{ "cells": [{ "cell_type": "markdown", "metadata": {}, "source": ["# Data Structure & Algoritms"] }, { "cell_type": "markdown", "metadata": {}, "source": ["## Basic"] }, { "cell_type": "markdown", "metadata": {}, "source": ["## String"] }, { "cell_type": "code", "execution_count": 1, "metadata": {}, "outputs": [], "source": ["s='String'"] }, { "cell_type": "markdown", "metadata": {}, "source": ["## List"] }, { "cell_type": "code", "execution_count": 2, "metadata": {}, "outputs": [], "source": ["l=list()"] }, { "cell_type": "markdown", "metadata": {}, "source": ["## Dictionary"] }, { "cell_type": "code", "execution_count": 3, "metadata": {}, "outputs": [], "source": ["d=dict()"] }, { "cell_type": "markdown", "metadata": {}, "source": ["## Other Primitive Data types"] }, { "cell_type": "code", "execution_count": null, "metadata": {}, "outputs": [], "source": [] }, { "cell_type": "markdown", "metadata": {}, "source": ["## Linked List"] }, { "cell_type": "code", "execution_count": 4, "metadata": {}, "outputs": [], "source": ["class Node:\n", " def **init**(self,val,next=None):\n", " self.val=val\n", " self.next=next\n", "\n", "class LinkedList:\n", " def **init**(self,val):\n", " self.head=Node(val)"] }, { "cell_type": "markdown", "metadata": {}, "source": ["## Tree"] }, { "cell_type": "code", "execution_count": 5, "metadata": {}, "outputs": [], "source": ["class Node:\n", " def **init**(self,val,left=None,right=None):\n", " self.val=val\n", " self.left=left\n", " self.right=right\n", "\n", "class LinkedList:\n", " def **init**(self,val):\n", " self.head=Node(val)"] }, { "cell_type": "markdown", "metadata": {}, "source": ["## Graph"] }, { "cell_type": "code", "execution_count": null, "metadata": {}, "outputs": [], "source": [] }, { "cell_type": "markdown", "metadata": {}, "source": ["## Traversing"] }, { "cell_type": "code", "execution_count": 6, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["1\n", "2\n", "3\n", "4\n", "5\n"] }], "source": ["l=[1,2,3,4,5]\n", "n=len(l)\n", "for i in range(n):\n", " print(l[i])"] }, { "cell_type": "markdown", "metadata": {}, "source": ["## Searching"] }, { "cell_type": "code", "execution_count": 7, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["3\n", "-1\n"] }], "source": ["l=[1,2,3,4,5]\n", "n=len(l)\n", "se= 4\n", "for i in range(n):\n", " if l[i]==se:\n", " print(i)\n", "print(-1)"] }, { "cell_type": "markdown", "metadata": {}, "source": ["## Sorting\n", "\n", "Sorting is a technique to sort any list/array either in increasing or decreasing order. There are multiple sorting techniques.\n"] }, { "cell_type": "markdown", "metadata": {}, "source": ["### Bubble Sort\n", "- **Mechanism** : Repeatedly swap adjacent elements if they are in the wrong order.\n", "- **Complexity**: `O(n^2)` for average and worst-case, O(n) for best case (already sorted).\n", "- **Key Point**: Simple but inefficient."] }, { "cell_type": "code", "execution_count": 8, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["List before sorting [39, 87, 29, 37, 55, 43, 23, 32, 73, 17, 16]\n", "List after sorting [16, 17, 23, 29, 32, 37, 39, 43, 55, 73, 87]\n"] }], "source": ["def bubbleSort(ar,n):\n", " for i in range(n):\n", " swapped = False\n", " for j in range(n-1-i):\n", " if ar[j]>ar[j+1]:\n", " ar[j],ar[j+1]=ar[j+1], ar[j]\n", " swapped = True\n", " if not swapped:\n", " return\n", "\n", "import random\n", "n = random.randrange(10,15)\n", "ar = [random.randrange(10, 100) for _ in range(n)]\n", "print(f"List before sorting {ar}")\n", "bubbleSort(ar,n)\n", "print(f"List after sorting {ar}")\n"] }, { "cell_type": "markdown", "metadata": {}, "source": ["### Selection Sort\n", "\n", "- **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.\n", "- **Complexity**: O(n^2) for average, worst, and best cases.\n", "- **Key Point**: Inefficient but simple."] }, { "cell_type": "code", "execution_count": 9, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["List before sorting [55, 96, 65, 17, 65, 98, 29, 97, 47, 41, 49, 52, 80, 71]\n", "List after sorting [17, 29, 41, 47, 49, 52, 55, 65, 65, 71, 80, 96, 97, 98]\n"] }], "source": ["def selectionSort(ar,n):\n", " for i in range(n):\n", " min_ind=i\n", " for j in range(i+1,n):\n", " if ar[j]<ar[min_ind]:\n", " min_ind=j\n", " ar[min_ind],ar[i]=ar[i],ar[min_ind]\n", "\n", "import random\n", "n = random.randrange(10,15)\n", "ar = [random.randrange(10, 100) for _ in range(n)]\n", "print(f"List before sorting {ar}")\n", "selectionSort(ar,n)\n", "print(f"List after sorting {ar}")\n"] }, { "cell_type": "markdown", "metadata": {}, "source": ["### Insertion Sort\n", "\n", "- *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.\n", "- **Complexity**: O(n^2) for average and worst-case, O(n) for best case (already sorted).\n", "- **Key Point**: Efficient for small lists or nearly sorted lists."] }, { "cell_type": "code", "execution_count": 10, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["List before sorting [62, 18, 93, 56, 62, 89, 97, 13, 19, 53]\n", "List after sorting [13, 18, 19, 53, 56, 62, 62, 89, 93, 97]\n"] }], "source": ["def insertionSort(ar,n):\n", " for i in range(1,n):\n", " min_ele=ar[i]\n", " j=i-1\n", " while j>=0 and min_ele<ar[j]:\n", " ar[j+1]=ar[j]\n", " j-=1\n", " ar[j+1]=min_ele\n", "\n", "\n", "import random\n", "n = random.randrange(10,15)\n", "ar = [random.randrange(10, 100) for _ in range(n)]\n", "print(f"List before sorting {ar}")\n", "insertionSort(ar,n)\n", "print(f"List after sorting {ar}")"] }, { "cell_type": "markdown", "metadata": {}, "source": ["### Merge Sort\n", "\n", "- **Mechanism**: Divide the array in half, sort each half, and then merge them

back together.\n", "- **Complexity**: O(n log n) for average, worst, and best cases.\n", "- **Key Point**: Divide and conquer approach. Stable sort. Requires O(n) additional space."] }, { "cell_type": "code", "execution_count": null, "metadata": {}, "outputs": [], "source": ["def merge(ar,low,mid,high):\n", " left=ar[low:mid+1]\n", " right=ar[mid+1:high+1]\n", " k=low\n", " i=j=0\n", " while i<len(left) and j<len(right):\n", " if left[i]<right[j]:\n", " ar[k]=left[i]\n", " i+=1\n", " else:\n", " ar[k]=right[j]\n", " j+=1\n", " k+=1\n", " \n", " while i<len(left):\n", " ar[k]=left[i]\n", " i+=1\n", " k+=1\n", " \n", " while j<len(right):\n", " ar[k]=right[j]\n", " j+=1\n", " k+=1\n", "\n", "def mergeSort(ar,low, high):\n", " if low<high:\n", " mid=(low+high)//2\n", " mergeSort(ar,low,mid)\n", " mergeSort(ar,mid+1,high)\n", " merge(ar,low,mid,high)\n", "\n", "\n", "\n", "import random\n", "n = random.randrange(10,15)\n", "ar = [random.randrange(10, 100) for _ in range(n)]\n", "print(f"List before sorting {ar}")\n", "mergeSort(ar,0,n-1)\n", "print(f"List after sorting {ar}")"] }, { "cell_type": "markdown", "metadata": {}, "source": ["## Back Tracking"] }, { "cell_type": "code", "execution_count": null, "metadata": {}, "outputs": [], "source": [] }, { "cell_type": "markdown", "metadata": {}, "source": ["## Dynamic Programming\n", "\n", "Dynamic Programming is nothing but recursion + optimization\n", "\n", "**Methods of Dynamic Programming**\n", "1. Memorization (Top > Bottom)\n", "2. Tabulation (Bottom > Up)\n", "\n", "### Fibonacci\n", "\n", "#### Using recursion\n", " `python\n", "def fib(n):\n", " if n<2:\n", " return n\n", " else:\n", " return fib(n-1)+fib(n-2)\n", " `\n", "\n", "#### Using Memorization\n", " `python\n", "def fib(n):\n", " memo = [None for _ in range(1000)]\n", " def fib_fun(n):\n", " if not memo[n]:\n", " if n<2:\n", " memo[n] = n\n", " else:\n", " memo[n] = fib_fun(n-1) + fib_fun(n-2)\n", " return memo[n]\n", " return fib_fun(n)\n", " `\n", "\n", "#### Using Tabulation\n", " `python\n", "def fib(n):\n", " DP = [None for _ in range(n)]\n", " DP[0]=0\n", " DP[1]=1\n", " for i in range(2,n+1):\n", " DP[i] = DP[i-1]+DP[i-2]\n", " return DP[n]\n", " `\n"] }, { "cell_type": "markdown", "metadata": {}, "source": ["### Longest Common Subsequence"] }, { "cell_type": "code", "execution_count": 60, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["4\n", "4\n", "4\n"] }], "source": ["# recursive solution\n", "def lcsRecursion(s1, s2, n, m):\n", " if n<1 or m<1:\n", " return 0\n", " else:\n", " if s1[n-1] == s2[m-1]:\n", " return 1 + lcsRecursion(s1, s2, n-1, m-1)\n", " else:\n", " return max(lcsRecursion(s1, s2, n-1, m),lcsRecursion(s1, s2, n, m-1))\n", "\n", "def lcsMemorization(s1,s2,n,m):\n", " memo = [\n", " [None for _ in range(n+1)] for _ in range(m+1)\n", "]\n", " def lcs(s1,s2,n,m):\n", " if not memo[m][n]:\n", " if m == 0 or n == 0:\n", " memo[m][n] = 0\n", " else:\n", " if s1[n-1] == s2[m-1]:\n", " memo[m][n] = 1 + lcs(s1,s2,n-1,m-1)\n", " else:\n", " memo[m][n] = max(lcs(s1,s2,n-1,m),lcs(s1,s2,n,m-1))\n", " return memo[m][n]\n", " return lcs(s1,s2,n,m)\n", "\n", "def lcsTabulation(s1,s2,n,m):\n", " DP = [\n", " [0 for _ in range(n+1)] for _ in range(m+1)\n", "]\n", "\n", " for i in range(1,m+1):\n", " for j in range(1,n+1):\n", " if s1[j-1] == s2[i-1]:\n", " DP[i][j] = 1 + DP[i-1][j-1]\n", " else:\n", " DP[i][j] = max(DP[i-1][j],DP[i][j-1])\n", " return DP[m][n]\n", "\n", "s1 = "AGGTAB"\n", "s2 = "GXTXAYB"\n", "n = len(s1)\n", "m = len(s2)\n", "# ans is 4\n", "\n", "print(lcsRecursion(s1, s2, n, m))\n", "print(lcsMemorization(s1, s2, n, m))\n", "print(lcsTabulation(s1, s2, n, m))"] }, { "cell_type": "markdown", "metadata": {}, "source": ["### Coin Change Problem"] }, { "cell_type": "code", "execution_count": 66, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["5\n", "5\n"] }], "source": ["def coinChangeRecursion(coins,sum,n):\n", " if n == 0 or sum <0:\n", " return 0\n", " if sum == 0:\n", " return 1\n", " else:\n", " return coinChangeRecursion(coins,sum-coins[n-1],n) + coinChangeRecursion(coins,sum,n-1)\n", "\n", "\n", "def coinChangeDP(coins,sum, n):\n", " DP = [0 for _ in range(sum+1)]\n", " DP[0] = 1\n", " for i in range(n):\n", " for j in range(coins[i],sum+1):\n", " DP[j] += DP[j-coins[i]]\n", " return DP[sum]\n", "\n", "coins = [1, 2, 3]\n", "sum = 5\n", "n = len(coins)\n", "# solution is 5\n", "print(coinChangeRecursion(coins,sum,n))\n", "print(coinChangeDP(coins,sum,n))"] }, { "cell_type": "markdown", "metadata": {}, "source": ["### Edit Distance"] }, { "cell_type": "code", "execution_count": 75, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["2\n", "2\n"] }], "source": ["def editDistanceRecursion(s,p,n,m):\n", " if n==0:\n", " return m\n", " if m == 0:\n", " return n\n", " if s[n-1] == p[m-1]:\n", " return editDistanceRecursion(s,p,n-1,m-1)\n", " else:\n", " return 1 + min(\n", " editDistanceRecursion(s,p,n-1,m),\n", " editDistanceRecursion(s,p,n,m-1),\n", " editDistanceRecursion(s,p,n-1,m-1)\n", ")\n", "\n", "def editDistanceDP(s,p,n,m):\n", " DP = [\n", " [0 for _ in range(n+1)] for _ in range(m+1)\n", "]\n", " for i in range(m+1):\n", " DP[i][0] = i\n", " for i in range(n+1):\n", " DP[0][i] = i\n", "\n", " for i in range(1,m+1):\n", " for j in range(1,n+1):\n", " if s[j-1] == p[i-1]:\n", " DP[i][j] = DP[i-1][j-1]\n", " else:\n", " DP[i][j] = 1 + min(\n", " DP[i-1][j],\n", " DP[i][j-1],\n", " DP[i-1][j-1]\n", ")\n", " return DP[m][n]\n", "\n", "s = "sit"\n", "p = "kiit"\n", "n = len(s)\n", "m = len(p)\n", "\n", "print(editDistanceRecursion(s,p,n,m))\n", "print(editDistanceDP(s,p,n,m))"] }, { "cell_type": "markdown", "metadata": {}, "source": ["## Subarray\n", "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."] }, { "cell_type": "markdown", "metadata": {}, "source": ["### Maximum Subarray\n", "Given an integer array nums, find the subarray with the largest sum, and return its sum."] }, { "cell_type": "code", "execution_count": 11, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["Maximum subarray sum: 6\n"] }], "source": ["# Return sum of maximum subarray\n", "def sumMaxSubarray(ar):\n", " n=len(ar)\n", " if n<1:\n", " return 0\n", " c_sum = m_sum = ar[0]\n", " for x in ar[1:]:\n", " if x>c_sum+x:\n", " c_sum=x\n", " else:\n", " c_sum+=x\n", " if c_sum>m_sum:\n", " m_sum=c_sum\n", " return m_sum\n", "\n", "# Example usage:\n", "nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]\n", "max_sum = sumMaxSubarray(nums)\n", "print("Maximum subarray sum:", max_sum)"] }, { "cell_type": "markdown", "metadata": {}, "source": ["Given an integer array nums, find the subarray with the largest sum, and return the subarray."] }, { "cell_type": "code", "execution_count": 13, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["Maximum subarray : [4, -1, 2, 1]\n"] }], "source": ["# Return the maximum subarray\n", "def maxSubarray(ar):\n", " n=len(ar)\n", " if n<1:\n", " return 0\n", " c_sum = m_sum = ar[0]\n", " c_start=0\n", " st=end=0\n", " for i in range(n):\n", " x=ar[i]\n", " if x>c_sum+x:\n", " c_sum=x\n", " c_start = i\n", " else:\n", " c_sum+=x\n", " if c_sum>m_sum:\n", " m_sum=c_sum\n", " st=c_start\n", " end=i\n", " return ar[st:end+1]\n", "\n", "# Example usage:\n", "nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]\n", "max_subarray = maxSubarray(nums)\n", "print("Maximum subarray :", max_subarray)"] }, { "cell_type": "markdown", "metadata": {}, "source": ["### Maximum Product Subarray\n", "Given an integer array nums, find a subarray, that has the largest product, and return the product."] }, { "cell_type": "code", "execution_count": 15, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["Maximum subarray prod: 6\n"] }], "source": ["def maxProdSubarray(ar):\n", " n = len(ar)\n", " if n < 1:\n", " return 0\n", " \n", " max_prod = min_prod = ans = ar[0]\n", " \n", " for i in range(1,n):\n", " num = ar[i]\n", " if num<0:\n", " max_prod, min_prod = min_prod, max_prod\n", " \n", " if num > max_prodnum:\n", " max_prod = num\n", " else:\n", " max_prod*=num\n", " if num < min_prod*num:\n", " min_prod = num\n", " else:\n", " min_prod*=num\n", " \n", " if ans<max_prod:\n", " ans=max_prod\n", " return ans\n", "\n", "# Example usage:\n", "nums = [2,3,-2,4]\n", "max_prod = maxProdSubarray(nums)\n", "print("Maximum subarray prod:", max_prod)"] }, { "cell_type": "markdown", "metadata": {}, "source": ["## Other Algorithms"] }, { "cell_type": "code", "execution_count": null, "metadata": {}, "outputs": [], "source": [] }, { "cell_type": "markdown", "metadata": {}, "source": ["\n", "## Competitive Programming\n", "\n", "### Basic Problems"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Largest element in the array\n", "\n", "#### Check if the array is sorted or not\n", "\n", "#### Return the second largest element in the array\n", "\n", "#### Remove duplicates from the sorted array\n", "\n", "#### Move all zeros at the end\n", "\n", "#### Reverse the array\n", "\n", "#### Left rotate an array by 1"] }, { "cell_type": "markdown", "metadata": {}, "source": ["### Advance Problems"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Left rotate an array by D O(n)"] }, { "cell_type": "code", "execution_count": 17, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["[3, 4, 5, 6, 7, 0, 1, 2]\n"] }], "source": ["def reverseArray(ar, s = None, l=None):\n", " l = l or len(ar)\n", " s = s or 0\n", " for i in range((l-s)//2):\n", " ar[s+i], ar[l-i-1] = ar[l-i-1], ar[s+i]\n", " return ar\n", "\n", "def leftRotateByD(ar,D):\n", " l = len(ar)\n", " ar = reverseArray(ar,0,D)\n", " ar = reverseArray(ar,D,l)\n", " ar = reverseArray(ar)\n", " return ar\n", "\n", "ar = leftRotateByD([0,1,2,3,4,5,6,7],3)\n", "'''\n", "[1,2,3,4,5,6,7,0]\n", "[2,3,4,5,6,7,0,1]\n", "[3,4,5,6,7,0,1,2]\n", "'''\n", "print(ar)\n", "\n"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Leaders of the array"] }, { "cell_type": "code", "execution_count": 19, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["[7, 4, 2, 1]\n"] }], "source": ["def leaders(ar):\n", " l = len(ar)\n", " ans = [ar[-1]]\n", " for i in range(l-2,-1,-1):\n", " if ar[i]>ans[-1]:\n", " ans.append(ar[i])\n", " return ans[::-1]\n", "\n", "print(leaders([6,7,1,2,3,4,2,1]))"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Max Diff ar[i]-ar[j] is max for i>j"] }, { "cell_type": "code", "execution_count": 21, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["3\n"] }], "source": ["def maxDiff(ar):\n", " res=ar[1]-ar[0]\n", " mn = ar[0]\n", " for i in range(1,len(ar)):\n", " res = max(res, ar[i]-mn)\n", " mn = min(ar[i],mn)\n", " return res\n", "print(maxDiff([6,7,1,2,3,4,2,1]))"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Frequency of Elements in a sorted array"] }, { "cell_type": "code", "execution_count": 23, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["{0: 3, 1: 6, 2: 5, 3: 4, 4: 1}\n"] }], "source": ["def frequency(ar):\n", " ans = {}\n", " ele= ar[0]\n", " cnt = 1\n", " for i in range(1,len(ar)):\n", " if ar[i]==ele:\n", " cnt+=1\n", " else:\n", " ans[ele] = cnt\n", " ele = ar[i]\n", " cnt =1\n", " ans[ele] = cnt\n", " return ans\n", "\n", "print(frequency([0,0,0,1,1,1,1,1,1,2,2,2,2,2,3,3,3,3,4]))\n"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Stock Buy and Sell | Max Profit"] }, { "cell_type": "code", "execution_count": 25, "metadata": {}, "outputs": [{ "name":

"stdout", "output_type": "stream", "text": ["17\n", "17\n"] }], "source": ["def maxProfitNaive(ar, st, nd):\n", " if nd<=st:\n", " return 0\n", " profit = 0\n", " for i in range(st,nd-1):\n", " for j in range(i+1,nd):\n", " if ar[j]>ar[i]:\n", " cur_profit = (ar[j]-ar[i]) + maxProfitNaive(ar,st,i-1) + maxProfitNaive(ar,j+1,nd)\n", " profit = max(profit, cur_profit)\n", " return profit\n", "\n", "def maxProfit(ar, st, nd): # sell if price graph is going up\n", " profit = 0\n", " for i in range(1, nd):\n", " if ar[i]>ar[i-1]:\n", " profit+=(ar[i]-ar[i-1])\n", " return profit\n", "\n", "price = [2,3,4,1,5,9,2,9]\n", "st = 0\n", "nd = len(price)\n", "\n", "print(maxProfitNaive(price,st,nd))\n", "print(maxProfit(price,st,nd))\n"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Trapping rain water"] }, { "cell_type": "code", "execution_count": 26, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["6\n"] }], "source": ["def trapWater(ar):\n", " l = len(ar)\n", " lmax = ar[:]\n", " rmax = ar[:]\n", " for i in range(1,l):\n", " lmax[i] = max(lmax[i-1],lmax[i])\n", " rmax[l-i-1] = max(rmax[l-i],rmax[l-i-1])\n", " ans = 0\n", " for i in range(l):\n", " ans += (min(lmax[i],rmax[i])-ar[i])\n", " return ans\n", "\n", "ar = [3,0,2,1,5]\n", "print(trapWater(ar))"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Caden's Algorithm\n", "\n", "Either extend the current or start a new\n"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Max consecutive 1 in a binary array"] }, { "cell_type": "code", "execution_count": 28, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["3\n"] }], "source": ["def maxCons1(ar):\n", " ans = 0\n", " cur = 0\n", " for i in range(len(ar)):\n", " if ar[i]==1:\n", " cur+=1\n", " else:\n", " ans = max(cur, ans)\n", " cur = 0\n", " ans = max(cur, ans)\n", " return ans\n", "\n", "ar = [0,1,1,0,1,1,1,0]\n", "\n", "print(maxCons1(ar))"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Max Subarray Sum"] }, { "cell_type": "code", "execution_count": 30, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["4\n", "4\n"] }], "source": ["def maxSubarraySumNaive(ar):\n", " ans = ar[0]\n", " for i in range(len(ar)):\n", " cur = 0\n", " for j in range(i,len(ar)):\n", " cur += ar[j]\n", " if cur > ans:\n", " ans = cur\n", " if cur > ans:\n", " ans = cur\n", " return ans\n", "\n", "def maxSubarraySum(ar):\n", " max_ending = ar[:]\n", " for i in range(1,len(ar)):\n", " max_ending[i] = max(max_ending[i-1]+ar[i],ar[i])\n", " return max(max_ending)\n", "\n", "ar = [1,-2,3,-1,2]\n", "\n", "print(maxSubarraySumNaive(ar))\n", "print(maxSubarraySum(ar))"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Max length even odd subarray"] }, { "cell_type": "code", "execution_count": 33, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["3\n", "3\n"] }], "source": ["def maxLengthEvenOddSubarrayNaive(ar):\n", " ans = 1\n", " for i in range(1,len(ar)):\n", " cur = 1\n", " for j in range(i,len(ar)):\n", " if (ar[j-1]%2==0 and ar[j]%2==1) or (ar[j-1]%2==1 and ar[j]%2==0):\n", " cur+=1\n", " else:\n", " ans = max(cur, ans)\n", " cur = 1\n", " ans = max(cur, ans)\n", " return ans\n", "\n", "def maxLengthEvenOddSubarray(ar):\n", " ans = 1\n", " cur = 1\n", " for i in range(1,len(ar)):\n", " if (ar[i-1]%2==1 and ar[i]%2==0) or (ar[i-1]%2==0 and ar[i]%2==1):\n", " cur +=1\n", " else:\n", " ans = max(cur,ans)\n", " cur =1\n", " ans = max(cur, ans)\n", " return ans\n", "\n", "ar = [5,10,6,20,3,8]\n", "print(maxLengthEvenOddSubarrayNaive(ar))\n", "print(maxLengthEvenOddSubarray(ar))"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Max Circular Subarray Sum"] }, { "cell_type": "code", "execution_count": 35, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["12\n", "12\n"] }], "source": ["def maxCircularSubarraySumNaive(ar):\n", " ans = ar[0]\n", " l = len(ar)\n", " for i in range(l):\n", " cur_sum = ar[i]\n", " cur_max = ar[i]\n", " for j in range(1,l):\n", " ind = (i+j)%l\n", " cur_sum += ar[ind]\n", " cur_max = max(cur_sum, cur_max)\n", " ans = max(ans, cur_max)\n", " return ans\n", "\n", "\n", "def maxCircularSubarraySum(ar):\n", " def maxSubarraySum(ar):\n", " max_ending = ar[:]\n", " for i in range(1, len(ar)):\n", " max_ending[i] = max(max_ending[i-1]+ar[i],ar[i])\n", " return max(max_ending)\n", "\n", " def minSubarraySum(ar):\n", " min_ending = ar[:]\n", " for i in range(1,len(ar)):\n", " min_ending[i] = min(min_ending[i-1]+ar[i],ar[i])\n", " return min(min_ending)\n", "\n", " total = sum(ar)\n", " max_sum = maxSubarraySum(ar)\n", " min_sum = minSubarraySum(ar)\n", " return max(max_sum, total-min_sum)\n", "\n", "ar = [5, -2, 3, 4]\n", "\n", "print(maxCircularSubarraySumNaive(ar))\n", "print(maxCircularSubarraySum(ar))"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Moore's Algorithm\n", "- Find in 1st part\n", "- Verify in 2nd Part\n", "\n", "#### Find majority element, count>n/2"] }, { "cell_type": "code", "execution_count": 40, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["3\n"] }], "source": ["def findMajority(ar):\n", " res = 0\n", " cnt = 1\n", " for i in range(1,len(ar)):\n", " if ar[res]==ar[i]:\n", " cnt+=1\n", " else:\n", " cnt -=1\n", " if cnt == 0:\n", " res = i\n", " cnt = 1\n", " return res\n", "\n", "def verifyMajority(ar,major):\n", " cnt =0\n", " for x in ar:\n", " if x == ar[major]:\n", " cnt+=1\n", " if cnt>len(ar)//2:\n", " return major\n", " else:\n", " return -1\n", "\n", "ar = [6,8,4,8,8]\n", "\n", "print(verifyMajority(ar,findMajority(ar)))"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Sliding window Technique\n", "##### Maximum sum of consecutive K numbers"] }, { "cell_type": "code", "execution_count": 41, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["45\n"] }], "source": ["def maxConsecutiveKsum(ar, k):\n", " ksum = 0\n", " for i in range(k):\n", " ksum+=ar[i]\n", " ans = ksum\n", " for i in range(k,len(ar)):\n", " ksum += ar[i]\n", " ksum -= ar[i-k]\n", " ans = max(ans,ksum)\n", " return ans\n", "\n", "ar = [1,8,30,-5,20,7]\n", "k =3\n", "print(maxConsecutiveKsum(ar, k))"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Prefix Sum\n", "\n", "Calculate sum prefix data, to answer multiple query on a fixed data\n", "\n", "#### Sum b/w 2 indexs of array\n"] }, { "cell_type": "code", "execution_count": 52, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["[0, 2, 10, 13, 22, 28, 33, 37]\n", "13\n", "20\n", "27\n"] }], "source": ["def getSum(ar,queries):\n", " prefix_sum = ar[:]\n", " for i in range(1,len(ar)):\n", " prefix_sum[i] += prefix_sum[i-1]\n", " prefix_sum = [0] +prefix_sum\n", " print(prefix_sum)\n", " for l,r in queries:\n", " print(prefix_sum[r+1]-prefix_sum[l])\n", " \n", "ar = [2,8,3,9,6,5,4]\n", "queries = [(0,2),(1,3),(2,6)]\n", "\n", "getSum(ar,queries)"] }, { "cell_type": "markdown", "metadata": {}, "source": ["#### Max occurring element b/w given queries"] }, { "cell_type": "code", "execution_count": 56, "metadata": {}, "outputs": [{ "name": "stdout", "output_type": "stream", "text": ["[0, 1, 2, 3, 2, 2, 1, 1, 0, 0]\n", "3\n"] }], "source": ["def maxOccuring(queries):\n", " ar = [0 for _ in range(10)]\n", " for l,r in queries:\n", " ar[l]+=1\n", " ar[r+1]-=1\n", " prefix_sum = ar[:]\n", " for i in range(1,len(ar)):\n", " prefix_sum[i] += prefix_sum[i-1]\n", " print(prefix_sum)\n", " mx = 0\n", " ans = 0\n", " for i in range(len(prefix_sum)):\n", " if prefix_sum[i]>mx:\n", " mx = prefix_sum[i]\n", " ans = i\n", " return ans\n", "\n", "queries =[(1,3),(2,5),(3,7)]\n", "\n", "print(maxOccuring(queries))"] }], "metadata": { "kernelspec": { "display_name": "Python 3", "language": "python", "name": "python3" }, "language_info": { "codemirror_mode": { "name": "ipython", "version": 3 }, "file_extension": ".py", "mimetype": "text/x-python", "name": "python", "nbconvert_exporter": "python", "pygments_lexer": "ipython3", "version": "3.10.7" } }, "nbformat": 4, "nbformat_minor": 2 }