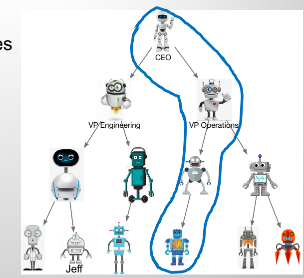
Getting a grip on recursion

"To understand recursion, one must first understand recursion"

Terence Parr
MSDS program
University of San Francisco

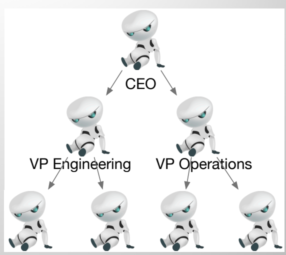
Recursion is just delegation

- Consider an org chart
- Bosses delegate to subordinates
- CEO wants work done:
  - sends emails to 2 VPs
  - who launch 2 other workers
  - those workers launch emails too
  - until potentially all *n* contacted
- Max work is *n* workers times work each worker performs
- Trace from CEO to mailroom cost is only height of org chart: *log(n)* in typical case



Recursion is more like startup org chart

- One person has to play every role, cloning themselves
- Person "pauses" current work, performs subtask(s), then continues where they left off, possibly using subtask results
- Each call or email to worker is analogous to a function call
- (A large company could launch multiple core/employees)



Let's start with math recurrence relations

- Factorial definition:
  - Let 0! = 1
  - Define n! = n * (n-1)! for n>=1
- Recurrent math function calls become recursive function call in Python
- Non-recursive version is harder to understand and less natural
- This has linear time *O(n)*; more clear in non-recursive version

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

```
def factloop(n):
    r = 1
    for i in range(1,n+1):
        r *= i
    return r
```


Fibonacci sequence

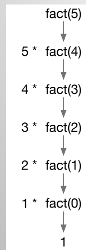
```
def fib(n):
    if n==0 or n==1: return 1
    return fib(n-1) + fib(n-2)
```

- Let F(0) = F(1) = 1
- Define F(n) = F(n-1) + F(n-2) for n>=2
- Recursive implementation is very natural but mucho inefficient!
- We do small bit of work to combine results of two subproblems, and each subproblem is only 1 and 2 elements smaller
- Compare this to fact(n) where we do small bit of work using result of a **single** subproblem, which has linear time

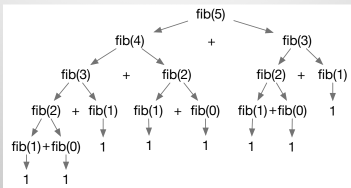
How fast is fib(n)?

- Seems like it should linear
- Computation of fib(n-1) and fib(n-2) overlap, repeating same computations
- Solving and combining results from two similarly-sized subproblems yields exponential complexity, O(k^n)
- fib(30) takes 0.5s
- fib(36) takes 9.0s
- fib(37) takes 16.7s
- fact(n) invokes a single similarly-sized subproblem so is linear

Compare fact, fib call trees visually



Difference comes down to two versus one subproblem!
Key idea: recursion traces out a tree of function calls, meaning recursion is most natural way to walk trees



An aside: Trade memory for speed

- Use *dynamic programming* to solve Fibonacci in O(n) not O(k^n)

```
def cachefib(n):
    F = [0 for i in range(n+1)]
    F[0] = F[1] = 1
    for i in range(2,n+1): # work up not down
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

- cachefib(1000) take 0.3s compared to fib(37) at 16.7s (wow)
- Algorithmic changes matter much more than code optimizations

Summary: formula for recursive functions

def f(input):
1. check termination condition
2. process the active input region / current node, etc…
3. invoke f on subregion(s)
4. combine and return results

Steps 2 and 4 are optional

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

```
def fib(n):
    if n==0 or n==1: return 1
    return fib(n-1) + fib(n-2)
```

Terminology: *currently-active region* is what function is currently trying to process. Here, that is argument n (the "region" is the numbers 0..n)

Recursion at its finest: Divide and conquer

- The big idea: break a big problem down into smaller subproblems via recursion until subproblems are so small we can solve in constant time; then, merge partial results in linear time as you climb back up the recursive calls
- Examples: merge sort, quick sort, decision tree construction
- Recursion is easiest way to describe algorithms that break problems into multiple subproblems
- In contrast, algorithms that use a single recursive call are easy to convert to loops

The nature of divide and conquer alg's

- Divide and conquer algorithms make 2 or more recursive calls where each subproblem is a **fraction** of the size of the currently active problem
- (Binary search splits problem in half each step but it descends into just one half with one recursive call not two and it doesn't merge results; technically, not divide and conquer…just divide)
- The cost of any recursive algorithms depends on:
  - the number of recursive subproblem calls per active region
  - the size of the subproblems; for example, n-1 or n/2 for active size n?
  - the work required for each active region

Example:Merge sort O(n log n)

- The idea is to split the currently active region in half, sorting both the left and right subregions, then merge two sorted pieces
- Eventually, the regions are so small we can sort in constant time; i.e., sorting 2 nums is easy
- Merging two sorted lists can be done in linear time

https://www.geeksforgeeks.org/merge-sort/