Lect 12 – Alg Analysis

Rob Capra

INLS 490-172

Algorithm Analysis

- Often there is more than one algorithm to solve a problem.

Algorithm Analysis

- How do we decide if one program is “better” than another?

Algorithm Analysis

- How do we decide if one program is “better” than another?
  - Runs faster
  - Uses less memory
  - Fewer lines of code
  - Easier to read/understand
  - …

Algorithm Analysis

- How do we decide if one program is “better” than another?
  - Runs faster ← **TIME**
  - Uses less memory ← **SPACE**
  - Fewer lines of code
  - Easier to read/understand
  - …

Sum of integers from 1..n

```
def sumofn(n):
    s = 0
    for i in range(1,n+1):
        s = s + i
    return s

print sumofn(4)
```

Keeping track of time

```
import time

def sumofn(n):
    start = time.time()
    s = 0
    for i in range(1,n+1):
        s = s + i
    end = time.time()
    return (s,end-start)

print "1000000"
for i in range(5):
    print "  Sum is %d;  time = %10.7f seconds." % sumofn(1000000)

print "10000000"
for i in range(5):
    print "  Sum is %d;  time = %10.7f seconds." % sumofn(10000000)
```

For n = n * 10, takes about 10 times longer. (linear)

Sum of integers from 1..n

```
def sumofn(n):
    s = 0
    for i in range(1,n+1):
        s = s + i
    return s

print sumofn(4)
```

$$\sum_{i=1}^{n} i = \frac{(n)(n+1)}{2}$$

```
def sumofn2(n):
    s = (n*(n+1))/2
    return s

print sumofn2(4)
```

Constant time

```
import time

def sumofn2(n):
    start = time.time()
    s = (n*(n+1))/2
    end = time.time()
    return (s,end-start)

print "1000000"
for i in range(5):
    print "  Sum is %d;  time = %10.7f seconds." % sumofn2(1000000)

print "10000000"
for i in range(5):
    print "  Sum is %d;  time = %10.7f seconds." % sumofn2(10000000)
```

Problems?

- My computer was made in 2007.
- Newer computers are faster.
- Your computer might run sumofn() faster than mine does sumofn2() !!!

- Maybe we need something other than wall-clock time to compare algorithms…

Solution: Count steps

- What steps should we count?
  - “Basic unit of computation”
  - Huh?
  - For now, we will count assignment statements
  - How many times does something get assigned to a variable?

T(n)

```
def sumofn(n):
    s = 0
    for i in range(1,n+1):
        s = s + i
    return s

print sumofn(4)
```

```
For sumofn(4):

s = 0
s = 0 + 1
s = 1 + 2
s = 3 + 3
s = 6 + 4
```

Happens once

Happens *n* times

- We can define the time (in terms of the number of steps) as a function T(n).
- N is the “size of the problem”.
- T(n) is the time it takes to solve a problem of size n, namely 1+n steps.
- T(n) = 1 + n

Big-O

- So we had figured out the sumofn was T(n) = 1 + n
- As *n* gets large, the 1 does not matter so much
  - For n=4, the 1 is 20% of the steps
  - For n=400, the 1 is 0.25% of the steps
- Big-O drops the lower-order parts of T(n)
- sumofn() is said to be O(*n*)

```
def sumofn(n):
    s = 0
    for i in range(1,n+1):
        s = s + i
    return s

print sumofn(4)
```

Big-O

- Suppose we had figured out that some algorithm took
  - $T(n) = 20 + 5n + 10 + 3n^2 + 12n$
- We could first reduce this:
  - $T(n) = 30 + 17n + 3n^2$
- And then conclude that it was $O(n^2)$
- O() is the “order of magnitude”

Big-O

Table 1: Common Functions for Big-O

f(n)	Name
1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Log Linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential

Big-O

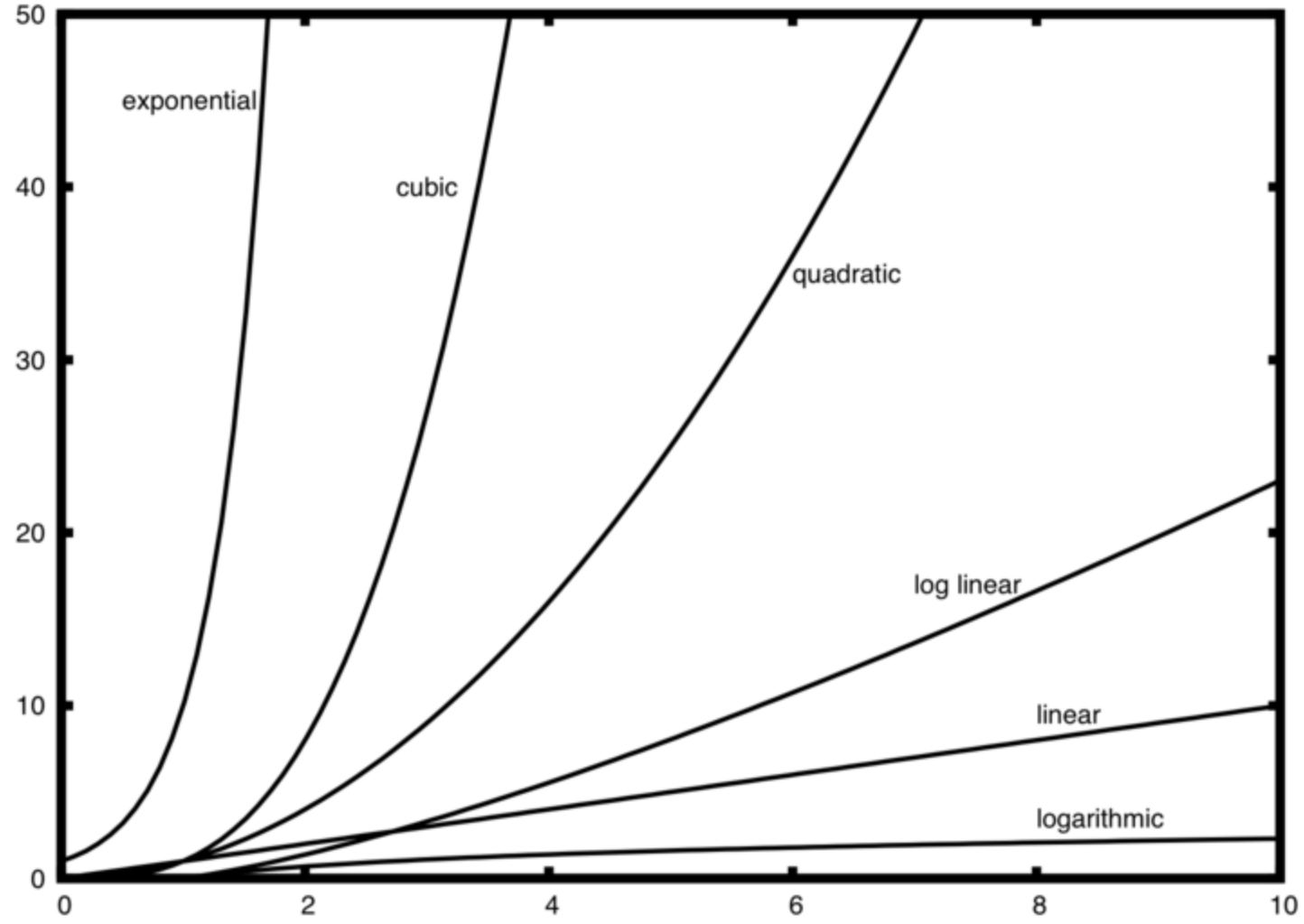

Figure 1: Plot of Common Big-O Functions

Big-O

- Best case
- Worst case
- Average case

Best, Worst, Average Case

```
import time

a = range(1,10000000)

start = time.time()
for b in a:
    #if b == 1:
    #if b == 9999999:
    if b == 5000000:
        break
end = time.time()
print end-start
```