Runtime and Big-O Notation

Kelly Rivers and Stephanie Rosenthal

15-110 Fall 2019

Announcements

- Homework 3 full is due next week!

Learning Objectives

- To define run time and Big-O Notation
- To compare the run time of different algorithms using Big-O Notation

Binary vs Linear Search

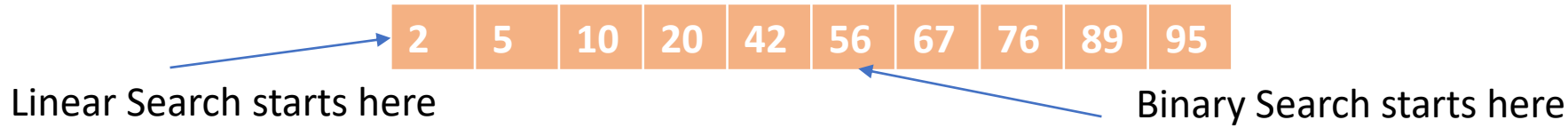
```
def binSearch(L,item):
    if len(L) == 0:
        return False
    mid = len(L)//2
    if L[mid] == item:
        return True
    elif L[mid] > item:
        return binSearch(L[:mid],item)
    else:
        return binSearch(L[mid+1:],item)
```

```
def linearSearch(L,item):
    if len(L) == 0:
        return False
    if L[0] == item:
        return True
    return linearSearch(L[1:],item)
```

Order of Comparisons

```
def binSearch(L,item):
    if len(L) == 0:
        return False
    mid = len(L)//2
    if L[mid] == item:
        return True
    elif L[mid] > item:
        return binSearch(L[:mid],item)
    else:
        return binSearch(L[mid+1:],item)
```

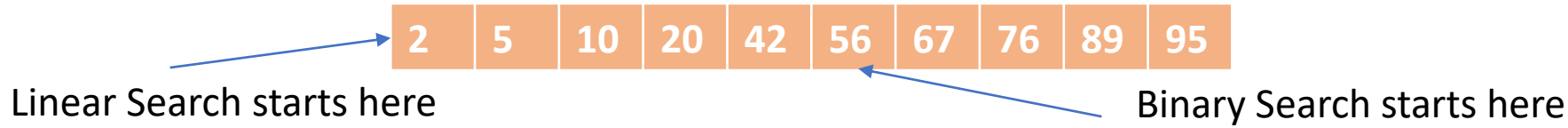
```
def linearSearch(L,item):
    if len(L) == 0:
        return False
    if L[0] == item:
        return True
    return linearSearch(L[1:],item)
```



What item should I search to return quickly?

```
def binSearch(L,item):
    if len(L) == 0:
        return False
    mid = len(L)//2
    if L[mid] == item:
        return True
    elif L[mid] > item:
        return binSearch(L[:mid],item)
    else:
        return binSearch(L[mid+1:],item)
```

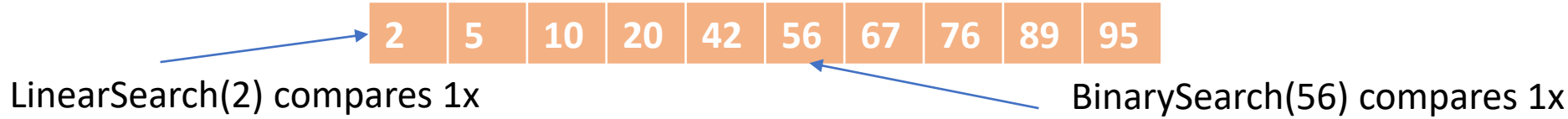
```
def linearSearch(L,item):
    if len(L) == 0:
        return False
    if L[0] == item:
        return True
    return linearSearch(L[1:],item)
```



What item should I search to return quickly?

```
def binSearch(L,item):
    if len(L) == 0:
        return False
    mid = len(L)//2
    if L[mid] == item:
        return True
    elif L[mid] > item:
        return binSearch(L[:mid],item)
    else:
        return binSearch(L[mid+1:],item)
```

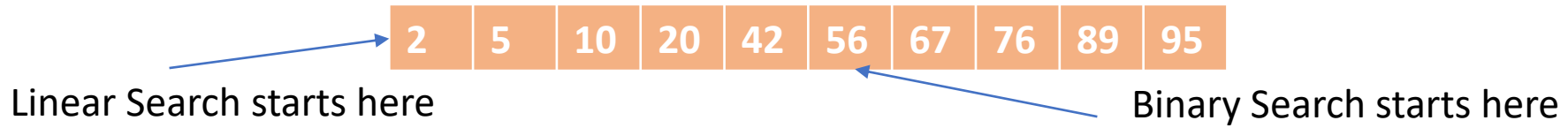
```
def linearSearch(L,item):
    if len(L) == 0:
        return False
    if L[0] == item:
        return True
    return linearSearch(L[1:],item)
```



How many comparisons when searching 95?

```
def binSearch(L,item):
    if len(L) == 0:
        return False
    mid = len(L)//2
    if L[mid] == item:
        return True
    elif L[mid] > item:
        return binSearch(L[:mid],item)
    else:
        return binSearch(L[mid+1:],item)
```

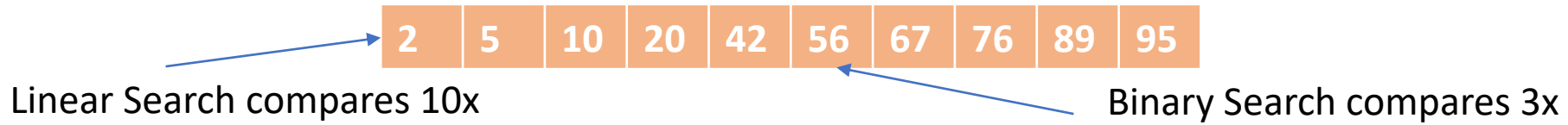
```
def linearSearch(L,item):
    if len(L) == 0:
        return False
    if L[0] == item:
        return True
    return linearSearch(L[1:],item)
```



How many comparisons when searching 95?

```
def binSearch(L,item):
    if len(L) == 0:
        return False
    mid = len(L)//2
    if L[mid] == item:
        return True
    elif L[mid] > item:
        return binSearch(L[:mid],item)
    else:
        return binSearch(L[mid+1:],item)
```

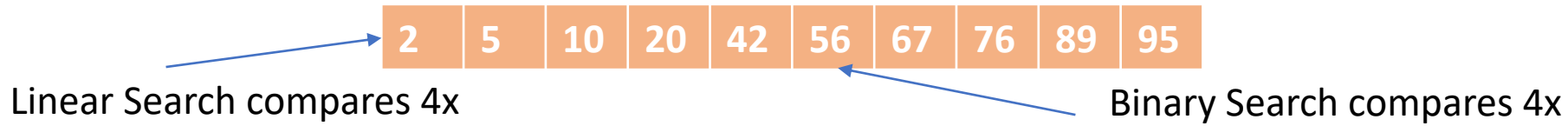
```
def linearSearch(L,item):
    if len(L) == 0:
        return False
    if L[0] == item:
        return True
    return linearSearch(L[1:],item)
```



How many comparisons when searching 15?

```
def binSearch(L,item):
    if len(L) == 0:
        return False
    mid = len(L)//2
    if L[mid] == item:
        return True
    elif L[mid] > item:
        return binSearch(L[:mid],item)
    else:
        return binSearch(L[mid+1:],item)
```

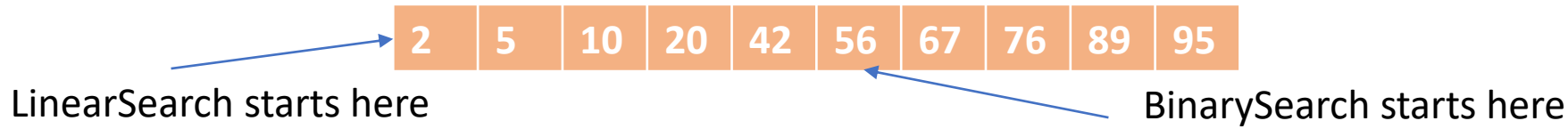
```
def linearSearch(L,item):
    if len(L) == 0:
        return False
    if L[0] == item:
        return True
    return linearSearch(L[1:],item)
```



What item should I search to take the longest?

```
def binSearch(L,item):
    if len(L) == 0:
        return False
    mid = len(L)//2
    if L[mid] == item:
        return True
    elif L[mid] > item:
        return binSearch(L[:mid],item)
    else:
        return binSearch(L[mid+1:],item)
```

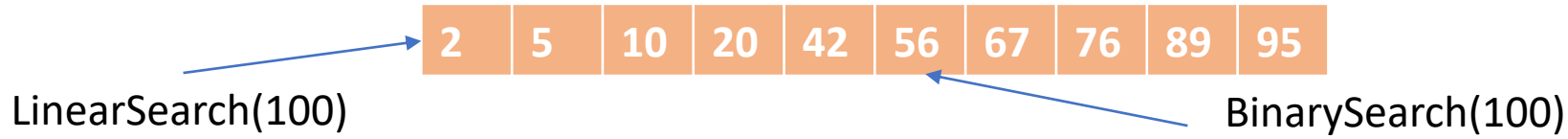
```
def linearSearch(L,item):
    if len(L) == 0:
        return False
    if L[0] == item:
        return True
    return linearSearch(L[1:],item)
```



What item should I search to take the longest?

```
def binSearch(L,item):
    if len(L) == 0:
        return False
    mid = len(L)//2
    if L[mid] == item:
        return True
    elif L[mid] > item:
        return binSearch(L[:mid],item)
    else:
        return binSearch(L[mid+1:],item)
```

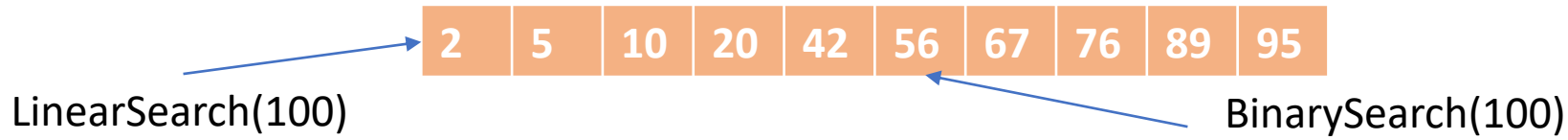
```
def linearSearch(L,item):
    if len(L) == 0:
        return False
    if L[0] == item:
        return True
    return linearSearch(L[1:],item)
```



How many comparisons to search 100?

```
def binSearch(L,item):
    if len(L) == 0:
        return False
    mid = len(L)//2
    if L[mid] == item:
        return True
    elif L[mid] > item:
        return binSearch(L[:mid],item)
    else:
        return binSearch(L[mid+1:],item)
```

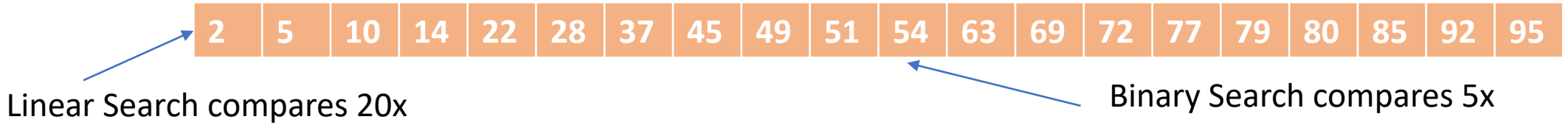
```
def linearSearch(L,item):
    if len(L) == 0:
        return False
    if L[0] == item:
        return True
    return linearSearch(L[1:],item)
```



How many comparisons to search 100?

```
def binSearch(L,item):
    if len(L) == 0:
        return False
    mid = len(L)//2
    if L[mid] == item:
        return True
    elif L[mid] > item:
        return binSearch(L[:mid],item)
    else:
        return binSearch(L[mid+1:],item)
```

```
def linearSearch(L,item):
    if len(L) == 0:
        return False
    if L[0] == item:
        return True
    return linearSearch(L[1:],item)
```



Number of Comparisons

Why did linearSearch comparisons double but binarySearch didn’t?

Hint: Think about the number of recursive calls

Number of Comparisons

Why did linearSearch comparisons double but binarySearch didn't?

Count the recursive calls

LinearSearch: compares and then reduces the list by 1

BinarySearch: compares and then reduces the list by half

To get to the end of a list of length n, how many calls do each make?

Number of Comparisons

Why did linearSearch comparisons double but binarySearch didn't?

Count the recursive calls

LinearSearch: compares and then reduces the list by 1

BinarySearch: compares and then reduces the list by half

To get to the end of a list of length n, how many calls do each make?

Linear: n

Binary: log(n)

Number of Comparisons

Why did linearSearch comparisons double but binarySearch didn't?

Count the recursive calls

LinearSearch: compares and then reduces the list by 1

BinarySearch: compares and then reduces the list by half

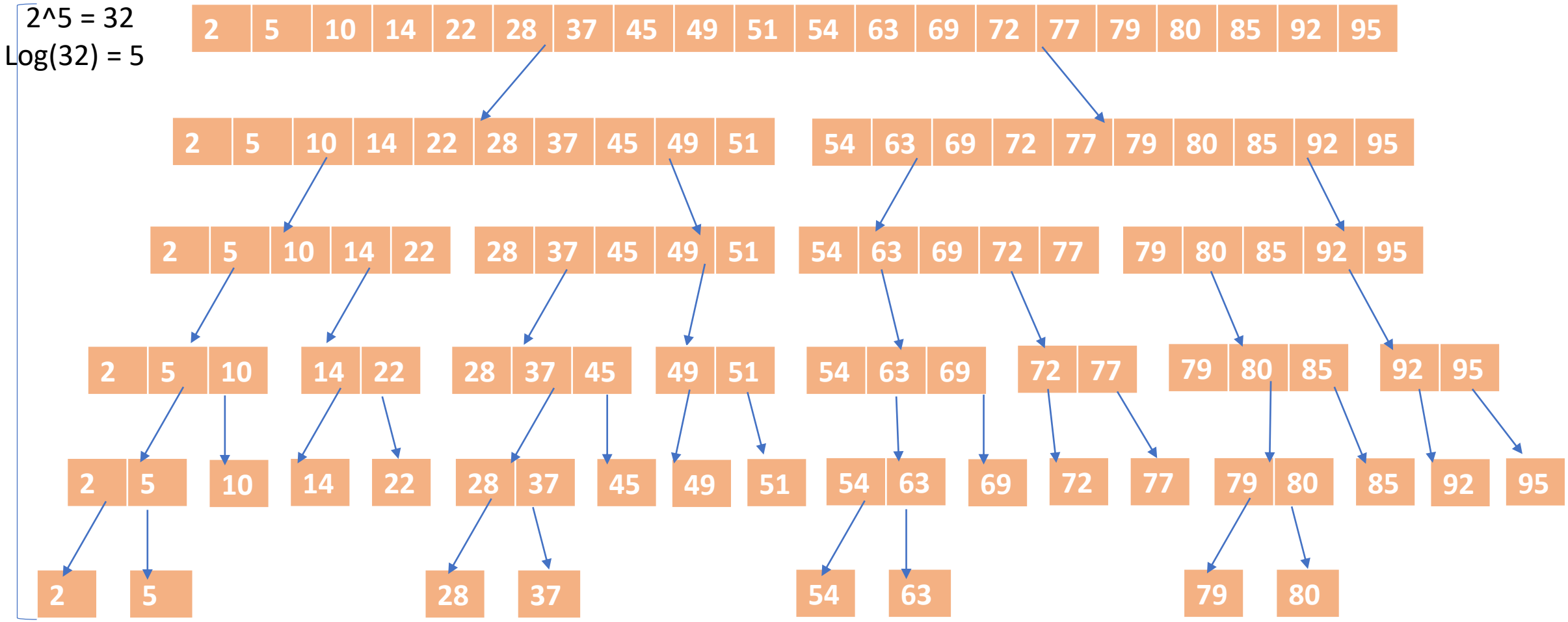
To get to the end of a list of length n, how many calls do each make?

Linear: n ← Doubling n doubles the number of calls

Binary: log(n) ← Doubling n increases number of calls by 1

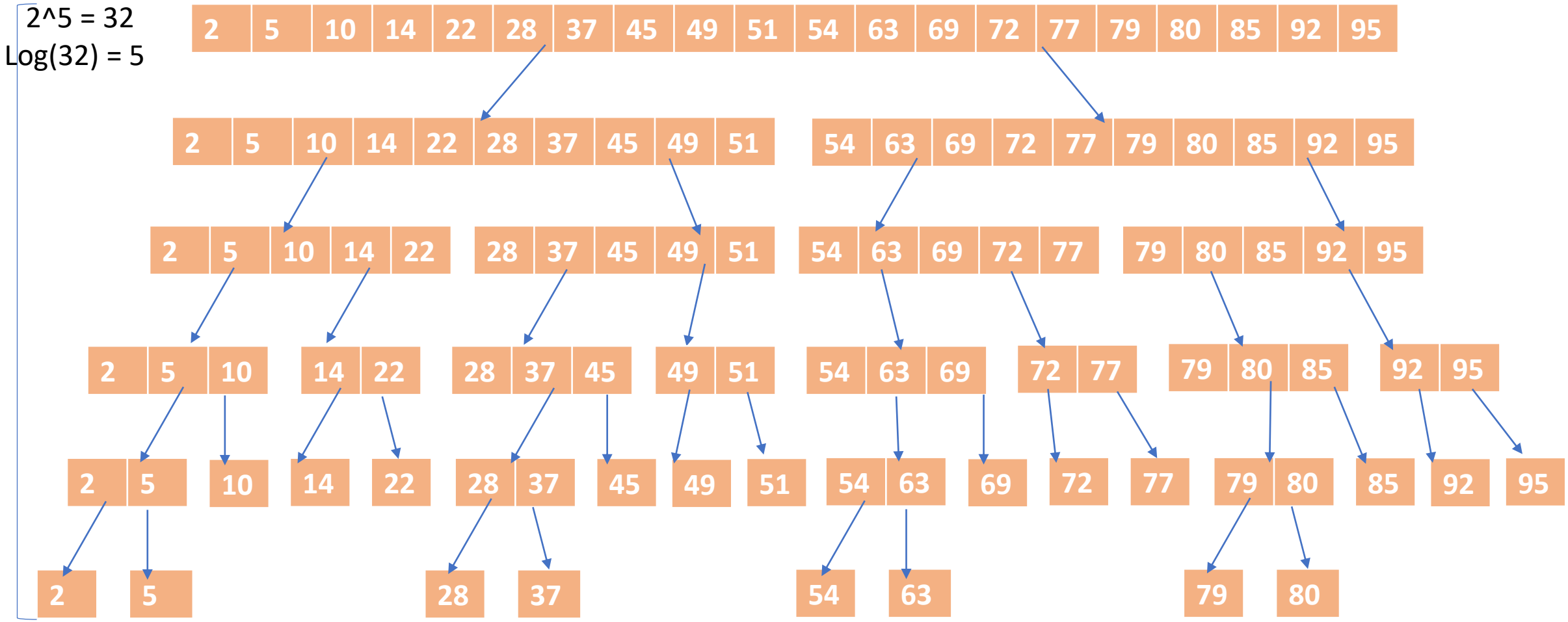
Log(N) relationships

Height = 5
2^5 = 32
Log(32) = 5



Log(N) relationships

Height = 5
2^5 = 32
Log(32) = 5



32 is the smallest power of 2 greater than 20

Approximation of Computation Steps (Runtime)

In LinearSearch, we do 2 comparisons + 1 recursive call = 3 steps

If our list is length n, we would do 3*n steps to get to the end

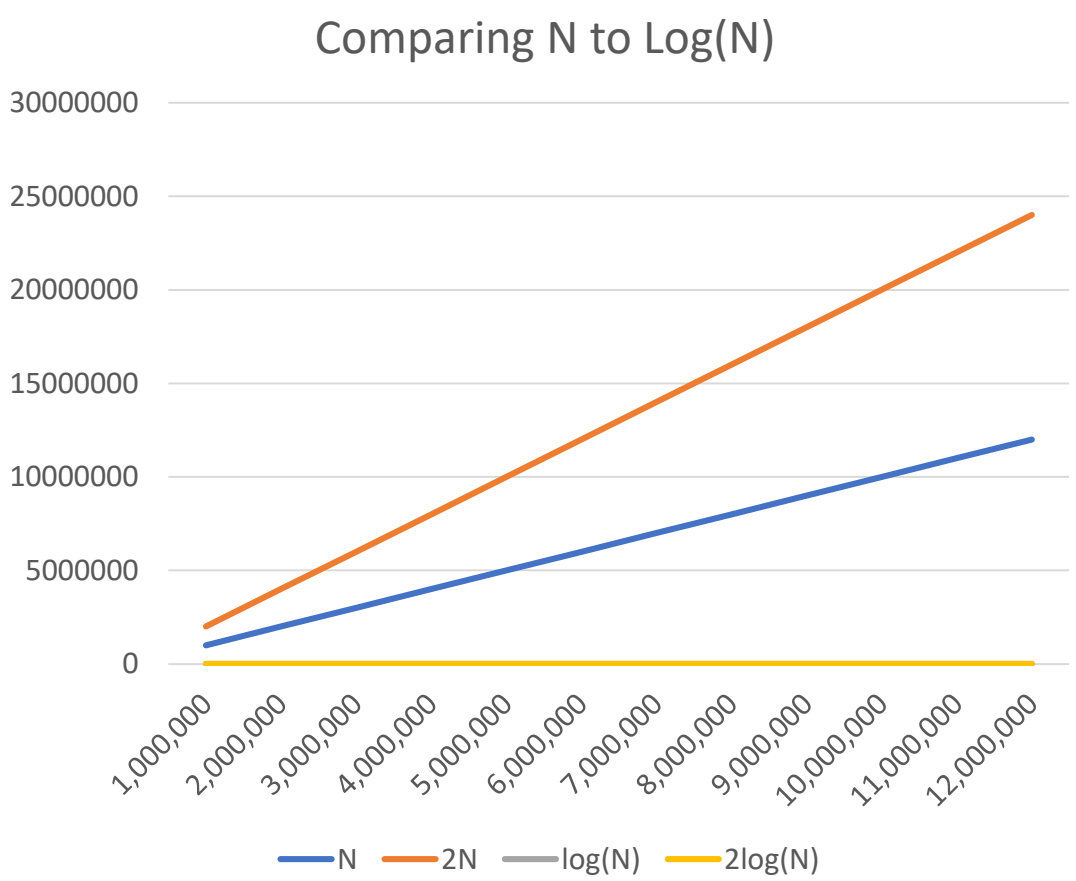
In BinarySearch, we do 4 comparisons + 1 function calls = 5 steps

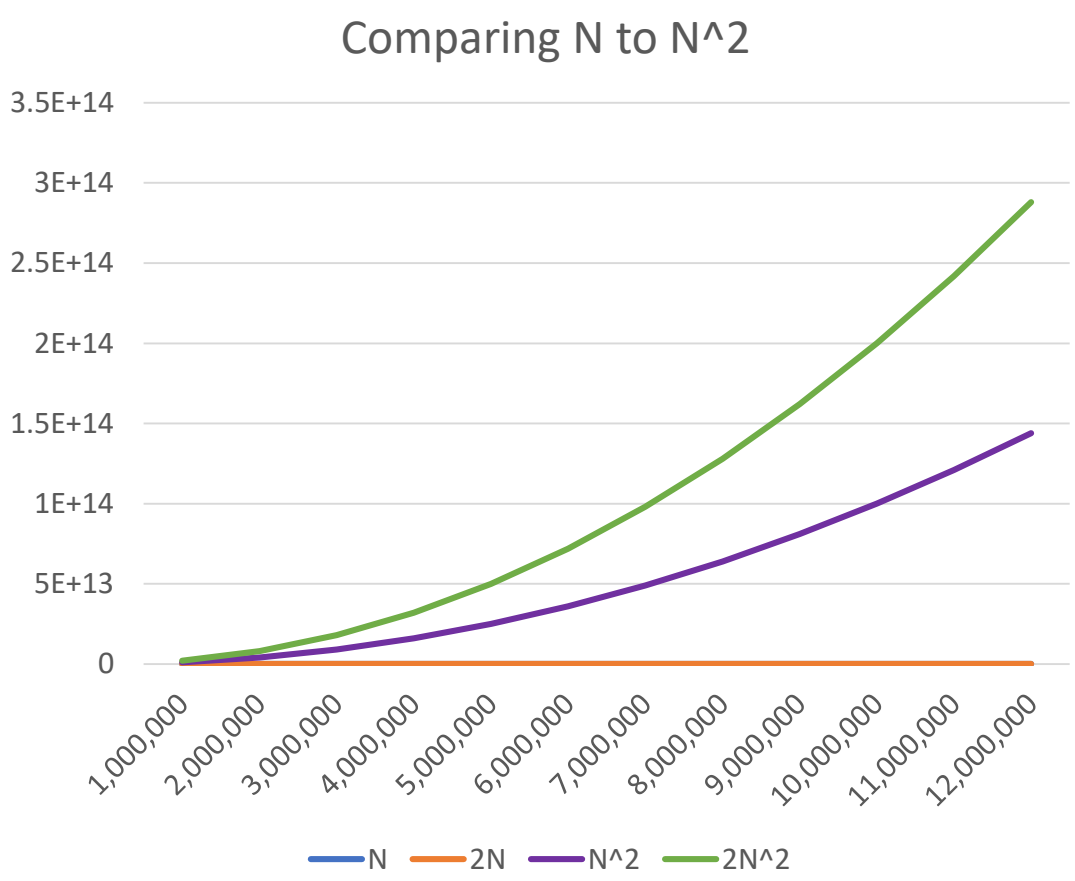
If our list is length n, we would do 5*log(n) steps to get to the end

When n gets very large, the constant is very small in comparison

In computer science, we approximate the number of steps by ignoring the constants.

Equations with Large N





Big-O Notation for Runtime Analysis

In computer science, we approximate the number of steps by ignoring the constants. The rest of the computation is only in terms of the input.

The worst case approximation of the number of steps is called Big-O Notation

$O(f(n)) = c*f(n)+b$ (b and c are constants not in terms of n)

In LinearSearch, $3*n$ steps = $O(n)$

In BinarySearch, $5*\log(n)$ = $O(\log(n))$

Runtime Analyses

Swapping two elements

```
def swap(a,b):
      temp = a
      a = b
      b = temp
```

How many steps? Does it change if a or b change?

Runtime Analyses

Swapping two elements = O(1)

```
def swap(a,b):
      temp = a
      a = b
      b = temp
```

How many steps? Does it change if a or b change?

3 steps, no matter what a and b are.

Runtime Analyses

Linear search with a for loop

```
def search(L,item):
      for elt in L:
            if elt == item:
                  return True
      return False
```

How many steps? Does it change depending on the size of L

Runtime Analyses

Linear search with a for loop: O(len(L))

```
def search(L,item):
      for elt in L:
            if elt == item:
                  return True
      return False
```

How many steps? Does it change depending on the size of L

1 comparison per time through the loop * len(L) times through the loop

Runtime Analyses

Linear search with a for loop: O(len(L))

```
def search(L,item):
      for elt in L:
            if elt == item:
                  return True
      return False
```

How many steps? Does it change depending on the size of L

1 comparison per time through the loop * len(L) times through the loop

Activity: Runtime Analysis

Searching a 2D list

```
def search(L,item):
    for row in L:
        for col in row:
            if col == item:
                return True
    return False
```

Assume the 2D list is square (n rows and n cols). What is the runtime?

Activity: Runtime Analysis

Searching a 2D list: O(n^2)

```
def search(L,item):
      for row in L:
            for col in row:
                  if col == item:
                        return True
      return False
```

Assume the 2D list is square (n rows and n cols). What is the runtime?

Activity: Runtime Analysis

Counting down by 5's

```
def countdown(n):
      for i in range(n,0,-5):
            print(i)
```

What is the runtime of countdown?

Activity: Runtime Analysis

Counting down by 5's: O(n)

```
def countdown(n):
      for i in range(n,0,-5):
            print(i)
```

What is the runtime of countdown?

Activity: Runtime Analysis

Counting down part 2:

```
def countdown(n):
      while n > 1:
            print(n)
            n = n//2
```

What is the runtime of countdown?

Activity: Runtime Analysis

Counting down part 2: O(logn)

```
def countdown(n):
      while n > 1:
            print(n)
            n = n//2
```

What is the runtime of countdown?

Activity: Runtime Analysis

Printing triangles runtime?

```
def triangle(n):
      for i in range(1,n):
            j = 1
            while j <= i:
                  print("*",end="")
                  j = j+1
            print()
```

Activity: Runtime Analysis

Printing triangles runtime? O(n^2)

```
def triangle(n):
      for i in range(1,n):
            j = 1
            while j <= i:
                  print("*",end="")
                  j = j+1
            print()
```

Takeaways

We can compute the runtime of an algorithm by counting the steps

Big-O notation is used to compute the worst case runtime of algorithms

- For large inputs, the constants don’t matter compared to input size
- We ignore the constants

We can use Big-O runtimes to compare different versions of the same algorithm