Algorithm Analysis

CSCI 112, Winter 2023, Lecture 1

December 30, 2022

Review Python

- Strings
- Lists
- Tuples
- Dictionaries
- Sets
- List comprehensions
- Functions, including `lambda`
- Classes
- Exception handling

- 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

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

Algorithms

- Classic multiply

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

- Peasant multiply

```
 238    13    -->   238
 476     6
 952     3    -->   952
1904     1    -->  1904
                   ----
                   3094
```

Programs

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

```
def peasant_multiply(a, b):
    sum = 0
    while b > 0:
        if b % 2 == 1:
            sum += a
        a, b = a*2, b//2
    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

```
a=5
b=6
c=10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a*k + 45
    v = b*b
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

```
a=5
b=6
c=10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a*k + 45
    v = b*b
d = 33
```

- Total number of assignments: $3n^2 + 2n^2 + 4$
- As $n \to \infty$, 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

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

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

Sum of i

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

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

Sum of i, another way

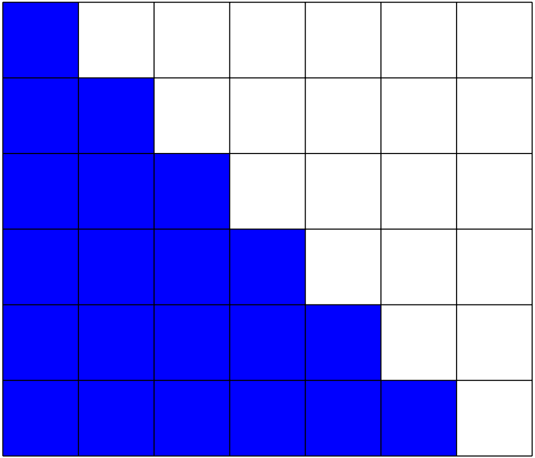
$$\sum_{i=m}^{n}(a_i - a_{i-1}) = a_n - a_{m-1}$$

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

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

Sum of i, easiest way

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



Sum of Odd Numbers

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

Geometric Sum

$$
\begin{aligned}
S_n &= \sum_{i=0}^{n} a^i \\
&= 1 + a + a^2 + \ldots + a^n \\
S_{n+1} &= (1 + a + a^2 + \ldots + a^n) + a^{n+1} \\
&= S_n + a^{n+1} \\
S_{n+1} &= 1 + (a + a^2 + \ldots + a^n + a^{n+1}) \\
&= 1 + a(1 + a + a^2 + \ldots + a^n) \\
&= 1 + aS_n
\end{aligned}
$$

$$
\begin{aligned}
S_n + a^{n+1} &= 1 + aS_n \\
&\Rightarrow \\
S_n &= \frac{a^{n+1} - 1}{a - 1} \\
&= \sum_{i=0}^{n} a^i
\end{aligned}
$$

Geometric sum, another way

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

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

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

$$1 + a + a^2 + \ldots + a^n = \frac{a^{n+1} - 1}{a - 1}$$

$$\approx \left(\frac{a}{a-1}\right) a^n$$
$$= ka^n$$

Memorizing the Geometric Sum

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

$$\approx a^n$$

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

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

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

$$
\begin{aligned}
f(x) &= 3x^2 + 10x + 2 \\
f(x) &= O(x^3) \\
f(x) &= O(x^2) \\
f(x) &\neq O(x) \\
f(x) &\neq O(1)
\end{aligned}
$$

- 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

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

implies

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

So we can use l'Hopital's rule!

Big O Hierarchy

$O(1)$

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

$O(\log n)$

$O(\sqrt{n})$

$O(n)$

$O(n \log n)$

$O(n^2)$

$O(n^2 \log n)$

$O(n^3)$

$O(2^n)$

$O(3^n)$

$O(n!)$

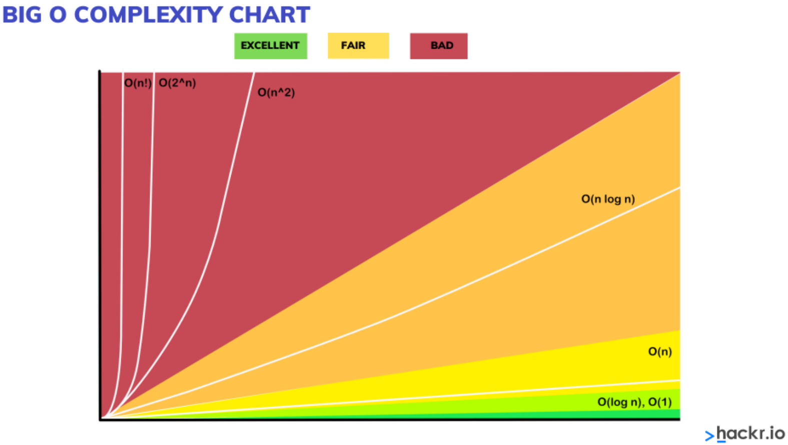
- Each function is Big-O of all the functions below it.
- No function is Big-O of any function above it.
- Since log to one base is a constant multiple of log to a different base:

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

for any bases a and b.

`https://www.bigocheatsheet.com/`

Big O Chart


BIG O COMPLEXITY CHART
EXCELLENT
FAIR
BAD
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
>hackr.io

Algorithm Analysis

How many assignment statements are made?

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

Algorithm Analysis

How many assignment statements are made?

```
for i in range(n):
    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?

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

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

Algorithm Analysis

How many assignment statements are made?

```python
for i in range(n):
    for j in range(n):
        x = i + j + f(x)
```

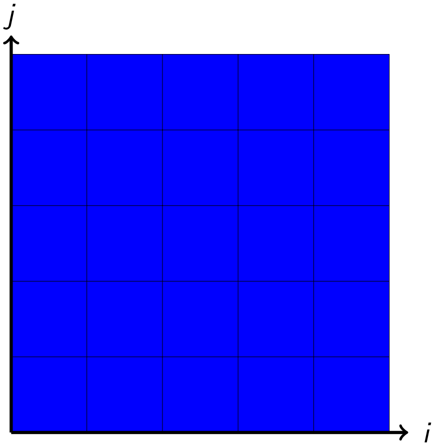
$$
\begin{aligned}
\sum_{i=0}^{n-1}\left(1+\sum_{j=0}^{n-1}(1+1)\right) &= \sum_{i=0}^{n-1}\left(1+\sum_{j=0}^{n-1}2\right) \\
&= \sum_{i=0}^{n-1}(1+2n) \\
&= \sum_{i=0}^{n-1}1+\sum_{i=0}^{n-1}2n \\
&= n+2\sum_{i=0}^{n-1}n \\
&= 2n^2+n
\end{aligned}
$$

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

Visualizing nested `for` loops

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

$O(n^2)$



Algorithm Analysis

How many assignment statements are made?

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

Algorithm Analysis

How many assignment statements are made?

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

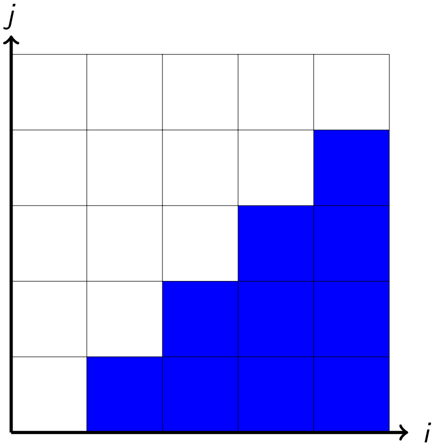
$$\sum_{i=0}^{n-1}\left(1+\sum_{j=0}^{i-1}(1+1)\right) = \sum_{i=0}^{n-1}\left(1+\sum_{j=0}^{i-1}2\right)$$
$$= \sum_{i=0}^{n-1}(1+2i)$$
$$= \sum_{i=0}^{n-1}1+2\sum_{i=0}^{n-1}i$$
$$= n+n(n+1)$$
$$= n^2+2n$$

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

Visualizing nested `for` loops

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

$O(n^2)$



Algorithm Analysis

How many lines does `doodle` draw?

```
def doodle(n, m):
  if n > 0:
    line(n, n, m, m)
    line(n, m, m, n)
    doodle(n-1, m)
```

Algorithm Analysis

How many lines does `doodle` draw?

```python
def doodle(n, m):
  if n > 0:
    line(n, n, m, m)
    line(n, m, m, n)
    doodle(n-1, m)
```

- $O(n)$

$$
\begin{aligned}
f(0) &= 0 \\
f(n) &= f(n-1) + 2 \\
&= (f(n-2) + 2) + 2 \\
&= ((f(n-3) + 2) + 2) + 2) \\
&= f(n-k) + 2k \\
&= \dots \\
&= f(n-n) + 2n \\
&= 2n
\end{aligned}
$$

Algorithm Analysis

How many additions does `quibble` make?

```
def quibble(n):
  if n == 0:
    return 1
  else:
    return quibble(n-1) + quibble(n-1)
```

$$
\begin{aligned}
f(0) &= 0 \\
f(n) &= 2f(n-1) + 1 \\
&= 2(2f(n-2) + 1) + 1 \\
&= 2^2 f(n-2) + 2 + 1 \\
&= 2^3 f(n-3) + 4 + 2 + 1 \\
&= 2^4 f(n-4) + 8 + 4 + 2 + 1
\end{aligned}
$$

Algorithm Analysis

How many additions does `quibble` make?

```
def quibble(n):
  if n == 0:
    return 1
  else:
    return quibble(n-1) + quibble(n-1)
```

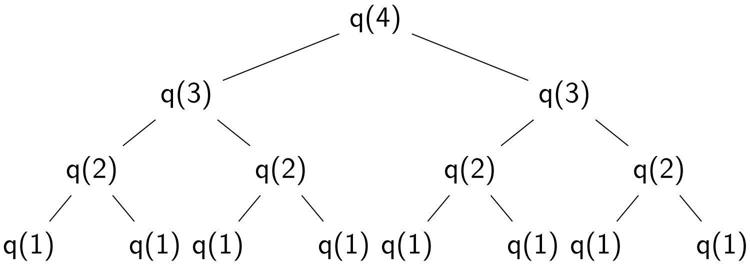
$$
\begin{aligned}
f(0) &= 0 \\
f(n) &= 2f(n-1) + 1 \\
&= 2(2f(n-2) + 1) + 1 \\
&= 2^2 f(n-2) + 2 + 1 \\
&= 2^3 f(n-3) + 4 + 2 + 1 \\
&= 2^4 f(n-4) + 8 + 4 + 2 + 1
\end{aligned}
$$

$$
\begin{aligned}
&= 2^k f(n-k) + \sum_{i=0}^{k-1} 2^i \\
&= 2^n f(n-n) + \sum_{i=0}^{n-1} 2^i \\
&= 2^n f(0) + \frac{2^n - 1}{2 - 1} \\
&= 2^n - 1 = O(2^n)
\end{aligned}
$$

Visualizing recursion

```python
def quibble(n):
  if n == 0:
    return 1
  else:
    return quibble(n-1) + quibble(n-1)
```

$O(2^n)$



Fibonacci

```python
def fibonacci(n):
  if n < 2:
    return n
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```

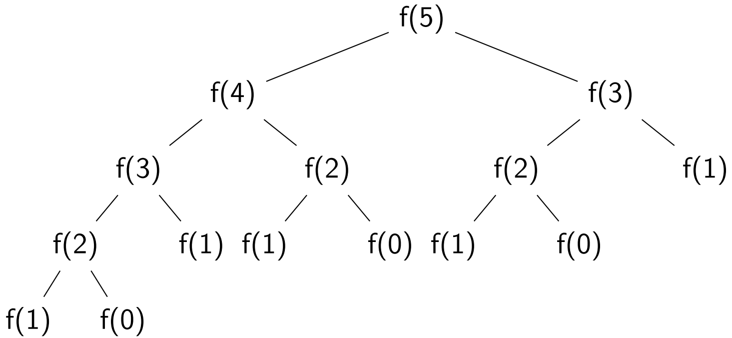
The mathematics is difficult, but the result is similar to `quibble`:

$$O(1.7)^n$$

Visualizing Fibonacci

```python
def fibonacci(n):
  if n < 2:
    return n
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```

$$O(1.7^n)$$



How many times can you divide an integer in half?

$$30 \to 15 \to 7 \to 3 \to 1 : 4$$
$$32 \to 16 \to 8 \to 4 \to 2 \to 1 : 5$$
$$40 \to 20 \to 10 \to 5 \to 2 \to 1 : 5$$
$$50 \to 25 \to 12 \to 6 \to 3 \to 1 : 5$$
$$64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1 : 6$$
$$70 \to 35 \to 17 \to 8 \to 4 \to 2 \to 1 : 6$$
$$128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1 : 7$$
$$200 \to 100 \to 50 \to 25 \to 12 \to 6 \to 3 \to 1 : 7$$
$$256 \to 128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1 : 8$$

How many times can you divide an integer in half?

$$30 \to 15 \to 7 \to 3 \to 1 : 4$$
$$32 \to 16 \to 8 \to 4 \to 2 \to 1 : 5$$
$$40 \to 20 \to 10 \to 5 \to 2 \to 1 : 5$$
$$50 \to 25 \to 12 \to 6 \to 3 \to 1 : 5$$
$$64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1 : 6$$
$$70 \to 35 \to 17 \to 8 \to 4 \to 2 \to 1 : 6$$
$$128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1 : 7$$
$$200 \to 100 \to 50 \to 25 \to 12 \to 6 \to 3 \to 1 : 7$$
$$256 \to 128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1 : 8$$

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

Algorithm Analysis

How many additions does `shoop` make at a maximum?

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

$$
\begin{aligned}
f(1) &= 0 \\
f(n) &= f(n/2) + 1 \\
&= (f(n/4) + 1) + 1 \\
&= ((f(n/8) + 1) + 1) + 1 \\
&= f(n/2^k) + k \\
&= f(n/2^{\log_2(n)}) + \log_2(n) \\
&= f(n/n) + \log_2(n) \\
&= \log_2(n) = O(\log n)
\end{aligned}
$$

Algorithm Analysis

How many additions does `snarfle` make at a maximum?

```
def snarfle(n):
  if n == 1:
    return 7
  else:
    return snarfle(n//2) + snarfle(n//2)
```

$$
\begin{aligned}
f(1) &= 0 \\
f(n) &= 2f(n/2) + 1 \\
&= 2(2f(n/4) + 1) + 1 \\
&= 2^2 f(n/2^2) + 2 + 1 \\
&= 2^3 f(n/2^3) + 4 + 2 + 1 \\
&= 2^k f(n/2^k) + \sum_{i=0}^{k-1} 2^i
\end{aligned}
$$

Algorithm Analysis

How many additions does `snarfle` make at a maximum?

```
def snarfle(n):
  if n == 1:
    return 7
  else:
    return snarfle(n//2) + snarfle(n//2)
```

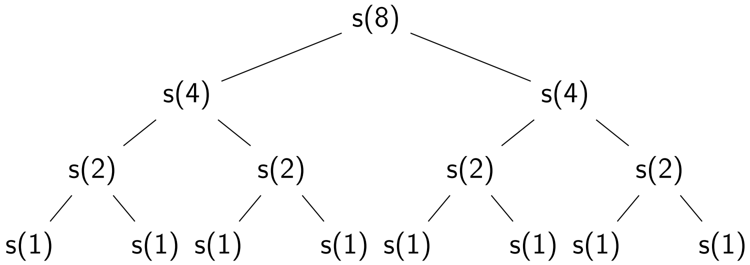
$$
\begin{aligned}
f(1) &= 0 \\
f(n) &= 2f(n/2) + 1 \\
&= 2(2f(n/4) + 1) + 1 \\
&= 2^2 f(n/2^2) + 2 + 1 \\
&= 2^3 f(n/2^3) + 4 + 2 + 1 \\
&= 2^k f(n/2^k) + \sum_{i=0}^{k-1} 2^i
\end{aligned}
$$

$$
\begin{aligned}
&= \dots \\
&= 2^{\log_2(n)} f(n/2^{\log_2(n)}) + \sum_{i=0}^{\log_2(n)-1} 2^i \\
&= 0 + (2^{\log_2(n)-1+1} - 1)/(2-1) \\
&= 2^{\log_2(n)} - 1 \\
&= n - 1 = O(n)
\end{aligned}
$$

Visualizing `snarfle`

```python
def snarfle(n):
  if n == 1:
    return 7
  else:
    return snarfle(n//2) + snarfle(n//2)
```

$O(n)$



Two algorithms for power

$a^b = a(a^{b-1})$ vs. $a^b = (a^{b/2})^2$

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

```
def powlog(a, b):
  if b == 0:
    return 1
  elif b % 2 == 1:
    return a * powlog(a, b-1)
  else:
    x = powlog(a, b//2)
    return x * x
```

What are their running times?

Fibonacci revisited

```python
def fibonacci(n):
  if n < 2:
    return n
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```

Running time?

Fibonacci revisited

```python
def fibonacci(n):
  if n < 2:
    return n
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```

Running time? $O(1.7^n)$

```python
def fibonacci(n):
  a,b = 0,1
  while n > 0:
    n,a,b = n-1, b, a+b
  return a
```

Runining time?

Fibonacci revisited

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

Running time? $O(1.7^n)$

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

Runining time? $O(n)$

Fibonacci revisited

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

Running time? $O(1.7^n)$

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

Runining time? $O(n)$ $1.7^{100} > 10^{23} \gg 100$

Fibonacci revisited

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

Running time? $O(1.7^n)$

```
def fibonacci(n):
  a,b = 0,1
  while n > 0:
    n,a,b = n-1, b, a+b
  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:

```
Linear: 8093750000
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

```python
def anagramSolution1(s1,s2):
    stillOK = True
    if len(s1) != len(s2):
        stillOK = False
    alist = list(s2)
    pos1 = 0
    while pos1 < len(s1) and stillOK:
        pos2 = 0
        found = False
        while pos2 < len(alist) and not found:
            if s1[pos1] == alist[pos2]:
                found = True
            else:
                pos2 = pos2 + 1
        if found:
            alist[pos2] = None
        else:
            stillOK = False
        pos1 = pos1 + 1
    return stillOK
```

Anagram algorithm 2: sort and compare

```python
def anagramSolution2(s1,s2):
    alist1 = list(s1)
    alist2 = list(s2)

    alist1.sort()
    alist2.sort()

    pos = 0
    matches = True

    while pos < len(s1) and matches:
        if alist1[pos]==alist2[pos]:
            pos = pos + 1
        else:
            matches = False

    return matches
```

Anagram algorithm 3: brute force

```
def anagramSolution3(s1, s2):
    return s1 in all_permutations(s2)

def all_permutations(s):
    if len(s) == 1:
        return [s]
    else:
        shorts = all_permutations(s[1:])
        longs = []
        for short in shorts:
            longs = longs + all_positions(s[0], short)
        return longs

def all_positions(c, s):
    strings = []
    for i in range(len(s)+1):
        strings.append(s[0:i] + c + s[i:])
    return strings
```

Anagram algorithm 4: count and compare

```
c1 = [0]*26
c2 = [0]*26

for i in range(len(s1)):
    pos = ord(s1[i])-ord('a')
    c1[pos] = c1[pos] + 1

for i in range(len(s2)):
    pos = ord(s2[i])-ord('a')
    c2[pos] = c2[pos] + 1

j = 0
stillOK = True
while j<26 and stillOK:
    if c1[j]==c2[j]:
        j = j + 1
    else:
        stillOK = False

return stillOK
```

Runtime of Builtin Python Data Structures

- See text.