

# DSA PYTHON MY PPT

Ananya Vemula 229310345

# WEEK 1

Basic GCD program:

```
def gcd(m,n):  
    fm = []  
    for i in range(1,m+1):  
        if(m%i) == 0:  
            fm.append(i)  
    fn = []  
    for j in range(1,n+1):  
        if (n%j) == 0:  
            fn.append(j)  
  
    cf = []  
    for f in fm:  
        if f in fn:  
            cf.append(f)  
    return(cf[-1])
```

## Simplified version

```
def gcd(m,n):  
    cf = []  
    for i in range(1, min(m,n) + 1):  
        if m%i == 0 and n%i == 0:  
            cf.append(i)  
  
    return(cf[-1])
```

MORE SIMPLIFIED VERSION:

```
def gcd(m,n):  
  
    for i in range(1,min(n,m)+1):  
        if m%i == 0 and n%i == 0:  
            mrcf = i  
  
    return(mrcf)
```

- MOST SIMPLIFIED VERSION(EUCLIDS ALGORITHM):

```
def gcd(m,n):
```

```
    if n>m:
```

```
        (m,n) = (n,m)
```

```
    while (m%n) != 0:
```

```
        (m,n) = (n,m%n)
```

```
    return(n)
```

- In python, we don't have to specify type. Int can become float and no error is thrown.
  - $A = 5$
  - $B = 7$
  - $C = 7/5$
  - $\text{Type}(C) = \text{float}$ ...while  $\text{type}(A)$  and  $\text{type}(B)$  are int only
- To find quotient, use `//` and not `/` (`/` will always give a float value) and modulo `'%'` for reminder
  - $\text{ie } 7/2 = 3.5$
  - $7//2 = 3$
  - $7\%2 = 1$
- You can import `log()` `sqrt()` `sin()` etc from math lib
  - Type down:- `from math import *`
- To find exponent, use `**`
  - $\text{ie, } 3^{**}4 = 81$
  - $2^{**}2 = 4$

- Bool values exist in python and they consist of “True” and “False”, not TRUE and FALSE or true and false

# WEEK 4

**MERGE SORT:-** Break it till you make it..... And rejoin

- Break the problem into disjoint parts
- Solve each part separately
- Combine the solutions effectively



## THE CODE FOR MERGING TWO SORTED LISTS (Of any length):

- `def merge(a,b):`
- `(c,m,n) = ([],len(a), len(b))`
- `(i,j) = (0,0)`
- `while i+j < m+n:`
- `if i == m:`
- `c.append(b[j])`
- `j = j + 1`
- `elif j == n:`
- `c.append(a[i])`
- `i = i + 1`
- `elif a[i] <= b[j]:`
- `c.append(a[i])`
- `i = i + 1`
- `elif a[i] > b[j]:`
- `c.append(b[j])`
- `j = j + 1`
- `return(c)`

# The code for actual sorting:

```
Def mergesort(A, left, right):  
    if right - left ≤ 1:  
        return(A[left:right])  
  
    if right - left > 1:  
        mid = (right + left)//2  
  
        L = mergesort(A, left, mid)  
        R = mergesort(A, mid, right)  
  
    return(merge(L, R))
```

# QUICK SORT CODE:

- `def Quicksort(A,l,r):`
- `if r-l <= 1:`
- `return()`
- `yellow = l + 1`
- `for green in range(l+1,r):`
- `if A[green] <= A[l]:`
- `(A[yellow], A[green]) = (A[green],A[yellow])`
- `yellow = yellow + 1`
- `(A[l], A[yellow - 1]) = (A[yellow -1], A[l])`
- `Quicksort(A,l,yellow - 1)`
- `Quicksort(A,yellow,r)`

# TUPLES

- There can be tuples inside a tuple.
- You can convert a list to a tuple, a set to a tuple etc.
- Tuples are immutable but you can access them and slice them and save in new variable.
- Represented using normal brackets
  - le:- `happy = (1,2,3,4,5,6,7,8,9,0)`
  - `print(happy[0:])` .....will execute
  - `happy[0] = 0`.....wont execute

# Dictionary

- Allows keys other than range(0,n)
- Key can be a string:
  - `le, test1["Dhawan"] = 84`
- Keys are immutable but values of the keys are mutable.
- Empty dictionary is `{}`
- Short form of dictionary is "dict"
- Dictionaries can be nester.
  - `le, Score["test1"]["dhoni"] = 84`
  - `Score["test2"]["kohli"] = 114`
  - `Score["test2"]["dhoni"] = 100`
- Or you can directly assign values to dictionaries:
  - `Score = {"Dhoni": 84, "Kohli": 100}`
  - `Score = {"test1" : {"Dhawan":84, "Kohli":100}, "test2":{"Kohli":50, "Dhoni":100}}`

- `d.keys()` returns the sequence of keys of dict `d` in random order:
  - for `k` in `d.keys()`:
    - #process `d[k]`
- `Sorted(L)` returns sorted copy of `L` , BUT, `L.sort()` sorts the `L` that is there
- `D.keys()` gives the data in a form that looks like a list but isn't. so use `list(D.keys())` instead.
- Values in a list can also be inserted like this:
  - `D = {}`
  - `D[0] = 7` .....{doesn't show an error, while in list, it does}
- `D.values()` is a sequence of values in `D`.
  - `Total = 0`
  - For `s` in `test.values()`:
    - `Total = total + test`

# Functions

- `int("A5",16)` means int value of A5 base 16 ie, 165
- `Def F(a,b,c):`
- -----
- `G = F.....`therefore `G(a,b,c)` can also be used
- Def can be conditional.
  - if condition:  
`def f(a,b,c)`
  - Else:  
`def f(a,z,x)`
- You can set up default arguments.
  - Ie, `def f (a,b,c=14,d=22)`
  - So in case you give an input of `def f(12,13)...` it's is assumed as `def f(12,13,14,22)`
  - Smly, `def(13,12,16) .....def(13,12,16,22)`. You can override defined values.

# List functions

- Built in function `map(f,l)` applies the function `f` in each element of `l`
- `Map(f,l)` is not a list, therefore `list(map(f,l,))` should be used
- Set comprehension is useful in initializing lists:  
lets say we need to initialize a 4x3 matrix  

```
l = [ [ 0 for i in range(3)]  
      for j in range(4)]
```
- Set comprehension in python can be done like this:
  - `[square(x) for i in range(0,100) if iseven(i)]`
  - `[(x,y,z) for x in range(100) for y in range(x,100) for z in range(y,100) if x*x + y*y = z*z]`
- `def select(property,l):`
  - `Sublist = []`
  - `For x in l:`  
    `if property(x):`  
        `sublist.append(x)`
  - `return(sublist)`
- `Filter(p,l)` checks `p` for each element of `l`.
  - Ex. `list(map(square,filter(iseven,range(0,100))))`.....where `square` and `iseven` are functions.



- THIS IS WRONG:-
  - Zerolist = [0 for l in range(3)]
  - l = [zerolist for j in range(4)]
  - This will give output l[1][1] = 7
  - You will get matrix [[0,7,0],[0,7,0],[0,7,0],[0,7,0]]

# WEEK 5

- What to do?:

```
try:
    #code with anticipated error
except IndexError:
    #code to avoid this error
except(NameError,KeyError):
    #code which can have either
except:
    #for errors other than metioned
else:
    #executes if no error in try block occurs
```

## Types of errors:

- SyntaxError:- wrong syntax; example:- range(1;100)
- NameError:- Name not defined
- ZeroDivisionError:- division by zero
- IndexError:- list assignment index out of range

- Positive use of this try block:

```
scores = {"Dhawan" : [3,22], "Kohli": [200,3]}
```

Batsman b already exists, append to the list:

```
scores[b].append(s)
```

New batsman, create fresh entry

```
scores[b] = [s]
```

This can be rewritten as->

```
try:
```

```
    scores[b].append(s)
```

```
except KeyError:
```

```
    scores[b] = [s]
```

# STDIO

- You can take input like this:  
    `userdata = input("Type a number: \n")`
- Using error exception:

`While(True):`

`try:`

`userdata = input("Enter a number: ")`

`usernum = int(userdata)`

`except ValueError:`

`print("Not a number: Try again")`

`else:`

`break`

- Each print line automatically starts in a new line. To avoid this problem. Use the end = “...” function.

Ex:-

```
print(“continue on the”, end = “ “)
print(“same line”, end = “.\n”)
print(“Next line.”)
```

output will be:  
continue on the same line.  
Next line.

- The separator works in the same way. The thing is, when we put a comma in the print statement, the comma is also assumed as a space gap given in between.

ie, print(“x is” ,x, “and y is” ,y, “.”) → x is 7 and y is 10 . {there is space near dot}

To avoid this, use sep = "..."

```
print("x is " ,x, " and y is " ,y, ".", sep = "") → x is 7 and y is 10.
```

- TO OPEN A FILE:

`fh = open("gcd.py","r").....`{instead of filename, we can give full path too}

Read: "r": opens a file for reading only

Write: "w": creates an empty file to write to

Append: "a": append to an existing file

- 3 ways to read through a file handle:

1) `Contents = fh.read()` {reads entire file into name as a single string}

2) `Contents = fh.readline()` {reads one line into name-lines end with '\n'.. String includes '\n' unlike `input()`}

3) `Contents = fh.readlines()` {reads entire file as list of strings. Each string is one line ending with '\n'}

- To read lines in a sequence:
  - when file is opened, point to position 0, the start
  - Each successive `readline()` moves forward
  - `fh.seek(n)` – moves pointer to position `n`
  - `block = fh.read(12)` – read a fixed number of characters
- To know the end of a file, `fh.read()` and `fh.readline()` return empty string `""`... these both signal the end of a file
- `fh.writelines(l)`
  - Write a list of lines `l` to file
  - Must include `'\n'` explicitly to each string
- `fh.write(s)` means to write a string `s` into a file.
  - This will return the number of characters written
  - We need to include `'\n'` explicitly to go to a new line



- To process a file line by line:  
    for l in fh.readlines():  
        #code
- fh.flush()
  - Manually forces write to disk
- fh.close()
  - Flushes output buffer and decouples file handle
  - All pending writes copied to disk

- To copy a file:
  - Infile = open("input.txt", "r")
  - Outfile = open("output.txt", "w")
  - For line in infile.readlines():
    - Outfile.write(line)
  - Infile.close()
  - Outfile.close()

Or instead of for loop, you can write:

- contents = infile.readlines()
- Outfile.writelines(contents)

# Continue writing strip line character thing

- W5E3
- Example given by sir after the strip line function thing

# String functions:

1. `s.rstrip()` -> removes trailing whitespaces
2. `s.lstrip()` -> removes leading whitespaces
3. `s.strip()` -> removes leading and trailing whitespaces
4. `s.split("smt")` -> if separated by this "smt" it will print the rest in a list form
  - To search for text:
    1. `S.find(pattern)` -> returns first positions in `s` where pattern occurs, -1, if no occurrence of pattern
    2. `s.find(pattern, start, end)` -> to find the pattern in the slice `s[start:end]`
    3. `s.index(pattern)` or `s.index(pattern, l, r)` -> like Find, but raise `ValueError` if pattern not found.

- String format() method

```
>>"First: {0}, Second: {1}".format(47,11)
```

```
'First: 47, Second: 11'
```

```
>>"Second{1}, First: {0}".format(47,11)
```

```
'Second: 11, First: 47'
```

(OR)

```
>>"One: {f}, Two: {s}".format(f = 47,s=11)
```

```
'One: 47, Two: 11'
```

- This is another way to use it:

```
>>"Value: {0:3d}".format(4)
```

....d means 4 is int value, 3 means that 3 spaces need to be given(1 contains 4, the other 2 were given, one was general after value).

```
'Value:  4'
```

- >>“Value: {0,6.2f}”.format(47.523)
  - Here, f means float val
  - 6 means that 6 spaces need to be given
  - 2 means that the decimal should be rounded to 2 decimal places
  - Therefore the answer will be ‘Value: 47.52’
  - Here, four spaces out of 6 are used to display the number.. Rounded to 2 decimal places. One extra space in the beginning as default
- Codes for string, octal number, hexadecimal, Left justify, Adding zeros etc are the same as in C language
- If you want your python code to do nothing in a loop, use “pass”

- You can use `del(stuList[4])`: to delete an element in a list
- `del(di[k])`: removes the key `k` and its associated value
- `del(x)`: just deletes a variable in general
  - `>>x=7`
  - `>>del(x)`
  - `>>y = 5 + x`
  - `NameError: name 'x' is not defined`

- None is a special value used to denote nothing
  - `x = None`
  - `#code`
  - If x is not None:
    - `Y = x`
- 'X is none' is the same value as 'x==None'



# Backtracking (WEEK 6)

- Systematically search for a solution
- Build a solution one step at a time
- If we hit a dead end, take a step back and try the next option
  
- To create a board in the Queen question, you can use a 2D array
- I.e, `board[i][j] == 1` can represent the presence of a queen at  $(i,j)$
- I.e, `board[i][j] == 0` can represent the absence of the queen at  $(i,j)$
- Also, `board[i] == j`: queen in row  $i$ , column  $j$  i.e,  $(i,j)$

- `///queenvids` notes

- Scope of name

Scope of the name is the portion of the code where it is available to read and update

```
def f():  
    y=x  
    print(y)  
  
x = 7  
  
f().....will print 7
```

```
Def f():  
    y = x  
    print(y)  
    x = 22  
  
x=7  
  
f().....gives error
```

- If x isn't found in f(), its considered global but if it's found and updated in the function, it is considered local.

- Global names that point to mutable values can be updated within a function.

le, def f():

    y = x[0]

    print(y)

    x[0] = 22

X = 7

f() .....(prints 7)

But, if you want a global integer, then make the integer global

- Def f():

```
    global x
```

```
    y=x
```

```
    print(y)
```

```
    x = 22
```

```
X = 7
```

```
F()
```

Print(x).....(prints 7 and 22 in next line)

This code will give val 22

```
def f():
```

```
    def g(a):
```

```
        return(a+1)
```

```
    def h(b):
```

```
        return(2*b)
```

```
    global x
```

```
    y = g(x) + h(x)
```

```
    print(y)
```

```
    x = 22
```

```
x = 7
```

```
f()
```

**\*\*If I were to use g(x) outside the function, an error will be thrown**

# Generating permutations


- In a,b,c,d,e,f,g,h,i,j,k,l,m it's the smallest permutation
- M,l,k,j,i,h,g,f,e,d,c,b,a is the largest permutation order
- Say we need to find the next permutation of dchbaeglkonmji, we will look at the ending and they are already arranged in descending order {onmji}
- So to find the next, we need to include one more alphabet, ie, k.
- Now, place it in the descending order sequence at the end, ie, monkji
- We basically replaced k and m.
- Now, keep everything same from start to letter m
- And inverse everything from there.. ie, dchbaeglmijkno
- This is the next permutation

- The process of writing it in code is that you first check from left to right and at one point it will stop incrementing by 1
- Then take the value right after and place it in a position such that from the back, it will end up being in ascending order if checked from the back.(therefore, compare the size next to next)
- Then change the positions of each to the inverse after m
- ???????paste code here

# Implementation


- From the right, identify first decreasing position

d c h b a e g l **k** o n m j i



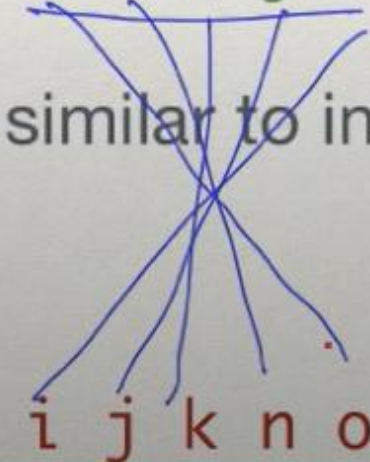
- Swap that value with its next larger letter to its right

d c h b a e g l **m** o n **k** j i



- Finding next larger letter is similar to insert
- Reverse the increasing suffix

d c h b a e g l m i j k n o







# Sets

- Denoted by {}
- They are unordered {hence, no index concept}
- To declare an empty set, use colors = set() and not {} as it means dict
- Set automatically removes duplicates
- Type “‘black’ in colours”, and it will print True if present
- You can convert list to a set using set function ie, set([0,1,1,2,2,3,4,5])
- And when you print, it will remove duplicates

IMP point

Letters = set('banana')

Print(letters).....it will print {'b','a','n'}

## SET FXNS ALSO WORK HERE

1. For union,  $a \mid b$
2. For intersection,  $a \& b$
3. For set difference,  $a - b$  {means in a but not in b}
4. For XOR,  $a \wedge b$  {removes the intersection of venn diag}

# Stacks

- `push(s,x)`... add x to s
- `Pop(s)`...return most recently added element
- Imagine stack as a list from left to right. Assume right is top.
- Ie, `push(s,x)` would be `s.append(x)`
- `S.pop()` returns last element(rightmost)
- Stacks are natural to keep track of recursive function calls
- Hence back tracking would be easier

# Queues

- First-in, first-out sequence
- `addq(q,x)` Adds `x` to rear of queue `q`
- `removeq(q)` removes element at head of `q`
- In python list, head is rightmost and rear is leftmost
- `addq(q,x)` is `q.insert(0,x)`
- `L.insert(j,x)` is to insert `x` before position `j`
- `removeq(q)` is like `q.pop()`.....removing last element

- In Systematic exploration
- Maintain a queue Q of cells to be explored
- Initially q contains only start node (sx,sy)
- Remove (ax,ay) from head of the queue
- Mark all squares reachable in one step from (ax,ay)
- Add all new marked squares to the queue
- When the queue is empty, we have finished

// finish the code part and write

# Priority queue

- A priority queue is when we need to target a job with higher priorities first even when there were jobs assigned before it
- `delete_max()`.....identify and remove job with highest priority
- `insert()`.....add a new job into the list

In linear, Unsorted list:

Deletion takes  $O(n)$  time and Insertion takes  $O(1)$

In linear, Sorted list:

Insertion takes  $O(n)$  time and deletion takes  $O(1)$  time

Processing a sequence of  $n$  jobs requires  $O(n^2)$  time

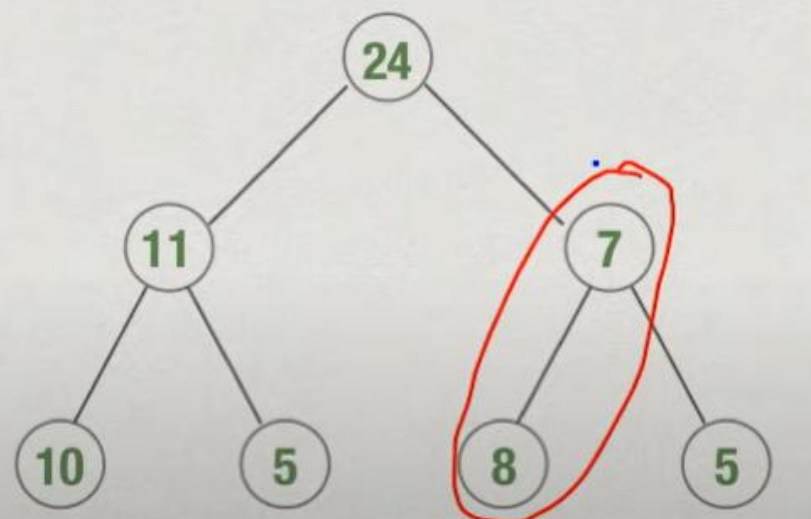
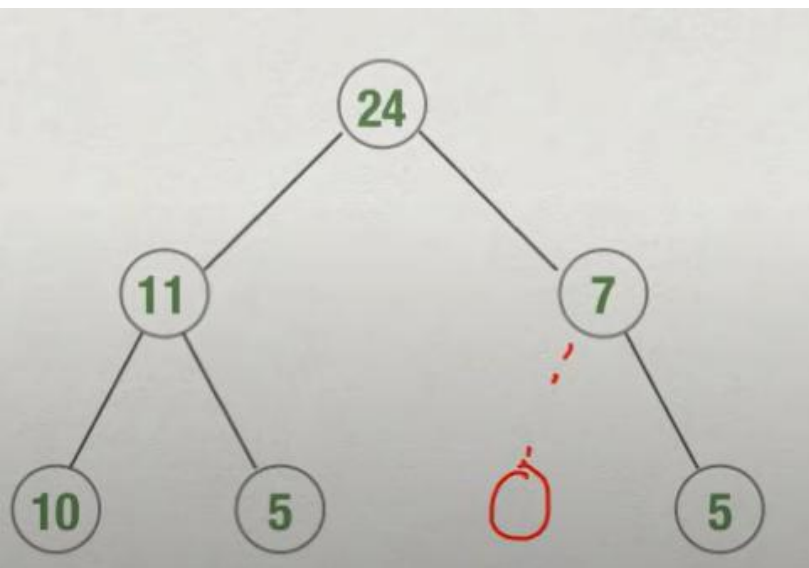
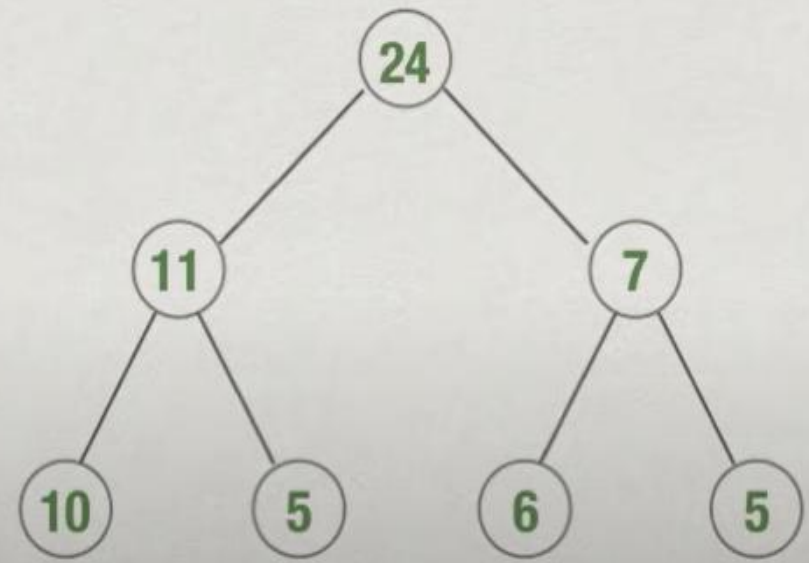
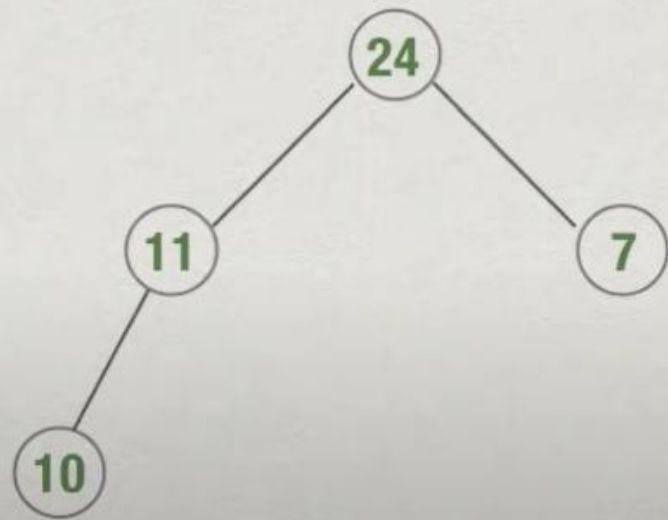
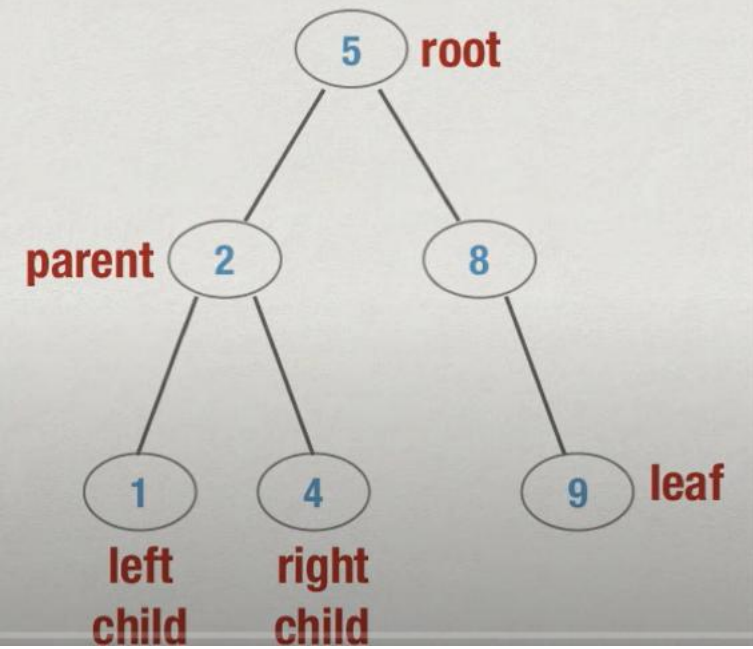
# Binary tree

- 2D structure, consists of nodes, has parent, left child, right child
- Top most is root, bottom most is leaf
- Taking priority queues as heaps can reduce the time taken to  $O(N \log N)$ .. Where both deletion and insertion take  $O(\log N)$  time



# HEAPS

- Binary trees filled level by level, left to right
- (Max) Heap Property Binary tree filled level by level left to right
- In max heap property, the tree is filled level by level as the root is the largest number and the ones below it are lesser than the root. The ones below (if exist) need to be smaller than that number itself. If the number has no kids, then it's ok.
- No holes are allowed
- Can't leave a level incomplete
- //give examples



- Lets say you are inserting a node into either min or max, first, it needs to start at the same/next level and then walk its way up.
- You cannot start at the top and go to the bottom, the node either stays at the bottom or comes to the top.

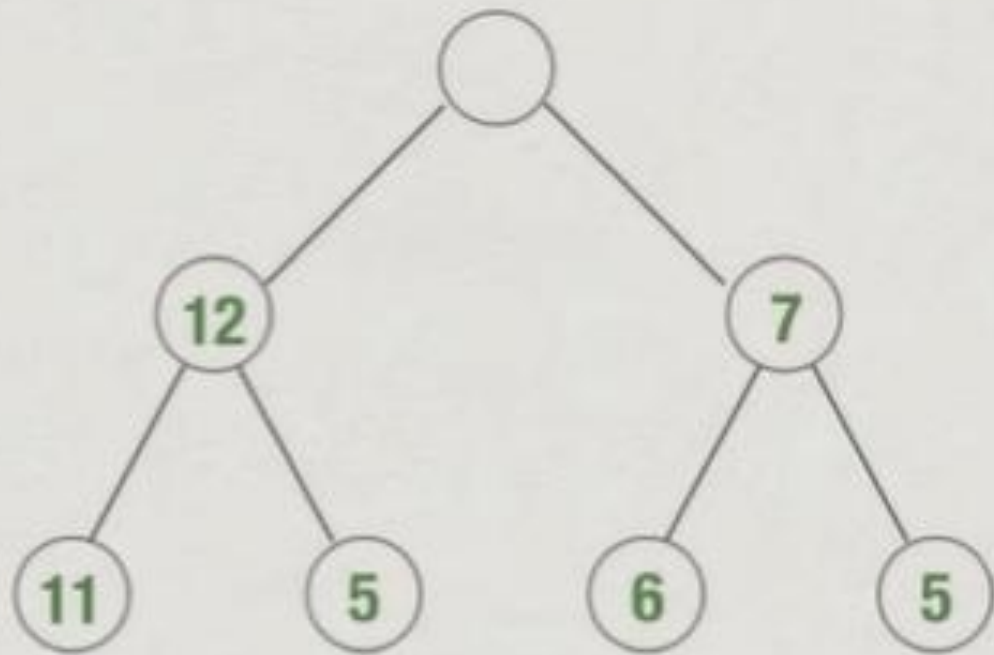
# Complexity of insert( )

- Need to walk up from the leaf to the root
  - Height of the tree
- Number of nodes at level 0,1,...,i is  $2^0, 2^1, \dots, 2^i$
- K levels filled :  $2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$  nodes
- N nodes : number of levels at most  $\log N + 1$
- insert( ) takes time  $O(\log N)$

- If you want to insert a number, first add it at the leaf position, whichever one is vacant.
- Next compare with parent and walk your way up. Push down the value which is the least heavy
- If you delete\_max() {lets say root node}, then you must replace with the one at the bottom most position
- Compare and work your way down. Pull up the one that is heavier child

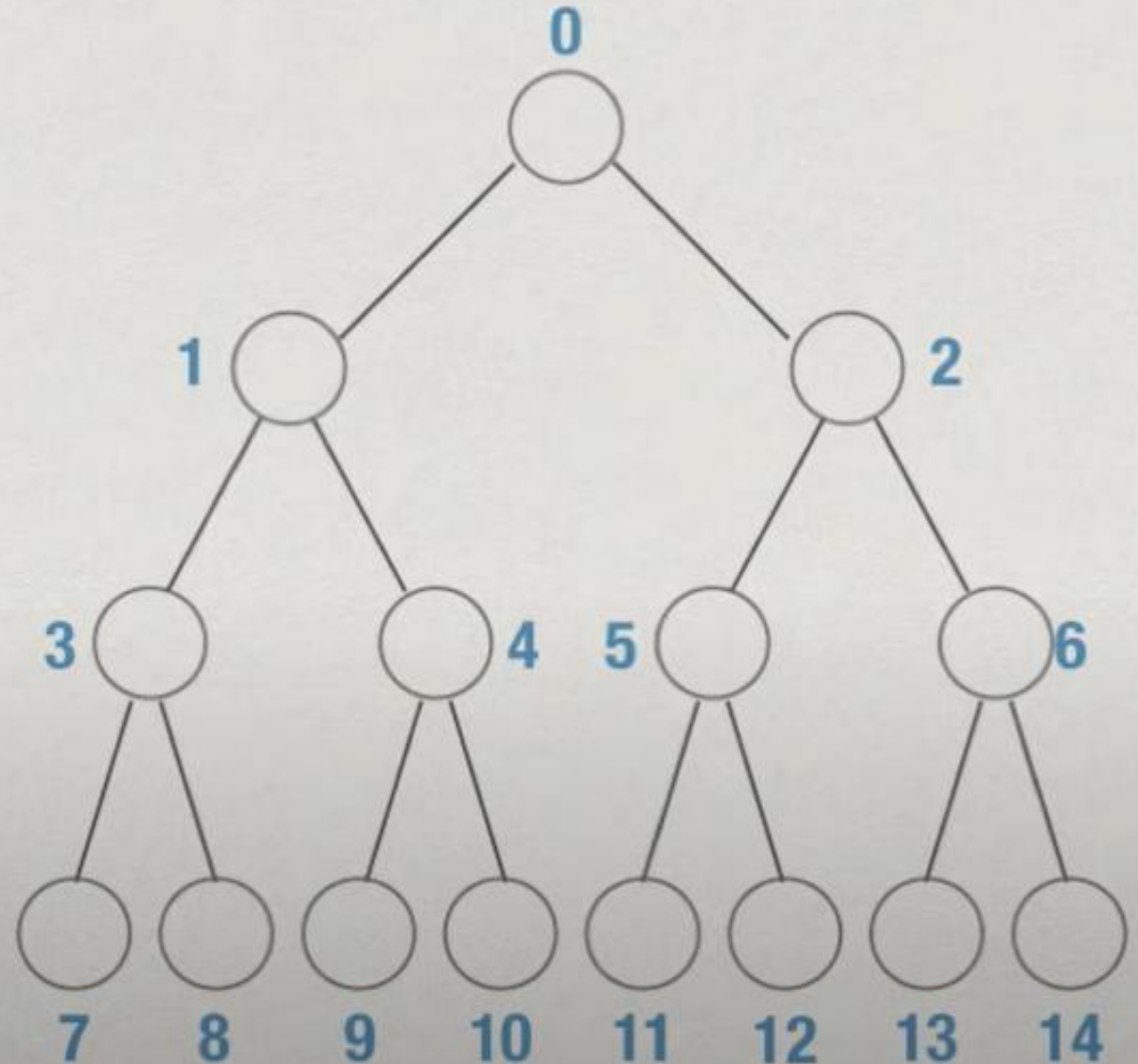
# delete\_max()

- Will follow a single path from root to leaf
- Cost proportional to height of tree
- $O(\log N)$



# Implementing using arrays

- ✱ Number the nodes left to right, level by level
- ✱ Represent as an array  $H[0..N-1]$
- ✱ **Children** of  $H[i]$  are at  $H[2i+1]$ ,  $H[2i+2]$
- ✱ **Parent** of  $H[j]$  is at  $H[\text{floor}((j-1)/2)]$  for  $j > 0$



# The steps to build a heap are:

- Set up an array  $[x_1, x_2, x_3, \dots, x_n]$
- Assume leaf nodes are at level  $k$
- For each node at level  $k, k-1, k-2, \dots, 0$ , fix heap property
- As we go up the number of steps per node goes up but the number of nodes is halved, therefore,  $O(N)$
- If we were to build a heap from root to the leaves, it would take complexity as  $O(N \log N)$  therefor, use the better heapify option



# Better heapify()

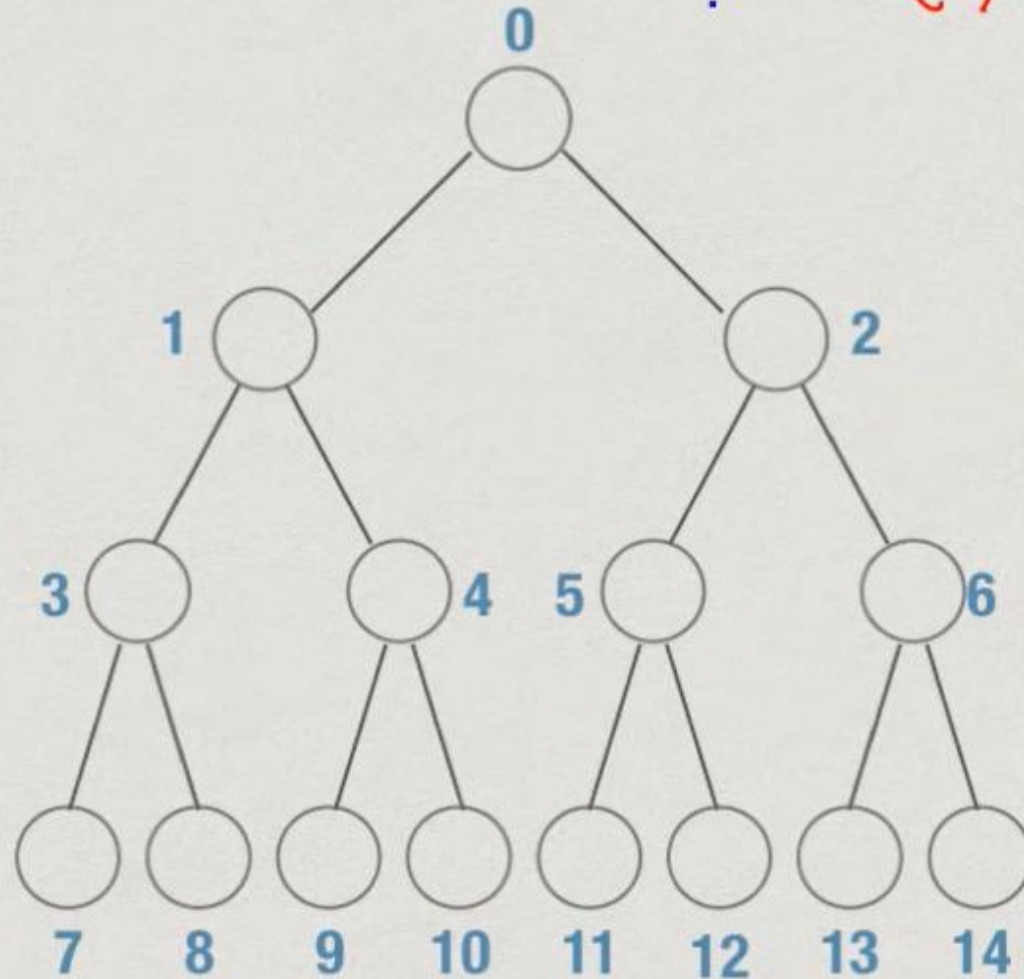
$O(n)$

1 node,  
height 3 repair

2 nodes,  
height 2 repair

4 nodes,  
height 1 repair

$N/2$  nodes  
already satisfy  
heap property



# Heap sort



- \* Start with an unordered list
- \* Build a heap —  $O(n)$
- \* Call `delete_max()`  $n$  times to extract elements in descending order —  $O(n \log n)$
- \* After each `delete_max()`, heap shrinks by 1
  - \* Store maximum value at the end of current heap
- \* In place  $O(n \log n)$  sort

# WEEK 7

- Behaviour defined through interface.
- `(s.push(v)).pop() == v`
- `((addq(u)).addq(v)).removeq() == u`
- Allowed set of operations are
  - Stack : `push()` and `pop()`
  - Queue: `addq()` and `removeq()`
  - Heap: `insert()` and `delete_max()`

- We need to use the built in data types
- `L = []`
- List operations `l.append()` and `l.extend()` can be used but not `l.keys()`
- Silly, `d = {}`, `d.values` is ok but not `d.append()`

## CLASSES

- Template for a data type

## OBJECTS

Concrete instance of template

- Here, `__init__` is a constructor... it will help refer to itself

```
class Heap:
    def __init__(self, l):
        # Create heap
        # from list l

    def insert(self, x):
        # insert x into heap

    def delete_max(self, x):
        # return max element

# Create object,
# calls __init__()
l = [14, 32, 15]
h = Heap(l)

# Apply operation
h.insert(17)
h.insert(28)
v = h.delete_max()
```

The diagram illustrates the relationship between the `Heap` class and its instance `h`. It features several hand-drawn annotations: a red circle around `__init__` in the class definition, a green circle around `self` in the `__init__` method signature, a green circle around `self` in the `insert` method signature, and a green circle around `self` in the `delete_max` method signature. A blue arrow points from the `h` in `h.insert(17)` to the `self` in the `insert` method. A red arrow points from the `h` in `h.insert(28)` to the `self` in the `insert` method. A red arrow points from the `h` in `h.delete_max()` to the `self` in the `delete_max` method. A green arrow points from the `self` in the `insert` method to the `self` in the `delete_max` method.



- Ex:- it's like struct of c or list1.next in java
- Here you can say `p = Point(3,2)` and also `p.translate(2,1)`

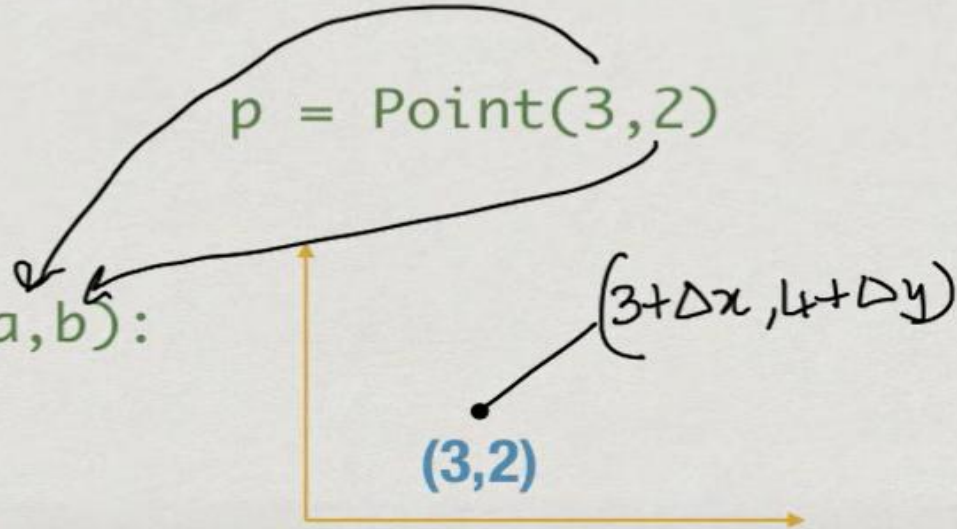
## Points on a plane

$$\begin{aligned} \textcircled{Z} &= \textcircled{Z} + b \\ Z &= Z - b \end{aligned}$$

```
class Point:
```

```
    def __init__(self, a, b):  
        self.x = a  
        self.y = b
```

```
    def translate(self, deltax, deltay):  
        # shift (x,y) to (x+deltax, y+deltay)  
        self.x += deltax # same as self.x =  
                          # self.x + deltax  
        self.y += deltay
```



- Another example

## Points on a plane

```
class Point:
    def __init__(self, a, b):
        self.r = sqrt(a*a + b*b)
        if a == 0:
            self.theta = 0
        else:
            self.theta = atan(b/a)

    def odistance(self):
        return(self.r)

    def translate(self, deltax, deltay):
        # Convert (r, theta) to (x, y) and back!
```

- \* Private implementation has changed
- \* Functionality of public interface remains same

# Special functions

self.x  
self.y

"(x, y)"

- `__init__()`

- Constructor, called when object is created

- `__str__()`

- Return string representation of object
- `str(o) == o.__str__()`
- Implicitly invoked by `print()`

```
def __str__(self): # For Point()
    return '(' + str(self.x) + ', ' + str(self.y) + ')'
```



# Special functions

- `__add__()`

- Invoked implicitly by +

- `p1 + p2 == p1.__add__(p2)`

```
def __add__(self,p):  # For Point()
    return(Point(self.x+p.x,self.y+p.y))
```

```
p1 = Point(1,2)
```

```
p2 = Point(2,5)
```

```
p3 = p1 + p2  # p3 is now (3,7)
```

# Special functions

- `__mult__()`
  - Called implicitly by `*`
- `__lt__()`, `__gt__()`, `__le__()`, . . .
  - Called implicitly by `<`, `>`, `<=`
- Many others, see Python documentation

- `#temp = self`
- `#while temp.next  $\neq$  None:`
- `#if temp.next.value == x:`
- `#temp.next = temp.next.next`
- `#return()`
- `#else:`
- `#temp = temp.next`
- `#return()`

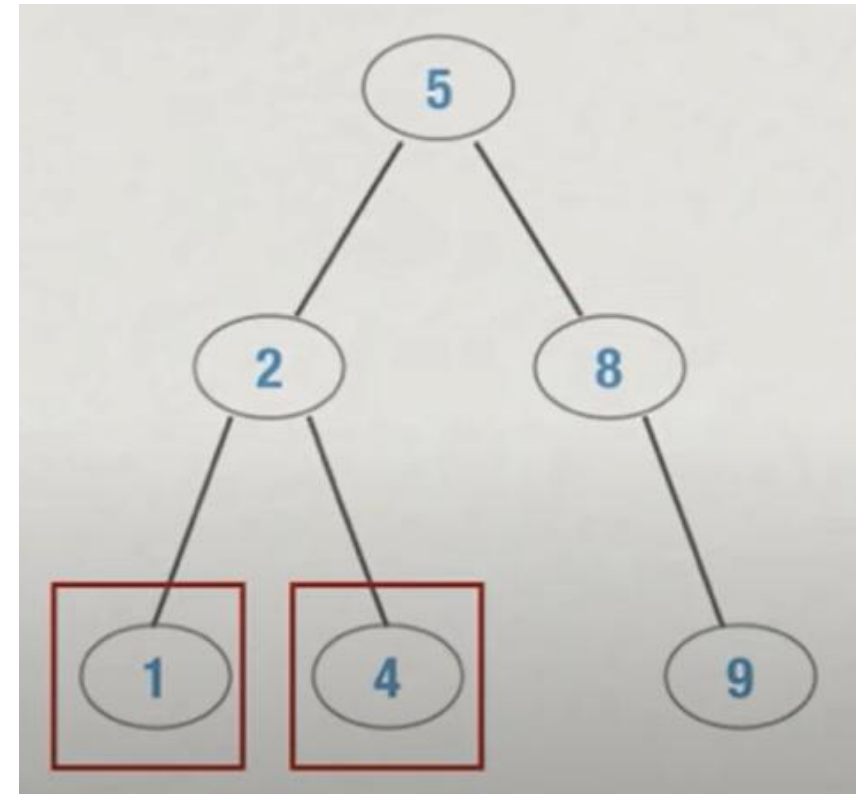
///Stack code

# Binary search trees

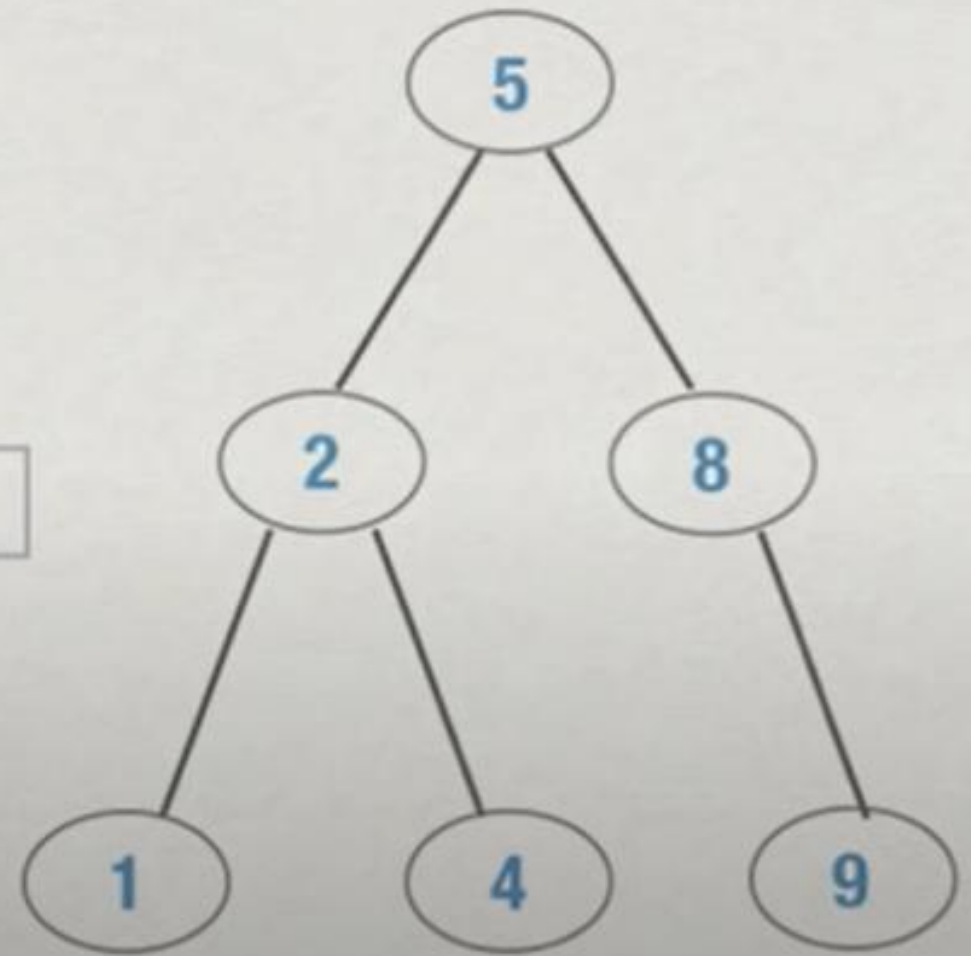
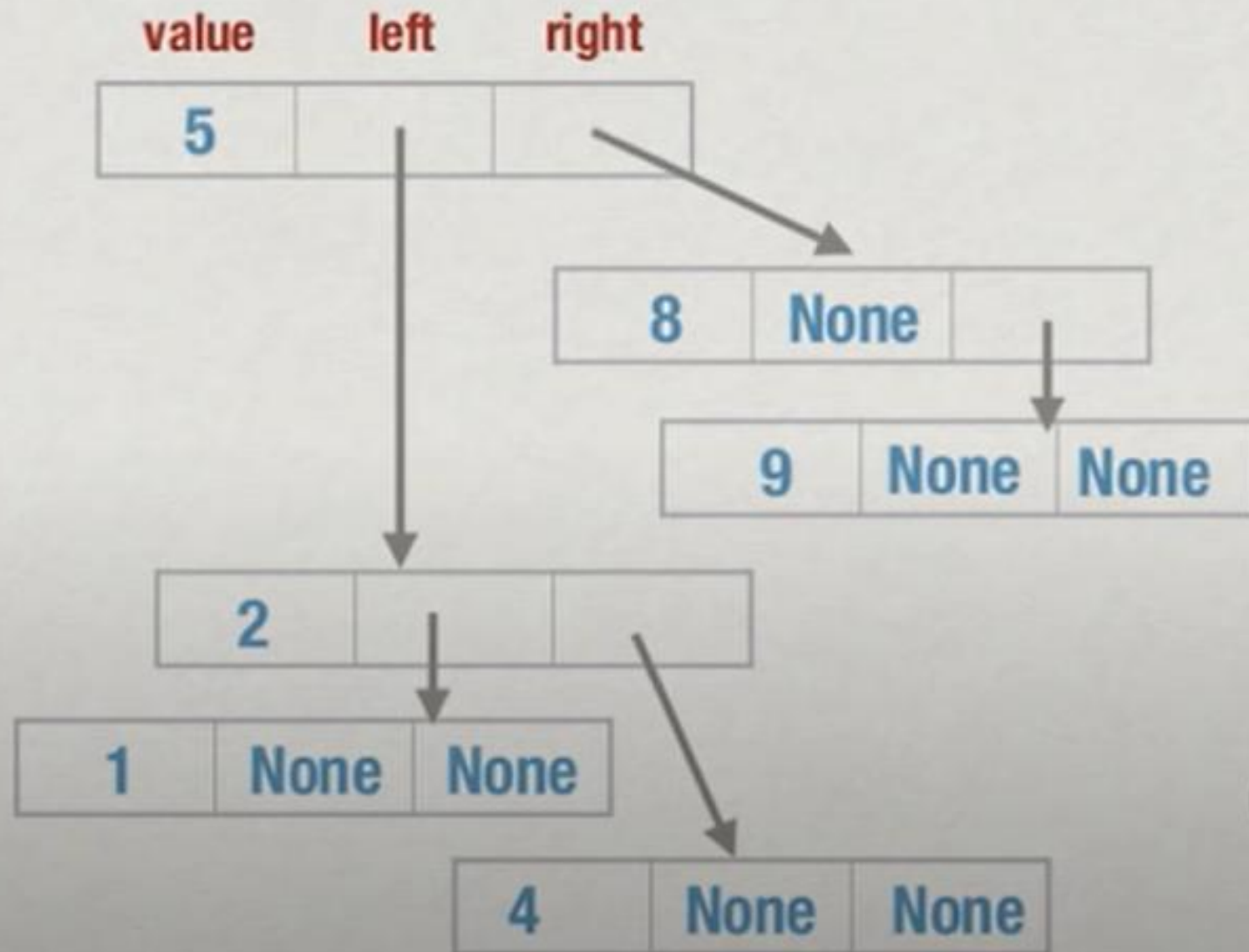
- For each node with value  $v$
- Values in left of subtree  $< v$
- Values in right of subtree  $> v$
- No duplicate values
- These are basically doubly linked lists

Which consist of left, right and data/value... Therefore

- Empty tree has a single empty node, leaf has both the nodes as None and other nodes can have either both filled or one filled exactly.

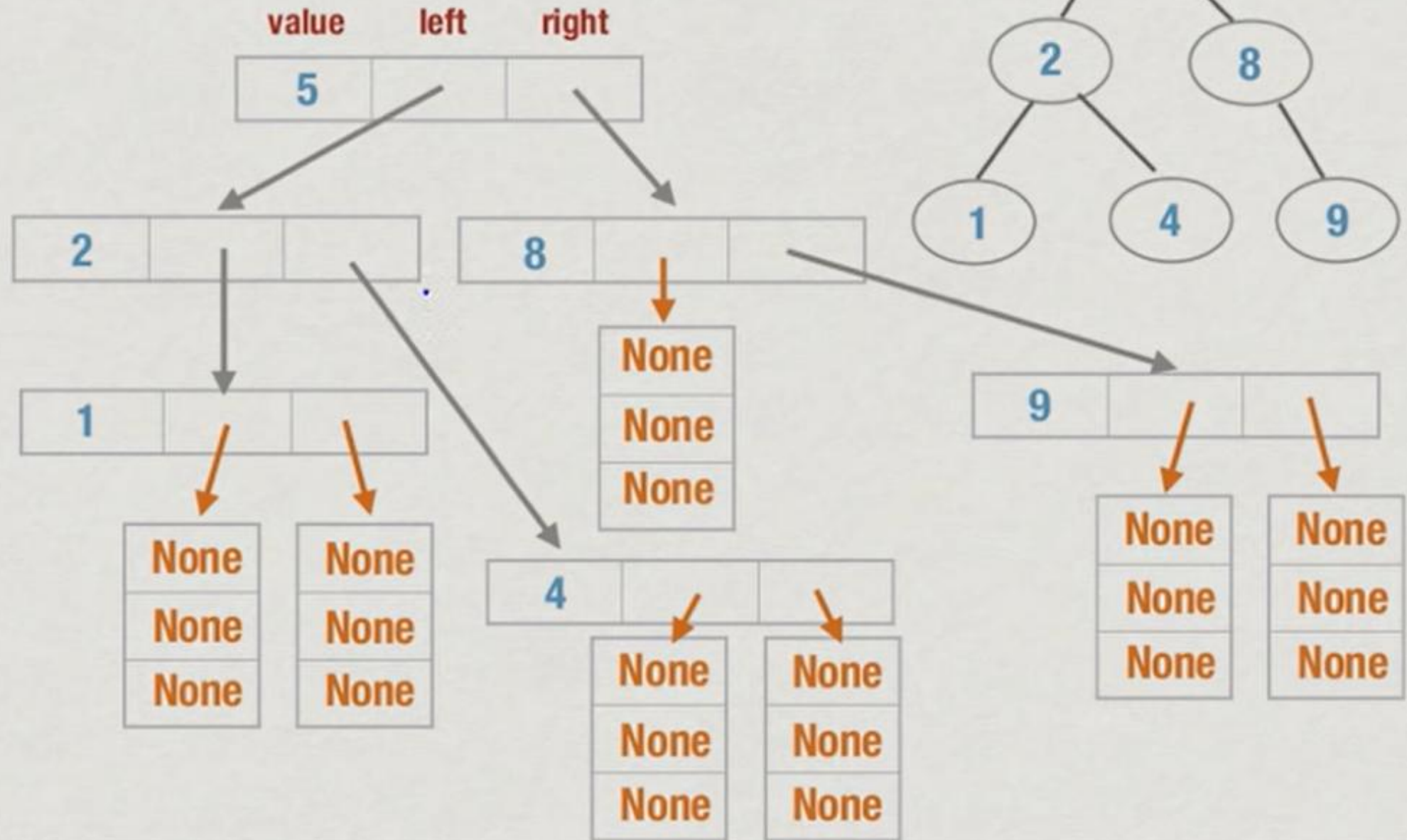


- Each node has a value and points to its children

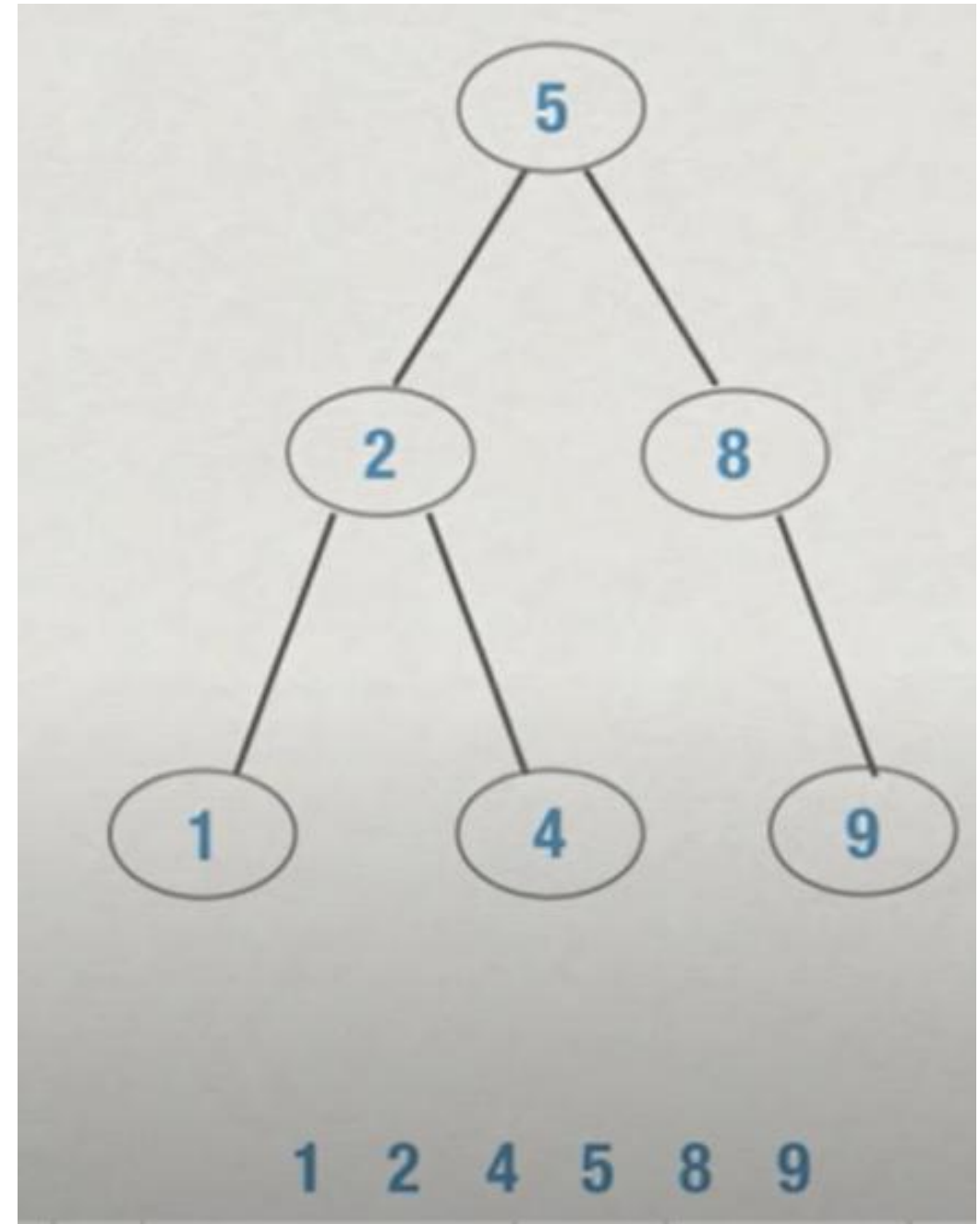




# Binary search tree



- It prints it in order itself





///`Type code and tell complexity`

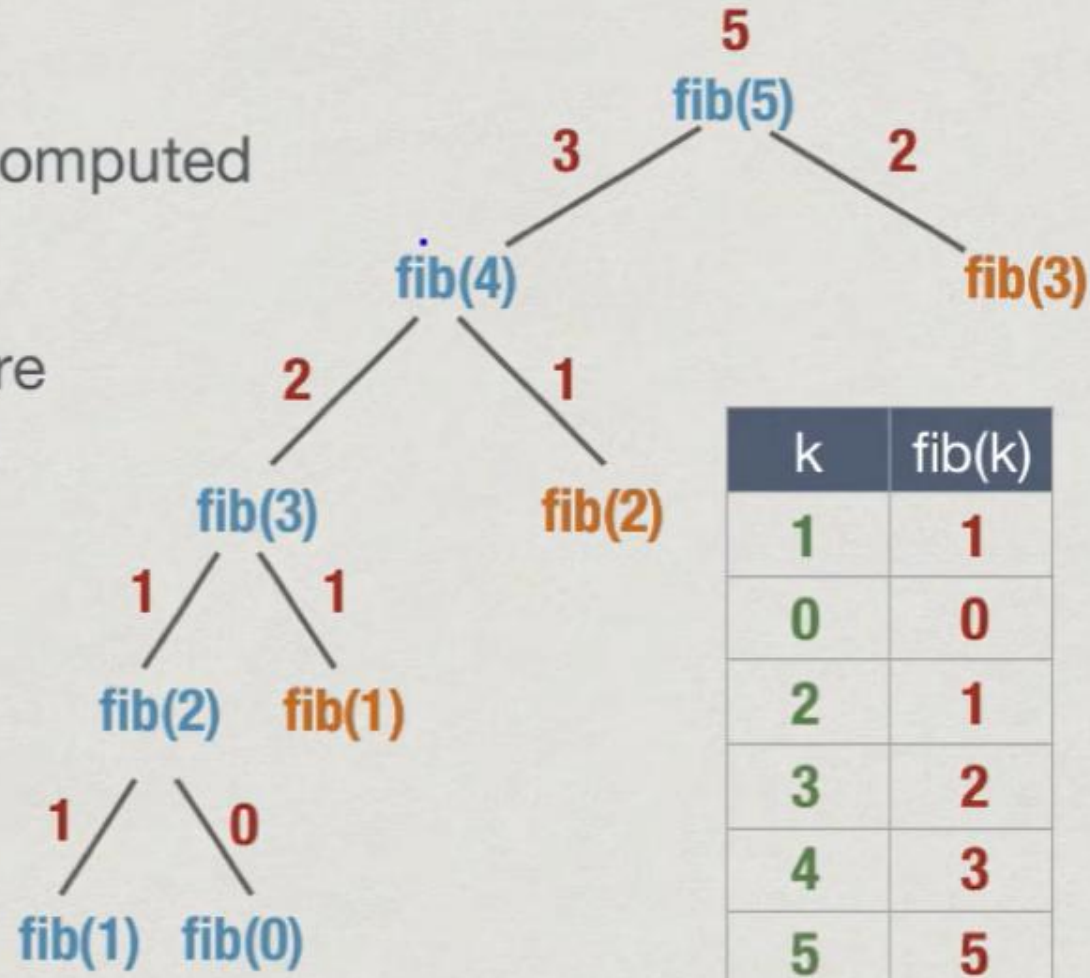
# WEEK 8 (Vid 1)

- Subproblems:  
factorial( $n-1$ ) is a subproblem of factorial( $n$ )
- Isort( $[x_2, \dots, x_n]$ ) is a subproblem of Isort( $[x_1, x_2, \dots, x_n]$ )
- We use recursion for these types of Q
- But it leads to wasteful recomputation and the computation tree grows exponentially
- To avoid that problem, Build a table of values already computed and store them. Re- evaluation space is saved. This is called Memoization

# Memoized fib(5)

## Memoization

- \* Store each newly computed value in a table
- \* Look up table before starting a recursive computation
- \* Computation tree is linear



# Memoized fibonnaci

```
def fib(n):  
    if fibtable[n]:  
        return(fibtable[n]):  
  
    if n==0 or n==1:  
        value = n  
  
    else:  
        value = fib(n-1) + fib(n-2)  
  
    fibtable[n] = value  
    return(value)
```

# In general

```
function f(x,y,z):
```

```
    if ftable[x][y][z]:
```

```
        return(ftable[x][y][z])
```

```
    value = expression in terms of  
              subproblems
```

```
    ftable[x][y][z] = value
```

```
    return(value)
```

# Dynamic programming for table

```
Def fib(n):  
    fibtable[0] = 0  
    fibtable[1] = 0  
    for i in range(2,n+1):  
        fibtable[i] = fibtable[i-1] + fibtable[i-2]  
    return(fibtable[n])
```

# Summary

## Memoization

- \* Store values of subproblems in a table
- \* Look up the table before making a recursive call

## Dynamic programming:

- \* Solve subproblems in order of dependency
  - \* Dependencies must be acyclic
- \* Iterative evaluation

# GRID PATHS

- In how many ways can you get from one point  $(0,0)$  to  $(m,n)$
- Ex, from  $(0,0)$  to  $(5,10)$ .

In general number of paths is  $15$ . usually  $m + n$

From which  $10$  are up moves and  $5$  are right moves

There number of combinations:  $(15!)/(10!)(5!) = 3003$

What if a path is blocked? Lets say  $(2,4)$  is blocked

Then, you have to count separately:

ie, Every path via  $(2,4)$  first goes from origin to  $(2,4)$  and then from  $(2,4)$  to  $(5,10)$



- $(4+2)$  choose 2 ie, 15 paths .....  $6!/(4!)(2!)$
- $(3 + 6)$  choose 3 ie, 84 paths .....  $9!/(3!)(6!)$
- {The reason for 2 and 3 is about choosing the right movement paths. Up useless in this case}
- Now, multiply the two paths to get the total number of paths through (2,4)
- ie, 1260
- Now  $3003 - 1260 = 1743$

What about multiple holes?

We can repeat the same process twice and get the answer or we can use the inductive method:

Path  $(i,j) = 0$ , if there is a hole there.

# Inductive formulation

- $\text{Paths}(i,j)$  : Number of paths from  $(0,0)$  to  $(i,j)$
- $\text{Paths}(i,j) = \text{Paths}(i-1,j) + \text{Paths}(i,j-1)$
- Boundary cases
  - $\text{Paths}(i,0) = \text{Paths}(i-1,0)$  # Bottom row
  - $\text{Paths}(0,j) = \text{Paths}(0,j-1)$  # Left column
  - $\text{Paths}(0,0) = 1$  # Base case

# Computing paths( $i,j$ )

- Naïve recursion will compute multiple values
- But if you use memoization, it will be much easier (or compute the subproblems directly in a subtle way)
- The value of numbers is the sum of left and down values.... NEXT PAGE

# Dynamic programming

- Start at (0,0)
- Fill row by row

1	11	51	181	526	1363
1	10	40	130	345	837
1	9	30	90	215	492
1	8	21	60	125	272
1	7	13	39	65	147
1	6	6	26	26	82
1	5	0	20	0	56
1	4	10	20	35	56
1	3	6	10	15	21
1	2	3	4	5	6
1	1	1	1	1	1

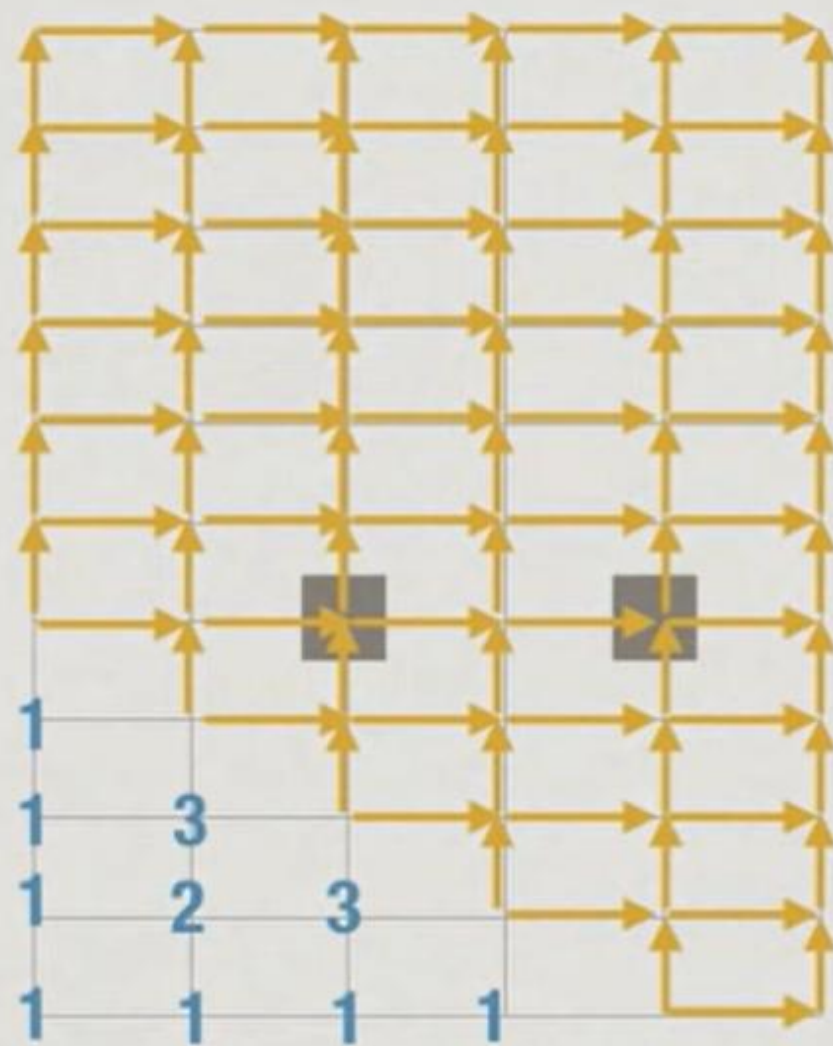
# Dynamic programming

- Start at (0,0)
- Fill by column

1	11	51	181	526	1363
1	10	40	130	345	837
1	9	30	90	215	492
1	8	21	60	125	272
1	7	13	39	65	147
1	6	6	26	26	82
1	5	0	20	0	56
1	4	10	20	35	56
1	3	6	10	15	21
1	2	3	4	5	6
1	1	1	1	1	1

# Dynamic programming

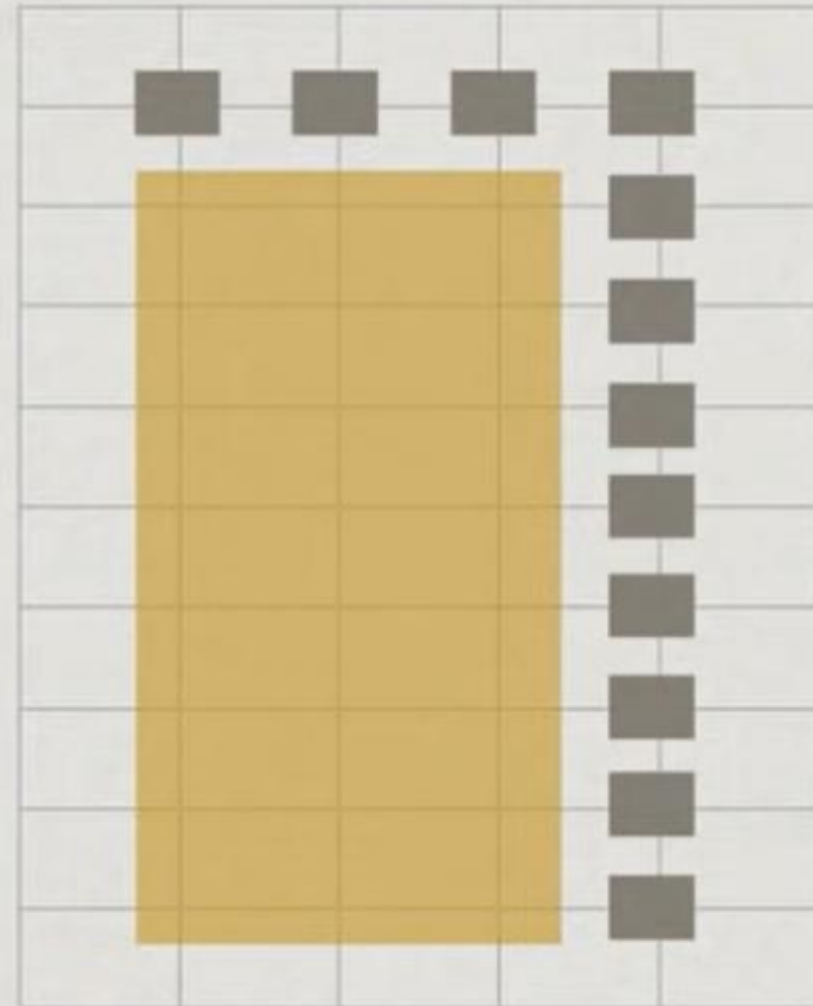
- Start at (0,0)
- Fill by diagonal





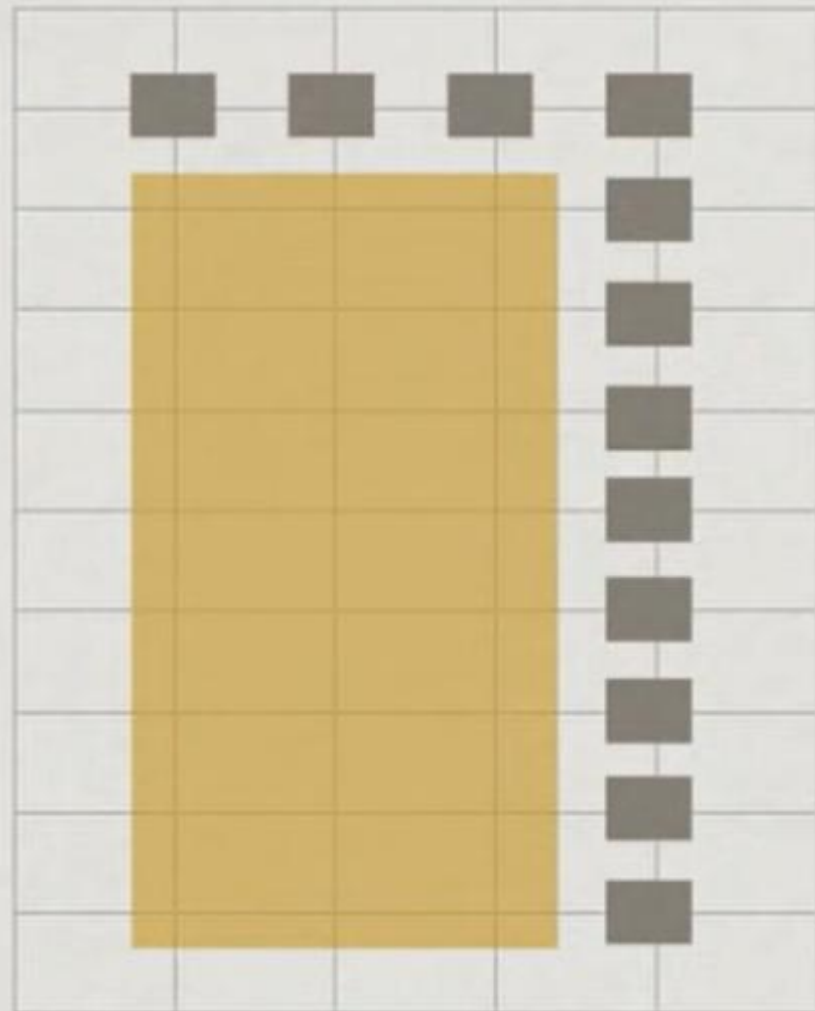
# Memoization vs dynamic programming

- Holes just inside the border
- Memoization never explores the shaded region



# Memoization vs dynamic programming

- Memo table has  $O(m+n)$  entries
- Dynamic programming blindly fills all  $O(mn)$  entries
- Iteration vs recursion — “wasteful”  
dynamic programming is still better, in general

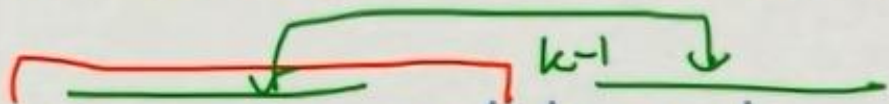




# Finding a set of common words in a string

- $LCW(i,j)$ : Length of the longest common subword

# Inductive structure

- 
- \*  $a_i a_{i+1} \dots a_{i+k-1} = b_j b_{j+1} \dots b_{j+k-1}$  is a common subword of length  $k$  at  $(i, j)$  iff
    - \*  $a_i = b_j$  and
    - \*  $a_{i+1} \dots a_{i+k-1} = b_{j+1} \dots b_{j+k-1}$  is a common subword of length  $k-1$  at  $(i+1, j+1)$
  - \*  $LCW(i, j)$ : length of the longest common subword starting at  $a_i$  and  $b_j$
  - \* If  $a_i \neq b_j$ ,  $LCW(i, j)$  is  $0$ , otherwise  $1 + LCW(i+1, j+1)$
  - \* Boundary condition: when we have reached the end of one of the words

# Reading off the solution

- Find  $(i,j)$  with largest entry
- $LCW(2,0) = 3$
- Read off the actual subword diagonally

		0	1	2	3	4	5	6
		s	e	c	r	e	t	.
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	.	0	0	0	0	0	0	0

```
def LCW(u,v):
    for r in range(len(u) + 1):
        LCW[r][len(v) + 1] = 0
    for c in range(len(v) + 1):
        LCW[len(u) + 1][c] = 0
    maxLCW = 0
    for c in range(len(v) +1,-1,-1):
        for r in range(len(u)+1,-1,-1):
            if u[r] == v[c]:
                LCW[r][c] = 1 + LCW[r+1][c + 1]
            else:
                LCW[r][c] = 0
            if LCW[r][c] > maxLCW:
                maxLCW = LCR[r][c]
    return(maxLCW)
```

# Longest common sequence

- le, “bisect” and “secret” .... “Sect” is the subword
- “dictionary” and “secretary” .... “ectr” and “retr” is the subword

Table and code below:-

# Recovering the sequence

- Trace back the path by which each entry was filled
- Each diagonal step is an element of the LCS
- “sect”

		0	1	2	3	4	5	6
		s	e	c	r	e	t	.
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	.	0	0	0	0	0	0	0

```
def LCS(u,v):  
    for r in range(len(u) + 1):  
        LCS[r][len(v)+ 1] = 0  
    for c in range(len(v) + 1):  
        LCS[len(u) + 1][c] = 0  
    for c in range(len(v),-1,-1):  
        for r in range(len(u),-1,-1):  
            if (u[r] == v[c])  
                LCS[r][c] = 1 + LCS[r+1][c+1]  
            else  
                LCS[r][c] = max(LCS[r+1][c],  
                                LCS[r][c+1])  
    return(LCS[0][0])
```



# MATRIX multiplication

- Check of  $m_1 n_1 m_2 n_2$  thing
- Each entry in AB take  $O(n)$  steps to compute
- Overall computing AB is  $O(mpn)$
- Matrix multiplication is associative is,  $ABC = (AB)C = A(BC)$
- Order produces the same answer but it the complexity is different
- Use proper ways to bracket you expression of matrices
- Final factor has dimensions  $(r_1, ck)$ , second  $(rk+1, cn)$
- Therefore final multiplication step will be  $O(r_1 ck cn)$
- Add cost of computing the two factors



# Subproblems

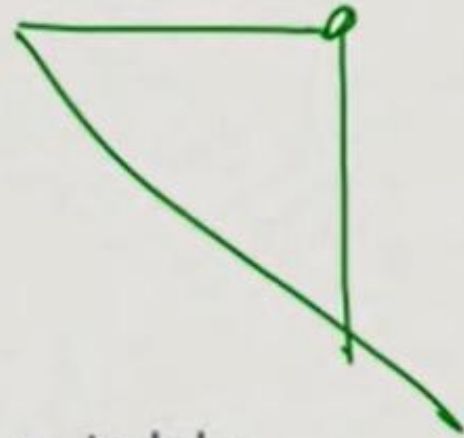
- Final step is  
 $(M_1 \times M_2 \times \dots \times M_k) \times (M_{k+1} \times M_{k+2} \times \dots \times M_n)$
- Subproblems are  $(M_1 \times M_2 \times \dots \times M_k)$  and  $(M_{k+1} \times M_{k+2} \times \dots \times M_n)$
- Total cost is  $\text{Cost}(M_1 \times M_2 \times \dots \times M_k) + \text{Cost}(M_{k+1} \times M_{k+2} \times \dots \times M_n) + r_1 c_k c_n$
- Which  $k$  should we choose?
- No idea! Try them all and choose the minimum!

- Watch the full explanation in the vid, WEEK 8 Vid 4
- $\text{Cost}(i,i) = 0$
- Note that we require  $\text{Cost}(i,j)$  only when  $i \leq j$
- $\text{Cost}(i,j) = \min \text{ over } i \leq k < j$ 
  - $[\text{Cost}(i,k) + \text{Cost}(k+1,j) + \text{rickc}j]$

```
def MMC(R,C):  
    for r in range(len(R)):  
        MMC[r][r] = 0  
    for c in range(1,len(R))  
        for r in range(c-1,-1,-1):  
            MMC[r][c] = infinity  
            for k in range(r,c):  
                subprob = MMC[r][k] + MMC[k][c] +  
                    R[r]C[k]C[c]  
  
            if subprob < MMC[r][c]:  
                MMC[r][c] = subprob
```

# Complexity

Complexity > Table Size



- As with LCS, ~~ED~~, we to fill an  $O(n^2)$  size table
- However, filling  $MMC[i][j]$  could require examining  $O(n)$  intermediate values
- Hence, overall complexity is  $O(n^3)$

READ Through last vid PPT before exam

# Node Code

```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None

    def isempty(self):
        if self.value == None:
            return True
        else:
            return False
```

```
def append(self, v):  
    if self.isempty():  
        self.value = v  
  
    elif self.next == None:  
        newnode = Node(v)  
        self.next = newnode  
  
    else:  
        self.next.append(v)  
  
    return()
```

```
def Traverse(self,v):  
    if self.isempty():  
        self.value = v  
        return()  
  
    temp = self  
    while temp.next  $\neq$  None:  
        temp = temp.next  
  
    newnode = Node(v)  
    temp.next = newnode  
  
    return()
```



```
def insert(self,v):  
  
    if self.isempty():  
        self.value = v  
        return()  
  
    newnode = Node(v)  
  
    (self.value,newnode.value) = (newnode.value,self.value)  
    (self.next,newnode.next) = (newnode,self.next)  
  
    return()
```

```
def delete(self,v):  
  
    if self.isempty():  
        return()  
  
    if self.value == v:  
        self.value = None  
        if self.next  $\neq$  None:  
            self.value = self.next.value  
            self.next = self.next.next  
            return  
    else:  
        if self.next  $\neq$  None:  
            self.next.delete(v)  
            if self.next.value == None:  
                self.next = None  
  
    return()
```

```
def __str__(self):  
  
    selflist = []  
    if self.value == None:  
        return(str(selflist))  
  
    temp = self  
    selflist.append(temp.value)  
  
    while temp.next  $\neq$  None:  
        temp = temp.next  
        selflist.append(temp.value)  
  
    return(str(selflist))
```

# TREE CODE

```
class Tree:
    def __init__(self, initval = None):
        self.value = initval
        if self.val:
            self.left = Tree()
            self.right = Tree()

        else:
            self.left = None
            self.right = None
        return()

    def isEmpty(self):
        return(self.value == None)
```

```
def inorder(self):
    if self.isEmpty():
        return([])
    else:
        return(self.left.inorder() + [self.value] + self.right.inorder)

def __str__(self):
    return(str(self.inorder()))

def find(self,v):
    if self.isEmpty():
        return False
```

```
def find(self,v):  
    if self.isEmpty():  
        return False  
  
    if self.value == v:  
        return True  
  
    if v<t.value:  
        return(self.left.find(v))  
  
    else:  
        return(self.right.find(v))
```

```
def minval(self):  
  
    if self.left == None:  
        return(self.value)  
    else:  
        return(self.left.minval())  
  
def maxval(self):  
  
    if self.right == None:  
        return(self.value)  
    else:  
        return(self.right.maxval())
```

```
def insert(self,v):  
  
    if self.isEmpty():  
        self.value = v  
        self.left = Tree()  
        self.right = Tree()  
  
    if self.value == v:  
        return  
  
    if v < self.value:  
        self.left.insert(v)  
        return  
  
    if v > self.value:  
        self.right.insert(v)  
        return
```



```
def delete(self,v):
    if self.isEmpty():
        return

    if v < self.value:
        self.left.delete(v)
        return

    if v > self.value:
        self.right.delete(v)
        return

    if v == self.value:
        if self.isleaf():
            self.makeEmpty()
        elif self.left.isEmpty():
            self.copyRight()
        else:
            self.value = self.left.maxval()
            self.left.delete(self.left.maxval())
    return
```

```
def makeEmpty(self):  
    self.left = None  
    self.right = None  
    self.value = None  
    return  
  
def copyRight(self):  
    self.value = self.right.value  
    self.left = self.right.left  
    self.right = self.right.right'''
```