

Algorithm Analysis

CSCI 112, Winter 2023, Lecture 1

January 1, 2023

Review Python

- Strings
 - Lists
 - Tuples
 - Dictionaries
 - Sets
 - List comprehensions
 - Functions, including `lambda`
 - Classes
 - Exception handling
 - Unit tests
- All of these are reviewed in your textbook.
 - Review the exercises at the end of the chapter.
 - There are also many online Python tutorials.

Algorithm Analysis

Algorithms

- Classic multiply

$$\begin{array}{r} 238 \\ \times 13 \\ \hline 714 \\ 238 \\ \hline 3094 \end{array}$$

Algorithms

- Classic multiply

```
  238
x  13
-----
 714
 238
-----
3094
```

- Peasant multiply

238	13	-->	238
476	6		
952	3	-->	952
1904	1	-->	1904

			3094

Programs

```
1 def classic_multiply(a, b):
2     bdigits = reversed([int(x) for x in list(str(b))])
3     sum = 0
4     for i, digit in enumerate(bdigits):
5         sum += digit * a * 10 ** i
6     return sum
```

```
1 def peasant_multiply(a, b):
2     sum = 0
3     while b > 0:
4         if b % 2 == 1:
5             sum += a
6         a, b = a*2, b//2
7     return sum
```

Which is better? What does that even mean?

Algorithm vs Program

- An **algorithm** is an explicit, step-by-step procedure for solving a problem.
 - There can be many algorithms to solve the same problem.
 - There are problems for which there exists no algorithm!
- A **program** is an explicit, step-by-step set of instructions to a computer.
 - Usually, a program is an **implementation** of an algorithm.

Measuring runtime

```
1 a=5
2 b=6
3 c=10
4 for i in range(n):
5     for j in range(n):
6         x = i * i
7         y = j * j
8         z = i * j
9 for k in range(n):
10     w = a*k + 45
11     v = b*b
12 d = 33
```

- We could time it.
- That would depend on processor, *etc.*
- Hard to compare this algorithm with another.

- Better to, e.g., count the number of assignments.
- There are 4 assignments outside the loops.
- There are 3 in the body of the first loop, which is done n^2 times.
- There are 2 in the body of the second loop, which is done n times.
- Total number of assignments: $3n^2 + 2n^2 + 4$

Measuring runtime

```
1 a=5
2 b=6
3 c=10
4 for i in range(n):
5     for j in range(n):
6         x = i * i
7         y = j * j
8         z = i * j
9 for k in range(n):
10     w = a*k + 45
11     v = b*b
12 d = 33
```

- Total number of assignments: $3n^2 + 2n^2 + 4$
- As $n \rightarrow \infty$, this function is dominated by the $3n^2$ term.
- We don't care about constants, because that is just units.
- We say that this program is **order** n^2 , or

$$O(n^2)$$

A refresher on some basic math

Logarithms

Logarithms, by shortening the labors, doubled the life of an astronomer.

– *Pierre-Simon, marquis de Laplace*

$$\log_b(a) = c \Leftrightarrow b^c = a$$

$$\log_b(ac) = \log_b(a) + \log_b(c)$$



Logarithm refresher

$$\log_b(a) = c \Leftrightarrow b^c = a$$

$$b^{\log_b(a)} = a$$

$$\log_b(ac) = \log_b(a) + \log_b(c)$$

$$\log_b(a) = \frac{\log_d(a)}{\log_d(b)}$$

$$\frac{a}{b} = \frac{a/d}{b/d}$$

$$\log_d(b) \log_b(a) = \log_d(a)$$

$$\frac{b}{d} \frac{a}{b} = \frac{a}{d}$$

$$\log_b(a) = \frac{1}{\log_a(b)}$$

$$\log_b(a) = k \log_d(a)$$

$$\log_{10}(1203248) \approx 6$$

$$\log_2(1203248) \approx 20$$

Summation rules

$$\sum_{i=m}^n c = (n - m + 1)c$$

$$\sum_{i=m}^n ca_i = c \sum_{i=m}^n a_i$$

$$\sum_{i=m}^n (a_i + b_i) = \sum_{i=m}^n a_i + \sum_{i=m}^n b_i$$

Summation rules

$$\sum_{i=m}^n a_{i+k} = \sum_{i=m+k}^{n+k} a_i$$

$$\sum_{i=m}^n a_i x^{i+k} = x^k \sum_{i=m}^n a_i x^i$$

$$\sum_{i=m}^n (a_i - a_{i-1}) = a_n - a_{m-1}$$

Sum of constant

$$\sum_{i=1}^n 1 = 1 + 1 + 1 \dots + 1$$
$$= n$$

$$\sum_{i=1}^n c = c + c + c \dots + c$$
$$= cn$$

Sum of i

$$\begin{array}{rcccccccc} & \sum_{i=1}^n i & = & 1 & + & 2 & + & \dots & + & n \\ + & \sum_{i=1}^n i & = & n & + & n-1 & + & \dots & + & 1 \\ \hline & & & = & n+1 & + & n+1 & + & \dots & + & n+1 \\ & & & = & n(n+1) & & & & & & \end{array}$$

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

Sum of i , another way

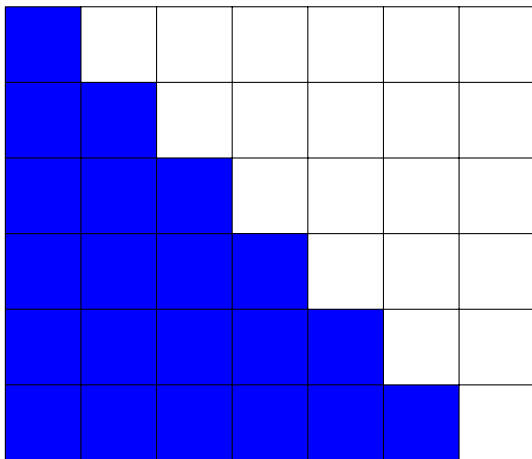
$$\sum_{i=m}^n (a_i - a_{i-1}) = a_n - a_{m-1}$$

$$n^2 = n^2 - 0^2$$

$$\begin{aligned} n^2 &= \sum_{i=1}^n (i^2 - (i-1)^2) \\ &= \sum_{i=1}^n (2i - 1) \\ &= 2 \sum_{i=1}^n i - \sum_{i=1}^n 1 \\ &= 2 \sum_{i=1}^n i - n \\ &\Rightarrow \sum_{i=1}^n i = \frac{n^2 + n}{2} \end{aligned}$$

Sum of i , easiest way

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



Sum of Odd Numbers

$$\begin{aligned}1 + 3 + \dots + (2n + 1) &= \sum_{i=0}^n (2i + 1) \\&= \sum_{i=0}^n 2i + \sum_{i=0}^n 1 \\&= 2 \sum_{i=0}^n i + (n + 1) \\&= 2n(n + 1)/2 + (n + 1) \\&= (n + 1)^2\end{aligned}$$

Geometric Sum

$$S_n = \sum_{i=0}^n a^i$$

$$= 1 + a + a^2 + \dots + a^n$$

$$S_{n+1} = (1 + a + a^2 + \dots + a^n) + a^{n+1}$$

$$= S_n + a^{n+1}$$

$$S_{n+1} = 1 + (a + a^2 + \dots + a^n + a^{n+1})$$

$$= 1 + a(1 + a + a^2 + \dots + a^n)$$

$$= 1 + aS_n$$

$$S_n + a^{n+1} = 1 + aS_n$$

\Rightarrow

$$S_n = \frac{a^{n+1} - 1}{a - 1}$$

$$= \sum_{i=0}^n a^i$$

Geometric sum, another way

$$\begin{aligned}(a-1) \sum_{i=0}^n a^i &= \sum_{i=0}^n (a^{i+1} - a^i) \\ &= \sum_{i=1}^{n+1} (a^i - a^{i-1}) \\ &= a^{n+1} - a^0 \\ &= a^{n+1} - 1\end{aligned}$$

$$\sum_{i=0}^n a^i = \frac{a^{n+1} - 1}{a - 1}$$

Geometric Sum

$$\sum_{i=0}^n a^i = \frac{a^{n+1} - 1}{a - 1}$$

If $a > 1$ and n is large, then

$$a^{n+1} - 1 \approx a^{n+1}$$

Therefore

$$\begin{aligned} 1 + a + a^2 + \dots + a^n &= \frac{a^{n+1} - 1}{a - 1} \\ &\approx \left(\frac{a}{a - 1} \right) a^n \\ &= ka^n \end{aligned}$$

Memorizing the Geometric Sum

$$\sum_{i=0}^n a^i = 1 + a + a^2 + \dots + a^n$$

$$\approx a^n$$

$$= \frac{a^{n+1}}{a}$$

$$\approx \frac{a^{n+1} - 1}{a - 1} = \sum_{i=0}^n a^i$$

Algorithm analysis

- We want to study the runtime of programs. How long does it take to solve a problem with the computer?
- Constants don't matter, since it is just a change in units. Two minutes or 120 seconds? Units are arbitrary.
- We also don't care about machine speed, programming language, etc. A computer that is twice as fast will just add a constant factor of $1/2$, anyway.
- We are also especially interested in how well programs scale to larger and larger problems.
- If the problem is ten times bigger, will it take ten times as long to solve? More? Less?
- This is the essence of what we want to measure. The **growth rate** of a program's execution time as the problem gets larger.
- This will be independent of units.

Big O notation

- We assume all our functions are positive (they measure elapsed time, after all).
- We say that a function $f(x)$ is Big-O $g(x)$,

$$f(x) = O(g(x))$$

if there is some constant, C , such that $f(x) \leq Cg(x)$ for x big enough.

- Example, let $f(x) = 3x^2 + 10x + 2$. Then $f(x) = O(x^2)$ because

$$\begin{aligned} f(x) &= 3x^2 + 10x + 2 \\ &\leq 3x^2 + 10x^2 + 2x^2 && \text{if } x \geq 1 \\ &= 15x^2 \end{aligned}$$

Big O notation

- Note that $f(x) = 3x^2 + 10x + 2$ is also $O(x^3)$, since

$$\begin{aligned} f(x) &= 3x^2 + 10x + 2 \\ &\leq 3x^3 + 10x^3 + 2x^3 && \text{if } x \geq 1 \\ &= 15x^3 \end{aligned}$$

- However, $f(x) \neq O(x)$, because, let C be any constant, then

$$\begin{aligned} Cx &< x^2 && \text{if } x > C \\ &< 3x^2 + 10x + 2 && \text{if } x > \max(1, C) \\ &= f(x) \end{aligned}$$

- So it is not possible to find C such that $f(x) \leq Cx$ for large x .
- If C is large, say $C = 10^{1000000}$, then $f(x) \leq Cx$ for quite a few values of x . But, eventually, ...

Summing up

$$f(x) = 3x^2 + 10x + 2$$

$$f(x) = O(x^3)$$

$$f(x) = O(x^2)$$

$$f(x) \neq O(x)$$

$$f(x) \neq O(1)$$

- We say $O(x^2)$ is a **tight** bound.
- $O(x^3)$ is a **loose** bound.
- Generally we prefer to find bounds as tight as possible.

Alternate method for showing Big-O

If $f(x)$ and $g(x)$ are continuous, then

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} < \infty$$

implies

$$f(x) = O(g(x))$$

So we can use l'Hopital's rule!

Big O Hierarchy

$O(1)$

$O(\log(\log(n)))$

$O(\log n)$

$O(\sqrt{n})$

$O(n)$

$O(n \log n)$

$O(n^2)$

$O(n^2 \log n)$

$O(n^3)$

$O(2^n)$

$O(3^n)$

$O(n!)$

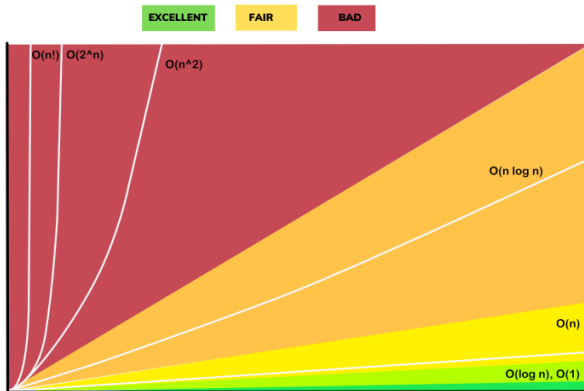
- Each function is Big-O of all the functions below it.
- No function is Big-O of any function above it.
- Since log to one base is a constant multiple of log to a different base:

$$\log_a n = O(\log_b n)$$

for any bases a and b .

Big O Chart

BIG O COMPLEXITY CHART



> hackr.io

Algorithm Analysis

How many assignment statements are made?

```
1 for i in range(n):  
2     x = i + f(x)
```

Algorithm Analysis

How many assignment statements are made?

```
1 for i in range(n):  
2     x = i + f(x)
```

$$\sum_{i=0}^{n-1} (1 + 1) = \sum_{i=0}^{n-1} 2 \\ = 2n$$

- This program is $O(n)$

Algorithm Analysis

How many assignment statements are made?

```
1 for i in range(n):  
2     for j in range(n):  
3         x = i + j + f(x)
```

- This program is $O(n^2)$

Algorithm Analysis

How many assignment statements are made?

```
1 for i in range(n):  
2     for j in range(n):  
3         x = i + j + f(x)
```

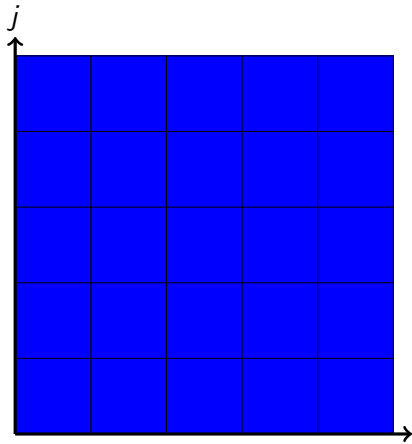
- This program is $O(n^2)$

$$\begin{aligned}\sum_{i=0}^{n-1} \left(1 + \sum_{j=0}^{n-1} (1 + 1) \right) &= \sum_{i=0}^{n-1} \left(1 + \sum_{j=0}^{n-1} 2 \right) \\ &= \sum_{i=0}^{n-1} (1 + 2n) \\ &= \sum_{i=0}^{n-1} 1 + \sum_{i=0}^{n-1} 2n \\ &= n + 2 \sum_{i=0}^{n-1} n \\ &= 2n^2 + n\end{aligned}$$

Visualizing nested for loops

```
1 for i in range(n):  
2     for j in range(n):  
3         x = i + j + f(x)
```

$$O(n^2)$$



Algorithm Analysis

How many assignment statements are made?

```
1 for i in range(n):  
2     for j in range(i):  
3         x = x + f(x) + j
```

Algorithm Analysis

How many assignment statements are made?

```
1 for i in range(n):  
2     for j in range(i):  
3         x = x + f(x) + j
```

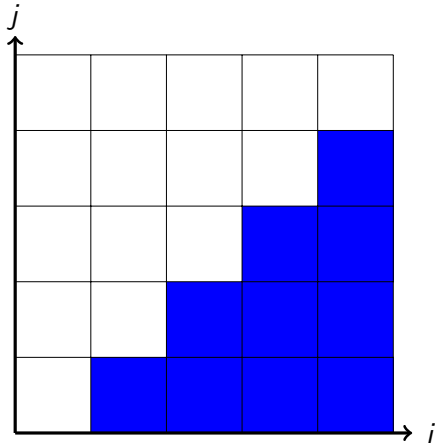
$$\begin{aligned}\sum_{i=0}^{n-1} \left(1 + \sum_{j=0}^{i-1} (1 + 1) \right) &= \sum_{i=0}^{n-1} \left(1 + \sum_{j=0}^{i-1} 2 \right) \\ &= \sum_{i=0}^{n-1} (1 + 2i) \\ &= \sum_{i=0}^{n-1} 1 + 2 \sum_{i=0}^{n-1} i \\ &= n + n(n + 1) \\ &= n^2 + 2n\end{aligned}$$

- This function is also $O(n^2)$

Visualizing nested for loops

```
1 for i in range(n):  
2     for j in range(i):  
3         x = x + f(x) + j
```

$$O(n^2)$$



Algorithm Analysis

How many lines does doodle draw?

```
1 def doodle(n, m):  
2     if n > 0:  
3         line(n, n, m, m)  
4         line(n, m, m, n)  
5         doodle(n-1, m)
```

Algorithm Analysis

How many lines does doodle draw?

```
1 def doodle(n, m):  
2     if n > 0:  
3         line(n, n, m, m)  
4         line(n, m, m, n)  
5         doodle(n-1, m)
```

- $O(n)$

$$f(0) = 0$$

$$f(n) = f(n-1) + 2$$

$$= (f(n-2) + 2) + 2$$

$$= ((f(n-3) + 2) + 2) + 2$$

$$= f(n-k) + 2k$$

$$= \dots$$

$$= f(n-n) + 2n$$

$$= 2n$$

Algorithm Analysis

How many additions does quibble make?

```
1 def quibble(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return quibble(n-1) + quibble(n-1)
```

$$f(0) = 0$$

$$\begin{aligned} f(n) &= 2f(n-1) + 1 \\ &= 2(2f(n-2) + 1) + 1 \\ &= 2^2f(n-2) + 2 + 1 \\ &= 2^3f(n-3) + 4 + 2 + 1 \\ &= 2^4f(n-4) + 8 + 4 + 2 + 1 \end{aligned}$$

Algorithm Analysis

How many additions does quibble make?

```
1 def quibble(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return quibble(n-1) + quibble(n-1)
```

$$f(0) = 0$$

$$f(n) = 2f(n-1) + 1$$

$$= 2(2f(n-2) + 1) + 1$$

$$= 2^2 f(n-2) + 2 + 1$$

$$= 2^3 f(n-3) + 4 + 2 + 1$$

$$= 2^4 f(n-4) + 8 + 4 + 2 + 1$$

$$= 2^k f(n-k) + \sum_{i=0}^{k-1} 2^i$$

$$= 2^n f(n-n) + \sum_{i=0}^{n-1} 2^i$$

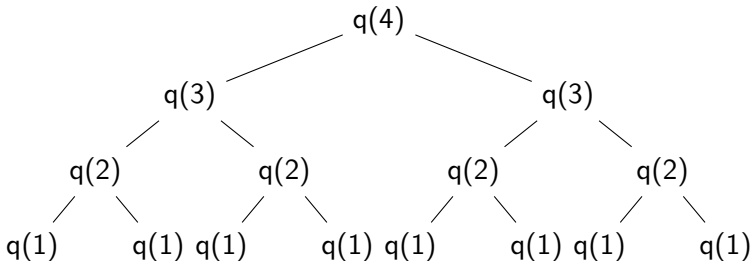
$$= 2^n f(0) + \frac{2^n - 1}{2 - 1}$$

$$= 2^n - 1 = O(2^n)$$

Visualizing recursion

```
1 def quibble(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return quibble(n-1) + quibble(n-1)
```

$O(2^n)$



Fibonacci

```
1 def fibonacci(n):  
2     if n < 2:  
3         return n  
4     else:  
5         return fibonacci(n-1) + fibonacci(n-2)
```

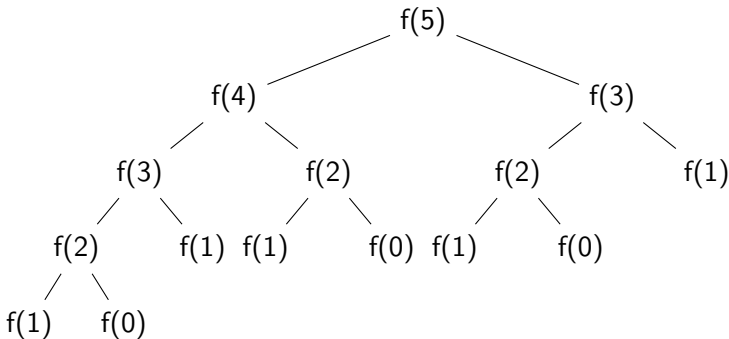
The mathematics is difficult, but the result is similar to quibble:

$$O(1.7)^n$$

Visualizing Fibonacci

```
1 def fibonacci(n):  
2     if n < 2:  
3         return n  
4     else:  
5         return fibonacci(n-1) + fibonacci(n-2)
```

$O(1.7^n)$



How many times can you divide an integer in half?

$$30 \rightarrow 15 \rightarrow 7 \rightarrow 3 \rightarrow 1 : 4$$

$$32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 : 5$$

$$40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 2 \rightarrow 1 : 5$$

$$50 \rightarrow 25 \rightarrow 12 \rightarrow 6 \rightarrow 3 \rightarrow 1 : 5$$

$$64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 : 6$$

$$70 \rightarrow 35 \rightarrow 17 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 : 6$$

$$128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 : 7$$

$$200 \rightarrow 100 \rightarrow 50 \rightarrow 25 \rightarrow 12 \rightarrow 6 \rightarrow 3 \rightarrow 1 : 7$$

$$256 \rightarrow 128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 : 8$$

How many times can you divide an integer in half?

$$30 \rightarrow 15 \rightarrow 7 \rightarrow 3 \rightarrow 1 : 4$$

$$32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 : 5$$

$$40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 2 \rightarrow 1 : 5$$

$$50 \rightarrow 25 \rightarrow 12 \rightarrow 6 \rightarrow 3 \rightarrow 1 : 5$$

$$64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 : 6$$

$$70 \rightarrow 35 \rightarrow 17 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 : 6$$

$$128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 : 7$$

$$200 \rightarrow 100 \rightarrow 50 \rightarrow 25 \rightarrow 12 \rightarrow 6 \rightarrow 3 \rightarrow 1 : 7$$

$$256 \rightarrow 128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 : 8$$

$$\lfloor \log_2(n) \rfloor \leq \log_2(n)$$

Algorithm Analysis

How many additions does shoop make at a maximum?

```
1 def shoop(n):  
2     if n == 1:  
3         return 7  
4     else:  
5         return shoop(n//2) + 3
```

$$f(1) = 0$$

$$f(n) = f(n/2) + 1$$

$$= (f(n/4) + 1) + 1$$

$$= ((f(n/8) + 1) + 1) + 1$$

$$= f(n/2^k) + k$$

$$= f(n/2^{\log_2(n)}) + \log_2(n)$$

$$= f(n/n) + \log_2(n)$$

$$= \log_2(n) = O(\log n)$$

Algorithm Analysis

How many additions does snarfile make at a maximum?

```
1 def snarfile(n):  
2     if n == 1:  
3         return 7  
4     else:  
5         return snarfile(n//2) + snarfile(n//2)
```

$$f(1) = 0$$

$$f(n) = 2f(n/2) + 1$$

$$= 2(2f(n/4) + 1) + 1$$

$$= 2^2 f(n/2^2) + 2 + 1$$

$$= 2^3 f(n/2^3) + 4 + 2 + 1$$

$$= 2^k f(n/2^k) + \sum_{i=0}^{k-1} 2^i$$

Algorithm Analysis

How many additions does snarfile make at a maximum?

```
1 def snarfile(n):  
2     if n == 1:  
3         return 7  
4     else:  
5         return snarfile(n//2) + snarfile(n//2)
```

$$f(1) = 0$$

$$f(n) = 2f(n/2) + 1$$

$$= 2(2f(n/4) + 1) + 1$$

$$= 2^2 f(n/2^2) + 2 + 1$$

$$= 2^3 f(n/2^3) + 4 + 2 + 1$$

$$= 2^k f(n/2^k) + \sum_{i=0}^{k-1} 2^i$$

$$= \dots$$

$$= 2^{\log_2(n)} f(n/2^{\log_2(n)}) + \sum_{i=0}^{\log_2(n)-1} 2^i$$

$$= 0 + (2^{\log_2(n)-1+1} - 1)/(2 - 1)$$

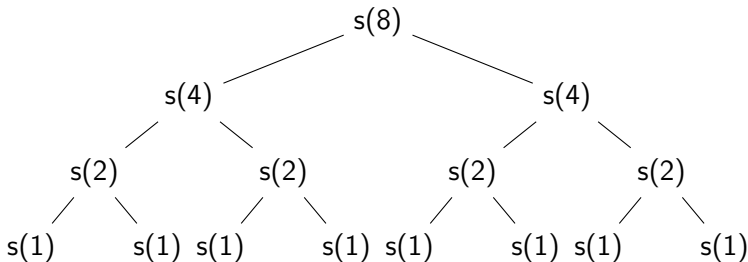
$$= 2^{\log_2(n)} - 1$$

$$= n - 1 = O(n)$$

Visualizing snarfile

```
1 def snarfile(n):  
2     if n == 1:  
3         return 7  
4     else:  
5         return snarfile(n//2) + snarfile(n//2)
```

$O(n)$



Two algorithms for power

$$a^b = a(a^{b-1})$$

vs.

$$a^b = (a^{b/2})^2$$

```
1 def powlinear(a, b):  
2     if b == 0:  
3         return 1  
4     else:  
5         return a * powlinear(a, b-1)
```

```
1 def powlog(a, b):  
2     if b == 0:  
3         return 1  
4     elif b % 2 == 1:  
5         return a * powlog(a, b-1)  
6     else:  
7         x = powlog(a, b//2)  
8         return x * x
```

What are their running times?

Fibonacci revisited

```
1 def fibonacci(n):  
2     if n < 2:  
3         return n  
4     else:  
5         return fibonacci(n-1) + fibonacci(n-2)
```

Running time?

Fibonacci revisited

```
1 def fibonacci(n):  
2     if n < 2:  
3         return n  
4     else:  
5         return fibonacci(n-1) + fibonacci(n-2)
```

Running time?

$O(1.7^n)$

```
1 def fibonacci(n):  
2     a,b = 0,1  
3     while n > 0:  
4         n,a,b = n-1, b, a+b  
5     return a
```

Runining time?

Fibonacci revisited

```
1 def fibonacci(n):  
2     if n < 2:  
3         return n  
4     else:  
5         return fibonacci(n-1) + fibonacci(n-2)
```

Running time?

$O(1.7^n)$

```
1 def fibonacci(n):  
2     a,b = 0,1  
3     while n > 0:  
4         n,a,b = n-1, b, a+b  
5     return a
```

Runining time?

$O(n)$

Fibonacci revisited

```
1 def fibonacci(n):  
2     if n < 2:  
3         return n  
4     else:  
5         return fibonacci(n-1) + fibonacci(n-2)
```

Running time?

$O(1.7^n)$

```
1 def fibonacci(n):  
2     a,b = 0,1  
3     while n > 0:  
4         n,a,b = n-1, b, a+b  
5     return a
```

Runining time?

$O(n)$

$1.7^{100} > 10^{23} \gg 100$

Fibonacci revisited

```
1 def fibonacci(n):  
2     if n < 2:  
3         return n  
4     else:  
5         return fibonacci(n-1) + fibonacci(n-2)
```

Running time?

$O(1.7^n)$

```
1 def fibonacci(n):  
2     a,b = 0,1  
3     while n > 0:  
4         n,a,b = n-1, b, a+b  
5     return a
```

Runining time?

$O(n)$

$1.7^{100} > 10^{23} \gg 100$

Can we do better?

Fibonacci revisited

$$(a, b) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = (b, a + b)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = (1, 1)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2 = (1, 2)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^3 = (2, 3)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^4 = (3, 5)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^5 = (5, 8)$$

Fibonacci revisited

$$(a, b) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = (b, a + b)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = (1, 1)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2 = (1, 2)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^3 = (2, 3)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^4 = (3, 5)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^5 = (5, 8)$$

- Does this give you an idea?

Fibonacci revisited

$$(a, b) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = (b, a + b)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = (1, 1)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2 = (1, 2)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^3 = (2, 3)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^4 = (3, 5)$$

$$(0, 1) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^5 = (5, 8)$$

- Does this give you an idea?
- Can you write a $O(\log n)$ fibonacci?

Lab: $O(\log n)$ fibonacci

- Develop a `Matrix` class that will handle matrices of any dimension.
- Your matrix class will handle matrix multiplication and matrix power by overloading the `__mul__` and `__pow__` methods.
- Also develop a `__str__` method.
- An example interaction is at the right. The initializing parameters for a matrix object are the number of rows, the number of columns, and a list of all the elements.
- Use this class to develop a $O(\log n)$ fibonacci function.
- Time and compare the three fibonacci's: exponential, linear, and log. To get meaningful results, compare exponential and linear in one comparison, and linear and log in another comparison.

Performance of log fibonacci

```
>>> m = Matrix(2,2,[0,1,1,1])  
>>> for i in range(4,7):  
    print(m ** i)
```

| 2 3|

| 3 5|

| 3 5|

| 5 8|

| 5 8|

| 8 13|

Time in nanoseconds for finding the 1000000th Fibonacci number:

1 Linear: 8093750000

2 Log: 203125000

Anagram problem

- Determine if two strings are anagrams:
 inch chin
 study dusty
 stressed desserts
 cried cider
- What are some algorithms?

Anagram algorithm 1: checking off

```
1 def anagramSolution1(s1,s2):
2     stillOK = True
3     if len(s1) != len(s2):
4         stillOK = False
5     alist = list(s2)
6     pos1 = 0
7     while pos1 < len(s1) and stillOK:
8         pos2 = 0
9         found = False
10        while pos2 < len(alist) and not found:
11            if s1[pos1] == alist[pos2]:
12                found = True
13            else:
14                pos2 = pos2 + 1
15        if found:
16            alist[pos2] = None
17        else:
18            stillOK = False
19        pos1 = pos1 + 1
20    return stillOK
```


Anagram algorithm 2: sort and compare

```
1 def anagramSolution2(s1,s2):
2     alist1 = list(s1)
3     alist2 = list(s2)
4
5     alist1.sort()
6     alist2.sort()
7
8     pos = 0
9     matches = True
10
11     while pos < len(s1) and matches:
12         if alist1[pos]==alist2[pos]:
13             pos = pos + 1
14         else:
15             matches = False
16
17     return matches
```

Anagram algorithm 3: brute force

```
1 def anagramSolution3(s1, s2):
2     return s1 in all_permutations(s2)
3
4 def all_permutations(s):
5     if len(s) == 1:
6         return [s]
7     else:
8         shorts = all_permutations(s[1:])
9         longs = []
10        for short in shorts:
11            longs = longs + all_positions(s[0], short)
12        return longs
13
14 def all_positions(c, s):
15     strings = []
16     for i in range(len(s)+1):
17         strings.append(s[0:i] + c + s[i:])
18     return strings
```

Anagram algorithm 4: count and compare

```
1  c1 = [0]*26
2  c2 = [0]*26
3
4  for i in range(len(s1)):
5      pos = ord(s1[i])-ord('a')
6      c1[pos] = c1[pos] + 1
7
8  for i in range(len(s2)):
9      pos = ord(s2[i])-ord('a')
10     c2[pos] = c2[pos] + 1
11
12  j = 0
13  stillOK = True
14  while j<26 and stillOK:
15      if c1[j]==c2[j]:
16          j = j + 1
17      else:
18          stillOK = False
19
20  return stillOK
```

Runtime of Builtin Python Data Structures

- See text.