

# **Getting a grip on recursion**

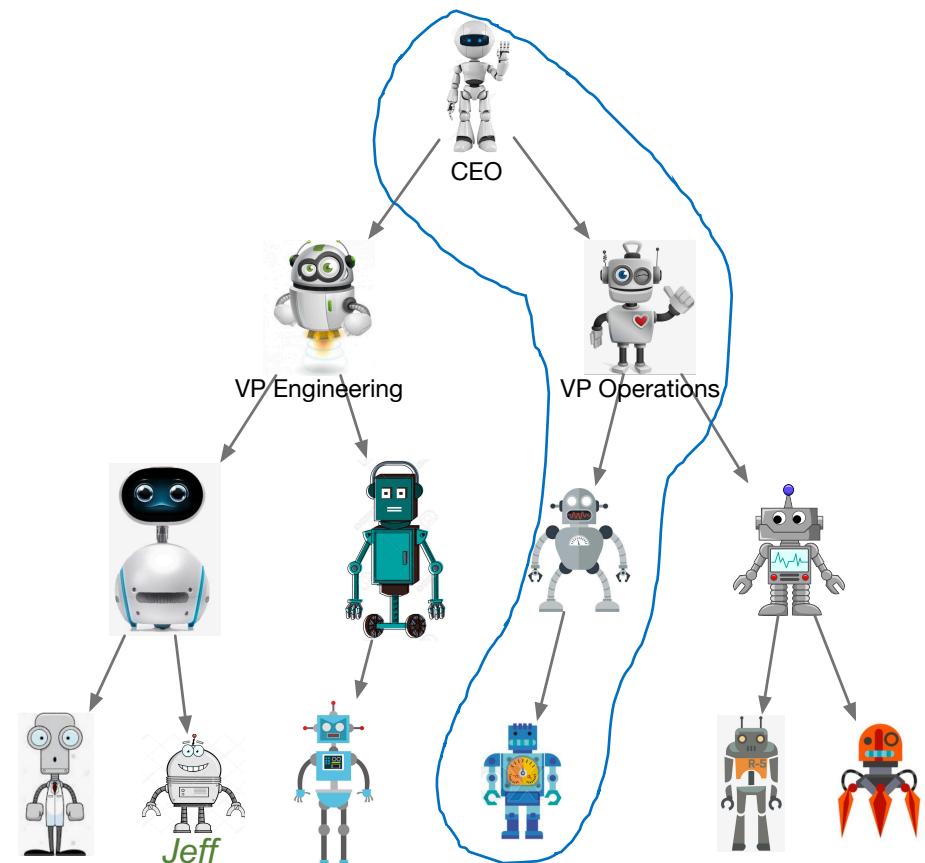
**“To understand recursion, one must first understand recursion”**

Terence Parr  
MSDS program  
**University of San Francisco**

Some of this recursion material is **review** from MSDS621, but we'll look much more deeply at recursion this time and consider the complexity analysis.

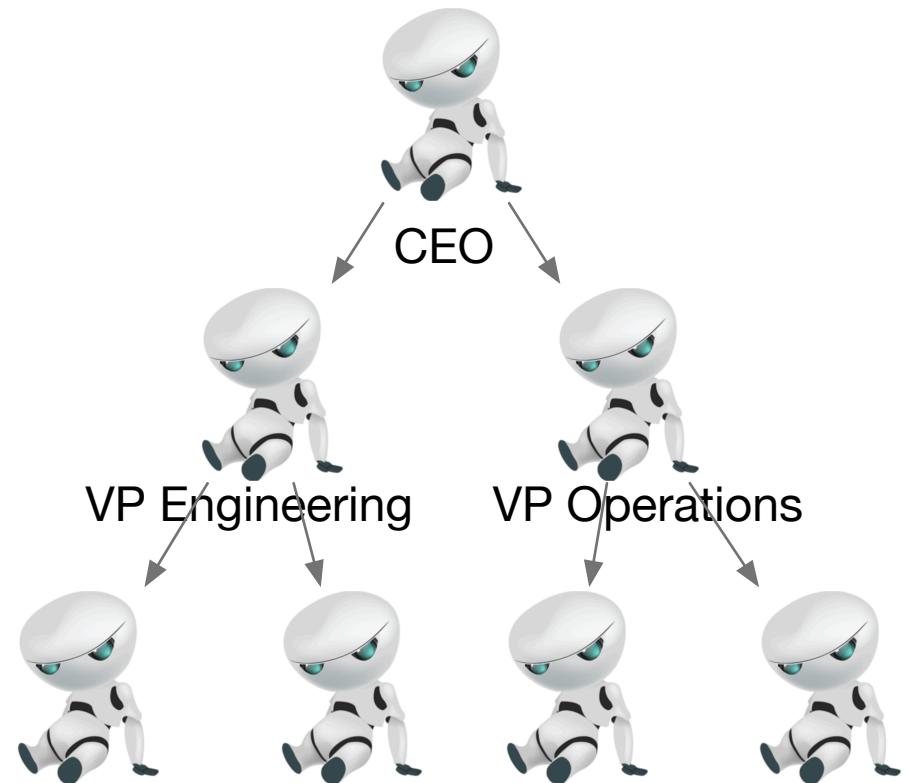
# Recursion is just delegation

- 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 just height of org chart:  $\log(n)$  in typical case
- Example: phone tree where each person calls  $k$  other people



# 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
- “Recursive leap of faith”
- You can pretend that you’re calling a different function
- (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  $n \geq 1$
- Recurrent math function calls become recursive function call in Python
- Non-recursive version is harder to understand and less natural
- This has linear time  $O(n)$ ;  $T(n) = k + T(n-1) = kn$ ; more clear in non-recursive version
- **Ex:** show it's  $O(n)$

```
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

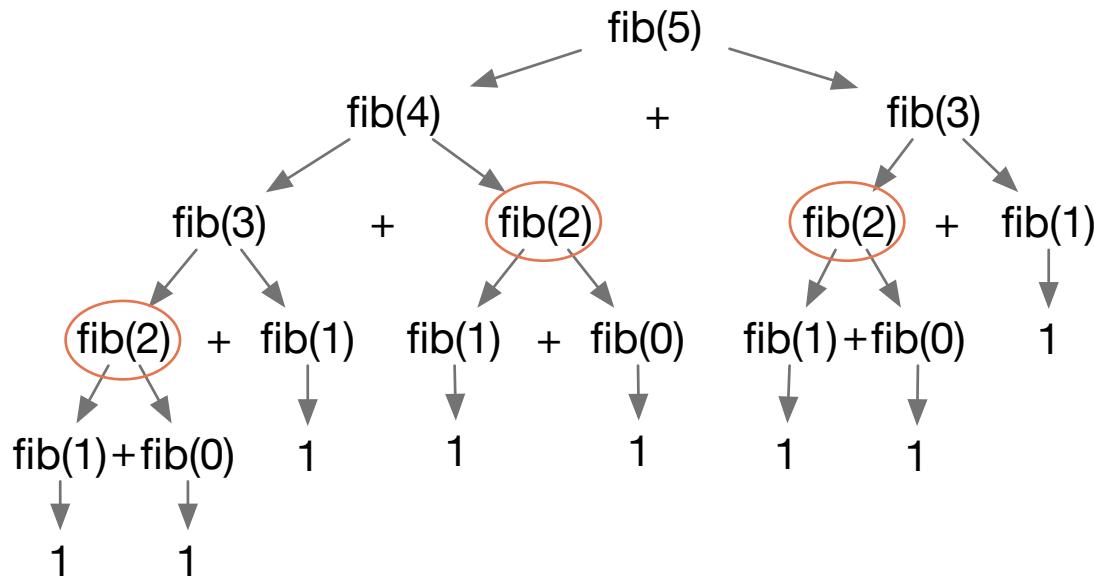
```
def fib(n):  
    if n==0 or n==1: return 1  
    return fib(n-1) + fib(n-2)
```

- Let  $F(0) = F(1) = 1$
- Define  $F(n) = F(n-1) + F(n-2)$  for  $n \geq 2$
- 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  $\text{fact}(n)$  where we do small bit of work using result of a **single** subproblem, which has linear time

# Compare fact, fib call trees visually

```
fact(5)
↓
5 * fact(4)
↓
4 * fact(3)
↓
3 * fact(2)
↓
2 * fact(1)
↓
1 * fact(0)
↓
1
```

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



# How fast is fib(n)?

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

- Seems like it should linear, right?
- BUT, computation of fib(n-1) and fib(n-2) overlap, repeating same redundant computations
- Solving and combining results from two similarly-sized subproblems yields exponential complexity,  $O(k^n)$
- fib(30) takes 0.5s on my fast mac
- fib(36) takes 9.0s
- fib(37) takes 16.7s

# Exercise

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

- What is  $T(n)$  for  $\text{fib}()$ ? Try to show it's exponential.
- $T(n) = k + T(n-1) + T(n-2)$
- $T(n) = k + (k + T(n-2) + T(n-3)) + (k + T(n-3) + T(n-4))$
- $T(n) = k + (k + (k + T(n-3) + T(n-4)) + (k + T(n-4) + T(n-5))) + (k + (k + T(n-4) + T(n-5)) + (k + T(n-5) + T(n-6)))$
- $T(n) = 7k + T(n-3) + 3T(n-4) + 3T(n-5) + T(n-6)$
- Yikes! I don't see the pattern, do you?
- Intuition: recursion tree is bushy; height  $n \Rightarrow 2^n$  nodes

# Proof by “complete induction”

- Try  $T(n) = 2^n$ ; it's a good bet given the explosion of computations then prove by induction
- Complete induction: Assume  $T(i)=2^i$  holds for all  $i < n$
- Observe that we can rearrange:  $T(n+2) = k + T(n+1) + T(n)$
- Complete induction assumption lets us directly substitute:
- $T(n+2) = k + 2^{n+1} + 2^n = k + 2^1 \cdot 2^n + 2^n = k + 2^n(2 + 1)$
- So  $T(n) = k + 3 \cdot 2^n$ , which is  $O(2^n)$

# Summary: formula for recursive functions

(review from MSDS621)

```
def f(input):
```

1. check termination condition
2. process the active input region / current node, etc...
3. invoke f on subregion or subregions
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* or *element* 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, closest pair of (x,y) points\*
- 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, and loops are usually more efficient

\*See [https://en.wikipedia.org/wiki/Closest\\_pair\\_of\\_points\\_problem](https://en.wikipedia.org/wiki/Closest_pair_of_points_problem)

# 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 algorithm 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

# Merge sort

## $O(n \log n)$

- The idea is to split currently active region in half, sorting both the left and right subregions, then merge two sorted subregions
- 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://youtu.be/XaqR3G\\_NVoo](https://youtu.be/XaqR3G_NVoo)

# Algorithm is simple but efficient ( $n \log n$ )

```
def msort(A):
    if len(A)==1: # sort one element in O(1)
        return A
    if len(A)==2: # sort two elements in O(1)
        return A if A[0]<A[1] else [A[1],A[0]]
    mid = int(len(A)/2)
    left = msort(A[:mid])
    right = msort(A[mid:])
    return sorted(left + right)
```

- “sorted(left + right)” is cheating but used for simplicity here; merging two sorted lists is  $O(n)$
- (Note: constant on this is huge)

# Exercise: write pow2(n)

- Compute  $2^n$  recursively by multiplying 2 together n times

```
def pow2(n):
    if n==0: return 1
    return 2 * pow2(n-1)
```

- Now do iterative version

```
def pow2(n):
    v = 1
    for i in range(n): v *= 2
    return v
```

- **Ex:** What is the complexity of this?

$$\begin{aligned}T(n) &= 1 + T(n-1) \\&= 1 + 1 + 1 + T(n-3) = n*1 + 1 = n+1 \Rightarrow O(n)\end{aligned}$$

# Can we do pow2 more efficiently?

- Divide and conquer; hint:  $2^n = 2^{\frac{n}{2}}2^{\frac{n}{2}}$  if n is even, else  $2 * 2^{\lfloor \frac{n}{2} \rfloor}$

```
def pow2(n):  
    if n==0: return 1  
    half = pow2(n//2)  
    if n % 2==0: # if even  
        return half * half  
    return 2 * half * half
```

- A non-recursive version would be awkward and hard to write!
- **Ex:** What is the complexity of this version?

$$\begin{aligned} T(n) &= k + T(n/2) \\ &= k + k + T(n/4) = k + k + k + T(n/8) = k \log_2(n) \Rightarrow O(\log n) \end{aligned}$$



UNIVERSITY OF SAN FRANCISCO

# Recursion summary

- Use recursion when implementing a recurrence relation (not always for efficiency reasons, such as gradient descent)
- Use recursion when you can break problem down into subproblems whose results can be merged to solve overall problem; divide and conquer
- Use recursion when the same operation and navigation procedure applies to any element of a data structure; binary search, walking trees and graphs; self-similar structures
- Recursion traces out a function call tree

# An Aside: Caching partial results with dynamic programming

# *Dynamic programming trades memory for speed*

- We can often decompose a problem into subproblems and combine the partial results to form the overall result
- If the same computation is repeated, it's a candidate for dynamic programming
- Dynamic programming is a terrible, meaningless name that simply means caching or *memoizing* partial results and using those to avoid redundant computations
- Idea: compute “1+1+1+1”. That’s 4. If I append “+1”, what is sum?
- You reused the 4 result hopefully; that’s dynamic programming

# Example: Use *dynamic programming* to solve Fibonacci in $O(n)$ not $O(k^n)$

- Make cache F of size n, working upwards, fill in F

```
def cachefib(n):
    F = [0 for i in range(n+1)]
    F[0] = F[1] = 1          # init cache
    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)
- Could even get away with saving 2 previous values I think
- **Algorithmic changes matter much more than code optimizations!!!**

# Example: use cache with recursive fib()

- The recursive version is more natural; to make efficient record partial results and look in cache before computing anything

```
def memofib(n, memo = {}):  
    if n==0 or n==1: return 1  
    # return if already computed  
    if n in memo: return memo[n]  
    f = memofib(n-1, memo) + memofib(n-2, memo)  
    memo[n] = f # memoize result  
    return f
```

(I pass memo around as arg to avoid a global variable)

# Memoized recursive function template

```
def f(input):
```

1. check termination condition as usual
2. **return memoized result if available**
3. process the active input region / current node, etc...
4. invoke f on subregion(s)
5. combine results and **memoize**
6. return results

Blue regions differ from normal recursive function template

# Check out the recursion notebook

<https://github.com/parrt/msds689/blob/master/notes/recursion-notebook.ipynb>