Hacker Rank Basics For Numpy

August 16, 2022

1 Numpy Basics

- Operations
  - type(arr) -> basic array is numpy.ndarray
  - arr.shape
  - np.arange(start, stop, stride)
  - np.zeros(amount), np.zeros((amount, amount))
  - np.ones(amount)
  - np.linspace(start, stop, number of linearly-spaced numbers)
  - np.eye(amount) -> identity matrix
  - np.random.rand(rows, columns) -> float
  - np.random.randint(start, stop, amount) -> int
  - array.reshape(rows, columns)
  - array.min(axis = 0) -> min of each column
  - array.max(axis = 1) -> max of each row
  - array.argmax(), array.argmin() -> find index of max/min values
  - np.log(array)
  - np.exp(array)
  - vect1.dot(vect2) or np.dot(vect1, vect2) -> inner product
  - np.cross(vect1, vect2)
  - np.dot(Matrix1, Matrix2)
  - np.multiply(Matrix1, Matrix2) -> element wise multiplication
  - np.linalg.inv(Matrix) -> inverse
  - np.linalg.det(Matrix) -> determinant
  - np.trace(Matrix)
  - np.sum(x, axis = 0) -> sum each column; np.sum(x, axis = 1) -> sum row
  - np.poly(array) -> returns coefficients of a polynomial with the given sequence of roots
  - np.roots(array) -> returns roots of a polynomial with the given coefficients
  - np.polyint(array) -> returns an antiderivative (indefinite integral)
  - np.polyder(array) -> returns the derivative of the specified order
  - np.polyval(array, val) -> evaluates polynomial at a specific value
  - np.polyfit(array1, array2, order) -> fits a polynomial of a specified order to a set of data using a least-squares approach
- Broadcasting Rules
  - The first rule of broadcasting is that if all input arrays do not have the same number of dimensions, a "1" will be repeatedly prepended to the shapes of the smaller arrays until all the arrays have the same number of dimensions.
  - The second rule of broadcasting ensures that arrays with a size of 1 along a particular

dimension act as if they had the size of the array with the largest shape along that dimension. The value of the array element is assumed to be the same along that dimension for the "broadcast" array.

[1]:
```
# Used in basics section
# !pip3 install numpy
# !pip3 install matplotlib
# !pip3 install scipy
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import pdist, squareform
# from scipy.misc import imread, imsave, imresize
```

[2]:
```
a = np.array([x for x in range(10)])
a + 2
```

[2]:
```
array([ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

[3]:
```
type(a)
```

[3]:
```
numpy.ndarray
```

[4]:
```
np.array((a, a)).ravel()  # returns the array, flattened
```

[4]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[5]:
```
# Important
# Simple assignments make no copy of objects
b = a
b is a
```

[5]:
```
True
```

1.1 Shared Memory in Numpy (POSIX)

[6]:
```
c = a.view()
print(f"C is A: {c is a}")
print(f"C is a view of data owned by A: {c.base is a}")
print(f"C owns data at A: {c.flags.owndata}")
c = c.reshape((2, 5))  # a's shape doesn't change
c[0, 4] = 1234         # a's data changes
print(f"A:\n {a}")
print(f"C:\n {c}")
```

```
C is A: False
C is a view of data owned by A: True
C owns data at A: False
A:
```

```
 [   0    1    2    3 1234    5    6    7    8    9]
C:
 [[   0    1    2    3 1234]
 [   5    6    7    8    9]]
```

[7]:
```
# Deep Copy
d = a.copy()
print(f"D is A: {d is a}\nD shares with A: {d.base is a}")
```

```
D is A: False
D shares with A: False
```

[8]:
```
x = np.array(a, dtype=np.float64)
y = np.array(a + 2, dtype=np.float64)
print(np.add(x,y))
(np.subtract(x,y))
```

```
[2.00e+00 4.00e+00 6.00e+00 8.00e+00 2.47e+03 1.20e+01 1.40e+01 1.60e+01
 1.80e+01 2.00e+01]
```

[8]:
```
array([-2., -2., -2., -2., -2., -2., -2., -2., -2., -2.])
```

[9]:
```
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

[10]:
```
# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
```

```python
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)

# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# Another solution is to reshape w to be a column vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))

# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
[[2 4 6]
 [5 7 9]]
[[ 5  6  7]
 [ 9 10 11]]
[[ 5  6  7]
 [ 9 10 11]]
[[ 2  4  6]
```

```
 [ 8 10 12]]
```

[11]:
```python
def f(x, y):
    return 10 * x + y

np.fromfunction(f, (5, 4), dtype=np.float32)
```

[11]:
```
array([[ 0.,  1.,  2.,  3.],
       [10., 11., 12., 13.],
       [20., 21., 22., 23.],
       [30., 31., 32., 33.],
       [40., 41., 42., 43.]], dtype=float32)
```

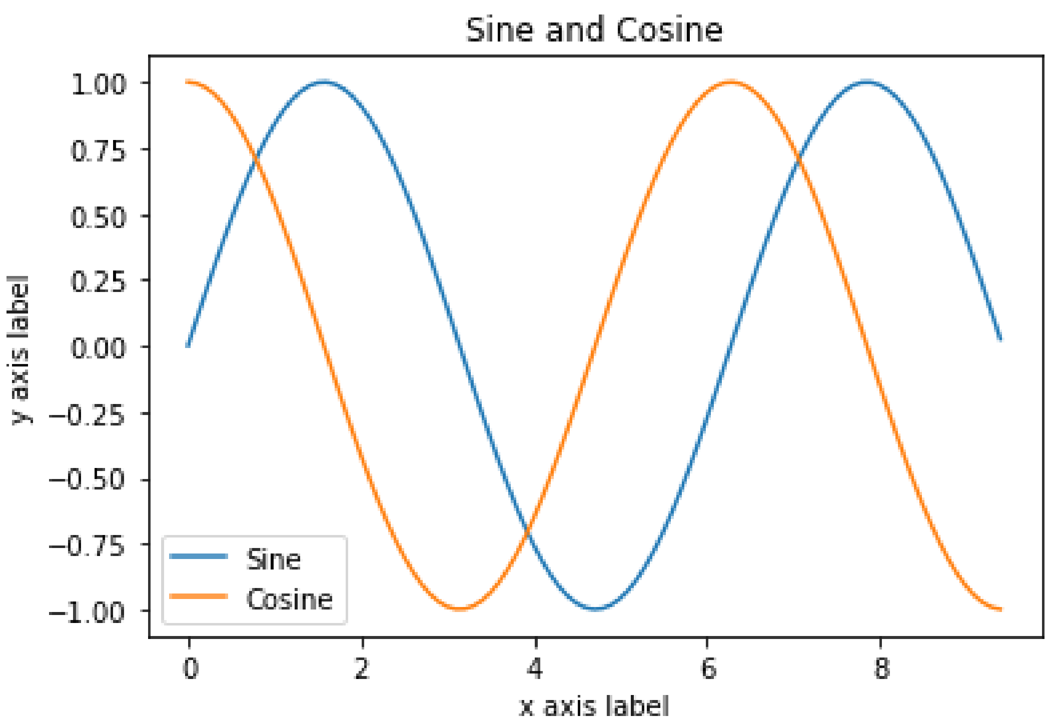
[ ]:
```python
# Read an JPEG image into a numpy array
img = imread('assets/cat.jpg')
print(img.dtype, img.shape)  # Prints "uint8 (400, 248, 3)"

# We can tint the image by scaling each of the color channels
# by a different scalar constant. The image has shape (400, 248, 3);
# we multiply it by the array [1, 0.95, 0.9] of shape (3,);
# numpy broadcasting means that this leaves the red channel unchanged,
# and multiplies the green and blue channels by 0.95 and 0.9
# respectively.
img_tinted = img * [1, 0.95, 0.9]

# Resize the tinted image to be 300 by 300 pixels.
img_tinted = imresize(img_tinted, (300, 300))

# Write the tinted image back to disk
imsave('assets/cat_tinted.jpg', img_tinted)
```

[12]:
```python
# Distance Between Points
# Create the following array where each row is a point in 2D space:
# [[0 1]
#  [1 0]
#  [2 0]]
x = np.array([[0, 1], [1, 0], [2, 0]])
print(x)

# Compute the Euclidean distance between all rows of x.
# d[i, j] is the Euclidean distance between x[i, :] and x[j, :],
# and d is the following array:
# [[ 0.          1.41421356  2.23606798]
#  [ 1.41421356  0.          1.        ]
#  [ 2.23606798  1.          0.        ]]
d = squareform(pdist(x, 'euclidean'))
print(d)
```

```
[[0 1]
 [1 0]
 [2 0]]
[[0.         1.41421356 2.23606798]
 [1.41421356 0.         1.        ]
 [2.23606798 1.         0.        ]]
```

[13]:

```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

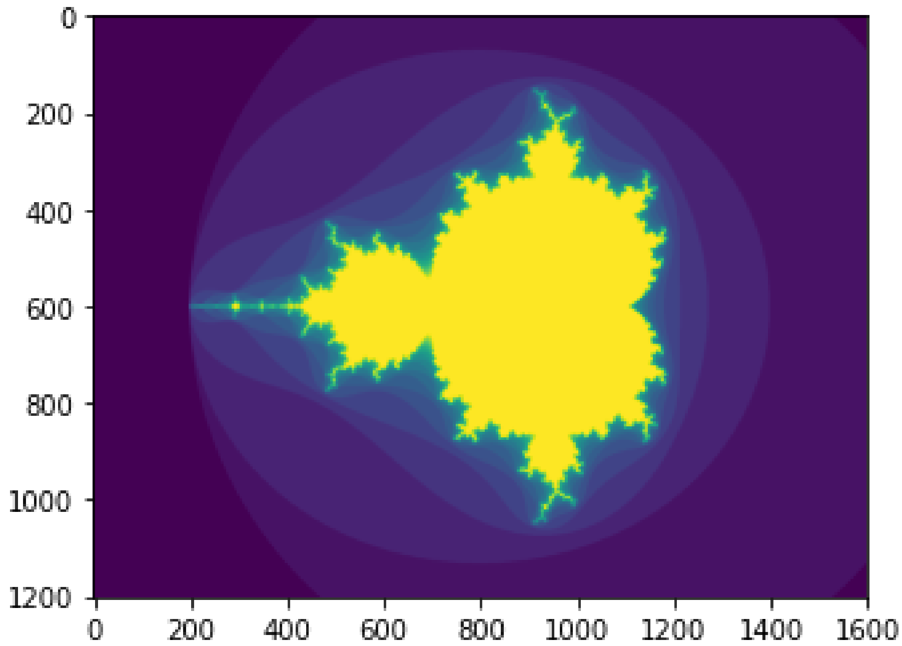


[14]:

```
def mandelbrot(h, w, maxit=20, r=2):

    """Returns an image of the Mandelbrot fractal of size (h,w)."""
```

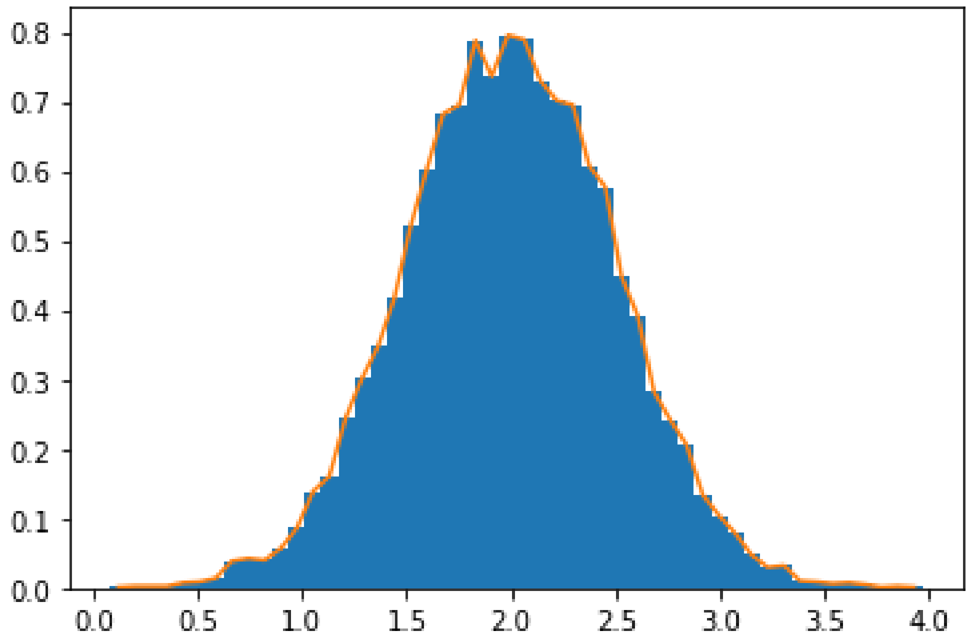
```
    x = np.linspace(-2.5, 1.5, 4*h+1)
    y = np.linspace(-1.5, 1.5, 3*w+1)
    A, B = np.meshgrid(x, y)
    C = A + B*1j
    z = np.zeros_like(C)
    divtime = maxit + np.zeros(z.shape, dtype=int)

    for i in range(maxit):
        z = z**2 + C
        diverge = abs(z) > r                    # who is diverging
        div_now = diverge & (divtime == maxit)  # who is diverging now
        divtime[div_now] = i                    # note when
        z[diverge] = r                          # avoid diverging too much

    return divtime
plt.clf()
plt.imshow(mandelbrot(400, 400))
```

[14]: <matplotlib.image.AxesImage at 0x7f377f87da30>



[15]:
```
import numpy as np

rg = np.random.default_rng(1)

import matplotlib.pyplot as plt

# Build a vector of 10000 normal deviates with variance 0.5^2 and mean 2

mu, sigma = 2, 0.5

v = rg.normal(mu, sigma, 10000)

# Plot a normalized histogram with 50 bins

plt.hist(v, bins=50, density=True)

# Compute the histogram with numpy and then plot it

(n, bins) = np.histogram(v, bins=50, density=True)  # NumPy version (no plot)

plt.plot(.5 * (bins[1:] + bins[:-1]), n)
```

[15]:
```
[<matplotlib.lines.Line2D at 0x7f377f7faee0>]
```



[17]:

```python
# Inner Product

A = np.array([0, 1])
B = np.array([3, 4])

print(np.inner(A, B))
# Output : 4
```

```
4
```

[18]:

```python
# Outter Product

A = np.array([0, 1])
B = np.array([3, 4])

print(np.outer(A, B))
```

```
[[0 0]
 [3 4]]
```

[19]:

```python
# Computes Eigenvalues And Right EigenVectors
# Of A Square Matrix
vals, vecs = np.linalg.eig([[1 , 2], [2, 1]])
print(vals)
print(vecs)
```

```
[ 3. -1.]
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

1.2 All Hacker Rank Numpy Problems

- Arrays
- Transpose
- Concatenate
- Shape
- Array Mathematics
- Eye and Identity
- Zeros and Ones
- Floor, Ceil, Rint -> Rint mean round to nearest int element-wise
- Min Max
- Sum
- Mean, Variance, Standard Deviation
- Dot and Cross Products
- Inner and Outer Products
- Polynomials
- Linear Algebra

[ ]:
```
# Arrays
def arrays(arr):
    return numpy.array(arr[::-1], dtype=float)

arr = "1 2 3 4 -8 -10".strip().split(' ')
result = arrays(arr)
print(result)
```

[ ]:
```
# Transpose
testCases = int(input().split()[0])

arr = numpy.array([input().split() for _ in range(testCases)], dtype=int)

print(arr.transpose())
print(arr.flatten())
```

[ ]:
```
# On the transpose problem, if you wanted to
# populate row indices first
rows, cols = map(int, input().split())
arr = [[] for _ in range(rows)]

for colIdx in range(cols):
    tmpVect = list(map(int, input().split()))
    idx = 0
    for rowIdx in range(rows):
        arr[rowIdx].append(tmpVect[idx])
```

```python
        idx += 1

arr = numpy.array(arr, dtype=int)

print(numpy.transpose(arr))
print(arr.flatten())
```

[ ]:
```python
# Concatenate
N, M, P = map(int, input().split())

A = numpy.array([list(map(int, input().split())) for _ in range(N)], dtype=int)
B = numpy.array([list(map(int, input().split())) for _ in range(M)], dtype=int)

C = numpy.concatenate((A, B), axis = 0)

print(C)
```

[ ]:
```python
# Shape
numpy.array(input().split(), dtype=int).reshape(3,3)
```

[ ]:
```python
# Array Mathematics
N, M = map(int, input().split())

A = numpy.array([input().split() for _ in range(N)], dtype=int)
B = numpy.array([input().split() for _ in range(N)], dtype=int)

print(numpy.add(A, B))
print(numpy.subtract(A, B))
print(numpy.multiply(A, B))
print(A // B)
print(numpy.mod(A, B))
print(A ** B)
```

[ ]:
```python
# Eye and Identity
# K = 0 equals diagonal, K > 0 == upper T,
# K < 0 == lower T
numpy.set_printoptions(legacy='1.13')

print(numpy.eye(*map(int, input().split())))
```

[ ]:
```python
# Zeros and Ones
args = tuple(map(int, input().split()))

print(numpy.zeros(args, dtype=int))
print(numpy.ones(args, dtype=int))
```

```
[ ]: # Floor, Ceil, Rint -> Rint mean round to nearest int element-wise
numpy.set_printoptions(legacy='1.13')

A = numpy.array(input().split(), dtype=float)
print(numpy.floor(A))
print(numpy.ceil(A))
print(numpy.rint(A))
```

```
[ ]: # Min Max
print(max(numpy.min(numpy.array([input().split() for _ in range(list(map(int,␣
↪input().split()))[0])], dtype=int), axis = 1)))
```

```
[ ]: # Sum
numpy.prod(numpy.sum(numpy.array([input().split() for _ in range(list(map(int,␣
↪input().split()))[0])], dtype=int), axis=0))
```

```
[ ]: # Mean, Variance, Standard Deviation
A = numpy.array([input().split() for _ in range(list(map(int, input().
↪split()))[0])], dtype=int)

print(numpy.round_(numpy.mean(A, axis = 1), 11))
print(numpy.round_(numpy.var(A, axis = 0), 11))
print(numpy.round_(numpy.std(A, axis = None), 11))
```

```
[ ]: # Dot and Cross Products
N = int(input().strip())

print(numpy.dot(numpy.array([input().split() for _ in range(N)], dtype=int),␣
↪numpy.array([input().split() for _ in range(N)], dtype=int)))
```

```
[ ]: # Inner and Outer Products
vectA = numpy.array(input().split(), dtype=int)
vectB = numpy.array(input().split(), dtype=int)

print(numpy.inner(vectA, vectB))
print(numpy.outer(vectA, vectB))
```

```
[ ]: # Polynomials
print(np.polyval(np.array(input().split(),dtype=float),float(input())))
```

```
[ ]: # Linear Algebra
print(numpy.round_(numpy.linalg.det(numpy.array([input().split() for _ in␣
↪range(int(input().strip()))], dtype=float)), 2))
```

1.3 More Later

- Vector Stacking: row_stack, vstack, hstack

- Vector Spliting: hsplit, vsplit
- Advanced Broadcasting
- Combine Different Vectors: the ix_() function

Arrays

August 17, 2022

1 Arrays

1.1 Important Points

- Python and C are 0-indexed
- Buffer overflows are common -> no bounds checking
  - That's why a large proportion of C functions from ANSI 89 are outdated -> use secure functions like snprintf, etc.
- Three ways to index in Python
  - Indexing

```
arr[idx]
```

  - Slicing

```
arr[start:stop]
arr[start:]
arr[:stop]
arr[:] -> deep copy
arr[start:stop:stride]
arr[-start:]
arr[-start:-stop]
```

  - Slicing (Nth Dimensions)

```
arr[1D][2D]..[NthD]
```

- Important functions and styles
  - Length

```
len(arr)
  - Use
  ```
  for idx in range(len(arr)):
      pass
  ```
```

  - Enumerate
    * Use

```
for idx, val in enumerate(arr):
    pass
```

  - Insertion
    * Use -> at idx. Front = 0, Back = -1

```
arr.insert(idx, val)
```

  - List comprehension -> famous one-liners
    * Used with sum a lot

```
return sum([num for num in arr])
```

  * Check one condition -> could use filter instead

```
return sum([1 for num in arr if num == 1])
```

  * Two decision points

```
return sum([1 if num % 2 == 0 else 0 for num in arr])
```

  * Used with sorting a lot -> 2D lambda sorting

```
return sorted(arr, key = lambda x: [x[1], x[0], reverse = True)
```

- Techniques
  - Single Pass
  - Back Populating Array
  - Two Pointer Array Overwrites -> Skipbeats
  - Sliding Window (Very Important)
  - Set Difference (Union, Intersection, Complement)

1.2 Single Pass

- Max Consecutive Ones

```
def findMaxConsecutiveOnes(nums: List[int]) -> int:
    max_ones = 0
    temp_ones = 0

    for num in nums:
        if num == 1:
            temp_ones += 1
        else:
            max_ones = max(temp_ones, max_ones)
            temp_ones = 0

    return max(temp_ones, max_ones)
```

1.3 One Liner List Comprehension

- Find Numbers

```
def findNumbers(nums: List[int]) -> int:
    return sum([1 for x in nums if len(str(x)) % 2 == 0])
```

1.4 One Liner Sorted List Comprehension (No Lambda)

- Sorted Squares

```
def sortedSquares(nums: List[int]) -> List[int]:
    return sorted([x**2 for x in nums])
```

1.5 Back Population Using Two Sorted Arrays

- Merge
- One array has extra storage -> common when securing passwords or memory buffers
- Operational assets/(weapon systems) need to maintain constant memory integrity in kernel of cached buffers.

- Unallocated space in kernel buffer means possibility for cyber exploitation.
- Write garbage (white noise/gaussian noise) after merging.

```
def merge(nums1: List[int], m: int, nums2: List[int], n: int) -> None:
    # Merge into nums1.
    # Size of nums1 is m + n.

    finalPtr = m + n - 1
    ptr1 = m-1
    ptr2 = n-1

    while ptr2 > -1:
        if ptr1 > -1 and nums2[ptr2] < nums1[ptr1]:
            nums1[finalPtr] = nums1[ptr1]
            ptr1 -= 1
        else:
            nums1[finalPtr] = nums2[ptr2]
            ptr2 -= 1
        finalPtr -= 1
```

1.6 Skipbeats

- Remove Element or Remove Duplicates

```
def removeElement(nums: List[int], val: int) -> int:
    skipPtr = 0

    for idx in range(len(nums)):
        if nums[idx] != val:
            nums[skipPtr] = nums[idx]
            skipPtr += 1

    return skipPtr
```

1.7 Sort By Parity

```
def sortArrayByParity(nums: List[int]) -> List[int]:
    arr = []

    for num in nums:
        if num % 2 == 0:
            arr.insert(0, num)
        else:
            arr.append(num)

    return arr
```

1.8 Sliding Window

- Find Max Consecutive Ones

- Numerous ways to implement -> dynamic programming section will have more examples

```
def maxOnesArr(nums: List[int]) -> List[int]:
    arr = []
    temp_ones = 0

    for num in nums:
        if num == 1:
            temp_ones += 1
        else:
            if len(arr):
                arr[-1] += temp_ones + 1
            arr.append(temp_ones)
            temp_ones = 0

    if temp_ones != 0:
        if len(arr):
            arr[-1] += temp_ones + 1
        arr.append(temp_ones)
    if nums[-1] == 0:
        arr[-1] += 1

    return arr
```

```
def findMaxConsecutiveOnes(nums: List[int]) -> int:
    return max(max(self.maxOnesArr(nums)), 1)
```

1.9 Set Difference

- Find Disappeared Numbers

```
def findDisappearedNumbers(nums: List[int]) -> List[int]:
    return set([x for x in range(1, len(nums) + 1)]) - set(nums)
```

[ ]:

ArraysAndStrings

August 17, 2022

1 Arrays & Strings

1.1 Important Points

- Techniques
  - Bisect
  - Adding One To A Binary (Important)
  - Adding Binaries (Important) -> XOR, AND, OR
  - Grepping It (Important) -> Tons Of Variations
  - Longest Common Prefix
  - Move Values To End In-Place
  - Pascal
  - Two Sum
  - Contiguous Subarray Sum (Important)
- Party/Cryptology Tricks
  - Spiral Order (Matrix Section Later)
  - Diagonal Traverse (Matrix Section Later)

1.2 Bisect

- Find Pivot Index

```
def pivotIndex(nums: List[int]) -> int:
    S = sum(nums)
    leftsum = 0
    for i, x in enumerate(nums):
        if leftsum == (S - leftsum - x):
            return i
        leftsum += x
    return -1
```

1.3 Adding One To A Binary

- Plus One
- Important when we look at linked lists

```
def plusOne(digits: List[int]) -> List[int]:
    backPtr = len(digits) - 1
    carry = 1
```

```
    while backPtr > -1:
        if carry == 1:
            if digits[backPtr] + 1 == 10:
                digits[backPtr] = 0
                carry = 1
            else:
                digits[backPtr] += 1
                carry = 0
            backPtr -= 1
        else:
            break

    if carry == 1:
        digits.insert(0, 1)

    return digits
```

1.4 Adding Binaries

- Add Binary

```
def finish(digits: List[int], backPtr: int, result: str, carry: str) -> str:
    # Back population
    while backPtr > -1:
        if carry == 1:
            if digits[backPtr] == '1':
                result += "0"
                carry = 1
            else:
                result += "1"
                carry = 0
        else:
            result += digits[backPtr]
        backPtr -= 1

    if carry == 1:
        result += "1"

    return result

def addBinary(a: str, b: str) -> str:
    length = min(len(a), len(b))
    aPtr = len(a)-1
    bPtr = len(b)-1
    result = ""
    carry = False

    while aPtr > -1 and bPtr > -1:
        if a[aPtr] == '1' and b[bPtr] == '1' and carry == False or (((a[aPtr] == '1') ^ (b[bPt
```

```
            result += "0"
            carry = True
        elif a[aPtr] == '1' and b[bPtr] == '1' and carry == True:
            result += "1"
            carry = True
        elif a[aPtr] == '1' or b[bPtr] == '1' or carry == True:
            result += "1"
            carry = False
        else:
            result += "0"
            carry = False

        aPtr -= 1
        bPtr -= 1
    if aPtr > -1:
        return self.finish(a, aPtr, result, int(carry))[::-1]
    elif bPtr > -1:
        return self.finish(b, bPtr, result, int(carry))[::-1]
    return result[::-1] if carry == False else "1" + result[::-1]
```

1.5 Grepping It

- Implement StrStr

```
def strStr(haystack: str, needle: str) -> int:
    # Two pointer
    haystackPtr = 0
    needleLength = len(needle)
    length = len(haystack) - needleLength + 1

    while haystackPtr < length:
        needlePtr = 0
        while needlePtr < needleLength:
            if haystack[haystackPtr + needlePtr] == needle[needlePtr]:
                needlePtr += 1
            else:
                break

        if needlePtr == needleLength:
            return haystackPtr

        haystackPtr += 1

    return -1
```

1.6 Longest Common Prefix

```
def strStr(haystack: str, needle: str) -> int:
    # Two pointer
```

```
    haystackPtr = 0
    needleLength = len(needle)
    length = len(haystack) - needleLength + 1
    needlePtr = 0

    while needlePtr < needleLength:
        if haystack[haystackPtr + needlePtr] == needle[needlePtr]:
            needlePtr += 1
        else:
            break

    return needle[:needlePtr]

def longestCommonPrefix(strs: List[str]) -> str:
    strs.sort(key = lambda x: len(x))
    if len(strs) == 1:
        return strs[0]

    commonPrefix = self.strStr(strs[1], strs[0])
    for idx in range(1, len(strs)-1):
        commonPrefix = min(self.strStr(strs[idx+1], strs[idx]), commonPrefix)

    return commonPrefix
```

1.7 Move Values To End In-Place

```
def moveZeroes(nums: List[int]) -> None:
    popPtr = 0
    actualPtr = 0
    length = len(nums)

    while popPtr < length:
        searching = True
        while searching and actualPtr < length:
        if nums[actualPtr] != 0:
            nums[popPtr] = nums[actualPtr]
            searching = False
        else:
            actualPtr += 1
        if searching == True:
            nums[popPtr] = 0

        popPtr += 1
        actualPtr += 1
```

1.8 Pascal

- Pascals Triangle

```
def generate(numRows: int) -> List[List[int]]:
    baseCase = [[1]]
    baseCase2 = [[1],[1,1]]
    if numRows == 1:
        return baseCase
    elif numRows == 2:
        return baseCase2
    else:
        row = 2
        while row < numRows:
            newRow = []
            for idx in range(len(baseCase2[row-1])-1):
                newRow.append(baseCase2[row-1][idx] + baseCase2[row-1][idx+1])
            newRow.insert(0, 1)
            newRow.append(1)
            baseCase2.append(newRow)
            row += 1
        return baseCase2
```

1.9 Two Sum

```
def twoSum(numbers: List[int], target: int) -> List[int]:
    # Optimize with binary search later
    newNumbers = sorted(list(set(numbers)))

    for idx in range(len(newNumbers)):
        x = newNumbers[idx]
        if x + x == target:
            index = numbers.index(x)
            return [index + 1, index + 2]
        for idx2 in range(idx+1, len(newNumbers)):
            y = newNumbers[idx2]
            if x + y == target:
                return [numbers.index(x) + 1, numbers.index(y) + 1]
            elif x + y > target:
                break
```

1.10 Contiguous Subarray Sum

- Minimum Size Subarray Sum

```
def minSubArrayLen(target: int, nums: List[int]) -> int:
    n = len(nums)
    ans = 10000000
    left = 0
    _sum = 0

    for i in range(n):
        _sum += nums[i]
```

```python
        while (_sum >= target):
            ans = min(ans, i + 1 - left)
            _sum -= nums[left]
            left += 1

    return ans if ans != 10000000 else 0
```

[ ]:

Hash Table

August 17, 2022

1 Hash Table

- Two kinds: hash set and hash map
  - Hash set -> no repeated values
  - Hash map (Python Dictionaries) -> key, val pairs
    - freq = {}
    - freq.keys()
    - freq.items()
    - for key, val in freq.items():
      - pass
    - freq.clear()
- Key component: hash function (used for mapping key to bucket)
  - Range of key values
  - Number of buckets
- If N is variable or large, use a height-balanced binary search tree instead
- Techniques
  - Two Sum
  - Intersection Of Two Arrays
  - Is Isomorphic
  - Logger

2 Hash Set Python Function

[1]:
```python
hashset = set()
hashset.add(3)
hashset.add(2)
hashset.remove(3)

if (3 not in hashset):
    print("Not here, finders keepers!")

hashset.clear()
```

```
Not here, finders keepers!
```

3 Single Number

[2]:
```
# Non hash map solution
def singleNumber(nums: list[int]) -> int:
    return sorted({num: nums.count(num) for num in set(nums)}.items(), \
                  key = lambda x: x[1])[0][0]
```

4 Intersection of Two Arrays

[3]:
```
# Set operations solution
def intersection(nums1: list[int], nums2: list[int]) -> list[int]:
    return set(nums1).intersection(set(nums2))
```

4.1 Two Sum

[4]:
```
# Solution
def twoSum(nums: list, target: int):
    buffer_dictionary = {}

    for i in range(len(nums)):
        if nums[i] in buffer_dictionary:
            return [buffer_dictionary[nums[i]], i]
        else:
            buffer_dictionary[target - nums[i]] = i
```

5 Is Isomorphic

[5]:
```
def isIsomorphic(s: str, t: str):
    s2t, t2s = {}, {}
    for i in range(len(s)):
        if s[i] in s2t and s2t[s[i]] != t[i]:
            return False
        if t[i] in t2s and t2s[t[i]] != s[i]:
            return False
        s2t[s[i]] = t[i]
        t2s[t[i]] = s[i]
    return True
```

5.1 First Unique Char

[6]:
```
# One Liner -> Dont do this!!!
def firstUniqChar(s: str) -> int:
    return sorted({letter: [s.count(letter), s.index(letter)] \
                   for letter in set(s)}.items(), \
```

```
                  key = lambda x: [x[1][0], x[1][1]])[0][1][1] \
if sorted({letter: [s.count(letter), s.index(letter)] for letter \
           in set(s)}.items(), \
          key = lambda x: [x[1][0], x[1][1]])[0][1][0] == 1 else -1
```

5.2 Logger

[7]:
```
class Logger:

    def __init__(self):
        self.mapper = {}

    def shouldPrintMessage(self, timestamp: int, message: str) -> bool:
        if message in self.mapper.keys() and timestamp < self.mapper[message]:
            return False
        else:
            self.mapper[message] = timestamp + 10
            return True
```

[8]:
```
class MyHashSet:
    def eval_hash(self, key):
        return ((key*1031237) & (1<<20) - 1)>>5

    def __init__(self):
        self.arr = [[] for _ in range(1<<15)]

    def add(self, key: int) -> None:
        t = self.eval_hash(key)
        if key not in self.arr[t]:
            self.arr[t].append(key)

    def remove(self, key: int) -> None:
        t = self.eval_hash(key)
        if key in self.arr[t]:
            self.arr[t].remove(key)

    def contains(self, key: int) -> bool:
        t = self.eval_hash(key)
        return key in self.arr[t]
```

6 Hash Map With Linked List

[9]:
```
class ListNode:
    def __init__(self, key, val, nxt):
        self.key = key
        self.val = val
```

```python
        self.next = nxt
class MyHashMap:
    def __init__(self):
        self.size = 19997
        self.mult = 12582917
        self.data = [None for _ in range(self.size)]
    def hash(self, key):
        return key * self.mult % self.size
    def put(self, key, val):
        self.remove(key)
        h = self.hash(key)
        node = ListNode(key, val, self.data[h])
        self.data[h] = node
    def get(self, key):
        h = self.hash(key)
        node = self.data[h]
        while node:
            if node.key == key: return node.val
            node = node.next
        return -1
    def remove(self, key: int):
        h = self.hash(key)
        node = self.data[h]
        if not node: return
        if node.key == key:
            self.data[h] = node.next
            return
        while node.next:
            if node.next.key == key:
                node.next = node.next.next
                return
            node = node.next
```

LinkedLists

August 17, 2022

1 Linked Lists

1.1 Important Points

- Single and double linked lists
- Can put more information in the node -> called reference field
  - Memory marks (Mark-Sweep)
  - Dimensional data
  - Permissions (reader writer locks)
- Add operation (middle)
  1. Initialize new node
  2. Link new node's nextPtr to the next node
  3. Link prev node's nextPtr to new node
- Add operation (head)
  1. Initialize new node
  2. Link new node's nextPtr to head node
- Add operation (tail)
  1. Initialize new node
  2. Link prev node's nextPtr to new node
- Delete operation (middle)
  1. Point prev node's nextPtr to new node's nextPtr
- Delete operation (head)
  1. Assign the curr node to head
- Delete operation (tail)
  1. Unlink prev node's nextPtr (assign to None)
- Techniques
  - Singly Linked List (Important)
  - Simple Cycle
  - Intersection Point
  - Remove Nth Node

[1]:
```python
# Code Used In Examples
class ListNode:
    def __init__(self, val):
        self.val = val
        self.next = None
```

1.2 Simple Cycle

[2]:
```python
def hasCycle(head: ListNode) -> bool:
    if head is None:
        return False

    slowPtr = head
    fastPtr = head

    while fastPtr.next is not None and fastPtr.next.next is not None:
        fastPtr = fastPtr.next.next
        slowPtr = slowPtr.next
        if fastPtr == slowPtr:
            return True

    return False
```

1.3 Detect Cycle Position

[3]:
```python
# Solution 1
def detectCycle(head: ListNode) -> ListNode:
    slow = fast = head
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
        if slow == fast: break
    else: return None  # if not (fast and fast.next): return None
    while head != slow:
        head, slow = head.next, slow.next
    return head

# Solution 2
def detectCycle(head: ListNode) -> ListNode:
    if head is None:
        return None

    slowPtr = head
    fastPtr = head

    while fastPtr.next is not None and fastPtr.next.next is not None:
        fastPtr = fastPtr.next.next
        slowPtr = slowPtr.next
        if fastPtr == slowPtr:
            _list = []
            fastPtr = fastPtr.next
            _list.append(slowPtr)
            _list.append(fastPtr)
            while fastPtr != slowPtr:
```

```
                fastPtr = fastPtr.next.next
                slowPtr = slowPtr.next
                _list.append(slowPtr)
                _list.append(fastPtr)
            newPtr = head
            while newPtr not in _list:
                newPtr = newPtr.next
            return newPtr
    return None
```

1.4 Intersection Point

[4]:
```
# Solution 1
def getIntersectionNode(headA, headB):
    if headA is None or headB is None:
        return None

    pa = headA # 2 pointers
    pb = headB

    while pa is not pb:
        # if either pointer hits the end, switch head and continue the second␣
↪traversal,
        # if not hit the end, just move on to next
        pa = headB if pa is None else pa.next
        pb = headA if pb is None else pb.next

    return pa # only 2 ways to get out of the loop, they meet or the both hit␣
↪the end=None

# Solution 2
def getLength(shadow):
    if shadow is None:
        return 0

    runner = shadow
    count = 1

    while runner.next is not None:
        count += 1
        runner = runner.next

    return count

def getIntersectionNode(headA, headB):
    """
    :type head1, head1: ListNode
```

```
    :rtype: ListNode
    """
    if headA is None or headB is None:
        return None

    aLength = self.getLength(headA)
    bLength = self.getLength(headB)

    while aLength and bLength:
        if aLength > bLength:
            headA = headA.next
            aLength -= 1
        elif bLength > aLength:
            headB = headB.next
            bLength -= 1
        else:
            if headA == headB:
                return headA
            if headA.next is not None:
                headA = headA.next
                aLength -= 1
            if headB.next is not None:
                headB = headB.next
                bLength -= 1
    return None
```

1.5 Remove Nth From End

[5]:
```
# Solution 1
def removeNthFromEnd(head: ListNode, n: int) -> ListNode:
    fast = slow = head
    for _ in range(n):
        fast = fast.next
    if not fast:
        return head.next
    while fast.next:
        fast = fast.next
        slow = slow.next
    slow.next = slow.next.next
    return head

# Solution 2
def getLength(shadow: ListNode) -> int:
    if shadow is None:
        return 0

    runner = shadow
```

```
    count = 1

    while runner.next is not None:
        count += 1
        runner = runner.next

    return count

def removeNthFromEnd(head: ListNode, n: int) -> ListNode:
    runner = head
    length = self.getLength(runner)

    if length == 1:
        head = None
        return head

    length -= n

    shadow = head
    prev = ListNode(-1000)
    count = 0

    while shadow.next is not None and count < length:
        prev = shadow
        shadow = shadow.next
        count += 1

    if shadow.next is None:
        prev.next = None
    elif prev.val == -1000:
        if head.next is not None:
            prev = head.next
            head = prev
        else:
            head = None
    else:
        prev.next = shadow.next

    return head
```

1.6 Reverse Linked List

[6]:
```
# Solution 1
def reverseList(head: ListNode) -> ListNode:
    prev = None
    curr = head
```

```python
    while curr:
        next = curr.next
        curr.next = prev
        prev = curr
        curr = next

    return prev

# Solution 2
def reverseList(head: ListNode) -> ListNode:
    new = None
    curr = head

    while curr is not None:
        node = ListNode(curr.val)
        node.next = new
        new = node
        curr = curr.next

    return new
```

1.7 Remove Linked List Elements

[7]:
```python
# Solution 1
def removeElements(head: ListNode, val: int) -> ListNode:
    dummy_head = ListNode(-1)
    dummy_head.next = head

    current_node = dummy_head
    while current_node.next != None:
        if current_node.next.val == val:
            current_node.next = current_node.next.next
        else:
            current_node = current_node.next

    return dummy_head.next

# Solution 2
def removeElements(head: ListNode, val: int) -> ListNode:
    shadow = head
    prev = ListNode(-1)
    ans = prev

    while shadow is not None:
        if shadow.val == val:
            if shadow.next is not None:
                prev.next = shadow.next.next
```

```
            else:
                prev.next = None
        else:
            prev.next = shadow
            prev = prev.next

        if shadow.next is not None:
            shadow = shadow.next
        else:
            break

    return ans.next if ans.next is not None else None
```

1.8 Odd Even List

[8]:
```
# Solution 1

def oddEvenList(head: ListNode) -> ListNode:
    if not head:
        return head
    odd = head
    even = head.next
    eHead = even
    while even and even.next:
        odd.next = odd.next.next
        even.next = even.next.next
        odd = odd.next
        even = even.next
    odd.next = eHead
    return head

# Solution 2

def oddEvenList(head: ListNode) -> ListNode:
    even = ListNode(-1)
    evenShadow = even
    odd = ListNode(-1)
    oddShadow = odd
    oddRunner = True

    while head is not None:
        if oddRunner:
            odd.next = ListNode(head.val)
            odd.next.next = None
            odd = odd.next
        else:
            even.next = ListNode(head.val)
```

```python
            even.next.next = None
            even = even.next
        head = head.next
        oddRunner = not oddRunner

    odd.next = evenShadow.next
    return oddShadow.next
```

1.9 Palindrome

[9]:
```python
def isPalindrome(head: ListNode) -> None:
    rev = None
    slow = fast = head
    while fast and fast.next:
        fast = fast.next.next
        rev, rev.next, slow = slow, rev, slow.next
    if fast:
        slow = slow.next
    while rev and rev.val == slow.val:
        slow = slow.next
        rev = rev.next
    return not rev
```

1.10 Merge Two Sorted Lists

[10]:
```python
# Solution 1

def mergeTwoLists(list1: ListNode, list2: ListNode) -> ListNode:
    cur = dummy = ListNode()
    while list1 and list2:
        if list1.val < list2.val:
            cur.next = list1
            list1, cur = list1.next, list1
        else:
            cur.next = list2
            list2, cur = list2.next, list2

    if list1 or list2:
        cur.next = list1 if list1 else list2

    return dummy.next

# Solution 2

def mergeTwoLists(list1: ListNode, list2: ListNode) -> ListNode:
    shadow = ListNode(-1)
```

```python
    ans = shadow

    while list1 is not None and list2 is not None:
        if list1.val > list2.val:
            shadow.next = ListNode(list2.val)
            list2 = list2.next
        else:
            shadow.next = ListNode(list1.val)
            list1 = list1.next
        shadow = shadow.next

    while list1 is not None:
        shadow.next = ListNode(list1.val)
        shadow = shadow.next
        list1 = list1.next
    while list2 is not None:
        shadow.next = ListNode(list2.val)
        shadow = shadow.next
        list2 = list2.next

    return ans.next
```

1.11 Add Two Numbers

[11]:
```python
# Solution 1

def addTwoNumbers(l1: ListNode, l2: ListNode) -> ListNode:
    carry = 0
    root = n = ListNode(0)
    while l1 or l2 or carry:
        v1 = v2 = 0
        if l1:
            v1 = l1.val
            l1 = l1.next
        if l2:
            v2 = l2.val
            l2 = l2.next
        carry, val = divmod(v1+v2+carry, 10)
        n.next = ListNode(val)
        n = n.next
    return root.next

# Solution 2

def addTwoNumbers(l1: ListNode, l2: ListNode) -> ListNode:
    carry = 0
    shadow = ListNode(-1)
```

```
    ans = shadow

    while l1 and l2:
        store = l1.val + l2.val + carry
        l1 = l1.next
        l2 = l2.next
        carry = store // 10
        shadow.next = ListNode(store % 10)
        shadow = shadow.next

    while l1 or l2:
        if l1:
            store = l1.val + carry
            l1 = l1.next
        else:
            store = l2.val + carry
            l2 = l2.next
        carry = store // 10
        shadow.next = ListNode(store % 10)
        shadow = shadow.next

    if carry:
        shadow.next = ListNode(1)

    return ans.next
```

1.12 Insert Into A Cyclic Sorted List

[12]:
```
# Solution 1
def insert(head: ListNode, insertVal: int) -> ListNode:

    #case1 - if head is None
    if head is None:
        node = Node()
        node.val = insertVal
        node.next = node
        return node

    #case 2a - if value is greater than max or lesser than min value of LL
    maximum = head
    while maximum.next!=head and maximum.val<=maximum.next.val:
        maximum=maximum.next
    minimum = maximum.next
    cur = minimum

    if insertVal>=maximum.val or insertVal<=minimum.val:
        node = Node(val=insertVal,next=minimum)
```

```
        maximum.next = node
    #case 2b - if value is in range of max and min values
    else:
        while cur.next.val<insertVal:
            cur=cur.next
        node = Node(val=insertVal,next=cur.next)
        cur.next=node
    return head

# Solution 2

def insert(head: ListNode, insertVal: int) -> ListNode:
    if head is None:
        new = ListNode(insertVal)
        new.next = new
        return new
    else:
        shadow = head
        prev = head
        while shadow.next != head and shadow.val < insertVal:
            prev = shadow
            shadow = shadow.next
        if shadow == head:
            new = ListNode(insertVal)
            new.next = head
            while shadow.next is not head:
                shadow = shadow.next
            shadow.next = new
            return new
        elif shadow.next == head:
            shadow.next = ListNode(insertVal)
            shadow.next.next = head
            return shadow.next
        else:
            prev.next = ListNode(insertVal)
            prev = prev.next
            prev.next = shadow
            return prev
```

1.13 Flatten a Multidimensional Doubly Linked List

[13]:

```
# Solution 1
def flatten(head: ListNode) -> ListNode:
    if not head: return None
    stack = [head]; p = None
    while stack:
        r = stack.pop()
```

```python
        if p:
            p.next,r.prev = r,p
        p = r
        if r.next:
            stack.append(r.next)
        if r.child:
            stack.append(r.child)
            r.child = None
    return head

# Solution 2
def getNodes(shadow: ListNode) -> ListNode:
    if shadow is None:
        return None

    runner = shadow

    while runner:
        if runner.child:
            tmp = self.getNodes(runner.child)
            hold = runner.next
            runner.next = tmp
            tmp.prev = runner
            runner.child = None
            while runner.next:
                runner = runner.next
            runner.next = hold
            if hold is not None:
                hold.prev = runner
        runner = runner.next
    return shadow

def flatten(head: ListNode) -> ListNode:
    ans = self.getNodes(head)

    return ans
```

1.14 Copy Random List

[14]:

```python
class Node:
    def __init__(self, x: int, next = None, random = None):
        self.val = int(x)
        self.next = next
        self.random = random

# Solution 1
def copyRandomList(head: Node) -> Node:
```

```python
    dic, prev, node = {}, None, head
    while node:
        if node not in dic:
            # Use a dictionary to map the original node to its copy
            dic[node] = Node(node.val, node.next, node.random)
        if prev:
            # Make the previous node point to the copy instead of the original.
            prev.next = dic[node]
        else:
            # If there is no prev, then we are at the head. Store it to return␣
↪later.
            head = dic[node]
        if node.random:
            if node.random not in dic:
                # If node.random points to a node that we have not yet␣
↪encountered, store it in the dictionary.
                dic[node.random] = Node(node.random.val, node.random.next, node.
↪random.random)
            # Make the copy's random property point to the copy instead of the␣
↪original.
            dic[node].random = dic[node.random]
        # Store prev and advance to the next node.
        prev, node = dic[node], node.next
    return head

# Solution 2
def copyRandomList(head: Node) -> Node:
    if head is None:
        return None
    shadow = head

    while shadow:
        node = Node(shadow.val)
        node.next = shadow.next
        node.random = shadow.random
        shadow.next = node
        shadow = shadow.next.next

    newShadow = head
    odd = True

    while newShadow:
        if not odd:
            if newShadow.random:
                newShadow.random = newShadow.random.next
        odd = not odd
        newShadow = newShadow.next
```

```
    newShadow2 = head.next
    ans = newShadow2

    while newShadow2:
        newShadow2.next = newShadow2.next.next if newShadow2.next and␣
↪newShadow2.next.next else None
        newShadow2 = newShadow2.next

    return ans
```

1.15 Rotate List

[15]:
```
def rotateRight(head: ListNode, k: int) -> ListNode:

    if not head:
        return None

    lastElement = head
    length = 1
    # get the length of the list and the last node in the list
    while ( lastElement.next ):
        lastElement = lastElement.next
        length += 1

    # If k is equal to the length of the list then k == 0
    # ElIf k is greater than the length of the list then k = k % length
    k = k % length

    # Set the last node to point to head node
    # The list is now a circular linked list with last node pointing to first␣
↪node
    lastElement.next = head

    # Traverse the list to get to the node just before the ( length - k )th␣
↪node.
    # Example: In 1->2->3->4->5, and k = 2
    #          we need to get to the Node(3)
    tempNode = head
    for _ in range( length - k - 1 ):
        tempNode = tempNode.next

    # Get the next node from the tempNode and then set the tempNode.next as None
    # Example: In 1->2->3->4->5, and k = 2
    #          tempNode = Node(3)
    #          answer = Node(3).next => Node(4)
    #          Node(3).next = None ( cut the linked list from here )
```

```
    answer = tempNode.next
    tempNode.next = None

    return answer
```

1.16 Singly Linked List

[16]:
```
class Node:

    def __init__(self, val):
        self.val = val
        self.next = None



class MyLinkedList:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.head = None
        self.size = 0

    def get(self, index):
        """
        Get the value of the index-th node in the linked list. If the index is␣
↪invalid, return -1.
        :type index: int
        :rtype: int
        """
        if index < 0 or index >= self.size:
            return -1

        if self.head is None:
            return -1

        curr = self.head
        for i in range(index):
            curr = curr.next
        return curr.val

    def addAtHead(self, val):
        """
        Add a node of value val before the first element of the linked list.
        After the insertion, the new node will be the first node of the linked␣
↪list.
```

```
        :type val: int
        :rtype: void
        """
        node = Node(val)
        node.next = self.head
        self.head = node

        self.size += 1

    def addAtTail(self, val):
        """
        Append a node of value val to the last element of the linked list.
        :type val: int
        :rtype: void
        """
        curr = self.head
        if curr is None:
            self.head = Node(val)
        else:
            while curr.next is not None:
                curr = curr.next
            curr.next = Node(val)

        self.size += 1

    def addAtIndex(self, index, val):
        """
        Add a node of value val before the index-th node in the linked list.
        If index equals to the length of linked list, the node will be appended␣
↪to the end of linked list.
        If index is greater than the length, the node will not be inserted.
        :type index: int
        :type val: int
        :rtype: void
        """
        if index < 0 or index > self.size:
            return

        if index == 0:
            self.addAtHead(val)
        else:
            curr = self.head
            for i in range(index - 1):
                curr = curr.next
            node = Node(val)
            node.next = curr.next
            curr.next = node
```

```python
            self.size += 1

    def deleteAtIndex(self, index):
        """
        Delete the index-th node in the linked list, if the index is valid.
        :type index: int
        :rtype: void
        """
        if index < 0 or index >= self.size:
            return

        curr = self.head
        if index == 0:
            self.head = curr.next
        else:
            for i in range(index - 1):
                curr = curr.next
            curr.next = curr.next.next

        self.size -= 1
```

Heaps

August 17, 2022

1 Heaps

- Important Points
  - Similar to a queue/stack but has a priority (quantum valum) associated with each node
  - Using a heap data structure allows both insertion and deletion to have a time complexity of O(log N). Retrieval is O(1)
  - Heap is a binary tree that is complete and has each node with a value greater than (or less than) the value of its child nodes (max/min heaps)
- Techniques
  - Heap Sort (Selection Sort Around Min/Max Value)
  - Top-K
  - Kth Element
  - Furthest Building Can Travel (DP Section Later)
  - Interval Scheduling (Graph Section Later)

1.1 Python Module (heapq)

[1]:
```python
import heapq

# minHeap = [3,2,1]
# heapq.heapify(minHeap)

# Construct max heap
# Time: O(N), Space: O(N)
maxHeap = [-x for x in [1,3,2,1]]
heapq.heapify(maxHeap)

# Insert
# Time: O(log N), Space: O(1)
heapq.heappush(maxHeap, -1 * 10)

# Get top element
# Time: O(1), Space: O(1)
-1 * maxHeap[0]

# Delete top element
# Time: O(log N), Space: O(1)
item = heapq.heappop(maxHeap)
```

1.2 K Largest

[2]:

```python
# Extra
def partition(nums, l, r):
    low = l
    while l < r:
        if nums[l] < nums[r]:
            nums[l], nums[low] = nums[low], nums[l]
            low += 1
        l += 1
    nums[low], nums[r] = nums[r], nums[low]
    return low

# Solution 2
def findKthLargest(nums: list[int], k: int) -> int:
    nums = [-x for x in nums]
    heapq.heapify(nums)
    count = 0
    item = None

    while count < k:
        count += 1
        item = -1 * heapq.heappop(nums)

    return item
```

1.3 Top K Frequent

[3]:

```python
# Solution
def topKFrequent(nums: list[int], k: int) -> list[int]:
        hashmap = {}
        for num in nums:
            if num in hashmap:
                hashmap[num] += 1
            else:
                hashmap[num] = 1
        heap = []
        for key in hashmap:
            heapq.heappush(heap, (-hashmap[key], key))

        res = []
        for _ in range(k):
            popped = heapq.heappop(heap)
            res.append(popped[1])

        return res

# Non heap solution
def topKFrequent(nums: list[int], k: int) -> list[int]:
```

```
    freq = {}

    for num in nums:
        if num not in freq.keys():
            freq[num] = 1
        else:
            freq[num] += 1

    return [x[0] for x in sorted(freq.items(), key = lambda x: x[1], reverse =␣
↪True)][:k]
```

1.4 Kth Largest From Data Stream

[4]:
```
# Solution
class KthLargest:
    def __init__(self, k: int, nums: list[int]):
        self.heap = []
        self.k = k
        for i in nums:
            if len(self.heap) < k:
                heapq.heappush(self.heap,i)
            else:
                if i > self.heap[0]:
                    heapq.heappushpop(self.heap,i)

    def add(self, val: int) -> int:
        if len(self.heap) < self.k:
            heapq.heappush(self.heap,val)
        else:
            if val > self.heap[0]:
                heapq.heappushpop(self.heap,val)
        return self.heap[0]

# Non heap solution
class KthLargest:

    def __init__(self, k: int, nums: list[int]):
        self.k = k
        self.nums = sorted(nums, reverse = True)[:self.k]

    def add(self, val: int) -> int:
        if len(self.nums) == 0:
            self.nums = [val]
        elif val > self.nums[-1] or len(self.nums) < self.k:
            self.nums = sorted(self.nums + [val], reverse = True)[:self.k]
        return self.nums[-1]
```

1.5 Last Stone

[5]:
```
# Solution
def lastStoneWeight(stones: list[int]) -> int:
    stones = [-x for x in stones]
    heapq.heapify(stones)

    while len(stones) > 1:
        item1, item2 = heapq.heappop(stones), heapq.heappop(stones)
        if item1 != item2:
            heapq.heappush(stones, -1 * ( (-item1) - (-item2) ) )

    return -stones[0] if len(stones) else 0
```

1.6 K Weakest Rows

[6]:
```
# Solution
def kWeakestRows(mat: list[list[int]], k: int) -> list[int]:
    self.scores = []
    heap = heapq.heapify(self.scores)

    for rowIdx in range(len(mat)):
        score = sum(mat[rowIdx][:])
        heapq.heappush(self.scores, (score, rowIdx))

    return [heapq.heappop(self.scores)[1] for _ in range(k)]

# Non heap solution
def kWeakestRows(mat: list[list[int]], k: int) -> list[int]:
    return sorted(range(len(mat)), key=lambda x: sum(mat[x]))[:k]
```

1.7 Kth Smallest

[7]:
```
# Binary Search
# Time Complexity: O( Nlog(max-min) * log(max-min) )
# Space Complexity: O(1)
def kthSmallest(self, matrix: list[list[int]], k: int) -> int:

    m = len(matrix)
    n = len(matrix[0])

    def count(m):
        c = 0
        i = n-1
        j = 0

        while i >= 0 and j < n:
```

```python
            if matrix[i][j] > m:
                i -= 1
            else:
                c += i+1
                j += 1
        return c


    low = matrix[0][0]
    high = matrix[n-1][n-1]

    while low <= high:
        m = (low+high)//2
        cnt = count(m)
        if cnt < k:
            low = m + 1
        else:
            cnt1 = count(m-1)
            if cnt1 < k:
                return m
            high = m-1
    return 0

# Heap solution
def kthSmallest(self, matrix: list[list[int]], k: int) -> int:
    heap = []
    heapq.heapify(heap)

    for x in range(len(matrix)):
        for y in range(len(matrix[0])):
            heapq.heappush(heap, matrix[x][y])

    val = None

    while k:
        val = heapq.heappop(heap)
        k -= 1

    return val

# Matrix solution if there aren't any repeating values
# Requires that the matrix be sorted by row and extended after cutoff
# Actually seen in the wild (skiplists, buffer caches, some ml backends, etc)
import math
def kthSmallest(self, matrix: list[list[int]], k: int) -> int:
    matrixRowLength = len(matrix)
    matrixColLength = len(matrix[0])
```

```
    row = math.floor(k / matrixColLength) % matrixColLength
    col = ((k - (matrixColLength * row)) % matrixRowLength - 1) %␣
↪matrixRowLength

    return matrix[row][col]
```

1.8 Min Heap

[8]:

```
class MinHeap:
    def __init__(self, heapSize):
        self.heapSize = heapSize
        self.minheap = [0] * (heapSize + 1)
        self.realSize = 0

    # Function to add an element
    def add(self, element):
        self.realSize += 1

        if self.realSize > self.heapSize:
            print("Added too many elements!")
            self.realSize -= 1
            return
        self.minheap[self.realSize] = element
        index = self.realSize
        # Parent node of the newly added element
        # Note if we use an array to represent the complete binary tree
        # and store the root node at index 1
        # index of the parent node of any node is [index of the node / 2]
        # index of the left child node is [index of the node * 2]
        # index of the right child node is [index of the node * 2 + 1]
        parent = index // 2

        while (self.minheap[index] < self.minheap[parent] and index > 1):
            self.minheap[parent], self.minheap[index] = self.minheap[index],␣
↪self.minheap[parent]
            index = parent
            parent = index // 2

    # Get the top element of the Heap
    def peek(self):
        return self.minheap[1]

    # Delete the top element of the Heap
    def pop(self):
        if self.realSize < 1:
```

```
            print("Don't have any element!")
            return sys.maxsize
        else:
            removeElement = self.minheap[1]
            self.minheap[1] = self.minheap[self.realSize]
            self.realSize -= 1
            index = 1

            while (index <= self.realSize // 2):
                left = index * 2
                right = (index * 2) + 1

                if (self.minheap[index] > self.minheap[left] or self.
↪minheap[index] > self.minheap[right]):
                    if self.minheap[left] < self.minheap[right]:
                        self.minheap[left], self.minheap[index] = self.
↪minheap[index], self.minheap[left]
                        index = left
                    else:
                        self.minheap[right], self.minheap[index] = self.
↪minheap[index], self.minheap[right]
                        index = right
                else:
                    break
            return removeElement

    def size(self):
        return self.realSize

    def __str__(self):
        return str(self.minheap[1 : self.realSize + 1])
```

1.9 Max Heap

[9]:
```
# Implementing "Max Heap"
class MaxHeap:
    def __init__(self, heapSize):
        self.heapSize = heapSize
        self.maxheap = [0] * (heapSize + 1)
        self.realSize = 0

    def add(self, element):
        self.realSize += 1
        if self.realSize > self.heapSize:
            print("Added too many elements!")
            self.realSize -= 1
```

```
            return
        self.maxheap[self.realSize] = element
        index = self.realSize
        # Parent node of the newly added element
        # Note if we use an array to represent the complete binary tree
        # and store the root node at index 1
        # index of the parent node of any node is [index of the node / 2]
        # index of the left child node is [index of the node * 2]
        # index of the right child node is [index of the node * 2 + 1]
        parent = index // 2

        # If the newly added element is larger than its parent node,
        # its value will be exchanged with that of the parent node
        while (self.maxheap[index] > self.maxheap[parent] and index > 1):
            self.maxheap[parent], self.maxheap[index] = self.maxheap[index],␣
↪self.maxheap[parent]
            index = parent
            parent = index // 2

    # Get the top element of the Heap
    def peek(self):
        return self.maxheap[1]

    # Delete the top element of the Heap
    def pop(self):
        if self.realSize < 1:
            print("Don't have any element!")
            return -sys.maxsize
        else:
            # When there are still elements in the Heap
            # self.realSize >= 1
            removeElement = self.maxheap[1]
            self.maxheap[1] = self.maxheap[self.realSize]
            self.realSize -= 1
            index = 1
            # When the deleted element is not a leaf node
            while (index <= self.realSize // 2):
                left = index * 2
                right = (index * 2) + 1

                if (self.maxheap[index] < self.maxheap[left] or self.
↪maxheap[index] < self.maxheap[right]):
                    if self.maxheap[left] > self.maxheap[right]:
                        self.maxheap[left], self.maxheap[index] = self.
↪maxheap[index], self.maxheap[left]
                        index = left
                    else:
```

```
                        self.maxheap[right], self.maxheap[index] = self.maxheap[index], self.maxheap[right]
                        index = right
                else:
                    break
            return removeElement

    def size(self):
        return self.realSize

    def __str__(self):
        return str(self.maxheap[1 : self.realSize + 1])
```

BinaryTrees

August 17, 2022

1 Binary Trees

- Important Points
  - Tree is an acyclic graphic which has N nodes and N-1 edges
    * Binary means each node has at most two children
  -
- Techniques
  - Traversal
    * Pre-order -> left then right subtree
    * In-order -> left subtree first, then root, then right subtree
    * Post-order -> left subtree, then right subtree, then root
    * Level-order -> grouped based on level
  - Maximum Depth of Tree
  - Symmetric Tree
  - Path Sum
  - Construct Tree From Given Traversal
  - Populate From Array
  - Serialize and Deserialize

[3]:
```
# Used in code later
class TreeNode:
    def __init__(self, val = 0, left = None, right = None):
        self.val = val
        self.left = left
        self.right = right
```

1.1 Traversal

1.1.1 Pre-order

[3]:
```
# Solution 1 - Iterative
def preorderTraversal(root: TreeNode) -> list[int]:
    ret = []
    stack = [root]
    while stack:
        node = stack.pop()
        if node:
            ret.append(node.val)
```

```
            stack.append(node.right)
            stack.append(node.left)
    return ret

# Solution 2 - Recursive
def preorderTraversal(root: TreeNode) -> list[int]:
    ans = []

    def helper(node):
        if node:
            ans.append(node.val)
            if node.left:
                helper(node.left)
            if node.right:
                helper(node.right)

    helper(root)

    return ans
```

1.2 In-order

[ ]:
```
# Solution 1 - Iterative
def inorderTraversal(root: TreeNode) -> list[int]:
    res, stack = [], [(root, False)]
    while stack:
        node, visited = stack.pop()  # the last element
        if node:
            if visited:
                res.append(node.val)
            else:  # inorder: left -> root -> right
                stack.append((node.right, False))
                stack.append((node, True))
                stack.append((node.left, False))
    return res

# Solution 2 - Recursive
def inorderTraversal(root: TreeNode) -> list[int]:
    ans = []

    def helper(node):
        if node:
            if node.left:
                helper(node.left)
            ans.append(node.val)
            if node.right:
                helper(node.right)
```

```
    helper(root)

    return ans
```

1.3 Post-order

[ ]:
```python
# Solution 1 - Iterative
def postorderTraversal(root: TreeNode) -> list[int]:
    res, stack = [], [(root, False)]
    while stack:
        node, visited = stack.pop()  # the last element
        if node:
            if visited:
                res.append(node.val)
            else:  # postorder: left -> right -> root
                stack.append((node, True))
                stack.append((node.right, False))
                stack.append((node.left, False))
    return res

# Solution 2 - Recursive
def postorderTraversal(root: TreeNode) -> list[int]:
    ans = []

    def helper(node):
        if node:
            if node.left:
                helper(node.left)
            if node.right:
                helper(node.right)
            ans.append(node.val)

    helper(root)

    return ans
```

1.4 Level-order

[5]:
```python
# Solution 1
def levelOrder(root):
    ans, level = [], [root]
    while root and level:
        ans.append([node.val for node in level])
        LRpair = [(node.left, node.right) for node in level]
        level = [leaf for LR in LRpair for leaf in LR if leaf]
    return ans
```

```python
# Solution 2
def levelOrder(root):
    ans, level = [], [root]
    while root and level:
        ans.append([node.val for node in level])
        level = [kid for n in level for kid in (n.left, n.right) if kid]
    return ans

# Solution 3
def levelOrder(root: TreeNode) -> list[list[int]]:

    if not root:
        return []

    result = {}

    def traverse(node, level):
        if node:
            if level not in result:
                result[level] = []
            result[level].append(node.val)

            traverse(node.left, level+1)
            traverse(node.right, level+1)

    traverse(root, 0)

    return [result[i] for i in result.keys()]
```

1.5 Maximum Depth of Binary Tree

[7]:
```python
# Solution
def levelOrder(root: TreeNode) -> list[list[int]]:
    ans, level = [], [root]
    while root and level:
        ans.append([node.val for node in level])
        level = [kid for n in level for kid in (n.left, n.right) if kid]
    return ans

def maxDepth(root: TreeNode) -> int:
    return len(self.levelOrder(root))
```

1.6 Symmetric Tree

[6]:
```python
# Solution 1 - Iterative
def isSymmetric(root: TreeNode) -> list[list[int]]:
    if not root:
        return True

    dq = collections.deque([(root.left,root.right),])
    while dq:
        node1, node2 = dq.popleft()
        if not node1 and not node2:
            continue
        if not node1 or not node2:
            return False
        if node1.val != node2.val:
            return False
        # node1.left and node2.right are symmetric nodes in structure
        # node1.right and node2.left are symmetric nodes in structure
        dq.append((node1.left,node2.right))
        dq.append((node1.right,node2.left))
    return True

# Solution 2 - Recursive
def isSymmetric(root: TreeNode) -> list[list[int]]:
    def isSym(L,R):
        if not L and not R: return True
        if L and R and L.val == R.val:
            return isSym(L.left, R.right) and isSym(L.right, R.left)
        return False
    return isSym(root, root)

# Solution 3 - Recursive
def levelOrder(root: TreeNode) -> list[list[int]]:

    if not root:
        return []

    result = {}

    def traverse(node, level):
        if level not in result:
            result[level] = []
        result[level].append(node.val if node else None)
        if node:
            traverse(node.left, level+1)
            traverse(node.right, level+1)
```

```python
    traverse(root, 0)

    return [result[i] for i in result.keys()]


def isSymmetric(root: TreeNode) -> bool:
    new = root
    if root is None or root.left is None and root.right is None:
        return True

    new = [nodes[::-1] for nodes in self.levelOrder(new)]

    return new == self.levelOrder(root)
```

1.7 Path Sum

[8]:
```python
# Solution 1 - Iterative
def hasPathSum(root: TreeNode, targetSum: int) -> bool:
    if not root:
        return 0
    stack = [(root,root.val)]
    while len(stack):
        node,sum_ = stack.pop()
        if node.left == None and node.right == None and sum_ == targetSum:
            return True
        if node.right:         # as right goes to bottom as stack is lifo
            stack.append((node.right,sum_+node.right.val))
        if node.left:
            stack.append((node.left,sum_+node.left.val))
    return False

# Solution 2 - Recursive
def hasPathSum(root: TreeNode, targetSum: int) -> bool:
    foundTargetSum = []

    def helper(node, runningTotal):
        if node:
            if node.left is None and node.right is None:
                if (runningTotal + node.val) == targetSum:
                    foundTargetSum.append(1)
            if node.left:
                helper(node.left, runningTotal + node.val)
            if node.right:
                helper(node.right, runningTotal + node.val)

    helper(root, 0)
```

```python
    return True if len(foundTargetSum) else False
```

[9]:
```python
## Count Univalue
```

[ ]:
```python

```

[10]:
```python
## Construct Binary Tree From Traversal
```

[12]:
```python
# GET CODE
```

[13]:
```python
## Populating Next
```

[ ]:
```python

```

[14]:
```python
## LCA
```

[15]:
```python

```

[16]:
```python
## Serialize and Deserialize
```

[ ]:
```python

```

N-ary Trees

August 17, 2022

1 N-ary Trees

- Traverse a Binary Tree
  - Preorder Traversal: Visit the root node, then traverse the left subtree and finally traverse the right subtree.
  - Inorder Traversal: Traverse the left subtree, then visit the root node and finally traverse the right subtree.
  - Postorder Traversal: Traverse the left subtree, then traverse the right subtree and finally visit the root node.
  - Level-order Traversal: Traverse the tree level by level.
- To generalize to n-ary trees
  - Replace: traverse left subtree… traverse right subtree
  - With: for each child; do traverse subtree rooted at that child by recursively calling the traversal function; done;
- Techniques
  - Preorder Traversal
  - Postorder Traversal
  - Levelorder Traversal
  - Maximum Depth of N-ary
  - Encode N-ary to Binary Tree

[1]:
```
# Needed for code
from typing import List

class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children

class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
```

1.1 Preorder Traversal

[2]:
```
# Solution 1
# Time: O(N)
# Space: O(N)
def preorder(root: Node) -> List[int]:
    if not root:
        return
    stack, output = [root, ], []
    while stack:
        root = stack.pop()
        output.append(root.val)
        stack.extend(root.children[::-1])

    return output

# Solution 2
def preorder(root: Node) -> List[int]:
    ret = []
    stack = [root]
    while stack:
        node = stack.pop()
        if node:
            ret.append(node.val)
            tmp = []
            for children in node.children:
                tmp.append(children)
            stack += tmp[::-1]
    return ret
```

1.2 Postorder Traversal

[3]:
```
# Solution 1
def postorder(root: Node) -> List[int]:
    if root is None:
        return []

    stack, output = [root, ], []
    while stack:
        root = stack.pop()
        if root is not None:
            output.append(root.val)
        for c in root.children:
            stack.append(c)

    return output[::-1]
```

```python
# Solution 2
def postorder(root: Node) -> List[int]:
    res, stack = [], [(root, False)]
    while stack:
        node, visited = stack.pop()  # the last element
        if node:
            if visited:
                res.append(node.val)
            else:  # postorder: left -> right -> root
                stack.append((node, True))
                tmp = []
                for children in node.children:
                    tmp.append((children, False))
                stack += tmp[::-1]
    return res
```

1.3 LevelOrder Traversal

[4]:
```python
# Solution 1
def levelOrder(root: Node) -> List[List[int]]:
    if not root:
        return []
    queue = deque([root])
    trav = []
    while queue:
        size = len(queue)
        lvl = []
        for i in range(size):
            node = queue.popleft()
            lvl.append(node.val)
            if node.children:
                for child in node.children:
                    queue.append(child)
        trav.append(lvl)
    return trav

# Solution 2
def levelOrder(root: Node) -> List[List[int]]:
    if not root:
        return []

    result = {}

    def traverse(node, level):
        if node:
            if level not in result:
                result[level] = []
```

```
            result[level].append(node.val)

            for children in node.children:
                traverse(children, level+1)

    traverse(root, 0)

    return [result[i] for i in result.keys()]
```

1.4 Maximum Depth of N-ary

[5]:
```
# Solution 1
def helper(root: Node) -> int:
    if root==None:
        return 0
    max_depth = 0
    for i in range(len(root.children)):
        max_depth = max(max_depth,self.helper(root.children[i]))
    return max_depth+1

def maxDepth(root: Node) -> int:
    return self.helper(root)

# Solution 2
def levelOrder(root: Node) -> List[List[int]]:
    ans, level = [], [root]
    while root and level:
        ans.append([node.val for node in level])
        level = [kid for n in level for kid in (n.children) if kid]
    return ans

def maxDepth(root: Node) -> int:
    return len(self.levelOrder(root))
```

1.5 Encode N-ary Tree to Binary Tree

[6]:
```
"""
# Definition for a Node.
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children
"""

"""
# Definition for a binary tree node.
```

```python
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
"""

class Codec:
    def encode(self, root: Node) -> TreeNode:
        if not root:
            return
        ans = TreeNode(root.val)
        if root.children:
            children = list(map(self.encode, root.children))
            ans.right = children[0]
            for i in range(len(children)-1):
                children[i].left = children[i+1]
        return ans

    def decode(self, data: TreeNode) -> Node:
        if not data:
            return
        ans = Node(data.val, [])
        if data.right:
            n = data.right
            while n:
                ans.children.append(self.decode(n))
                n = n.left
        return ans

# Your Codec object will be instantiated and called as such:
# codec = Codec()
# codec.decode(codec.encode(root))
```

Tries

August 17, 2022

1 Tries (Prefix Tree)

- Techniques
  - Trie Implementation
  - Map Sum Pairs
  - Replace Words
  - Autocomplete System (Question Was Too Long)
  - Add and Search Word
  - Palindrome Pairs (Combinatorics/Permutations Section)
  - Word Search II (Matrix Section)
  - Word Squares (Matrix Section)
  - Maximum XOR of Two Numbers in Array (DP Section)

1.1 Trie/Prefix Tree Implementation

[3]:
```
# Solution 1
class TrieNode:
    def __init__(self):
        self.word=False
        self.children={}

class Trie:

    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node=self.root
        for i in word:
            if i not in node.children:
                node.children[i]=TrieNode()
            node=node.children[i]
        node.word=True

    def search(self, word):
        node=self.root
        for i in word:
            if i not in node.children:
```

```python
                return False
            node=node.children[i]
        return node.word

    def startsWith(self, prefix):
        node=self.root
        for i in prefix:
            if i not in node.children:
                return False
            node=node.children[i]
        return True


# Solution 2
# Remembrance Trie (RealCronus PyPi)
class Trie:
    def __init__(self):
        self.child = {}
    def insert(self, word, obj = 1):
        current = self.child
        for l in word:
            if l not in current:
                current[l] = {}
            current = current[l]
        if "#" in current.keys():
            current["#"].insert(0, [obj,time()])
        else:
            current['#']=[[obj,time()]]

    def search(self, word):
        current = self.child
        for l in word:
            if l not in current:
                return False
            current = current[l]

        if "#" in current:
            return current['#']
        else:
            return False

    def startsWith(self, prefix):
        current = self.child
        for l in prefix:
            if l not in current:
                return False
            current = current[l]
```

```
        return True
```

1.2 Map Sum Pairs

[4]:
```
# Solution 1: Dictionary Approach
class MapSum:

    def __init__(self):
        self.d = {}

    def insert(self, key, val):
        self.d[key] = val

    def sum(self, prefix):
        return sum(self.d[i] for i in self.d if i.startswith(prefix))

# Solution 2
class TrieNode:
    def __init__(self):
        self.child = defaultdict(TrieNode)
        self.sum = 0  # Store the sum of values of all strings go through this␣
 ↪node.

class MapSum:  # 24 ms, faster than 97.01%
    def __init__(self):
        self.trieRoot = TrieNode()
        self.map = defaultdict(int)

    def insert(self, key: str, val: int) -> None:
        diff = val - self.map[key]
        curr = self.trieRoot
        for c in key:
            curr = curr.child[c]
            curr.sum += diff
        self.map[key] = val

    def sum(self, prefix: str) -> int:
        curr = self.trieRoot
        for c in prefix:
            if c not in curr.child: return 0
            curr = curr.child[c]
        return curr.sum

# Solution 3
class Trie:
    def __init__(self):
        self.child = {}
```

```
    def insert(self, word, val):
        current = self.child
        for l in word:
            if l not in current:
                current[l] = {}
            current = current[l]
        current["#"] = val

    def search(self, word):
        current = self.child
        for l in word:
            if l not in current:
                return False
            current = current[l]

        if "#" in current:
            return current['#']
        else:
            return False

    def startsWith(self, prefix):
        runningTotal = []
        current = self.child
        for l in prefix:
            if l not in current:
                return 0
            current = current[l]

        def findAllSubTrees(node):
            if '#' in node:
                runningTotal.append(node['#'])
            for l in node:
                if isinstance(node[l], int) != True:
                    findAllSubTrees(node[l])

        findAllSubTrees(current)
        return sum(runningTotal)

class MapSum:

    def __init__(self):
        self.prefixTree = Trie()

    def insert(self, key: str, val: int) -> None:
        self.prefixTree.insert(key, val)
```

```python
    def sum(self, prefix: str) -> int:
        return self.prefixTree.startsWith(prefix)

# Your MapSum object will be instantiated and called as such:
# obj = MapSum()
# obj.insert(key,val)
# param_2 = obj.sum(prefix)
```

1.3 Replace Words

[7]:
```python
# Solution 1: Set Approach (Prefix Hash)
def replaceWords(roots, sentence):
    rootset = set(roots)

    def replace(word):
        for i in range(1, len(word)):
            if word[:i] in rootset:
                return word[:i]
        return word

    return " ".join(map(replace, sentence.split()))


# Solution 2
class Trie:
    def __init__(self):
        self.child = {}

    def insert(self, word):
        current = self.child
        for l in word:
            if l not in current:
                current[l] = {}
            current = current[l]
        current["#"] = word

    def startsWith(self, prefix):
        runningTotal = []
        current = self.child
        for l in prefix:
            if "#" in current and current["#"]:
                return current["#"]
            if l not in current:
                return None
            current = current[l]

class Solution:
```

```
    def replaceWords(self, dictionary, sentence):
        trie = Trie()
        for word in dictionary:
            trie.insert(word)

        result = ""

        for idx, word in enumerate(sentence.split()):
            tmp = trie.startsWith(word)
            if tmp:
                result += tmp
            else:
                result += word
            if idx != len(sentence.split()) - 1:
                result += " "

        return result
```

1.4 Add and Search Words Data Structure (Study)

[ ]:
```
# Solution 1
class WordDictionary:

    def __init__(self):
        self.root = {}

    def addWord(self, word):
        node = self.root
        for char in word:
            node = node.setdefault(char, {})
        node[None] = None

    def search(self, word):
        def find(word, node):
            if not word:
                return None in node
            char, word = word[0], word[1:]
            if char != '.':
                return char in node and find(word, node[char])
            return any(find(word, kid) for kid in node.values() if kid)
        return find(word, self.root)

# Solution 2
class WordDictionary:

    def __init__(self):
        """
```

```
        Initialize your data structure here.
        """
        self.trie = {}


    def addWord(self, word: str) -> None:
        """
        Adds a word into the data structure.
        """
        node = self.trie

        for ch in word:
            if not ch in node:
                node[ch] = {}
            node = node[ch]
        node['$'] = True

    def search(self, word: str) -> bool:
        """
        Returns if the word is in the data structure. A word could contain the␣
↪dot character '.' to represent any letter.
        """
        def search_in_node(word, node) -> bool:
            for i, ch in enumerate(word):
                if not ch in node:
                    # if the current character is '.'
                    # check all possible nodes at this level
                    if ch == '.':
                        for x in node:
                            if x != '$' and search_in_node(word[i + 1:],␣
↪node[x]):
                                return True
                    # if no nodes lead to answer
                    # or the current character != '.'
                    return False
                # if the character is found
                # go down to the next level in trie
                else:
                    node = node[ch]
            return '$' in node

        return search_in_node(word, self.trie)
```

Binary Search Tree

August 17, 2022

1 Binary Search Tree

1.1 BST Insert

[1]:
```
class GFG :
    @staticmethod
    def main( args) :
        tree = BST()
        tree.insert(30)
        tree.insert(50)
        tree.insert(15)
        tree.insert(20)
        tree.insert(10)
        tree.insert(40)
        tree.insert(60)
        tree.inorder()
class Node :
    left = None
    val = 0
    right = None
    def __init__(self, val) :
        self.val = val
class BST :
    root = None
    def insert(self, key) :
        node = Node(key)
        if (self.root == None) :
            self.root = node
            return
        prev = None
        temp = self.root
        while (temp != None) :
            if (temp.val > key) :
                prev = temp
                temp = temp.left
            elif(temp.val < key) :
                prev = temp
                temp = temp.right
```

```
        if (prev.val > key) :
            prev.left = node
        else :
            prev.right = node
    def inorder(self) :
        temp = self.root
        stack =  []
        while (temp != None or not (len(stack) == 0)) :
            if (temp != None) :
                stack.append(temp)
                temp = temp.left
            else :
                temp = stack.pop()
                print(str(temp.val) + " ", end ="")
                temp = temp.right

if __name__=="__main__":
    GFG.main([])
```

```
10 15 20 30 40 50 60
```

1.2 BST Delete

[2]:
```
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# A utility function to do
# inorder traversal of BST
def inorder(root):
    if root is not None:
        inorder(root.left)
        print(root.key, end=" ")
        inorder(root.right)

# A utility function to insert a
# new node with given key in BST
def insert(node, key):

    # If the tree is empty,
    # return a new node
    if node is None:
        return Node(key)
```

```
    # Otherwise recur down the tree
    if key < node.key:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)

    # return the (unchanged) node pointer
    return node


# Given a binary search tree
# and a key, this function
# delete the key and returns the new root
def deleteNode(root, key):

    # Base Case
    if root is None:
        return root

    # Recursive calls for ancestors of
    # node to be deleted
    if key < root.key:
        root.left = deleteNode(root.left, key)
        return root

    elif(key > root.key):
        root.right = deleteNode(root.right, key)
        return root

    # We reach here when root is the node
    # to be deleted.

    # If root node is a leaf node

    if root.left is None and root.right is None:
          return None

    # If one of the children is empty

    if root.left is None:
        temp = root.right
        root = None
        return temp

    elif root.right is None:
        temp = root.left
```

```
        root = None
        return temp

    # If both children exist

    succParent = root

    # Find Successor

    succ = root.right

    while succ.left != None:
        succParent = succ
        succ = succ.left

    # Delete successor.Since successor
    # is always left child of its parent
    # we can safely make successor's right
    # right child as left of its parent.
    # If there is no succ, then assign
    # succ->right to succParent->right
    if succParent != root:
        succParent.left = succ.right
    else:
        succParent.right = succ.right

    # Copy Successor Data to root

    root.key = succ.key

    return root


# Driver code
""" Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80 """

root = None
root = insert(root, 50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
```

```
root = insert(root, 60)
root = insert(root, 80)

print("Inorder traversal of the given tree")
inorder(root)

print("\nDelete 20")
root = deleteNode(root, 20)
print("Inorder traversal of the modified tree")
inorder(root)

print("\nDelete 30")
root = deleteNode(root, 30)
print("Inorder traversal of the modified tree")
inorder(root)

print("\nDelete 50")
root = deleteNode(root, 50)
print("Inorder traversal of the modified tree")
inorder(root)
```

```
Inorder traversal of the given tree
20 30 40 50 60 70 80
Delete 20
Inorder traversal of the modified tree
30 40 50 60 70 80
Delete 30
Inorder traversal of the modified tree
40 50 60 70 80
Delete 50
Inorder traversal of the modified tree
40 60 70 80
```

1.3 Construct BST From Pre-Order Traversal

[3]:
```
# Construct a BST from given pre-order traversal
# for example if the given traversal is {10, 5, 1, 7, 40, 50},
# then the output should be the root of the following tree.
#     10
#    /   \
#   5     40
# /  \      \
# 1    7      50
class Node:
    data = 0
    left = None
    right = None
```

```
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class CreateBSTFromPreorder:
    node = None
    # This will create the BST
    @staticmethod
    def createNode(node,  data):
        if (node == None):
            node = Node(data)
        if (node.data > data):
            node.left = CreateBSTFromPreorder.createNode(node.left, data)
        if (node.data < data):
            node.right = CreateBSTFromPreorder.createNode(node.right, data)
        return node

    # A wrapper function of createNode
    @staticmethod
    def create(data):
        CreateBSTFromPreorder.node = CreateBSTFromPreorder.createNode(
            CreateBSTFromPreorder.node, data)

    # A function to print BST in inorder
    @staticmethod
    def inorderRec(root):
        if (root != None):
            CreateBSTFromPreorder.inorderRec(root.left)
            print(root.data)
            CreateBSTFromPreorder.inorderRec(root.right)

    # Driver Code
    @staticmethod
    def main(args):
        nodeData = [10, 5, 1, 7, 40, 50]
        i = 0
        while (i < len(nodeData)):
            CreateBSTFromPreorder.create(nodeData[i])
            i += 1
        CreateBSTFromPreorder.inorderRec(CreateBSTFromPreorder.node)

if __name__ == "__main__":
    CreateBSTFromPreorder.main([])
```

1

5
7
10
40
50

1.4 Binary Tree To BST

[4]:
```python
# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Helper function to store the inorder traversal of a tree
def storeInorder(root, inorder):

    # Base Case
    if root is None:
        return

    # First store the left subtree
    storeInorder(root.left, inorder)

    # Copy the root's data
    inorder.append(root.data)

    # Finally store the right subtree
    storeInorder(root.right, inorder)

# A helper function to count nodes in a binary tree
def countNodes(root):
    if root is None:
        return 0

    return countNodes(root.left) + countNodes(root.right) + 1

# Helper function that copies contents of sorted array
# to Binary tree
def arrayToBST(arr, root):

    # Base Case
    if root is None:
        return
```

```python
    # First update the left subtree
    arrayToBST(arr, root.left)

    # now update root's data delete the value from array
    root.data = arr[0]
    arr.pop(0)

    # Finally update the right subtree
    arrayToBST(arr, root.right)

# This function converts a given binary tree to BST
def binaryTreeToBST(root):

    # Base Case: Tree is empty
    if root is None:
        return

    # Count the number of nodes in Binary Tree so that
    # we know the size of temporary array to be created
    n = countNodes(root)

    # Create the temp array and store the inorder traversal
    # of tree
    arr = []
    storeInorder(root, arr)

    # Sort the array
    arr.sort()

    # copy array elements back to binary tree
    arrayToBST(arr, root)

# Print the inorder traversal of the tree
def printInorder(root):
    if root is None:
        return
    printInorder(root.left)
    print (root.data,end=" ")
    printInorder(root.right)

# Driver program to test above function
root = Node(10)
root.left = Node(30)
root.right = Node(15)
root.left.left = Node(20)
root.right.right = Node(5)
```

```
# Convert binary tree to BST
binaryTreeToBST(root)

print ("Following is the inorder traversal of the converted BST")
printInorder(root)
```

```
Following is the inorder traversal of the converted BST
5 10 15 20 30
```

1.5 All Possible BSTs For Keys 1 To N

[5]:
```
# Python3 program to construct all unique
# BSTs for keys from 1 to n

# Binary Tree Node
""" A utility function to create a
new BST node """
class newNode:

    # Construct to create a newNode
    def __init__(self, item):
        self.key=item
        self.left = None
        self.right = None

# A utility function to do preorder
# traversal of BST
def preorder(root) :

    if (root != None) :

        print(root.key, end = " " )
        preorder(root.left)
        preorder(root.right)

# function for constructing trees
def constructTrees(start, end):

    list = []

    """ if start > end then subtree will be
        empty so returning None in the list """
    if (start > end) :

        list.append(None)
        return list
```

```
    """ iterating through all values from
        start to end for constructing
        left and right subtree recursively """
    for i in range(start, end + 1):

        """ constructing left subtree """
        leftSubtree = constructTrees(start, i - 1)

        """ constructing right subtree """
        rightSubtree = constructTrees(i + 1, end)

        """ now looping through all left and
            right subtrees and connecting
            them to ith root below """
        for j in range(len(leftSubtree)) :
            left = leftSubtree[j]
            for k in range(len(rightSubtree)):
                right = rightSubtree[k]
                node = newNode(i)   # making value i as root
                node.left = left    # connect left subtree
                node.right = right    # connect right subtree
                list.append(node)    # add this tree to list
    return list

# Driver Code
if __name__ == '__main__':

    # Construct all possible BSTs
    totalTreesFrom1toN = constructTrees(1, 3)

    """ Printing preorder traversal of
        all constructed BSTs """
    print("Preorder traversals of all",
                "constructed BSTs are")
    for i in range(len(totalTreesFrom1toN)):
        preorder(totalTreesFrom1toN[i])
        print()
```

```
Preorder traversals of all constructed BSTs are
1 2 3
1 3 2
2 1 3
3 1 2
3 2 1
```

1.6 BST To Min-Heap

[6]:
```python
# Python3 program to construct all unique
# BSTs for keys from 1 to n

# Binary Tree Node
""" A utility function to create a
new BST node """
class newNode:

    # Construct to create a newNode
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Utility function to print Min-heap
# level by level
def printLevelOrder(root):

    # Base Case
    if (root == None):
        return

    # Create an empty queue for level
    # order traversal
    q = []
    q.append(root)

    while (len(q)):
        nodeCount = len(q)
        while (nodeCount > 0) :

            node = q[0]
            print(node.data, end = " " )
            q.pop(0)
            if (node.left) :
                q.append(node.left)
            if (node.right) :
                q.append(node.right)
            nodeCount -= 1
        print()

# A simple recursive function to convert a
# given Binary Search tree to Sorted Linked
# List root     -. Root of Binary Search Tree
def BSTToSortedLL(root, head_ref):
```

```
    # Base cases
    if(root == None) :
        return

    # Recursively convert right subtree
    BSTToSortedLL(root.right, head_ref)

    # insert root into linked list
    root.right = head_ref[0]

    # Change left pointer of previous
    # head to point to None
    if (head_ref[0] != None):
        (head_ref[0]).left = None

    # Change head of linked list
    head_ref[0] = root

    # Recursively convert left subtree
    BSTToSortedLL(root.left, head_ref)

# Function to convert a sorted Linked
# List to Min-Heap.
# root -. root[0] of Min-Heap
# head -. Pointer to head node of
#           sorted linked list
def SortedLLToMinHeap( root, head) :

    # Base Case
    if (head == None) :
        return

    # queue to store the parent nodes
    q = []

    # The first node is always the
    # root node
    root[0] = head[0]

    # advance the pointer to the next node
    head[0] = head[0].right

    # set right child to None
    root[0].right = None

    # add first node to the queue
```

```
    q.append(root[0])

    # run until the end of linked list
    # is reached
    while (head[0] != None) :

        # Take the parent node from the q
        # and remove it from q
        parent = q[0]
        q.pop(0)

        # Take next two nodes from the linked
        # list and Add them as children of the
        # current parent node. Also in push them
        # into the queue so that they will be
        # parents to the future nodes
        leftChild = head[0]
        head[0] = head[0].right     # advance linked list to next node
        leftChild.right = None # set its right child to None
        q.append(leftChild)

        # Assign the left child of parent
        parent.left = leftChild

        if (head) :
            rightChild = head[0]
            head[0] = head[0].right # advance linked list to next node
            rightChild.right = None # set its right child to None
            q.append(rightChild)

            # Assign the right child of parent
            parent.right = rightChild

# Function to convert BST into a Min-Heap
# without using any extra space
def BSTToMinHeap(root):

    # head of Linked List
    head = [None]

    # Convert a given BST to Sorted Linked List
    BSTToSortedLL(root, head)

    # set root as None
    root = [None]

    # Convert Sorted Linked List to Min-Heap
```

```
    SortedLLToMinHeap(root, head)
    return root

# Driver Code
if __name__ == '__main__':

    """ Constructing below tree
                8
            / \
            4     12
        / \ / \
        2 6 10 14
    """
    root = newNode(8)
    root.left = newNode(4)
    root.right = newNode(12)
    root.right.left = newNode(10)
    root.right.right = newNode(14)
    root.left.left = newNode(2)
    root.left.right = newNode(6)

    root = BSTToMinHeap(root)

    """ Output - Min Heap
                2
            / \
            4     6
        / \ / \
        8 10 12 14
    """
    printLevelOrder(*root)
```

```
2
4 6
8 10 12 14
```

1.7 Iterative Searching

[7]:
```
# Python program to demonstrate searching operation
# in binary search tree without recursion
class newNode:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

```python
# Function to check the given
# key exist or not
def iterativeSearch(root, key):

    # Traverse until root reaches
    # to dead end
    while root != None:

        # pass right subtree as new tree
        if key > root.data:
            root = root.right

        # pass left subtree as new tree
        elif key < root.data:
            root = root.left
        else:
            return True # if the key is found return 1
    return False

# A utility function to insert a
# new Node with given key in BST
def insert(Node, data):

    # If the tree is empty, return
    # a new Node
    if Node == None:
        return newNode(data)

    # Otherwise, recur down the tree
    if data < Node.data:
        Node.left = insert(Node.left, data)
    elif data > Node.data:
        Node.right = insert(Node.right, data)

    # return the (unchanged) Node pointer
    return Node

# Driver Code
if __name__ == '__main__':

    # Let us create following BST
    # 50
    # 30     70
    # / \ / \
    # 20 40 60 80
    root = None
```

```
    root = insert(root, 50)
    insert(root, 30)
    insert(root, 20)
    insert(root, 40)
    insert(root, 70)
    insert(root, 60)
    insert(root, 80)
    if iterativeSearch(root, 15):
        print("Yes")
    else:
        print("No")
```

```
No
```

1.8 Convert BST To Balanced BST

[8]:
```
# Python3 program to convert a left
# unbalanced BST to a balanced BST
import sys
import math

# A binary tree node has data, pointer to left child
# and a pointer to right child
class Node:
    def __init__(self,data):
        self.data=data
        self.left=None
        self.right=None

# This function traverse the skewed binary tree and
# stores its nodes pointers in vector nodes[]
def storeBSTNodes(root,nodes):

    # Base case
    if not root:
        return

    # Store nodes in Inorder (which is sorted
    # order for BST)
    storeBSTNodes(root.left,nodes)
    nodes.append(root)
    storeBSTNodes(root.right,nodes)

# Recursive function to construct binary tree
def buildTreeUtil(nodes,start,end):

    # base case
```

```
    if start>end:
        return None

    # Get the middle element and make it root
    mid=(start+end)//2
    node=nodes[mid]

    # Using index in Inorder traversal, construct
    # left and right subtress
    node.left=buildTreeUtil(nodes,start,mid-1)
    node.right=buildTreeUtil(nodes,mid+1,end)
    return node

# This functions converts an unbalanced BST to
# a balanced BST
def buildTree(root):

    # Store nodes of given BST in sorted order
    nodes=[]
    storeBSTNodes(root,nodes)

    # Constructs BST from nodes[]
    n=len(nodes)
    return buildTreeUtil(nodes,0,n-1)

# Function to do preorder traversal of tree
def preOrder(root):
    if not root:
        return
    print("{} ".format(root.data),end="")
    preOrder(root.left)
    preOrder(root.right)

# Driver code
if __name__=='__main__':
    # Constructed skewed binary tree is
    #         10
    #         /
    #         8
    #         /
    #     7
    #     /
    #     6
    #     /
    # 5
    root = Node(10)
    root.left = Node(8)
```

```python
    root.left.left = Node(7)
    root.left.left.left = Node(6)
    root.left.left.left.left = Node(5)
    root = buildTree(root)
    print("Preorder traversal of balanced BST is :")
    preOrder(root)
```

```
Preorder traversal of balanced BST is :
7 5 6 8 10
```

[ ]:

```python

```

Graphs

August 17, 2022

1 Graphs

[6]:
```python
# Needed For Code
from typing import List
```

1.1 Union Find

[ ]:
```python
class UnionFind:
    def __init__(self, size):
        self.root = [i for i in range(size)]
        # Use a rank array to record the height of each vertex, i.e., the␣
 ↪"rank" of each vertex.
        # The initial "rank" of each vertex is 1, because each of them is
        # a standalone vertex with no connection to other vertices.
        self.rank = [1] * size
        self.count = size

    # The find function here is the same as that in the disjoint set with path␣
 ↪compression.
    def find(self, x):
        if x == self.root[x]:
            return x
        self.root[x] = self.find(self.root[x])
        return self.root[x]

    # The union function with union by rank
    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
                self.root[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.root[rootX] = rootY
            else:
                self.root[rootY] = rootX
                self.rank[rootX] += 1
```

```
            self.count -= 1

    def getCount(self):
        return self.count
```

1.2 Number Of Provinces

[ ]:
```
# Solution 1
def findCircleNum(self, A):
    N = len(A)
    seen = set()
    def dfs(node):
        for nei, adj in enumerate(A[node]):
            if adj and nei not in seen:
                seen.add(nei)
                dfs(nei)

    ans = 0
    for i in range(N):
        if i not in seen:
            dfs(i)
            ans += 1
    return ans

# Solution 2
class UnionFind(object):
    def __init__(self, n):
        self.u = list(range(n))

    def union(self, a, b):
        ra, rb = self.find(a), self.find(b)
        if ra != rb: self.u[ra] = rb

    def find(self, a):
        while self.u[a] != a: a = self.u[a]
        return a

class Solution(object):
    def findCircleNum(self, M: List[List[int]]) -> int:
        if not M: return 0
        s = len(M)

        uf = UnionFind(s)
        for r in range(s):
            for c in range(r,s):
                if M[r][c] == 1: uf.union(r,c)
```

```python
        return len(set([uf.find(i) for i in range(s)]))
```

1.3 Number Of Connected Components In An Undirected Graph

[ ]:
```python
# Solution 1
class Solution(object):
    def countComponents(self, n, edges):
        adj = [[] for i in range(n)]
        for [first, second] in edges:
            if second not in adj[first]:
                adj[first].append(second)
            if first not in adj[second]:
                adj[second].append(first)

        unvisited = set()
        for i in range(n):
            unvisited.add(i) # Put all nodes to start
        ret = 0
        while len(unvisited) != 0:
            # Get first elem AND remove from set
            q = [unvisited.pop()]
            while q:
                cur = q.pop()
                neighbors = adj[cur]
                for neighbor in neighbors:
                    if neighbor in unvisited:
                        q.append(neighbor)
                if cur in unvisited: # key error thrown if not removed
                    unvisited.remove(cur)
            ret += 1

        return ret

# Solution 2
class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        if n == 1:
            return 1

        uf = UnionFind(n)
        for item in edges:
            uf.union(item[0], item[1])
        return uf.getCount()
```

1.4 Valid Tree

[ ]:
```python
# Come back to
def validTree(self, n: int, edges: List[List[int]]) -> bool:
    visited = {idx: 0 for idx in range(n)}
    ins = {}
    outs = {}

    for edge in edges:
        if visited[edge[1]] == 1:
            return False
        visited[edge[1]] = 1

    return True if sum(visited.values()) == n-1 else False
```

1.5 Find If Path Exists in Graph

[ ]:
```python
class Solution:
    def validPath(self, n: int, edges: List[List[int]], start: int, end: int)␣
 ↪-> bool:

        adjacency_list = [[] for _ in range(n)]
        for a, b in edges:
            adjacency_list[a].append(b)
            adjacency_list[b].append(a)

        stack = [start]
        seen = set()

        while stack:
            # Get the current node.
            node = stack.pop()

            # Check if we have reached the target node.
            if node == end:
                return True

            # Check if we've already visited this node.
            if node in seen:
                continue
            seen.add(node)

            # Add all neighbors to the stack.
            for neighbor in adjacency_list[node]:
                stack.append(neighbor)

        # Our stack is empty and we did not reach the end node.
```

```
        return False
```

1.6 All Paths Source Target

[2]:
```
class Solution:
    def allPathsSourceTarget(self, graph):
        def dfs(node):
            path.append(node)
            if node == len(graph) - 1:
                paths.append(path.copy())
                return

            next_nodes = graph[node]
            for next_node in next_nodes:
                dfs(next_node)
                path.pop()

        paths = []
        path = []
        if not graph or len(graph) == 0:
            return paths
        dfs(0)
        return paths
```

1.7 Earliest Moment When Everyone Became Acquainted (Friends of Friends)

[4]:
```
class Solution:
    def earliestAcq(self, logs, n):
        logs = sorted(logs, key = lambda x: x[0])
        uf = UnionFind(n)
        for time, x, y in logs:
            uf.union(x, y)
            if uf.getCount() == 1:
                return time

        return -1
```

1.8 Min Cost To Connect All Points

[8]:
```
import heapq
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        if not points or len(points) == 0:
            return 0
        size = len(points)
        pq = []
```

```
        uf = UnionFind(size)

        for i in range(size):
            x1, y1 = points[i]
            for j in range(i + 1, size):
                x2, y2 = points[j]
                # Calculate the distance between two coordinates.
                cost = abs(x1 - x2) + abs(y1 - y2)
                edge = Edge(i, j, cost)
                pq.append(edge)

        # Convert pq into a heap.
        heapq.heapify(pq)

        result = 0
        count = size - 1
        while pq and count > 0:
            edge = heapq.heappop(pq)
            if not uf.connected(edge.point1, edge.point2):
                uf.union(edge.point1, edge.point2)
                result += edge.cost
                count -= 1
        return result

class Edge:
    def __init__(self, point1, point2, cost):
        self.point1 = point1
        self.point2 = point2
        self.cost = cost

    def __lt__(self, other):
        return self.cost < other.cost

class UnionFind:
    def __init__(self, size):
        self.root = [i for i in range(size)]
        self.rank = [1] * size

    def find(self, x):
        if x == self.root[x]:
            return x
        self.root[x] = self.find(self.root[x])
        return self.root[x]

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)
```

```
        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
                self.root[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.root[rootX] = rootY
            else:
                self.root[rootY] = rootX
                self.rank[rootX] += 1

    def connected(self, x, y):
        return self.find(x) == self.find(y)

if __name__ == "__main__":
    points = [[0,0],[2,2],[3,10],[5,2],[7,0]]
    solution = Solution()
    print(f"points = {points}")
    print(f"Minimum Cost to Connect Points = {solution.
 ↪minCostConnectPoints(points)}")
```

```
points = [[0, 0], [2, 2], [3, 10], [5, 2], [7, 0]]
Minimum Cost to Connect Points = 20
```

1.9 Cheapest Flights Within K Stops

[9]:
```
class Solution:
    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst:
 ↪ int, k: int) -> int:
        if src == dst:
            return 0

        INF = sys.maxsize
        previous = [INF] * n
        current = [INF] * n
        previous[src] = 0

        for i in range(1, k + 2):
            current[src] = 0
            for flight in flights:
                previous_flight, current_flight, cost = flight

                if previous[previous_flight] < INF:
                    current[current_flight] = min(current[current_flight],
                                                  previous[previous_flight] +␣
 ↪cost)

            previous = current.copy()
```

```python
        return -1 if current[dst] == INF else current[dst]
```

1.10 Course Schedule II

[10]:
```python
class Solution:
    def findOrder(self, num_courses: int, prerequisites: List[List[int]]) ->
 ↪List[int]:
        result = [0] * num_courses
        if num_courses == 0:
            return result

        if not prerequisites:
            result = [i for i in range(num_courses)]
            return result

        indegree = [0] * num_courses
        zero_degree = deque()
        for pre in prerequisites:
            indegree[pre[0]] += 1
        for i in range(len(indegree)):
            if indegree[i] == 0:
                zero_degree.append(i)
        if not zero_degree:
            return []

        index = 0
        while zero_degree:
            course = zero_degree.popleft()
            result[index] = course
            index += 1
            for pre in prerequisites:
                if pre[1] == course:
                    indegree[pre[0]] -= 1
                    if indegree[pre[0]] == 0:
                        zero_degree.append(pre[0])

        if any(i for i in indegree):
            return []

        return result
```

1.11 Alien Dictionary

[11]:
```python
class Solution:
    def alienOrder(self, words: List[str]) -> str:
        adjList = {c: [] for w in words for c in w}
```

```
        for i in range(len(words) -1):
            w1, w2 = words[i], words[i + 1]
            minLen = min(len(w1),len(w2))
            if len(w1) > len(w2) and w1[:minLen] == w2[:minLen]:
                return ''
            for j in range(minLen):
                if w1[j] != w2[j]:
                    adjList[w1[j]].append(w2[j])
                    break


        visit = set()
        cycle = set()
        output = ''

        def dfs(node):
            nonlocal output
            if node in cycle:
                return False
            if node in visit:
                return True

            cycle.add(node)
            for nei in adjList[node]:
                if not dfs(nei):
                    return False
            cycle.remove(node)
            visit.add(node)
            if len(output) == 0:
                output += node
            else:
                output = node + output
            return True

        for char in adjList:
            if not dfs(char):
                return ''
        return output
```

1.12 Valid Tree

[12]:
```
class UnionFind:

    # For efficiency, we aren't using makeset, but instead initialising
    # all the sets at the same time in the constructor.
    def __init__(self, n):
```

```python
        self.parent = [node for node in range(n)]
        # We use this to keep track of the size of each set.
        self.size = [1] * n

    # The find method, with path compression. There are ways of implementing
    # this elegantly with recursion, but the iterative version is easier for
    # most people to understand!
    def find(self, A):
        # Step 1: Find the root.
        root = A
        while root != self.parent[root]:
            root = self.parent[root]
        # Step 2: Do a second traversal, this time setting each node to point
        # directly at A as we go.
        while A != root:
            old_root = self.parent[A]
            self.parent[A] = root
            A = old_root
        return root

    # The union method, with optimization union by size. It returns True if a
    # merge happened, False if otherwise.
    def union(self, A, B):
        # Find the roots for A and B.
        root_A = self.find(A)
        root_B = self.find(B)
        # Check if A and B are already in the same set.
        if root_A == root_B:
            return False
        # We want to ensure the larger set remains the root.
        if self.size[root_A] < self.size[root_B]:
            # Make root_B the overall root.
            self.parent[root_A] = root_B
            # The size of the set rooted at B is the sum of the 2.
            self.size[root_B] += self.size[root_A]
        else:
            # Make root_A the overall root.
            self.parent[root_B] = root_A
            # The size of the set rooted at A is the sum of the 2.
            self.size[root_A] += self.size[root_B]
        return True

class Solution:
    def validTree(self, n: int, edges: List[List[int]]) -> bool:
        # Condition 1: The graph must contain n - 1 edges.
        if len(edges) != n - 1: return False
```

```python
    # Create a new UnionFind object with n nodes.
    unionFind = UnionFind(n)

    # Add each edge. Check if a merge happened, because if it
    # didn't, there must be a cycle.
    for A, B in edges:
        if not unionFind.union(A, B):
            return False

    # If we got this far, there's no cycles!
    return True
```

Dynamic Programming

August 17, 2022

1 Dynamic Programming

[2]:
```
from typing import List
```

1.1 Minimum Difficulty Of A Job Schedule

[3]:
```
class Solution:
    def minDifficulty(self, jobDifficulty: List[int], d: int) -> int:
        n = len(jobDifficulty)
        # If we cannot schedule at least one job per day,
        # it is impossible to create a schedule
        if n < d:
            return -1

        dp = [[float("inf")] * (d + 1) for _ in range(n)]

        # Set base cases
        dp[-1][d] = jobDifficulty[-1]

        # On the last day, we must schedule all remaining jobs, so dp[i][d]
        # is the maximum difficulty job remaining
        for i in range(n - 2, -1, -1):
            dp[i][d] = max(dp[i + 1][d], jobDifficulty[i])

        for day in range(d - 1, 0, -1):
            for i in range(day - 1, n - (d - day)):
                hardest = 0
                # Iterate through the options and choose the best
                for j in range(i, n - (d - day)):
                    hardest = max(hardest, jobDifficulty[j])
                    # Recurrence relation
                    dp[i][day] = min(dp[i][day], hardest + dp[j + 1][day + 1])

        return dp[0][1]
```

1.2 Coin Change

[4]:
```
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        # determine the number of coins required to get every value
        #   between 0 and amount inclusive
        #   Least number of coins calculate before larger for a value
        q = deque([(0,0)]) # no value, no coinNum
        visited = set()

        while q:
            cur, coinNum = q.popleft()
            if cur == amount:
                return coinNum
            if cur > amount:
                continue

            for c in coins:
                addCoin = cur + c
                if addCoin not in visited:
                    visited.add(addCoin)
                    q.append((addCoin, coinNum+1))

        return -1
```

1.3 Word Break

[6]:
```
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        words = set(wordDict)
        memo = {}
        def dp(index):
            if index == len(s):
                return True
            if index in memo:
                return memo[index]

            string = ""
            for i in range(index, len(s)):
                string += s[i]
                if string in words:
                    if dp(i + 1):
                        memo[index] = True
                        return True
            memo[index] = False
            return False
```

```
        return dp(0)

# time and space complexity
# time: O(n)
# space: O(n)
```

1.4 Longest Increasing Subsequence

[7]:
```
class Solution:  # 2516 ms, faster than 64.96%
    def lengthOfLIS(self, nums: List[int]) -> int:
        n = len(nums)
        dp = [1] * n
        for i in range(n):
            for j in range(i):
                if nums[i] > nums[j] and dp[i] < dp[j] + 1:
                    dp[i] = dp[j] + 1
        return max(dp)
```

1.5 Best Time To Buy And Sell Stock IV

[9]:
```
    def maxProfit(self, k, prices):
        """
        :type k: int
        :type prices: List[int]
        :rtype: int
        """
        #The problem is hard
        #Time complexity, O(nk)
        #Space complexity, O(nk)
        length = len(prices)
        if length < 2:
            return 0
        max_profit = 0
        #if k>= n/2, then it can't complete k transactions. The problem becomes␣
 ↪buy-and-sell problem 2
        if k>=length/2:
            for i in range(1,length):
                max_profit += max(prices[i]-prices[i-1],0)
            return max_profit

        #max_global[i][j] is to store the maximum profit, at day j, and having␣
 ↪i transactions already
        #max_local[i][j] is to store the maximum profit at day j, having i␣
 ↪transactions already, and having transaction at day j
        max_global = [[0]*length for _ in range(k+1)]
        max_local = [[0]*length for _ in range(k+1)]
```

```
        #i indicates the transaction times, j indicates the times
        for j in range(1,length):
            cur_profit = prices[j]-prices[j-1] #variable introduced by the␣
↪current day transaction
            for i in range(1,k+1):
                #max_global depends on max_local, so updata local first, and␣
↪then global.
                max_local[i][j] = max( max_global[i-1][j-1]+max(cur_profit,0),␣
↪max_local[i][j-1] + cur_profit)
                #if cur_profit <0, then the current transaction loses money, so␣
↪max_local[i][j] = max_global[i-1][j-1]
                #else, it can be max_global[i-1][j-1] + cur_profit, by␣
↪considering the current transaction
                #or it can be max_local[i][j-1] + cur_profit, this is to CANCEL␣
↪the last day transaction and moves to the current transaction. Note this␣
↪doesn't change the total number of transactions. Also, max_local[i-1] has␣
↪already been considered by max_global[i-1] term
                max_global[i][j] = max(max_global[i][j-1], max_local[i][j])
                #This is more obvious, by looking at whether transaction on day␣
↪j has influenced max_global or not.
        return max_global[k][-1] #the last day, the last transaction
```

1.6 Unique Paths II

[10]:
```
class Solution:
    # in place
    def uniquePathsWithObstacles(self, obstacleGrid):
        if not obstacleGrid:
            return
        r, c = len(obstacleGrid), len(obstacleGrid[0])
        obstacleGrid[0][0] = 1 - obstacleGrid[0][0]
        for i in range(1, r):
            obstacleGrid[i][0] = obstacleGrid[i-1][0] * (1 - obstacleGrid[i][0])
        for i in range(1, c):
            obstacleGrid[0][i] = obstacleGrid[0][i-1] * (1 - obstacleGrid[0][i])
        for i in range(1, r):
            for j in range(1, c):
                obstacleGrid[i][j] = (obstacleGrid[i-1][j] +␣
↪obstacleGrid[i][j-1]) * (1 - obstacleGrid[i][j])
        return obstacleGrid[-1][-1]
```

1.7 Minimum Falling Path Sum

[12]:
```python
#this problem will use DP
#by taking the minimum value from itself plus one of the 3 values right above it

#EX:
# 1  2  3
# 4  5  6
# 7  8  9

# new value for number at A[1][1] will be  min(5 + 1, 5 + 2, 5 + 3)
# therefore it will be 5 + 1 = 6, and 6 will then replace the value at A[1][1]

#new value for number at A[1][0] will be  min(4 + 1, 4 + 2) = 5
#it will only have two values to compare since there is no upper left value

#new value for number at A[1][2] will be  min(6 + 2, 6 + 3) = 8
#it will only have two values to compare since there is no upper right value

def minFallingPathSum(A: List[List[int]]) -> int:
    for i in range(1,len(A)):
        for j in range(len(A[0])):

            #edge cases are first column and last column which only have two␣
 ↪paths from above
            if j == 0:
                A[i][j]  = min((A[i][j] + A[i - 1][j]), (A[i][j] + A[i - 1][j +␣
 ↪1]) )

            elif (j == len(A[0]) - 1):
                A[i][j]  = min((A[i][j] + A[i - 1][j]), (A[i][j] + A[i - 1][j -␣
 ↪1]) )

            #every other column will have three paths coming from above
            else:
                A[i][j] = min(A[i][j] + A[i - 1][j],A[i][j] + A[i - 1][j + 1],␣
 ↪A[i][j] + A[i - 1][j - 1])

    # Now that minimum falling sums for each value at the bottom row have been␣
 ↪computer
    # We can just take the min of the bottow row to get the smallest overall␣
 ↪path sum
    return min(A[len(A) - 1])
```

1.8 Erect The Fence

[13]:

```python
def outerTrees(self, trees: List[List[int]]) -> List[float]:
    def circle_less_than_3pts(pts): # draw circle for <=3 points
        if not pts:
            return 0,0,0
        if len(pts)==1:
            return pts[0][0],pts[0][1],0
        elif len(pts)==2:
            (x0, y0), (x1, y1) = pts
            return ((x0+x1)/2, (y0+y1)/2, sqrt((x0-x1)**2+(y0-y1)**2)/2)
        elif len(pts)==3:
            (x0, y0), (x1, y1), (x2, y2) = pts
            A = x0*(y1-y2)-y0*(x1-x2)+x1*y2-x2*y1
            B = (x0*x0+y0*y0)*(y2-y1)+(x1*x1+y1*y1)*(y0-y2)+(x2*x2+y2*y2)*(y1-y0)
            C = (x0*x0+y0*y0)*(x1-x2)+(x1*x1+y1*y1)*(x2-x0)+(x2*x2+y2*y2)*(x0-x1)
            D = (x0*x0+y0*y0)*(x2*y1-x1*y2) \
                +(x1*x1+y1*y1)*(x0*y2-x2*y0) \
                +(x2*x2+y2*y2)*(x1*y0-x0*y1)
            return (-B/(2*A),-C/(2*A),sqrt((B*B+C*C-4*A*D)/(4*A*A)))

    def welzl(pts, pt_on_edge):
        if len(pt_on_edge)==3 or not pts:
            return circle_less_than_3pts(pt_on_edge)
        exclude_pt = pts.pop() # exclude one random point.
        x,y,r = welzl(pts, pt_on_edge)
        if (exclude_pt[0]-x)**2+(exclude_pt[1]-y)**2<=r**2:
            res = x,y,r
        else:
            res = welzl(pts,pt_on_edge+[exclude_pt]) # 'exclude_pt' must lie on circle edge
        pts.append(exclude_pt) # backtracking putting removed point back.
        return res

    trees = list(set((x,y) for x,y in trees))
    shuffle(trees)
    return welzl(trees,[])
```