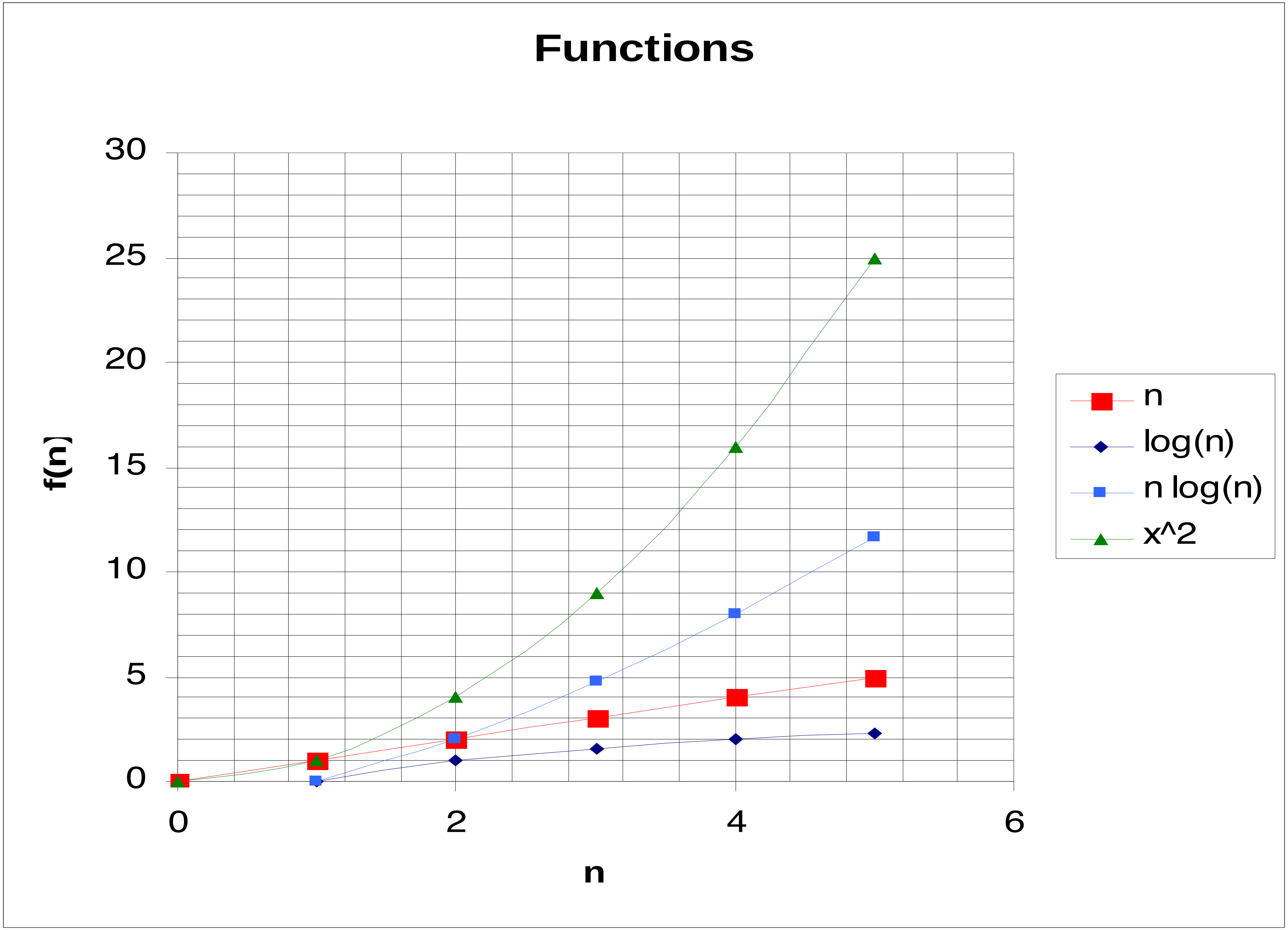
```
public static int gcd (int x, int y) {
  while (y != 0) {
    int temp = y;
    y = x%y;
    x = temp;
  }
  return (x);
}
```

How many times is the while loop executed?

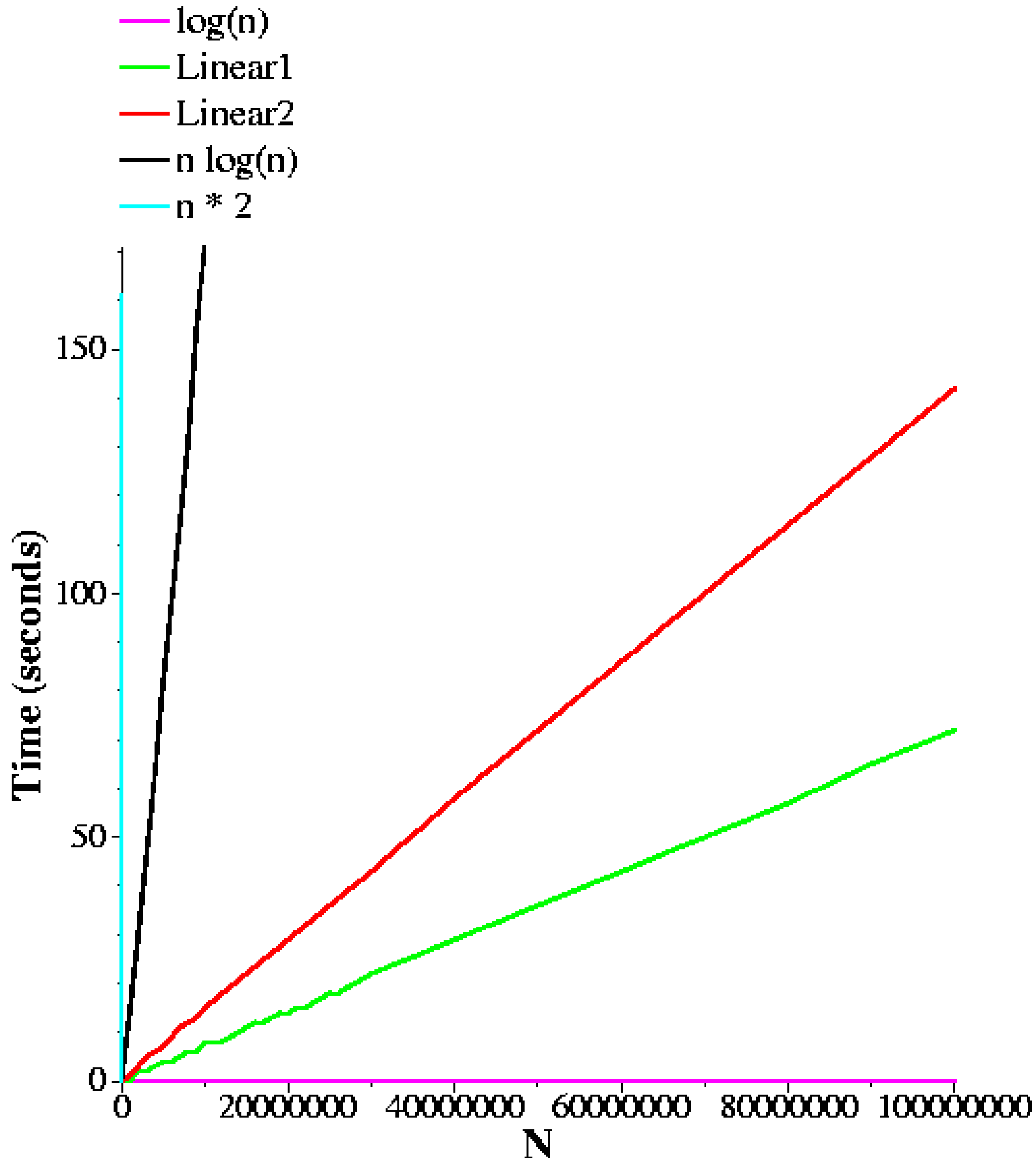
- The program executes in ***O(y)*** time

Examples

- Linear search: O(n)
- Binary search: **O(log(n))**
- Shuffling Cards ??



Functions
f(n)
30
25
20
15
10
5
0
0
2
4
6
n
n
log(n)
n log(n)
x^2



log(n)
Linear1
Linear2
n log(n)
n * 2
Time (seconds)
150
100
50
0
0
20000000
40000000
60000000
80000000
100000000
N