Data Structure & Algorithms

Sorting

Sorting is a technique to sort any list/array either in increasing or decreasing order. There are multiple sorting techniques.

Bubble Sort

- **Mechanism**: Repeatedly swap adjacent elements if they are in the wrong order.
- **Complexity**: `O(n^2)` for average and worst-case, O(n) for best case (already sorted).
- **Key Point**: Simple but inefficient.

```
def bubbleSort(ar,n):
    for i in range(n):
        swapped = False
        for j in range(n-1-i):
            if ar[j]>ar[j+1]:
                ar[j],ar[j+1]=ar[j+1], ar[j]
                swapped = True
        if not swapped:
            return

import random
n = random.randrange(10,15)
ar = [random.randrange(10, 100) for _ in range(n)]
print(f"List before sorting {ar}")
bubbleSort(ar,n)
print(f"List after sorting {ar}")
```

Selection Sort

- **Mechanism**: Divide the array into a sorted and an unsorted region. Repeatedly pick the smallest (or largest) element from the unsorted region and add it to the sorted region.
- **Complexity**: O(n^2) for average, worst, and best cases.
- **Key Point**: Inefficient but simple.

```
def selectionSort(ar,n):
    for i in range(n):
        min_ind=i
        for j in range(i+1,n):
            if ar[j]<ar[min_ind]:
                min_ind=j
        ar[min_ind],ar[i]=ar[i],ar[min_ind]

import random
n = random.randrange(10,15)
ar = [random.randrange(10, 100) for _ in range(n)]
print(f"List before sorting {ar}")
```

```
selectionSort(ar,n)
print(f"List after sorting {ar}")
```

Insertion Sort

- **Mechanism**: Build the final sorted array one item at a time by repeatedly removing one element from the input and inserting it into its correct position within the sorted list.
- **Complexity**: O(n^2) for average and worst-case, O(n) for best case (already sorted).
- **Key Point**: Efficient for small lists or nearly sorted lists.

```
def insertionSort(ar,n):
    for i in range(1,n):
        min_ele=ar[i]
        j=i-1
        while j>=0 and min_ele<ar[j]:
            ar[j+1]=ar[j]
            j-=1
        ar[j+1]=min_ele


import random
n = random.randrange(10,15)
ar = [random.randrange(10, 100) for _ in range(n)]
print(f"List before sorting {ar}")
insertionSort(ar,n)
print(f"List after sorting {ar}")
```

Merge Sort

- **Mechanism**: Divide the array in half, sort each half, and then merge them back together.
- **Complexity**: O(n log n) for average, worst, and best cases.
- **Key Point**: Divide and conquer approach. Stable sort. Requires O(n) additional space.

```
def merge(ar,low,mid,high):
    left=ar[low:mid+1]
    right=ar[mid+1:high+1]
    k=low
    i=j=0
    while i<len(left) and j<len(right):
        if left[i]<right[j]:
            ar[k]=left[i]
            i+=1
        else:
            ar[k]=right[j]
            j+=1
        k+=1

    while i<len(left):
        ar[k]=left[i]
        i+=1
        k+=1
```

```
    while j<len(right):
        ar[k]=right[j]
        j+=1
        k+=1

def mergeSort(ar,low, high):
    if low<high:
        mid=(low+high)//2
        mergeSort(ar,low,mid)
        mergeSort(ar,mid+1,high)
        merge(ar,low,mid,high)



import random
n = random.randrange(10,15)
ar = [random.randrange(10, 100) for _ in range(n)]
print(f"List before sorting {ar}")
mergeSort(ar,0,n-1)
print(f"List after sorting {ar}")
```

Back Tracking

Dynamic Programming

Dynamic Programming is nothing but recursion + optimization

Methods of Dynamic Programming

1. Memorization (Top > Bottom)
2. Tabulation (Bottom > Up)

Fibonacci

Using recursion

```
def fib(n):
    if n<2:
        return n
    else:
        return fib(n-1)+fib(n-2)
```

Using Memorization

```
def fib(n):
    memo = [None for _ in range(1000)]
    def fib_fun(n):
        if not memo[n]:
            if n<2:
                memo[n] = n
```

```
            else:
                memo[n] = fib_fun(n-1) + fib_fun(n-2)
        return memo[n]
    return fib_fun(n)
```

Using Tabulation

```
def fib(n):
    DP = [None for _ in range(n)]
    DP[0]=0
    DP[1]=1
    for i in range(2,n+1):
        DP[i] = DP[i-1]+DP[i-2]
    return DP[n]
```

Longest Common Subsequence

```
# recursive solution
def lcsRecursion(s1, s2, n, m):
    if n<1 or m<1:
        return 0
    else:
        if s1[n-1] == s2[m-1]:
            return 1 + lcsRecursion(s1, s2, n-1, m-1)
        else:
            return max(lcsRecursion(s1, s2, n-1, m),lcsRecursion(s1, s2, n, m-1))

def lcsMemorization(s1,s2,n,m):
    memo = [
        [None for _ in range(n+1)] for _ in range(m+1)
    ]
    def lcs(s1,s2,n,m):
        if not memo[m][n]:
            if m == 0 or n == 0:
                memo[m][n] = 0
            else:
                if s1[n-1] == s2[m-1]:
                    memo[m][n] = 1 + lcs(s1,s2,n-1,m-1)
                else:
                    memo[m][n] = max(lcs(s1,s2,n-1,m),lcs(s1,s2,n,m-1))
        return memo[m][n]
    return lcs(s1,s2,n,m)

def lcsTabulation(s1,s2,n,m):
    DP = [
        [0 for _ in range(n+1)] for _ in range(m+1)
    ]

    for i in range(1,m+1):
        for j in range(1,n+1):
```

```
            if s1[j-1] == s2[i-1]:
                DP[i][j] = 1 + DP[i-1][j-1]
            else:
                DP[i][j] = max(DP[i-1][j],DP[i][j-1])
    return DP[m][n]

s1 = "AGGTAB"
s2 = "GXTXAYB"
n = len(s1)
m = len(s2)
# ans is 4

print(lcsRecursion(s1, s2, n, m))
print(lcsMemorization(s1, s2, n, m))
print(lcsTabulation(s1, s2, n, m))
```

Coin Change Problem

```
def coinChangeRecursion(coins,sum,n):
    if n == 0 or sum <0:
        return 0
    if sum == 0:
        return 1
    else:
        return coinChangeRecursion(coins,sum-coins[n-1],n) +
coinChangeRecursion(coins,sum,n-1)


def coinChangeDP(coins,sum, n):
    DP = [0 for _ in range(sum+1)]
    DP[0] = 1
    for i in range(n):
        for j in range(coins[i],sum+1):
            DP[j] += DP[j-coins[i]]
    return DP[sum]

coins = [1, 2, 3]
sum = 5
n = len(coins)
# solution is 5
print(coinChangeRecursion(coins,sum,n))
print(coinChangeDP(coins,sum,n))
```

Edit Distance

```
def editDistanceRecursion(s,p,n,m):
    if n==0:
        return m
    if m == 0:
        return n
```

```
    if s[n-1] == p[m-1]:
        return editDistanceRecursion(s,p,n-1,m-1)
    else:
        return 1 + min(
            editDistanceRecursion(s,p,n-1,m),
            editDistanceRecursion(s,p,n,m-1),
            editDistanceRecursion(s,p,n-1,m-1)
        )

def editDistanceDP(s,p,n,m):
    DP = [
        [0 for _ in range(n+1)] for _ in range(m+1)
    ]
    for i in range(m+1):
        DP[i][0] = i
    for i in range(n+1):
        DP[0][i] = i

    for i in range(1,m+1):
        for j in range(1,n+1):
            if s[j-1] == p[i-1]:
                DP[i][j] = DP[i-1][j-1]
            else:
                DP[i][j] = 1 + min(
                    DP[i-1][j],
                    DP[i][j-1],
                    DP[i-1][j-1]
                )
    return DP[m][n]

s = "sit"
p = "kiit"
n = len(s)
m = len(p)

print(editDistanceRecursion(s,p,n,m))
print(editDistanceDP(s,p,n,m))
```

Subarray

A subarray is a contiguous or non-empty portion of an array. In the context of an array, a subarray is a subset of the original array that maintains the relative order of the elements.

Maximum Subarray

Given an integer array nums, find the subarray with the largest sum, and return its sum.

```
# Return sum of maximum subarray
def sumMaxSubarray(ar):
    n=len(ar)
    if n<1:
```

```
        return 0
    c_sum = m_sum = ar[0]
    for x in ar[1:]:
        if x>c_sum+x:
            c_sum=x
        else:
            c_sum+=x
        if c_sum>m_sum:
            m_sum=c_sum
    return m_sum

# Example usage:
nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
max_sum = sumMaxSubarray(nums)
print("Maximum subarray sum:", max_sum)
```

Given an integer array number, find the subarray with the largest sum, and return the subarray.

```
# Return the maximum subarray
def maxSubarray(ar):
    n=len(ar)
    if n<1:
        return 0
    c_sum = m_sum = ar[0]
    c_start=0
    st=end=0
    for i in range(n):
        x=ar[i]
        if x>c_sum+x:
            c_sum=x
            c_start = i
        else:
            c_sum+=x
        if c_sum>m_sum:
            m_sum=c_sum
            st=c_start
            end=i
    return ar[st:end+1]

# Example usage:
nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
max_subarray = maxSubarray(nums)
print("Maximum subarray :", max_subarray)
```

Maximum Product Subarray

Given an integer array nums, find a subarray, that has the largest product, and return the product.

```
def maxProdSubarray(ar):
    n = len(ar)
    if n < 1:
```

```
        return 0

    max_prod = min_prod = ans = ar[0]

    for i in range(1,n):
        num = ar[i]
        if num<0:
            max_prod, min_prod = min_prod, max_prod

        if num > max_prod*num:
            max_prod = num
        else:
            max_prod*=num
        if num < min_prod*num:
            min_prod = num
        else:
            min_prod*=num

        if ans<max_prod:
            ans=max_prod
    return ans

# Example usage:
nums = [2,3,-2,4]
max_prod = maxProdSubarray(nums)
print("Maximum subarray prod:", max_prod)
```

Competitive Programming

Basic Problems

Largest element in the array

Check if the array is sorted or not

Return the second largest element in the array

Remove duplicates from the sorted array

Move all zeros at the end

Reverse the array

Left rotate an array by 1

Advance Problems

Left rotate an array by D

```
def reverseArray(ar, s = None, l=None):
    l = l or len(ar)
    s = s or 0
    for i in range((l-s)//2):
        ar[s+i], ar[l-i-1] = ar[l-i-1], ar[s+i]
    return ar
```

```
def leftRotateByD(ar,D):
    l = len(ar)
    ar = reverseArray(ar,0,D)
    ar = reverseArray(ar,D,l)
    ar = reverseArray(ar)
    return ar

ar = leftRotateByD([0,1,2,3,4,5,6,7],3)
'''
[1,2,3,4,5,6,7,0]
[2,3,4,5,6,7,0,1]
[3,4,5,6,7,0,1,2]
'''
print(ar)
```

Leaders of the array

```
def leaders(ar):
    l = len(ar)
    ans = [ar[-1]]
    for i in range(l-2,-1,-1):
        if ar[i]>ans[-1]:
            ans.append(ar[i])
    return ans[::-1]

print(leaders([6,7,1,2,3,4,2,1]))
```

Max Diff ar[i]-ar[j] is max for i>j

```
def maxDiff(ar):
    res=ar[1]-ar[0]
    mn = ar[0]
    for i in range(1,len(ar)):
        res = max(res, ar[i]-mn)
        mn = min(ar[i],mn)
    return res
print(maxDiff([6,7,1,2,3,4,2,1]))
```

Frequency of Elements in a Sorted Array

```
def frequency(ar):
    ans = {}
    ele= ar[0]
    cnt = 1
    for i in range(1,len(ar)):
        if ar[i]==ele:
            cnt+=1
        else:
```

```
            ans[ele] = cnt
            ele = ar[i]
            cnt =1
    ans[ele] = cnt
    return ans

print(frequency([0,0,0,1,1,1,1,1,1,2,2,2,2,2,3,3,3,3,4]))
```

Stock Buy and Sell | Max Profit

```
def maxProfitNaive(ar, st, nd):
    if nd<=st:
        return 0
    profit = 0
    for i in range(st,nd-1):
        for j in range(i+1,nd):
            if ar[j]>ar[i]:
                cur_profit = (ar[j]-ar[i]) + maxProftNaive(ar,st,i-1) +
maxProftNaive(ar,j+1,nd)
                profit = max(profit, cur_profit)
    return profit

def maxProfit(ar, st, nd): # sell if price graph is going up
    profit = 0
    for i in range(1, nd):
        if ar[i]>ar[i-1]:
            profit+=(ar[i]-ar[i-1])
    return profit

price = [2,3,4,1,5,9,2,9]
st = 0
nd = len(price)

print(maxProfitNaive(price,st,nd))
print(maxProfit(price,st,nd))
```

Trapping rain water

```
def trapWater(ar):
    l = len(ar)
    lmax = ar[:]
    rmax = ar[:]
    for i in range(1,l):
        lmax[i] = max(lmax[i-1],lmax[i])
        rmax[l-i-1] = max(rmax[l-i],rmax[l-i-1])
    ans = 0
    for i in range(l):
        ans += (min(lmax[i],rmax[i])-ar[i])
    return ans
```

```
ar = [3,0,2,1,5]
print(trapWater(ar))
```

Caden's Algorithm

Either extend the current or start a new

Max consecutive 1 in a binary array

```
def maxCons1(ar):
    ans = 0
    cur = 0
    for i in range(len(ar)):
        if ar[i]==1:
            cur+=1
        else:
            ans = max(cur, ans)
            cur = 0
    ans = max(cur, ans)
    return ans

ar = [0,1,1,0,1,1,1,0]

print(maxCons1(ar))
```

Max Subarray Sum

```
def maxSubarraySumNaive(ar):
    ans = ar[0]
    for i in range(len(ar)):
        cur = 0
        for j in range(i,len(ar)):
            cur += ar[j]
            if cur > ans:
                ans = cur
    return ans

def maxSubarraySum(ar):
    max_ending = ar[:]
    for i in range(1,len(ar)):
        max_ending[i] = max(max_ending[i-1]+ar[i],ar[i])
    return max(max_ending)

ar = [1,-2,3,-1,2]

print(maxSubarraySumNaive(ar))
print(maxSubarraySum(ar))
```

Max length even odd subarray

```
def maxLengthEvenOddSubarrayNaive(ar):
    ans = 1
    for i in range(1,len(ar)):
        cur = 1
        for j in range(i,len(ar)):
            if (ar[j-1]%2==0 and ar[j]%2==1) or (ar[j-1]%2==1 and ar[j]%2==0):
                cur+=1
            else:
                ans = max(cur, ans)
                cur = 1
        ans = max(cur, ans)
    return ans

def maxLengthEvenOddSubarray(ar):
    ans = 1
    cur = 1
    for i in range(1,len(ar)):
        if (ar[i-1]%2==1 and ar[i]%2==0) or (ar[i-1]%2==0 and ar[i]%2==1):
            cur +=1
        else:
            ans = max(cur,ans)
            cur =1
    ans = max(cur, ans)
    return ans

ar = [5,10,6,20,3,8]
print(maxLengthEvenOddSubarrayNaive(ar))
print(maxLengthEvenOddSubarray(ar))
```

Max Circular Subarray Sum

```
def maxCircularSubarraySumNaive(ar):
    ans = ar[0]
    l = len(ar)
    for i in range(l):
        cur_sum = ar[i]
        cur_max = ar[i]
        for j in range(1,l):
            ind = (i+j)%l
            cur_sum += ar[ind]
            cur_max = max(cur_sum, cur_max)
        ans = max(ans, cur_max)
    return ans


def maxCircularSubarraySum(ar):
    def maxSubarraySum(ar):
        max_ending = ar[:]
        for i in range(1, len(ar)):
            max_ending[i] = max(max_ending[i-1]+ar[i],ar[i])
```

```
        return max(max_ending)

    def minSubarraySum(ar):
        min_ending = ar[:]
        for i in range(1,len(ar)):
            min_ending[i] = min(min_ending[i-1]+ar[i],ar[i])
        return min(min_ending)

    total = sum(ar)
    max_sum = maxSubarraySum(ar)
    min_sum = minSubarraySum(ar)
    return max(max_sum, total-min_sum)

ar = [5, -2, 3, 4]

print(maxCircularSubarraySumNaive(ar))
print(maxCircularSubarraySum(ar))
```

Moore's Algorithm

- Find in 1st part
- Verify in 2nd Part

Find majority element, count>n/2

```
def findMajority(ar):
    res = 0
    cnt = 1
    for i in range(1,len(ar)):
        if ar[res]==ar[i]:
            cnt+=1
        else:
            cnt -=1
        if cnt == 0:
            res = i
            cnt = 1
    return res

def verifyMajority(ar,major):
    cnt =0
    for x in ar:
        if x == ar[major]:
            cnt+=1
    if cnt>len(ar)//2:
        return major
    else:
        return -1

ar = [6,8,4,8,8]

print(verifyMajority(ar,findMajority(ar)))
```

Sliding window Technique

Maximum sum of consecutive K numbers

```
def maxConsecutiveKsum(ar, k):
    ksum = 0
    for i in range(k):
        ksum+=ar[i]
    ans = ksum
    for i in range(k,len(ar)):
        ksum += ar[i]
        ksum -= ar[i-k]
        ans = max(ans,ksum)
    return ans

ar = [1,8,30,-5,20,7]
k =3
print(maxConsecutiveKsum(ar, k))
```

Prefix Sum

Calculate sum prefix data, to answer multiple queries on a fixed data

Sum b/w 2 indexs of array

```
def getSum(ar,queries):
    prefix_sum = ar[:]
    for i in range(1,len(ar)):
        prefix_sum[i] += prefix_sum[i-1]
    prefix_sum = [0] +prefix_sum
    print(prefix_sum)
    for l,r in queries:
        print(prefix_sum[r+1]-prefix_sum[l])

ar = [2,8,3,9,6,5,4]
queries = [(0,2),(1,3),(2,6)]

getSum(ar,queries)
```

Max occurring element b/w given queries

```
def maxOccuring(queries):
    ar = [0 for _ in range(10)]
    for l,r in queries:
        ar[l]+=1
        ar[r+1]-=1
    prefix_sum = ar[:]
    for i in range(1,len(ar)):
        prefix_sum[i] += prefix_sum[i-1]
    print(prefix_sum)
    mx = 0
```

```
    ans = 0
    for i in range(len(prefix_sum)):
        if prefix_sum[i]>mx:
            mx = prefix_sum[i]
            ans = i
    return ans

queries =[(1,3),(2,5),(3,7)]

print(maxOccuring(queries))
```