# CS 61A Discussion 7

## Orders of Growth

Raymond Chan
Discussion 121
UC Berkeley

# Agenda

- Quiz
- Announcements
- Orders of Growth

# Quiz - WWPP

```
def f(l, n):
    s = '< '
    while n > 0 and l != Link.empty:
        s += str(l.first) + ' '
        l = l.rest
        n -= 1
    print(s + '>')

link = Link(1, Link(2, Link(3, Link(4, Link(5, Link(6))))))
linkA = link.rest
linkB = link.rest.rest
linkC = link.rest.rest.rest.rest

link.rest.rest, linkB.rest = linkB.rest, link.rest.rest
link.rest.rest.rest = linkC.rest.rest
linkC.rest.rest = linkC
link.rest = linkC.rest
linkC.rest = link
f(link, 5)
f(linkA, 5)
f(linkB, 5)
f(linkC, 5)
```

Credit goes to Kristin

# Quiz Solutions

- The function f prints out the first 5 elements of a linked list.
- Try drawing out the linked list.

```
< 1 6 5 1 6 >
< 2 4 >
< 3 3 3 3 3 >
< 5 1 6 5 1>
```

# Announcements

- HW 5 due Wednesday 3/16
- Ants Project due Thursday 3/17
- Midterm 2 Wednesday 3/30 (after spring break)
- Submit Midterm 2 Alternative Petition by 3/18

# Orders of Growth

- When we have really large inputs, we need to worry about **efficiency**.
- We measure efficiency by runtime (Time complexity).
- How long does the functions take to run in terms of the size of the input?
- If the size of the input grows, how does the runtime change?

# Orders of Growth

- We use Big-O notation means an upper bound on time complexity.
- $O(n^2)$ means that the function’s runtime is **no larger than quadratic** of the input.

# Orders of Growth

```
def square(n):
  return n * n
```

1 primitive operation *

# Orders of Growth

```
def square(n):
  return n * n
```

1 primitive operation *

| input | function call | number of | number of operations |
|---|---|---|---|
| 1 | square(1) | 1*1 | 1 |
| 2 | square(2) | 2*2 | 1 |
| … | … | … | … |
| 100 | square(100) | 100*100 | 1 |
| … | … | … | … |
| n | square(n) | n*n | 1 |

# Orders of Growth

```
def square(n):
  return n * n
```

1 primitive operation *

**O(1)**

| input | function call | return value | number of operations |
|---|---|---|---|
| 1 | square(1) | 1*1 | 1 |
| 2 | square(2) | 2*2 | 1 |
| … | … | … | … |
| 100 | square(100) | 100*100 | 1 |
| … | … | … | … |
| n | square(n) | n*n | 1 |

# Orders of Growth

```
def factorial(n):
  if n == 0:
    return 1
  return n * factorial(n - 1)
```

Each recursive call has
a constant amount operations.

# Orders of Growth

```
def factorial(n):
  if n == 0:
    return 1
  return n * factorial(n - 1)
```

Each recursive call has
a constant amount operations.

| input | function call | return value | number of operations |
|---|---|---|---|
| 1 | factorial(1) | 1*1 | 1 |
| 2 | factorial(2) | 2*1*1 | 2 |
| … | … | … | … |
| 100 | factorial(100) | 100*99*…*1*1 | 100 |
| … | … | … | … |
| n | factorial(n) | n*(n-1)*…*1*1 | n |

# Orders of Growth

```
def factorial(n):
  if n == 0:
    return 1
  return n * factorial(n - 1)
```

Each recursive call has
a constant amount operations.

**O(n)**

| input | function call | return value | number of operations |
|---|---|---|---|
| 1 | factorial(1) | 1*1 | 1 |
| 2 | factorial(2) | 2*1*1 | 2 |
| … | … | … | … |
| 100 | factorial(100) | 100*99*…*1*1 | 100 |
| … | … | … | … |
| n | factorial(n) | n*(n-1)*…*1*1 | n |

# Orders of Growth

- $O(1)$ - constant time; same time regardless of input size.
- $O(\log n)$ - logarithmic time; e.g. usually dividing the problem down by some factor.
- $O(n)$ - linear time; e.g. usually 1 loop
- $O(n^2)$, $O(n^3)$, etc - polynomial time; e.g. nested loops
- $O(2^n)$ - exponential time; can change 2 to some other constant; really horrible time complexity; e.g. tree recursion

# Orders of Growth

- Constant time is the best and exponential is the worse.
- Any polynomial is worse than any logarithmic.
- Higher degree polynomial worse than lower degree.

# Orders of Growth

- Since we care about the runtime when ***n*** gets infinitely large, we can **drop** lower order terms and constants.
    - $O(2n^3 + 6n + \log(n)) = O(n^3)$
- Should always provide the tightest bound.
    - Factorial is $O(n^2)$ and $O(n)$. But the tightest bound is $O(n)$.

# Orders of Growth

- Count the number of iterations and/or recursive calls.
- Find the number of operations per iteration or recursive call.

# Recap

- Orders of growth tells us how long the running time of the function approach as n approach infinity.
- Constant is better than logarithmic, which is better than polynomial, which is better than exponential.
- Lower polynomial is better than higher polynomial.
- Try drawing a call stack or tree to count the # of operations.