Math 260: Python programming in math

Fall 2020

Data structures, part I:
Recursion, stacks and queues

Recursion

Recursion

- **Recursion** refers to a function calling itself
- A **recursive algorithm** has (a simpler case of) itself as a step
  - We always need 'base cases' that are resolved explicitly
- So how does this work in code?

Recursive code puts function calls on the (call) stack:

```
def fact(n):
    if n == 0:  # base case
        return 1 # (*)
    elif n > 0:
        return n*fact(n-1)
    else:
        raise(ValueError('oh no!'))

a = fact(3) # **
```

Call stack at the base case (*):
(most recent at bottom)

- `fact(3)`
- `fact(2)`
- `fact(1)`
- `fact(0)`
- `return 1`

- Functions resolve in 'last in, first out' (LIFO) order
- `fact(0)` finishes, then `fact(1)`, ...
- computing $n!$ puts $n + 1$ calls on the stack **at once**

Recursion: good and bad

Was that really necessary? No - recursion is excessive here.

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

Call stack:

- `fact(n)`
- ⋮
- `fact(0)`

```
def fact(n):
    f = 1
    for k in range(n+1):
        f *= k
    return f
```

Call stack:

- `fact(n)`
  - (for loop)

Both are different implementations of the same algorithm!

Key point: **Recursion isn't free**

- The call stack takes memory/time to set up
- This is called 'overhead' (costs not part of the algorithm steps)
- Stack size is limited! `fact(3000)` → `stack size exceeded!`

Recursion: good and bad

- Common usage:
  - Derive a recursive **algorithm** (elegant)
  - Code an equivalent but **non-recursive implementation**
- Algorithms can often be nicely expressed recursively (e.g. binary search...)
- Good recursion (implementation) - less common:
  - 'naturally' recursive (can't easily be made non-recursive)
  - or where the function overhead is not relevant
  - Stack limits mean recursive algorithms don't scale up well

Example (bad): overlap can make a recursive algorithm inefficient:

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

How many function calls are made here?

Answer: 2^n - compared to the $O(n)$ steps it should take.

Recursion

A naturally recursive type of algorithm is 'divide and conquer':

- To compute something over a set:
- Break the set into (disjoint) pieces...
  - then compute for each piece,
  - ...combine and return the result

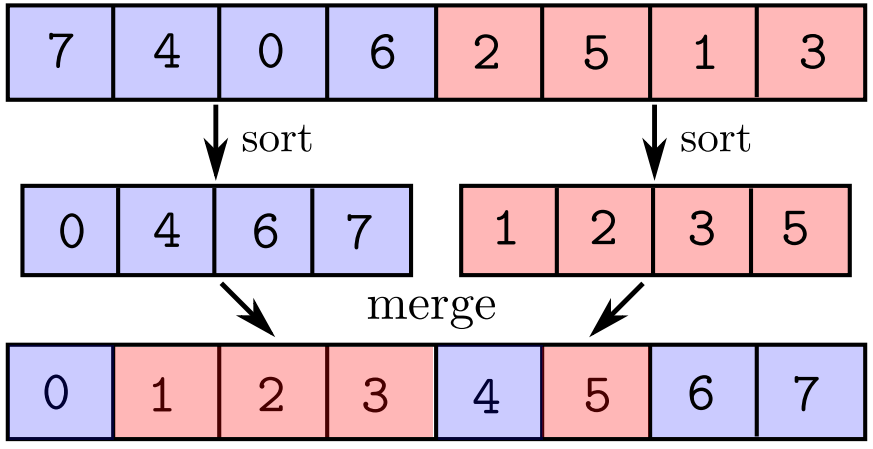
Example: sorting a list. Suppose `a` is a length `n` list of numbers:

a) Split the list into two $n/2$ sized lists

b) Sort each half-list

c) Combine the sorted half-lists into one sorted list

This is **mergesort**.



Recursion: mergesort

Mergesort:

a) Split the list into two $n/2$ sized lists

b) Sort each half-list

c) Combine the sorted half-lists into one sorted list

implementation (sketch):

```
def mergesort(j, k, arr, work):

    if j==k:  # base case
        return

    m = (j + k)//2
    mergesort(j, m, arr, work)
    mergesort(m+1, k, arr, work)

    # now [j,m] and [m+1,k] are sorted
    # merge them together...
    # (use work for temp space)
```

- This sorts the sub-list from $[j, k]$ (inclusive).
- Note that `work` is passed: shared space for calculations

Recursion: mergesort

Mergesort:

a) Split the list into two $n/2$ sized lists

b) Sort each half-list

c) Combine the sorted half-lists into one sorted list

How efficient is mergesort?

- Let $M(n)$ be the work required to run mergesort on a length n list
- Step (b) requires $2M(n/2)$ work
- Step (a) is trivial and (c) requires cn work (exercise)

Assume $n = 2^k$ is a power of 2 for simplicity. Then

$$\begin{aligned} M(n) &= 2M(n/2) + cn \\ &= 2(2M(n/4) + c(n/2)) + cn \\ &= \cdots \\ &= cn + 2c\frac{n}{2} + 4c\frac{n}{4} + \cdots \end{aligned}$$

$\implies$ $M(n) \approx kcn$ so $O(n \log n)$ work is required.

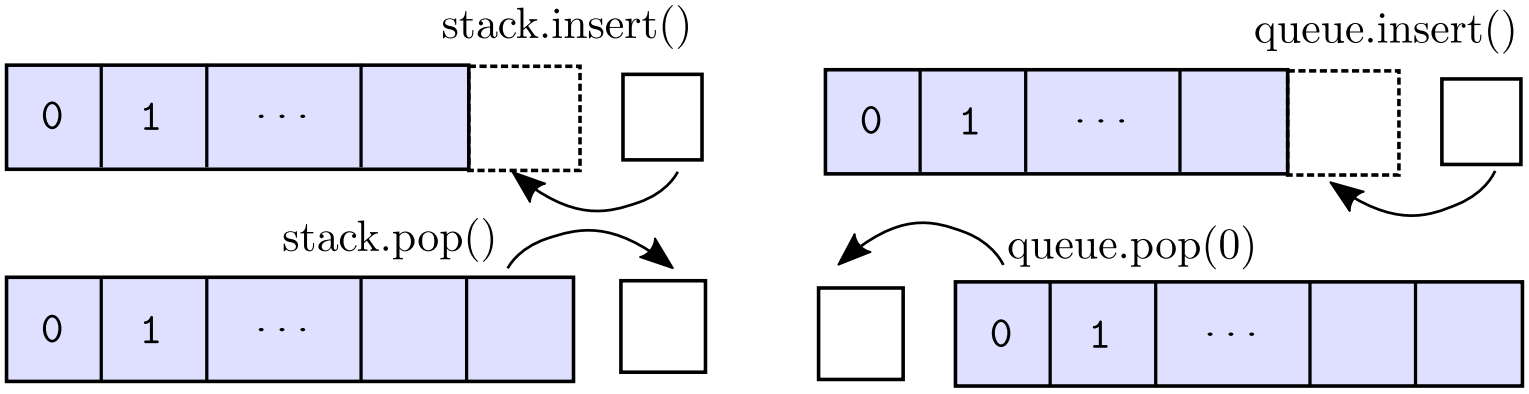
(Aside: the $O(n \log n)$ is essentially optimal for Big-O. The popular algorithm with this Big-O is **quicksort**, which is a bit faster than mergesort.)

Data structures: stacks and queues

Stacks and queues

The simplest data structures are **stacks** and **queues**. For both:

- A container with ordered data (a 'list', in the English sense of the word)
- Two operations are available:
  - An **insert** function that adds an item to the container
  - A **pop** function that removes an item (by some rule)



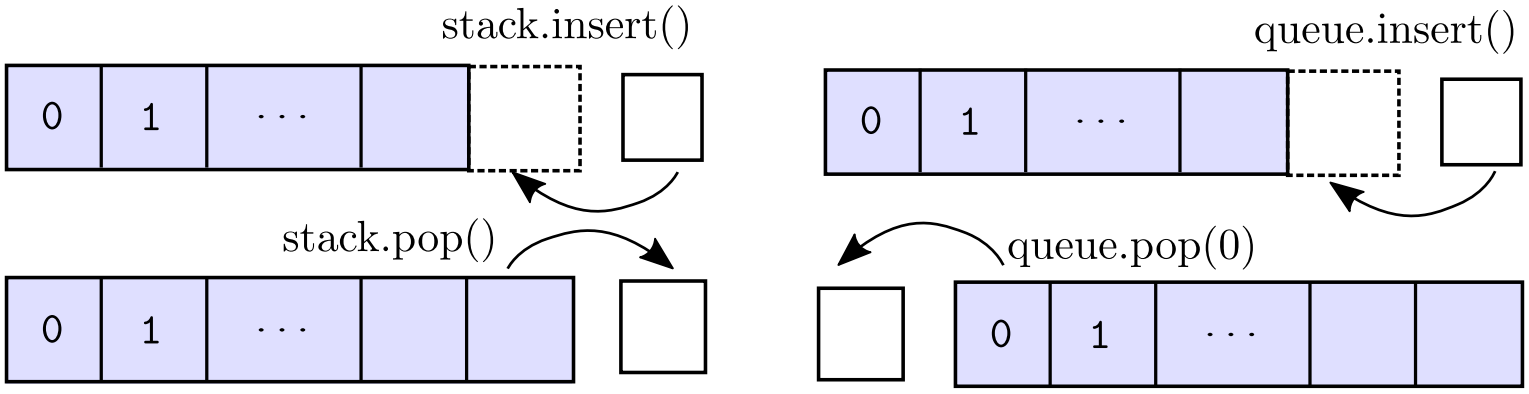
The differences:

- A **stack** is a **first in, first out** (FIFO) container:
  - items are inserted to and popped from the top of the stack
- A **queue** is a **first in, last out** (LIFO) container
  - items are inserted to the top, removed from the bottom

Stacks and queues

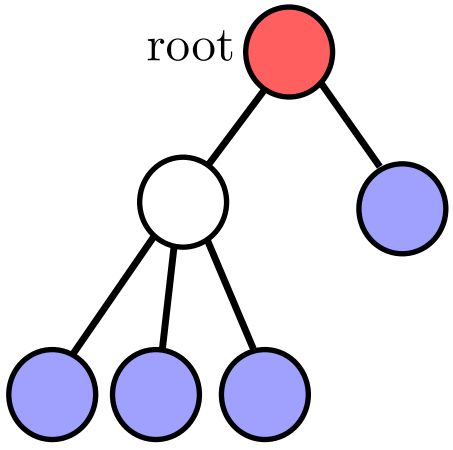
Python lists have commands to do this (so we don't need a new class):

- `a.pop()` pops from the end; `a.pop(0)` from the start
- `a.append()` adds to the end, `a.insert(0)` inserts at the start



- We'll use `append` for simplicity; insert/pop can both be implemented to be $O(1)$ operations (very efficient) for stacks and queues.

Aside: trees



A useful structure is a **tree**:

- a graph consisting of nodes connected by edges
- Has the tree property: contains no cycles (closed paths from a node to itself)

We can build a simple tree in python with a `Node` class

- It contains data for that node (e.g. names in a family tree)
- It has a list `children` of nodes below it in the tree
- A tree is a **root** node with children, which have children, etc.
- A node with no children: '**leaf node**'

Node class:

```
class Node:

    def __init__(self, data, children):
        self.data = data
        self.children = children
    # ... and other features ...
```

The tree above, with k^2's as data:

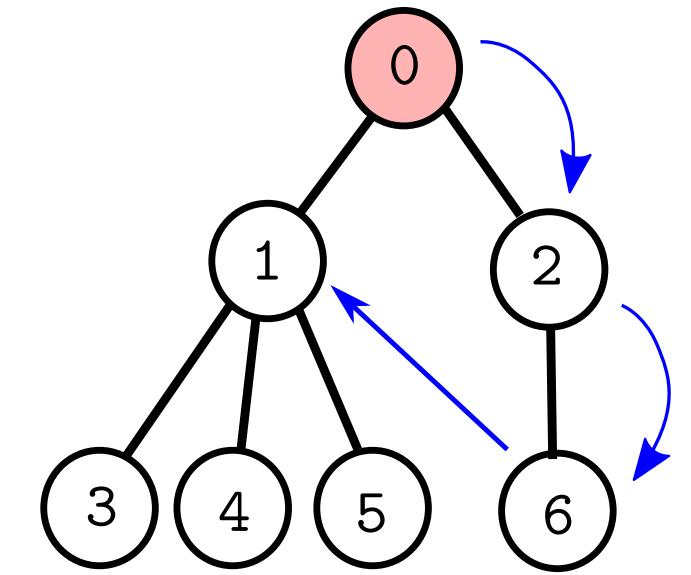
```
n = [Node(k**2, []) for k in range(7)]
n[0].children = [n[1], n[2]]
n[1].children = n[3:6]
n[2].children = n[6]
```

Recursion and stacks: searching trees

Now suppose we want to find a value `val` in a tree.
If found, we return a reference to the corresponding node. One approach:

- Check if the root has `val`
- If not, search its children recursively

Recursive algorithm:



This is called a **depth-first search** (DFS).

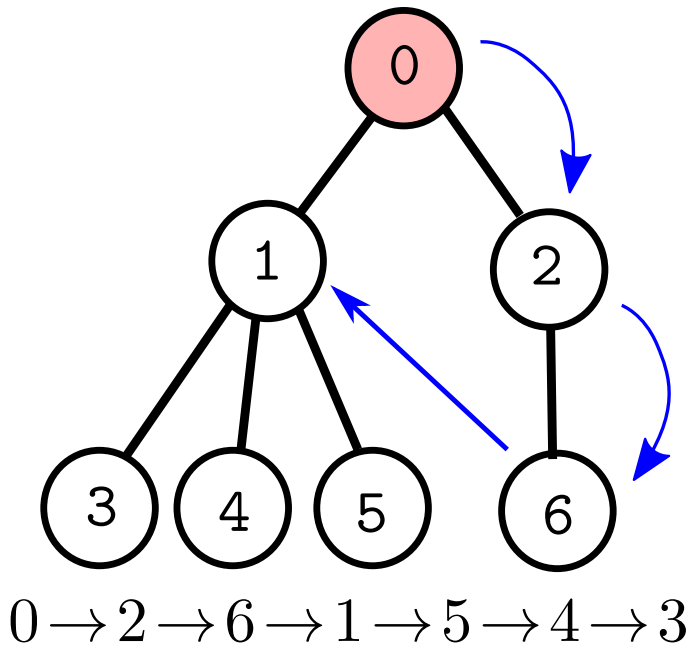
Recursion and stacks: searching trees

Now suppose we want to find a value `val` in a tree.
If found, we return a reference to the corresponding node. One approach:

- Check if the root has `val`
- If not, search its children recursively

Recursive algorithm:

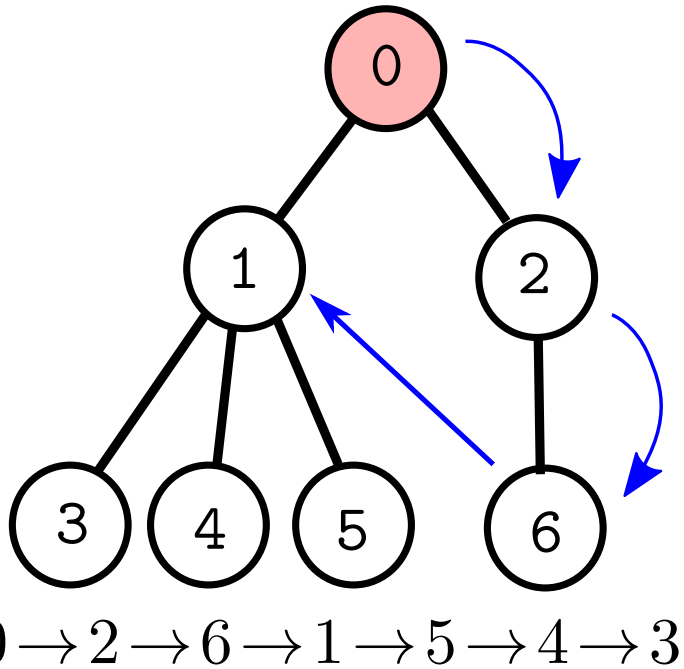
```
def search(val, root):
    if(root.data==val):
        return root

    for child in root.children:
        t = search(val, child)
        if t: #if t != None
            return t

    return None
```



This is called a **depth-first search** (DFS).

Depth-first search

At the start, initialize `stack` to contain node 0.
Replace recursive calls with 'add node to the stack'.



$0 \to 2 \to 6 \to 1 \to 5 \to 4 \to 3$

steps:

- Check 0, add $[1, 2]$ to stack
  `stack` is `[1,2]`
- Check 2, add 6 to stack
  `stack` is `[1,6]`
- Check 6 (no children!)
  `stack` is `[1]`
- Check 1, add $[3, 4, 5]$ to stack
  `stack` is `[3,4,5]`

This does the same thing as the recursive method!

Recursion and stacks: searching trees

Recursive algorithm:

```
def search(val, root):
    if(root.data==val):
        return root

    for child in root.children:
        t = search(val, child)
        if t: #if t != None
            return t

    return None
```

Using a stack:

```
def dfs(val, root):
    stack = [root]

    while stack:  # empty list is False
        node = stack.pop()
        if(node.data == val):
            return node

        stack.extend(node.children)

    return None
```

- The recursive method puts function calls on the (special) call stack
- The stack method uses a simpler stack of nodes to track un-searched nodes
- The explicit code is more efficient (but the same algorithm)!

Breadth-first search

Replacing the stack in DFS with a **queue** also works...

Using a stack:

```
def dfs(val, root):
    stack = [root]

    while stack:
        node = stack.pop()
        if(node.data == val):
            return node

        stack.extend(node.children)

    return None
```

Using a queue:

```
def bfs(val, root):
    q = [root]

    while q:
        node = q.pop(0)
        if(node.data == val):
            return node

        q.extend(node.children)

    return None
```

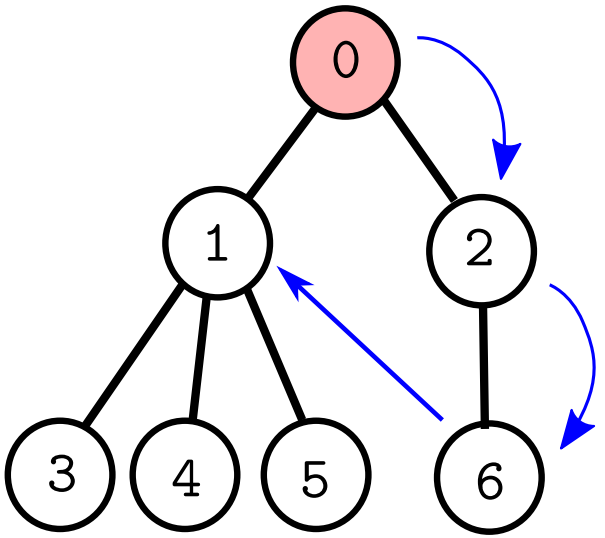
This time, the earliest nodes added are checked first! (closest to root).
We call this a **breadth-first search**.

(Aside: for a cyclic graph, more effort is required for both DFS and BFS to avoid going in circles!)

Breadth-first search

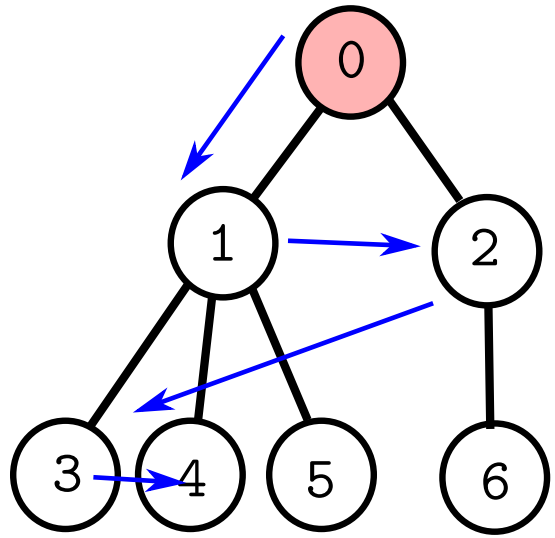
Example (comparing DFS and BFS):

Depth-first:



$0 \to 2 \to 6 \to 1 \to 5 \to 4 \to 3$

Breadth-first:



$0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6$

(Order may vary a bit by implementation)