The Art of Data Structures

Runtime Analysis

Alan Beadle
CSC 162: The Art of Data Structures

Acknowledgment:
Slides and course materials originally prepared by Richard E Sarkis for the previous offering of this course

UNIVERSITY of ROCHESTER

Agenda

- Why algorithm analysis is important
- Use Big-O to describe execution time
- Examine the Big-O execution time of common operations on Python lists and dictionaries
- Understand how the implementation of Python data impacts algorithm analysis
- Benchmark simple Python programs

Algorithm Analysis

Algorithm Analysis

- Given an algorithm can you estimate how much time and memory it will take to process a given amount of data?

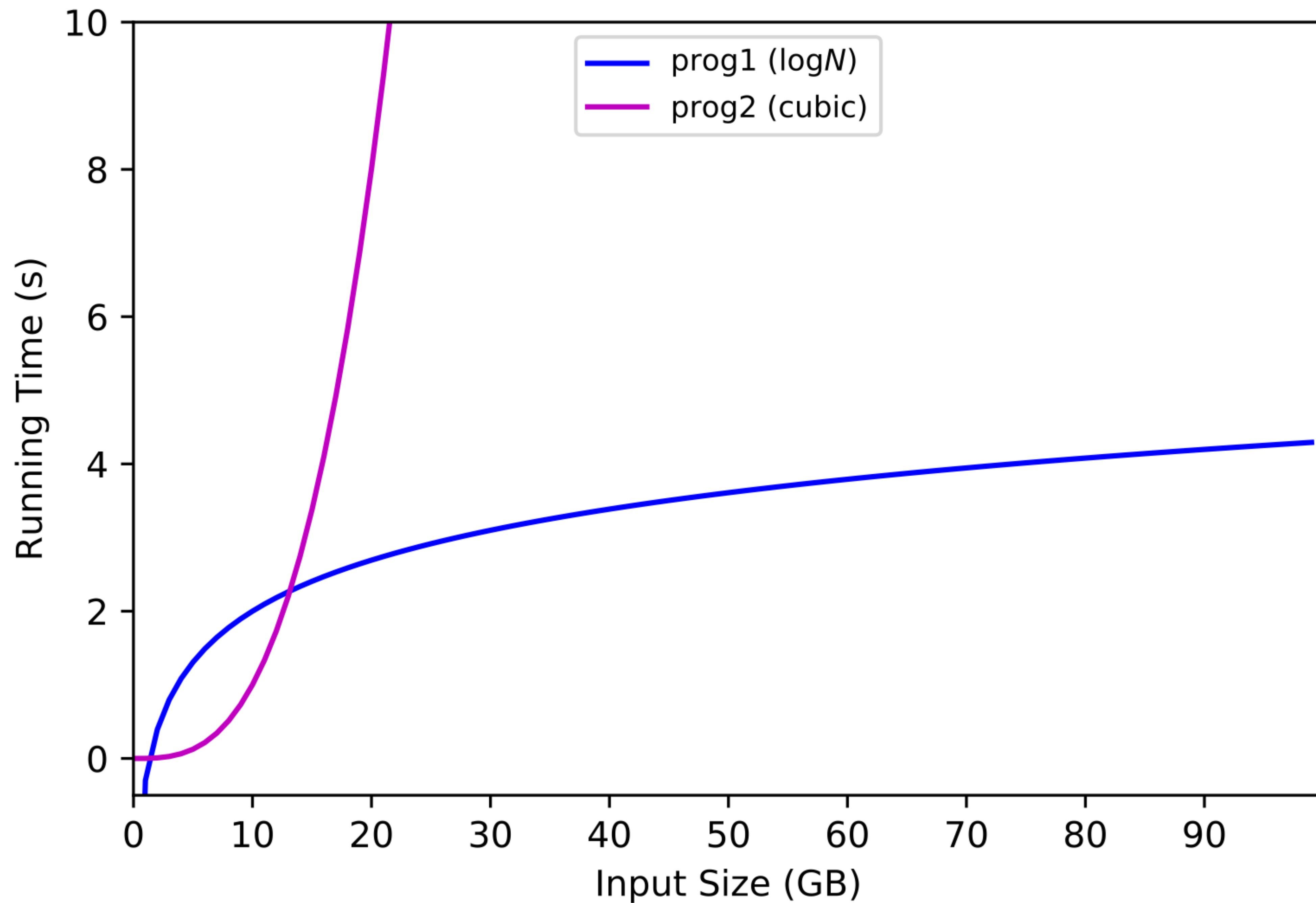
Algorithm Analysis

- I have two programs
- They both do the same thing
- I measure their run time

Algorithm Analysis

- prog1 takes 3 sec to process a 1GB array and 3.585 sec to process a 1.5GB array.
- prog2 takes 1 sec to process a 1GB array and 2.25 sec to process a 1.5GB array.
- Which should I use on my 100GB array?

Algorithm Analysis



Algorithm Analysis

- This example captures the notion of **asymptotic complexity**
  - how much time (and sometimes space) it takes to solve a problem of a given input size
- If you keep taking CS, you'll learn a lot more about this in 173, 280, or 282

Algorithm Analysis

- Sorting is a canonical example
- For lists of length *N*, we'll look at sorts that take time proportional to 2^N, N^2, Nlog(N), and N (in special cases)
  - Note that if N is big enough, 1000000Nlog(N) is still less than N^2
- We will cover sorting later in the course

Algorithm Analysis

- The following solves a familiar problem, computing the sum of the first *n* integers

```
def loop_sum(n):
    the_sum = 0
    for i in range(1, n+1):
        the_sum = the_sum + i

    return the_sum
```

Algorithm Analysis

```
import time

def loop_sum(n):
    start = time.time()

    the_sum = 0
    for i in range(1, n+1):
        the_sum = the_sum + i

    end = time.time()
    return the_sum, end-start
```

Algorithm Analysis

```
def formula_sum(n):
    start = time.time()

    the_sum = (n*(n+1))//2

    end = time.time()
    return the_sum, end-start
```

Algorithm Analysis

- Let’s evaluate the execution (running) time of these algorithms as a good benchmarking of whether an algorithm is “good”

Algorithm Analysis

- Notice how the time it takes for loop_sum() to run when we scale the value of *n* to larger values
- Compare that with the results for formula_sum() using the same scaled values of *n*
    - It seems to be unaffected the the size of our input

Algorithm Analysis

- While this is illustrative, it isn't concretely useful for analysis
- Many factors could affect the performance these same algorithms
  - A different (faster, slower) computer
  - A different language is used

Algorithm Analysis

- A characterization is needed that can describe algorithm performance regardless of these kinds of variability

Big-O

Big-O

- Try to quantify an algorithm on the number of operations, or steps taken
- A basic unit of computation needs to be considered

Big-O

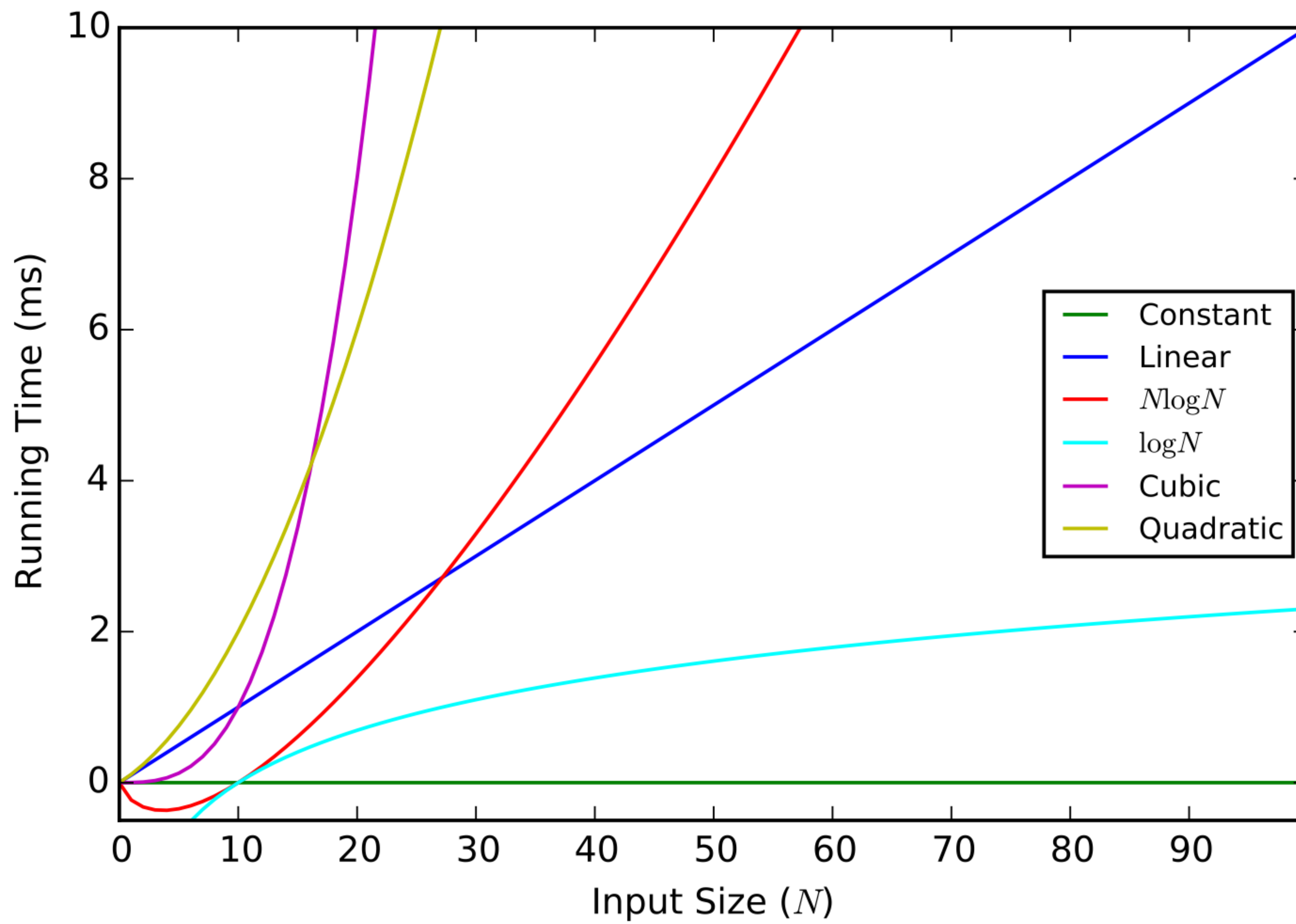
- For example, we can consider assignment statements as a basic unit of computation
- In our function sum_of_n2, the number of assignments is 1 plus the value of *n*
- Call this $T(n) = 1 + n$
- *“T(n) is the time it takes to solve a problem of size n, name 1+n steps”*

Big-O

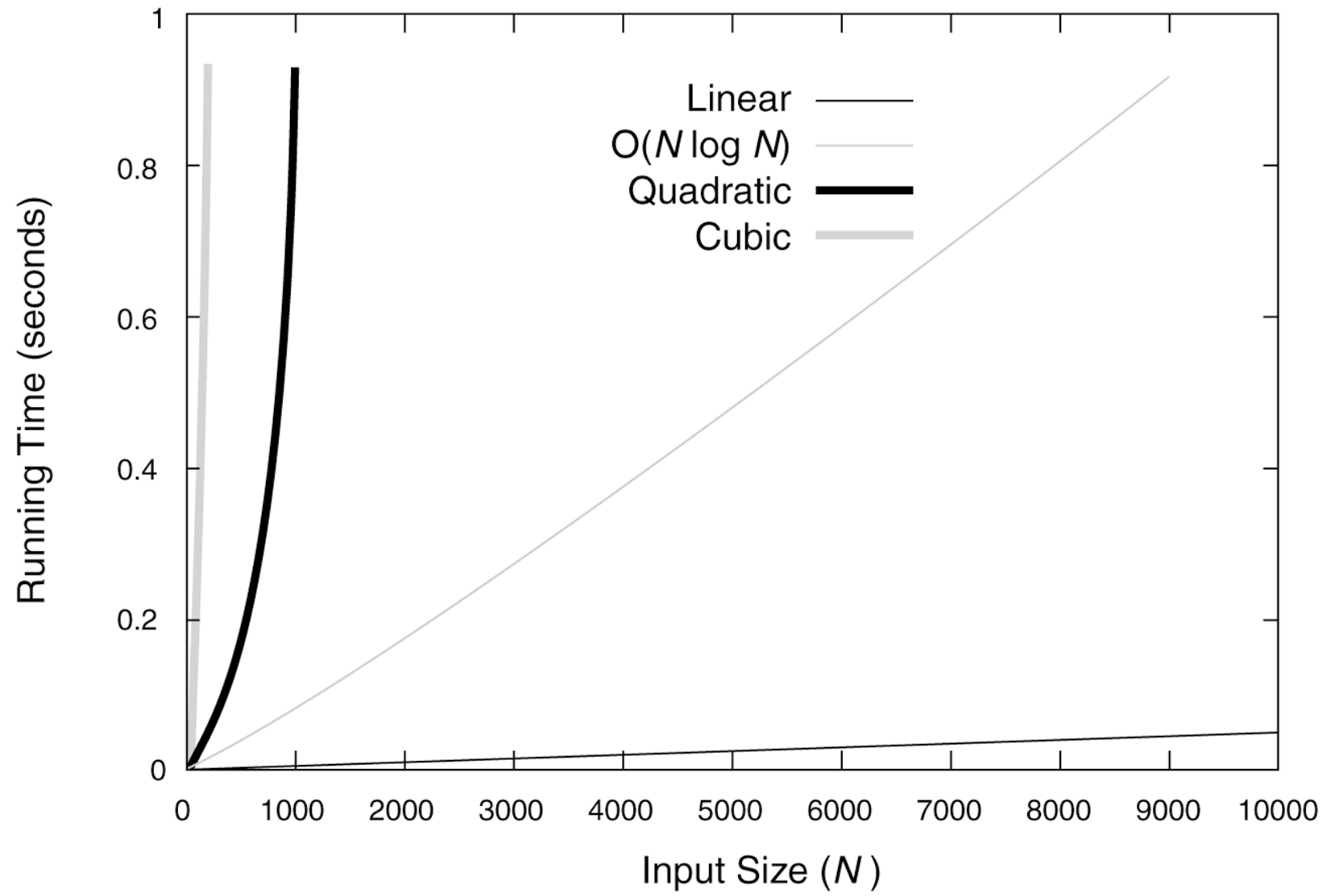
- The characterization of the algorithm using a $T(n)$ relation shows something interesting
- If the problem size increases, certain terms dominate

Big-O

- The **order of magnitude** function describes the part of *T(n)* that increases the fastest as *n* increases
- **Big-O** notation, written as *O(f(n))*
- This is a useful approximation
- *f(n)* is a simple representation of the dominate part of *T(n)*



Running Time (ms)
Input Size (N)
Constant
Linear
$N\log N$
$\log N$
Cubic
Quadratic
0
2
4
6
8
10
0
10
20
30
40
50
60
70
80
90



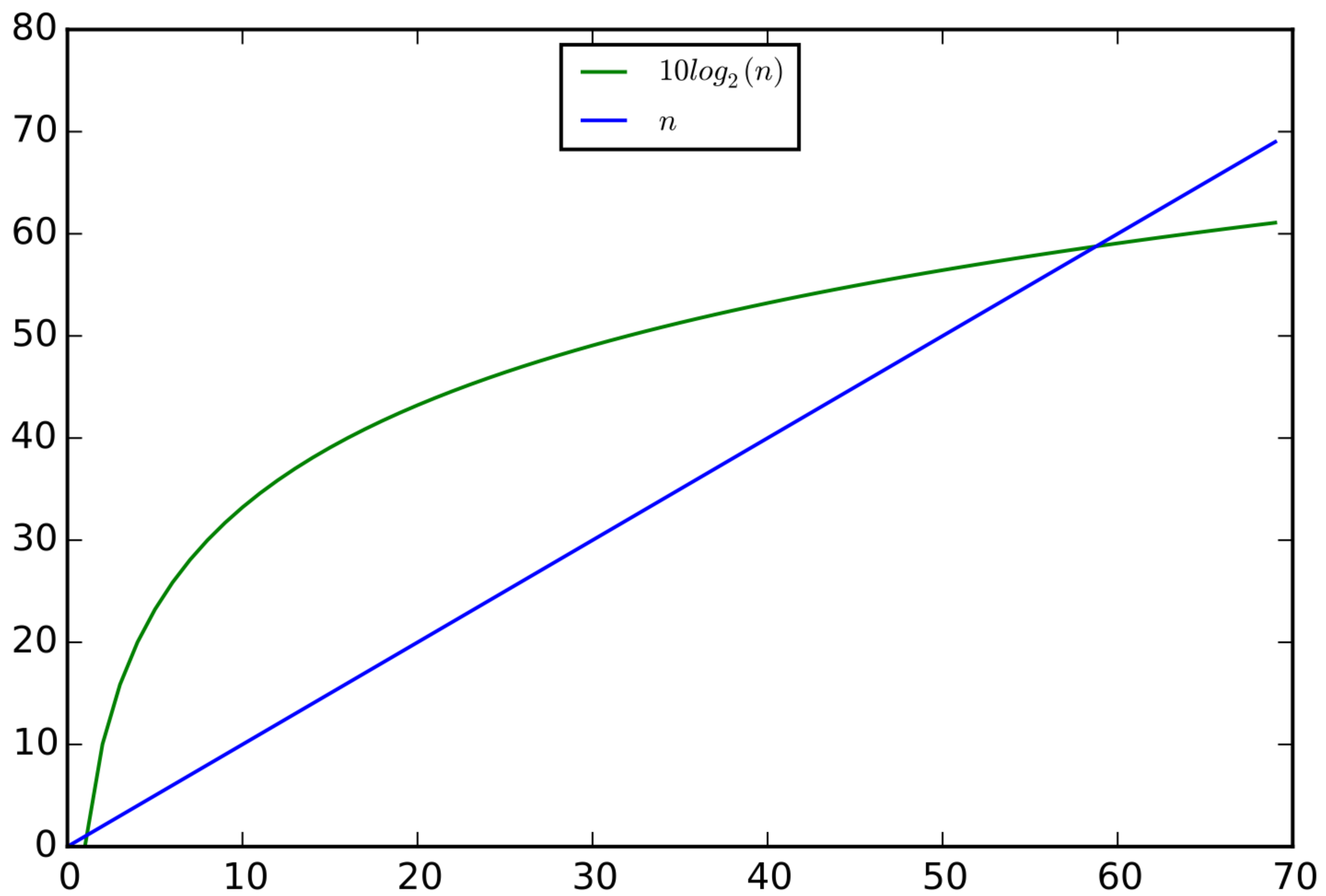
Running Time (seconds)
1
0.8
0.6
0.4
0.2
0
Linear
O(N log N)
Quadratic
Cubic
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Input Size (N)

Mathematical Expression	Relative Rates of Growth
$T(N) = O(F(N))$	Growth of $T(N)$ is $\leq$ growth of $F(N)$.
$T(N) = \Omega(F(N))$	Growth of $T(N)$ is $\geq$ growth of $F(N)$.
$T(N) = \Theta(F(N))$	Growth of $T(N)$ is $=$ growth of $F(N)$.
$T(N) = o(F(N))$	Growth of $T(N)$ is $<$ growth of $F(N)$.

- **Some examples:**

- *$10n^2 + 50n + 100$ is $O(n^2)$*
- *$\log(n)+7$ is $O(\log(n))$*

- *We only care about the term that increases the fastest, since eventually it will take most of our resources!*


$10log_2(n)$
n
80
70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
70

- Any Polynomial is big-O of its leading term with coefficient of 1
- The base of a logarithm doesn’t matter.
- $\log_a(n)$ is $O(\log_b n)$ for any bases a and b because
- $\log_a(n) = (\log_b n)(\log_a b)$

- Logs grow slower than powers:
  $\log(n)$ is $O(n^{1/10})$
- Exponentials (c^n, $c>1$) grow faster than poly n^{10} is $O(1.0001^n)$
- Generally, polynomial time is tolerable
- Generally, exponential time is intolerable

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	$N \log N$
N^2	Quadratic
N^3	Cubic
2^N	Exponential

The Tale of Two Algorithms

- Slow Algorithm, Fast Computer
- Fast Algorithm, Slow Computer

DEC Alpha workstation 500/400
(1995)
400 MHz Alpha 21164A

Radio Shack TRS-80
(1977)
1.78 MHz Zilog Z80

Two Algorithms

n	Alpha21164A, “C”, O(n^3)	TRS-80, “BASIC”, O(n)
10	0.6 microsec	200 millisecs
100	0.6 millisecs	2.0 sec
1000	0.6 sec	20 sec
10,000	10 min	3.2 min
100,000	7 days	32 min
1,000,000	19 years	5.4 hrs

Programming Exercise

Programming Exercise

- Write a Python function to convert ints to strings containing the binary representation
- if (n == 5) return “101”;
- if (n==12) return “1100”;
- def int2bin(n)

Programming Exercise

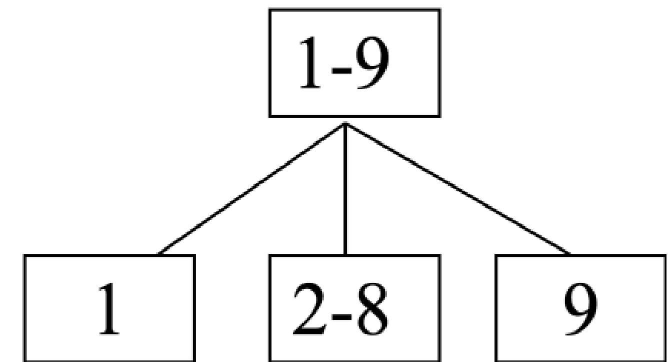
```
def int2bin(n):
    rval = ""
    while (n>0) :
        if((n%2) == 1) :
            rval += “1”
        else :
            rval +=“0”
        n = n // 2

    return rval
```

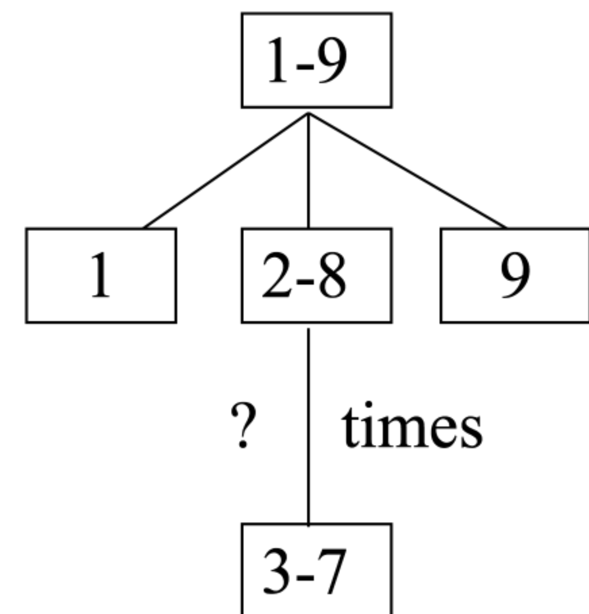
```
1 def int2bin(n):
2     rval = ""
3     while (n>0) :
4         if((n%2) == 1) :
5             rval += “1”
6         else :
7             rval +=“0”
8         n = n // 2
9
10    return rval
```

1-9

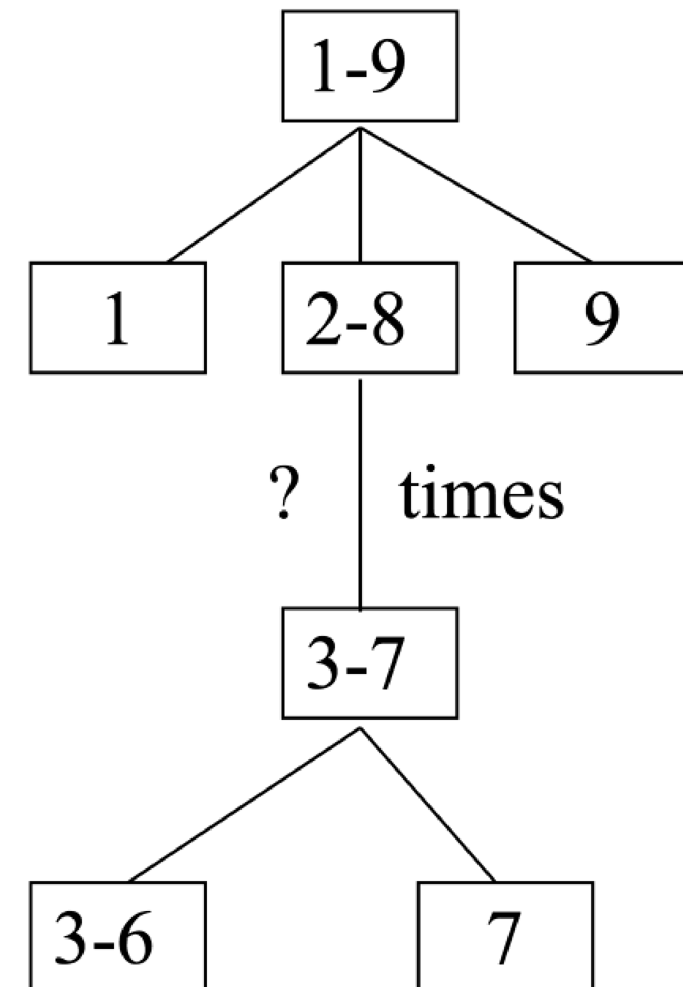
```
1 def int2bin(n):
2     rval = ""
3     while (n>0) :
4         if((n%2) == 1) :
5             rval += “1”
6         else :
7             rval +=“0”
8         n = n // 2
9
10    return rval
```



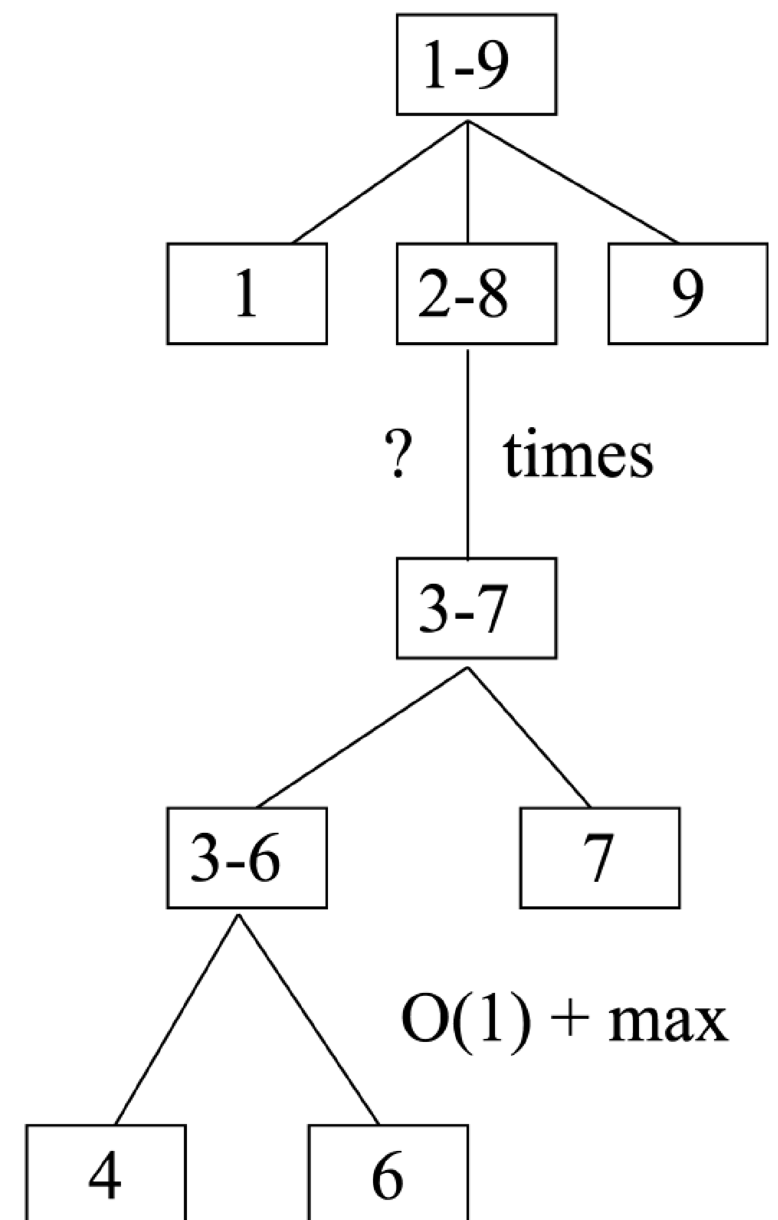
```
1  def int2bin(n):
2      rval = ""
3      while (n>0) :
4          if((n%2) == 1) :
5              rval += “1”
6          else :
7              rval +=“0”
8          n = n // 2
9
10     return rval
```



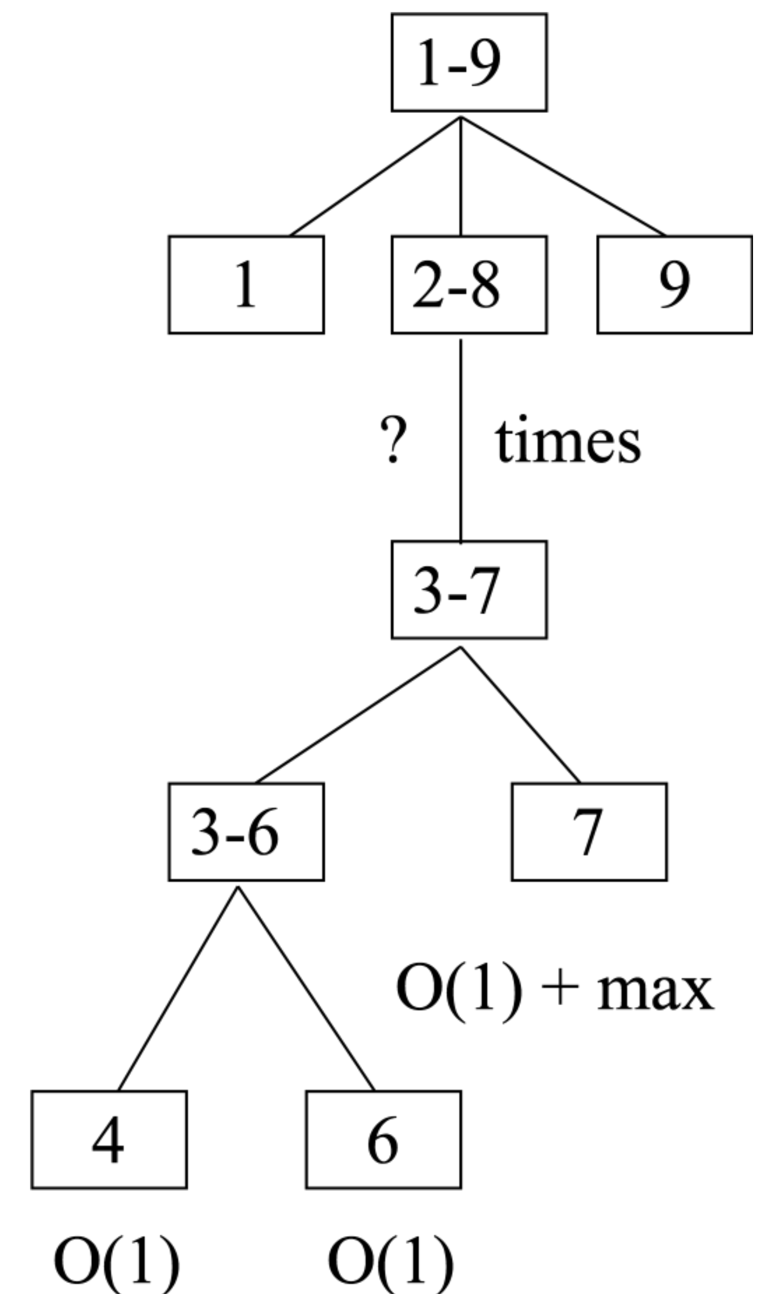
```
1  def int2bin(n):
2      rval = ""
3      while (n>0) :
4          if((n%2) == 1) :
5              rval += “1”
6          else :
7              rval +=“0”
8          n = n // 2
9
10     return rval
```



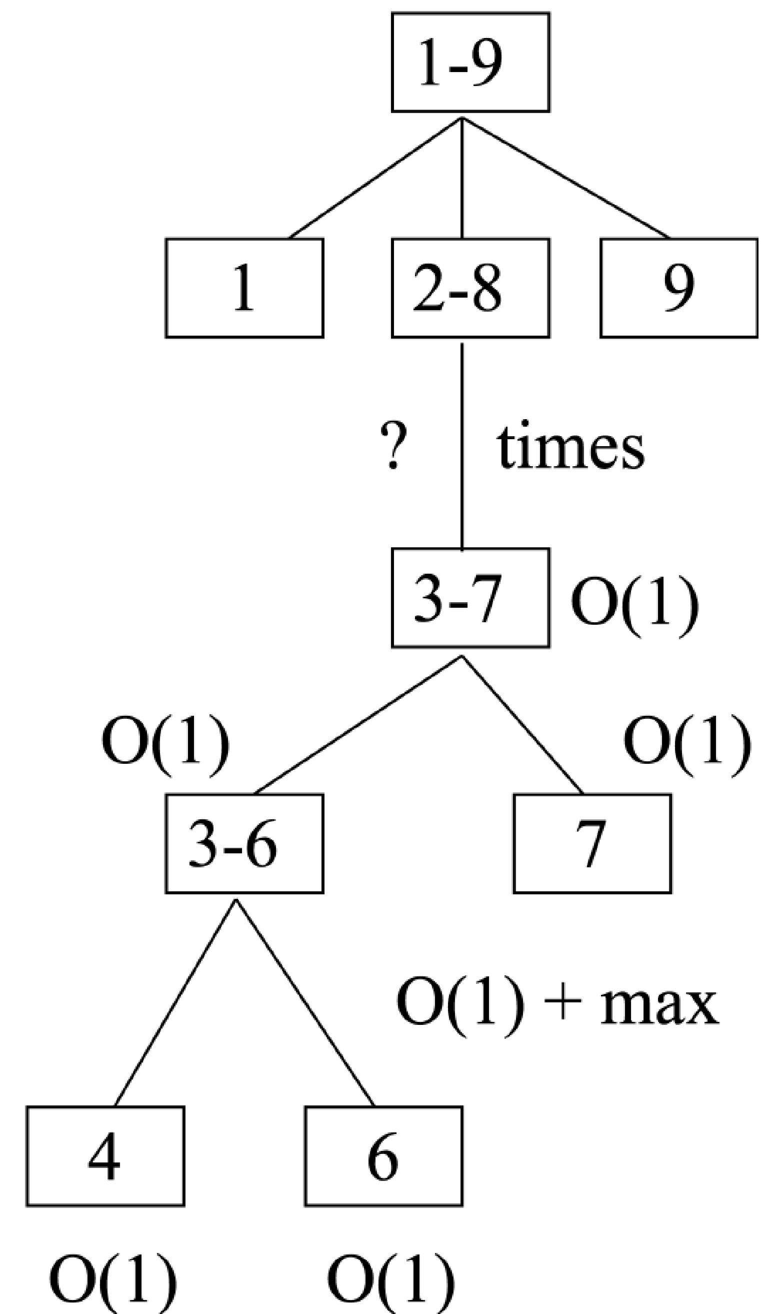
```
1 def int2bin(n):
2     rval = ""
3     while (n>0) :
4         if((n%2) == 1) :
5             rval += “1”
6         else :
7             rval +=“0”
8         n = n // 2
9
10    return rval
```



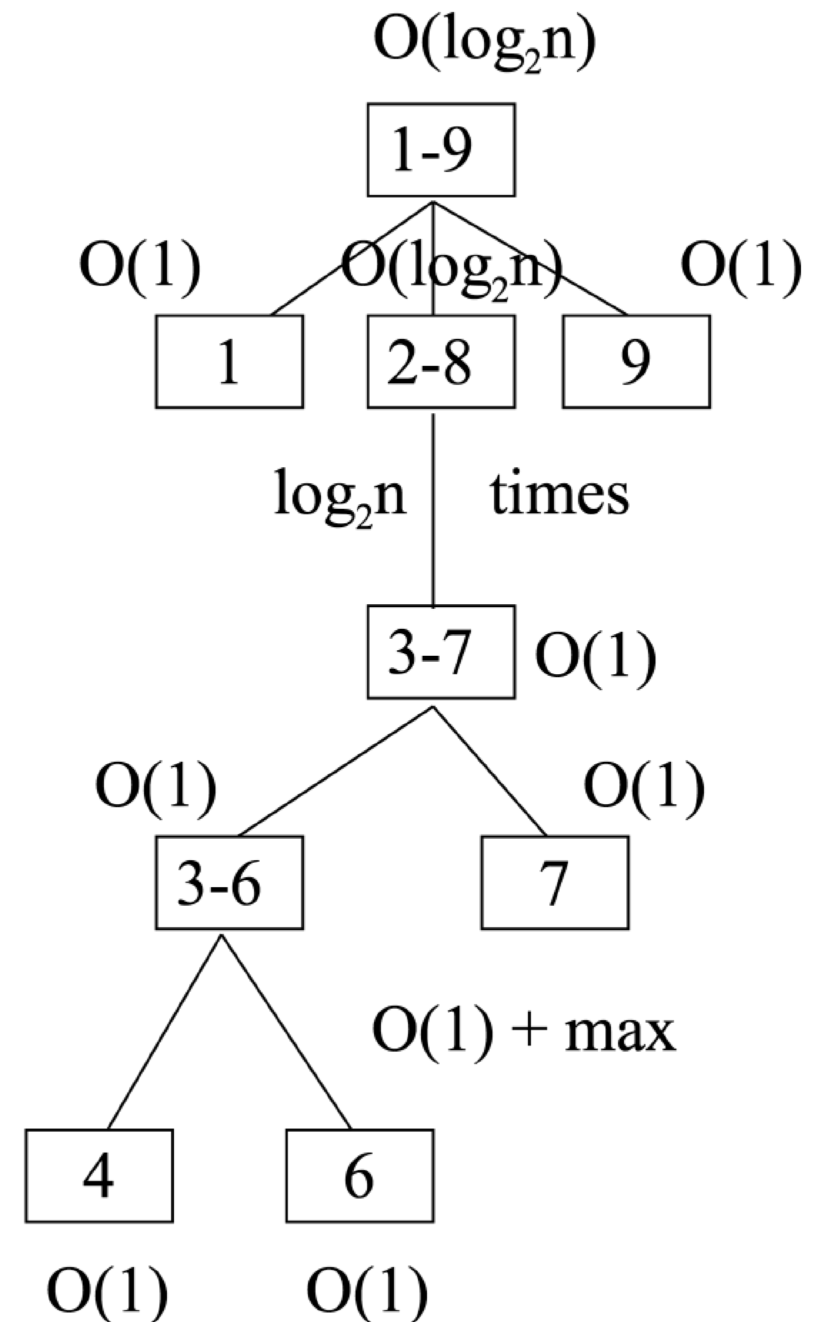
```
1 def int2bin(n):
2     rval = ""
3     while (n>0) :
4         if((n%2) == 1) :
5             rval += “1”
6         else :
7             rval +=“0”
8         n = n // 2
9
10    return rval
```



```python
1 def int2bin(n):
2     rval = ""
3     while (n>0) :
4         if((n%2) == 1) :
5             rval += "1"
6         else :
7             rval +="0"
8         n = n // 2
9
10    return rval
```



```
1  def int2bin(n):
2      rval = ""
3      while (n>0) :
4          if((n%2) == 1) :
5              rval += "1"
6          else :
7              rval +="0"
8          n = n // 2
9
10     return rval
```



- If we have time, discuss anagram examples in textbook
- We have a workshop to do during class, due date is a couple of days from now just in case
- There will also be a new lab assignment on Blackboard after class, due in 1 week

Questions?