amazon_oa2_prep

September 10, 2020

At Amazon, it's important that SDEs demonstrate both technical skills and our Leadership Principles. Part 2 of the Online Assessment includes both a Coding Assessment and Workstyles Assessment. The Coding Assessment asks you to solving two coding problems and you will have the choice of coding in Java, Python, Python3, C, C#, or C++. The Coding Assessment takes approximately 70 minutes to complete. Next, you will complete the Workstyles Assessment, which is built around Amazon's Leadership Principles and asks you to choose statements that represent your work style. The Workstyles Assessment takes 10-20 minutes to complete. In this section, we ask you to choose what extent a provided statement represents your work style.

[4]:
```python
# distinct productes after removing "rem" itemsSolution - o(n)

def findLeastNumOfUniqueInts(self, arr, k) -> int:
    # Get the count of our elements.
    cnts = collections.Counter(arr)
    # Create a heap with our counts.
    heap = [(v, k) for k, v in cnts.items()]
    heapq.heapify(heap)
    # Remove k from the cnts of the elements in our heap, always popping the␣
 ↪lowest cnts.
    for _ in range(k):
        cnt, val = heapq.heappop(heap)
        cnt -= 1

        if cnt != 0:
            heapq.heappush(heap, (cnt, val))
    # The len of whats left in the heap is our answer.
    return len(heap)

        cnt = collections.Counter(arr)
        heap = [(count, num) for num, count in cnt.items()]
        heapq.heapify(heap)

        while k > 0:
            cnt, val = heapq.heappop(heap)
            k -= cnt
        if k < 0:
            return len(heap) + 1
        else:
            return len(heap)

o(n)
import collections
class Solution:
    def findLeastNumOfUniqueInts(self, arr: List[int], k: int) -> int:
        buckets = [[] for _ in range(len(arr) + 1)]
        counter = collections.Counter(arr)
        for key, count in counter.items():
            buckets[count].append(key)
        for count in range(len(arr) + 1):
            if k == 0: break
            while buckets[count] and k >= count:
                del counter[buckets[count].pop()]
                k -= count
        return len(counter)
```

[ ]:
```
# Split String Into Unique Primes
def countPrimes(self, n: int) -> int:
    if n < 2:
        return 0
    isPrime = [True]*n
    isPrime[0] = isPrime[1] = False

    i = 2
    while i*i < n:
        if isPrime[i]:
            for j in range(i*i, n, i):
                isPrime[j] = False

        i += 1

    return sum(isPrime)

def primeSetCount(num):
    def sieve(n):
        isPrime = [False, False] + [True]*n
        for p in range(2,n):
            if isPrime[p]:
                for kp in range(2*p,n,p):
                    isPrime[kp] = False
        return isPrime

    isPrime = sieve(1000000)

    MOD = 1000000007
    def rec(num, i, dp):
        if dp[i] != -1:
            return dp[i]
        cnt = 0
        for j in range(1, 7):
            if i - j >= 0 and num[i-j] != '0' and isPrime[int(num[i-j:i])]:
                cnt += rec(num, i - j, dp)
                cnt %= MOD
        dp[i] = cnt
        return dp[i]

    n = len(num)
    dp = [-1] * (n+1)
    dp[0] = 1
    return rec(num, n, dp)
```

[ ]:
```
def countTeams(num, skills, minAssociates, minLevel, maxLevel):
    candidates_cnt = 0
    for skill in skills:
        if minLevel <= skill <= maxLevel:
            candidates_cnt += 1

    combination_dict = {candidates_cnt: 1}
    def combination(m):
        if m in combination_dict:
            return combination_dict[m]
        combination_dict[m] = combination_dict[m+1]*(m+1)//(candidates_cnt-m)
        return combination_dict[m]
    res = 0
    for i in range(candidates_cnt, minAssociates-1, -1):
        res += combination(i)
    return res
```

[ ]:
```
"""
Subtree With Maximum Average
   ,  Tree   ,        ,    List<Node> childs
       (        )
"""
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.children = []
import collections
def subtreeMaxAvg(root: TreeNode) -> TreeNode:
    table = {}
    def helper(root):
        if not root:
            return 0, 0
        children = root.children
        if not children:
            table[root]=root.val
            return root.val, 1

        summ = root.val
        num = 1
        for child in children:
            subsum, cnt = helper(child)
            summ += subsum
            num += cnt

        table[root]=summ/num
        return summ, num

    helper(root)
    max_node = None
    max_val = float('-inf')
    for key in table:
        if table[key] > max_val:
            max_node = key
            max_val = table[key]

    return max_node
```

[ ]:
```
# power grid: old edges, new edges
def compute_min_cost(num_nodes, base_mst, poss_mst):
    uf = {}

    # create union find for the initial edges given
    def find(edge):
        uf.setdefault(edge, edge)
        if uf[edge] != edge:
            uf[edge] = find(uf[edge])
        return uf[edge]

    def union(edge1, edge2):
        uf[find(edge1)] = find(edge2)

    for e1, e2 in base_mst:
        if find(e1) != find(e2):
            union(e1, e2)

    # sort the new edges by cost
    # if an edge is not part of the minimum spanning tree, then include it,␣
 ↪else continue
    cost_ret = 0
    for c1, c2, cost in sorted(poss_mst, key=lambda x : x[2]):
        if find(c1) != find(c2):
            union(c1, c2)
            cost_ret += cost

    if len({find(c) for c in uf}) == 1 and len(uf) == num_nodes:
        return cost_ret
    else:
        return -1
```

[ ]:
```
# only new edges
def minCostConnectNodes(N, connections):
    if N == 0:
        return 0
    uf = {}

    # create union find for the initial edges given
    def find(edge):
        uf.setdefault(edge, edge)
        if uf[edge] != edge:
            uf[edge] = find(uf[edge])
        return uf[edge]

    def union(edge1, edge2):
        uf[find(edge1)] = find(edge2)


    # sort the new edges by cost
    # if an edge is not part of the minimum spanning tree, then include it,␣
↪else continue
    cost_ret = 0
    for c1, c2, cost in sorted(connections, key=lambda x : x[2]):
        if find(c1) != find(c2):
            union(c1, c2)
            cost_ret += cost

    if len({find(c) for c in uf}) == 1 and len(uf) == N:
        return cost_ret
    else:
        return -1
```

[32]:
```
import collections
def findLargestGroup(items):
    uf = {}
    def find(edge):
        uf.setdefault(edge, edge)
        if uf[edge] != edge:
            uf[edge] = find(uf[edge])
        return uf[edge]

    def union(edge1, edge2):
        uf[find(edge1)] = find(edge2)

    for item in items:
        if len(item) > 1:
            i = 1
            while i < len(item):
                if find(item[i]) != find(item[i-1]):
                    union(item[i], item[i-1])
                i += 1

    group = collections.defaultdict(list)
    for c in uf:
        group[find(uf[c])].append(c)

    maxcnt = 0
    res = []
    for g in group:
        if len(group[g]) > maxcnt:
            maxcnt = len(group[g])
            res = group[g]

    return res
```

[35]:

```
```

[42]:

```
import re
def getTopToys(numToys, topToys, toys, numQuotes, quotes):
    def count_in_string(string, toy):
        string_list = re.split('\W+', string)
#          string_list = string.split()
        print(string_list)
        cnt = collections.Counter(string_list)
        return cnt[toy]
    total_cnt = []
    for i, toy in enumerate(toys):
        string_count = 0
        word_cnt = 0
        for j, quote in enumerate(quotes):
            cnt = count_in_string(quote, toy)
            if cnt > 0:
                string_count+= 1
                word_cnt += cnt
        total_cnt.append([word_cnt,string_count, toy])

    total_cnt = sorted(total_cnt)
    res = []
    while topToys:
        _, _, name = total_cnt.pop()
        res.append(name)
        topToys -= 1

    return res
```

[31]:
```
def load_balance(arr):
    n = len(arr)
    if n < 5:
        return False
    l, r = 1, n - 2
    left = arr[0]
    right = arr[-1]
    mid = sum(arr[2:-2])
#     print(left,  mid, right)
    while l +1  < r:
#         print(l,r, left, right, mid)
        if left == mid and mid == right:
            return True
        if left > mid or right > mid:
#             print(left, mid, right)
            return False
        if left < right:

            left += arr[l]
            l += 1
            mid -= arr[l]
        else:
            right += arr[r]
            r -= 1
            mid -= arr[r]

    return False



print(load_balance([1, 1, 1, 1]))
```

```
False
```

[14]:
```
a = [1,2,3,4]
print(a[:-2])
```

```
[1, 2]
```

[ ]:

[ ]:

[ ]:
```
# zombie
def humanDays(matrix):
    """
    :type matrix: List[List[int]]
    :rtype: int
    """

    rows = len(matrix)
    columns = len(matrix[0])
    if not rows or not columns:
        return 0

    q = [[i,j] for i in range(rows) for j in range(columns) if matrix[i][j]==1]
    directions = [[1,0],[-1,0],[0,1],[0,-1]]
    time = 0
    if not q:
        return -1

    while True:
        new = []
        for [i,j] in q:
            for d in directions:
                ni, nj = i + d[0], j + d[1]
                if 0 <= ni < rows and 0 <= nj < columns and matrix[ni][nj] == 0:
                    matrix[ni][nj] = 1
                    new.append([ni,nj])
        q = new
        if not q:
            break
        time += 1

    return time
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:
```
# max profit
def maxProfit(numSuppliers, order, inventory):
    price = collections.Counter(inventory)
    # print(price)
    max_price = max(price)
    profit = 0
    while order > 0:
        cnt = min(order, price[max_price])
        profit += max_price * cnt
        order -= cnt
        price[max_price] -= cnt
        price[max_price - 1] += cnt
        if price[max_price] == 0:
            max_price -= 1

    return profit
```

[ ]:
```
class Solution:
    def breakPalindrome(self, palindrome: str) -> str:
        if len(palindrome) == 1:
            return ""

        n = len(palindrome)

        for i in range(n):
            letter = palindrome[i]
            if letter != 'a':
                if n%2 == 0 or (n%2 == 1 and i != n//2):
                    return palindrome[:i] + 'a' + palindrome[i+1:]

        return palindrome[:-1] + 'b'
```

[ ]:
```
def cluster(numOfRows, grid) -> int:
    cnt = 0
    cols = len(grid[0])
    def dfs(i, j, letter):
        if i + 1 < numOfRows and grid[i+1][j] == letter and visited[i+1][j] ==␣
 ↪False:
            visited[i+1][j] = True
            dfs(i+1, j, letter)
        if i - 1 >= 0 and grid[i-1][j] == letter and visited[i-1][j] == False:
            visited[i-1][j] = True
            dfs(i-1, j, letter)
        if j + 1 < cols and grid[i][j+1] == letter and visited[i][j+1] == False:
            visited[i][j+1] = True
            dfs(i, j+1, letter)
        if j - 1>= 0 and grid[i][j-1] == letter and visited[i][j-1] == False:
            visited[i][j-1] = True
            dfs(i, j-1, letter)

        return None
    visited = [[False for j in range(cols)] for i in range(numOfRows)]

    for i in range(numOfRows):
        for j in range(cols):

            if visited[i][j] == False:
                letter = grid[i][j]
                cnt += 1
                visited[i][j] = True
                dfs(i, j, letter)
    return cnt
```

[ ]:
```
# ordereddict
d = {'banana': 3, 'apple': [1,4], 'pear': 1, 'orange': 2}
dic = OrderedDict(sorted(d.items(), key=lambda t: t[0]))
ls = list(d.items())
# print(ls)

#OrderedDict
def itemToDisplay(numOfltems, items, sortParameter, sortOrder, itemsPerPage,␣
 ↪pageNumber):
    """
    items: name : relevance, price.
    sort order (0: ascending, 1: descending),
    sortParameter (name: 0, relevance: 1, price: 2)
    """
    if sortParameter != 2:
        items_dict = collections.OrderedDict(sorted(items.items(), key = lambda␣
 ↪x:x[sortParameter], reverse = (sortOrder == 0)))
    else:
        items_dict = collections.OrderedDict(sorted(items.items(), key = lambda␣
 ↪x:x[1][1], reverse = (sortOrder == 0)))
    # print(items_dict)
    if len(items)%itemsPerPage == 0:
        totalPages = len(items)//itemsPerPage
    else:
        totalPages = len(items)//itemsPerPage + 1

    if pageNumber >= totalPages:
        return []
    else:
        start = pageNumber*itemsPerPage
        end = min(len(items), (pageNumber+1)*itemsPerPage)
        i = 0
        while i < start:
            items_dict.popitem()
            i += 1
        items_to_display = []
        # detail_to_display = []
        while i < end:
            item_detail = items_dict.popitem()
            i += 1
            items_to_display.append(item_detail[0])
            # detail_to_display.append(item_detail[1])
        # res = dict.fromkeys(items_to_display, detail_to_display)
    return items_to_display

# list
def itemToDisplay2(numOfItems, items, sortParameter, sortOrder, itemsPerPage,␣
 ↪pageNumber):
    """
    items: name : relevance, price.
    sort order (0: ascending, 1: descending),
    sortParameter (name: 0, relevance: 1, price: 2)
    """
    items_dict = list(items.items())
    if sortParameter != 2:
        items_dict = sorted(items_dict, key = lambda x:x[sortParameter],␣
 ↪reverse = (sortOrder == 1))
    else:
        items_dict = sorted(items_dict, key = lambda x:x[1][1], reverse =␣
 ↪(sortOrder == 1))

    if len(items)%itemsPerPage == 0:
        totalPages = len(items)//itemsPerPage
    else:
        totalPages = len(items)//itemsPerPage + 1

    if pageNumber >= totalPages:
        return []
    else:
        start = pageNumber*itemsPerPage
        end = min(len(items), (pageNumber+1)*itemsPerPage)
        res = []
        i = start
        while i < end:
            res.append(items_dict[i][0])
            i += 1
        return res

    # return items_to_display
# print(itemToDisplay2(4, {"item1": [10,15], "item2":[3,4], "item3":[17, 8],␣
↪"item4":[12,5]}, 1,1,2,1))
```

[ ]:
```
#
```

[ ]:
```
#
def nearestCity(numOfCities, cities, xCoordinates, yCoordinates, numOfQueries,␣
↪queries):
    city_map = []
    for i in range(numOfCities):
        city_map.append([cities[i], xCoordinates[i], yCoordinates[i]])
    # sorted by x, name
    sorted_city_x = sorted(city_map, key = lambda x: (x[1], x[2],x[0]))
    # sorted by y, name
    sorted_city_y = sorted(city_map, key = lambda x: (x[2], x[1],x[0]))
    map_x = collections.defaultdict(list)
    map_y = collections.defaultdict(list)

    for city in sorted_city_x:
        name, x, y = city
        map_x[x].append([name, y])
    for city in sorted_city_y:
        name, x, y = city
        map_y[y].append([name, x])

    def binarySearchClosest(ls, target):
        l, r = 0, len(ls) - 1
        while l <= r:
            m = l + (r-l)//2
            if ls[m][1] == target:
                break
            elif ls[m][1] > target:
                r = m - 1
            else:
                l = m + 1
        cand = []
        if m - 1 >= 0:
            cand.append([abs(ls[m-1][1] - target), ls[m-1][0]])
        if m + 1 < len(ls):
            cand.append([abs(ls[m+1][1] - target), ls[m+1][0]])
        # print(cand)
        cand = sorted(cand)

        return cand[0]
    res = [None]* numOfQueries
    for i, query in enumerate(queries):
        idx = cities.index(query)
        x, y = xCoordinates[idx], yCoordinates[idx]
        sameX = map_x[x]
        sameY = map_y[y]
        candidates = []
        if len(sameX) > 1:
            candidates.append(binarySearchClosest(sameX, y))
        if len(sameY) > 1:
            candidates.append(binarySearchClosest(sameY, x) )
        print(candidates)
        if candidates:
            candidates = sorted(candidates)
            res[i] = candidates[0][1]

    return res
```

[ ]:
```
# sliding window
def diskSpaceAnalysis(nums, k):
    d = collections.deque()
    res = -float('Inf')
    for i, num in enumerate(nums):

        while d and nums[d[-1]] > num:
            d.pop()

        d.append(i)

        if d[0] == i-k:
            d.popleft()

        if i >= k - 1:
            res = max(res, nums[d[0]])
        print(i,d, res)
    return res

# print(diskSpaceAnalysis([8,2,2,5,6,1,3,8],3))
```

[ ]:
```
#movie
def flight(arr, k):
    k -= 30
    table = {}
    for i, movie in enumerate(arr):
        table[i] = movie
    table = sorted(table.items(), key = lambda x:x[1])
    def binarySearch(l, r, target):
        while l + 1 < r:
            m = l + (r-l)//2
            if table[m][1] < target:
                l = m
            elif table[m][1] > target:
                r = m
            else:
                return table[m][0]

        if table[r][1] <= target:
            return table[r][0]
        elif table[l][1] <= target:
            return table[l][0]
        else:
            return -1

    candidates = []
    for idx, movie in table:
        loc = binarySearch(idx, len(table) - 1, k - movie)
        if loc == -1:
            continue
        else:
            candidates.append([idx, loc, movie + table[loc][1], max(movie,␣
 ↪table[loc][1])])

    candidates = sorted(candidates, key = lambda x: (x[2], x[3]))
    if candidates:
        return candidates[-1][:2]
    else:
        return []

# print(flight([90, 85, 75, 60, 120, 150, 125], 50))
```

```
[ ]: # fresh promotion
def matchSecretLists(secretFruitList: List[List[str]], customerPurchasedList:␣
 ↪List[str]) -> bool:

    if not secretFruitList:
        return True

    if not customerPurchasedList:
        return False

    i, j = 0, 0
    while i <len(secretFruitList) and j + len(secretFruitList[i]) <=␣
↪len(customerPurchasedList):
        match = True
        for k in range(len(secretFruitList)):
            if secretFruitList[i][k] != "anything" and␣
↪customerPurchasedList[j+k] != secretFruitList[i][k]:
                match = False
                break

        if match:
            j += len(secretFruitList[i])
            i += 1

        else:
            j += 1
    return i == len(secretFruitList)
```

[ ]:
```
#     list of forward
# list of return   a pair of routes id, where the values sum is the floor of␣
↪max distance
#      two sum + map distance   pair + 2 for loops
```

[ ]:
```
# https://leetcode.com/discuss/interview-question/699973/
↪Goldman-Sachs-or-OA-or-Turnstile
def turnstile(numCustomers, arrTime, direction):
    cust_dict = collections.defaultdict(list)
    for i in range(numCustomers):
        time = arrTime[i]
        dire = direction[i]
        cust_dict[time].append([dire, i])  # dictinary structure: time:␣
↪[[direction1, customer 1], [direction2, customer 2]]

    res = [0]*numCustomers
    # print(cust_dict)
    curTime = 0
    cnt = 0
    prev_used = -1 # -1: not used, 1: used as exit, 0: used as entrance
    while cnt < numCustomers:
        # if curTime in cust_dict:
        curCust = cust_dict[curTime]
        print(curTime, curCust, prev_used)
        if len(curCust) > 0:
            if prev_used == -1 or prev_used == 1:
                curCust = sorted(curCust, key = lambda x: (-x[0], x[1]))
            else:
                curCust = sorted(curCust, key = lambda x: (x[0], x[1]))
            res[curCust[0][1]] = curTime
            cust_dict[curTime+1] += curCust[1:]
            prev_used = curCust[0][0]
            cnt += 1
        else:
            prev_used = -1

        curTime += 1
    # print(cnt)
    return res

# print(turnstile(3, [5,5,6], [0,0,1]))
```

[ ]:
```
def getMaxUnit(num, boxes, unitSize, unitsPerBox, truckSize):
    unit_dict = collections.defaultdict(int)
    for i in range(num):
        unit_dict[unitsPerBox[i]] = boxes[i]

    unit_dict = sorted(unit_dict.items(), key = lambda x:x[0], reverse = True)

    res = 0
    for i in range(num):
        if truckSize == 0:
            return res
        else:
            maxUnit = unit_dict[i][0]
            cnt = min(unit_dict[i][1], truckSize)
            res += cnt*maxUnit
            truckSize -= cnt

    return res


# print(getMaxUnit(3, [2,5,3], 3, [3,2,1], 50))
```

[ ]:
```
# negative balance debt
def balance(numRows, numCols, debts):
    balance = collections.defaultdict(int)
    for debt in debts:
        borrower, lender, amount = debt
        balance[borrower] -= amount
        balance[lender] += amount
    print(balance)
    # for team in balance:
    sorted_team = sorted(balance.items(), key = lambda x:(x[1], x[0]), reverse␣
 ↪= True)
    smallest_balance = sorted_team[-1][1]
    if smallest_balance >= 0:
        return "Nobody has a negative balance"
    else:
        res = []
        while sorted_team[-1][1] == smallest_balance:
            res.append(sorted_team[-1][0])
            sorted_team.pop()

    return res
```

[ ]:
```
# baseball
def scorekeep(blocks):
    n = len(blocks)
    curScore = [0]*n
    totalScore = [0]*n
    for i, record in enumerate(blocks):
        if record == 'X':
            curScore[i] = 2*curScore[i-1]

        elif record == '+':
            curScore[i] = curScore[i-1] + curScore[i-2]
        elif record == 'Z':
            curScore = totalScore[i-2]
            totalScore[i-1] = 0

        else:
            curScore = record

        totalScore[i] = totalScore[i-1] + curScore

    return totalScore[-1]
```

[36]:
```
def partitionLabels(S):
    table = collections.defaultdict(list)
    for i, s in enumerate(S):
        table[s].append(i)

    sorted_list = sorted(table.items(), key = lambda x:x[1][0])

    ptr = 0
    i = 0
    res = []
    start = 0

    while ptr < len(S):
        while i < len(sorted_list) and ptr >= table[sorted_list[i][0]][0]:
            ptr = max(ptr, table[sorted_list[i][0]][-1])
            i += 1

        res.append(S[start:ptr+1])

        ptr += 1
        start = ptr

    return res

print(partitionLabels("bbeadcxede"))
```

```
['bb', 'eadcxede']
```

[ ]:
```
def maximumMinimumPath(A):
        m, n = len(A), len(A[0])
        pq, score, A[m-1][n-1] = [(-A[m-1][n-1], m-1, n-1)], A[0][0], -1
        while pq:
                s, i, j = heapq.heappop(pq)
                score = min(-s, score)
                for x, y in ((i-1,j),(i+1,j),(i,j-1),(i,j+1)):
                        if not (x or y):
                                return score
                        if 0 <= x < m and 0 <= y < n and A[x][y] >= 0:
                                heapq.heappush(pq, (-A[x][y], x, y))
                                A[x][y] = -1


def path(matrix): #
    if not matrix:
        return 0
    rows, cols = len(matrix), len(matrix[0])

    heap = []
    heapq.heappush(heap, [-matrix[rows - 1][cols-1], rows - 1, cols - 1])
    matrix[rows - 1][cols-1] = -1
    score = float('inf')
    while heap:
        val, i, j = heapq.heappop(heap)
        if not (i == rows-1 and j == cols-1) and not (i == 0 and j == 0):
            score = min(score, -val)

        for x,y in [[i+1, j], [i-1, j], [i, j+1], [i, j-1]]:
            if  x == 0 and y == 0:
                return score
            if 0<= x < rows and 0 <= y < cols:
                heapq.heappush(heap, [-matrix[x][y], x, y])
                matrix[x][y] = -1
```

```
```

[41]:
```
def maxPathScore(nums): #
    N = len(nums)
    M = len(nums[0])

    # nums[0][0] = 1e9
    # nums[N - 1][M - 1] = 1e9

    dp = [[1e9] * M for i in range(N)]
    dp[0][0] = nums[0][0]
    for j in range(1, M):
        dp[0][j] = min(dp[0][j - 1], nums[0][j])
    for i in range(1, N):
        dp[i][0] = min(dp[i - 1][0], nums[i][0])

    for i in range(1, N):
        for j in range(1, M):
            cur = max(dp[i - 1][j], dp[i][j - 1])
            dp[i][j] = min(cur, nums[i][j])
    print(dp)
    return dp[N-1][M-1]

input = [[0, 1], [0, 2], [1, 3], [2, 3], [2, 5], [5, 6], [3, 4]]
print(maxPathScore(input))
```

```
[[0, 0], [0, 0], [0, 0], [0, 0], [0, 0], [0, 0], [0, 0]]
0
```

[44]:
```
import heapq
def maxPathScore(A):  # four directions, negative integers

        m, n = len(A), len(A[0])
        if m == 1 and n == 1:
            return A[0][0]
        visited = [[False ]* n for i in range(m)]
        pq, score = [(-A[m-1][n-1], m-1, n-1)], A[0][0]
        visited[m-1][n-1] = True
        while pq:
                s, i, j = heapq.heappop(pq)
                score = min(-s, score)
                for x, y in ((i-1,j),(i+1,j),(i,j-1),(i,j+1)):
                        if not (x or y):
                                return score
                        if 0 <= x < m and 0 <= y < n and not visited[x][y]:
                                heapq.heappush(pq, (-A[x][y], x, y))
                                visited[x][y] = True
```

```
A = [[0, 1], [0, 2], [1, 3], [2, 3], [2, 5], [5, 6], [3, 4]]
print(maxPathScore(A))
```

```
0
```

[46]:
```
def longestVowelsOnlySubstring(S):
    temp, aux, vowels = 0, [], set('aeiou')
    # Count the length of each vowel substring
    for c in S + 'z':
        if c in vowels:
            temp += 1
        elif temp:
            aux.append(temp)
            temp = 0

    # If the first letter is not vowel, you must cut the head
    if S[0] not in vowels: aux = [0] + aux
    # If the last letter is not vowel, you must cut the tail
    if S[-1] not in vowels: aux += [0]
    # Max length = max head + max tail + max middle
    print(aux)
    return aux[0] + aux[-1] + max(aux[1:-1]) if len(aux) >= 3 else sum(aux)
s = "earthproblem"
print(longestVowelsOnlySubstring(s))
```

```
[2, 1, 1, 0]
3
```

[54]:
```
import heapq
def connectRopes(ropes):
    # ropes = sorted(ropes)
    heapq.heapify(ropes)
    minsum = 0
    i = 0

    while len(ropes)>1:
#         print(ropes, minsum)
        a, b = heapq.heappop(ropes), heapq.heappop(ropes)
        minsum += a + b
        heapq.heappush(ropes, a + b)
#         i += 1
        # minsum += ropes[i]

    return minsum
ropes = [20, 4, 8, 2]
print(connectRopes(ropes))
```

```
[2, 4, 8, 20] 0
[6, 20, 8] 6
[14, 20] 20
54
```

[ ]:
```

```

[ ]:
```

```

[ ]:
```
# print(maxProfit(3, 10, [3,4,4]))



# print(balance(4, 3, [('a', 'b', 11), ('b', 'c', 9), ('c','a', 10)]))


# print(nearestCity2(4, ["c1", "c2", "c3","c4"], [3,2,1,2],[3,2,3, 3],3,["c1",␣
↪"c2", "c3"]))




# disk space analysis
# def diskSpaceAnalysis(n, nums, k):

#     d = collections.deque()
#     out = []
#     res = -float('inf')
#     for i, n in enumerate(nums):
#         print(i, d)
#         while d and nums[d[-1]] > n:
#             d.pop()
#         d.append(i)
#         if d[0] == i - k:
#             d.popleft()

#         if i>=k-1:
#             res = max(res, nums[d[0]])

#     return res




def uniquePairs(nums, target):
    res = {}
    out = set()

    for index, value in enumerate(nums):
        if target - value in res:
            out.add((value, target-value))
        else:
            res[value]=index

    return len(out)

def slowestKey(keyTimes):
    longest_key = None
    longest_duration = None
    for i in range(len(keyTimes)):
        if i == 0:
            start = 0
        else:
            start = keyTimes[i-1][1]
        end = keyTimes[i][1]
        char = keyTimes[i][0]
        interval = end - start
        print(i, char, interval)
        if not longest_duration or interval > longest_duration:
            longest_duration = interval
            longest_key = char
    return chr(longest_key+ord("a"))

# output1 = slowestKey([[0, 2], [1, 5], [0, 9], [2, 15]])
# print(output1)



class Solution:
    def kClosest(self, points: List[List[int]], K: int) -> List[List[int]]:
        dist = lambda i : points[i][0]**2 + points[i][1]**2

        def sort(i,j,K):
            if i >= j:
                return

            k = random.randint(i,j)
            points[i], points[k] = points[k], points[i]

            mid = partition(i,j)
            if K<mid - i + 1:
                sort(i, mid - 1, K)
            elif K > mid - i + 1:
                sort(mid + 1, j, K - (mid-i+1))

        def partition(i,j):
            oi = i
            pivot = dist(i)
            i += 1

            while True:
                while i < j and dist(i) < pivot:
                    i += 1
                while i <= j and dist(j) >= pivot:
                    j -= 1
                if i >= j:
                    break
                points[i], points[j] = points[j], points[i]

            points[oi], points[j] = points[j], points[oi]

            return j

        sort(0, len(points)-1, K)
        return points[:K]

    import heapq

class Solution:
    def kClosest(self, points: List[List[int]], K: int) -> List[List[int]]:

        heap = []

        for (x, y) in points:
            dist = -(x*x + y*y)
            if len(heap) == K:
                heapq.heappushpop(heap, (dist, x, y))
            else:
                heapq.heappush(heap, (dist, x, y))

        return [(x,y) for (dist,x, y) in heap]



# print(countTeams(6, [12, 4, 6, 13, 5, 10], 3,4,10))



def numIslands(self, grid: List[List[str]]) -> int:
    # DFS
    if not grid:
        return 0
    rows, cols = len(grid), len(grid[0])
    count = 0

    def dfs(i,j):
        if i + 1 < rows and grid[i+1][j] == '1':
            grid[i+1][j] = '0'
            dfs(i+1,j)
        if i - 1>= 0 and grid[i-1][j] == '1':
            grid[i-1][j] = '0'
            dfs(i-1,j)
        if j + 1 < cols and grid[i][j+1] == '1':
            grid[i][j+1] = '0'
            dfs(i, j+1)
        if j - 1 >= 0 and grid[i][j-1] == '1':
            grid[i][j-1] = '0'
            dfs(i, j-1)
        return None

    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == '1':
                count += 1
                grid[i][j] = '0'
                dfs(i,j)


    return count


"""
ID: Integer
VALUE: double
 List<ID, VALUE>,     ID   5 VALUE   ,    Map<ID, Double>

 :   ,   ID   HEAP,    5 poll(): Map<ID, PriorityQueue<Double>>,      Map<ID,␣
 ↪Double> .
"""
def avg_value(studentScores):
    """
    Given a List<ID, VALUE>, return the five largest value of each id.
    """
    score_table = collections.defaultdict(list)
    for record in studentScores:
        name, score = record
        score_table[name].append(score)
    # print(score_table)
    def avg_five_largest(ls):
        """
        return the average of five largest values from a list ls.
        """
        if len(ls) <= 5:
            return sum(ls)/len(ls)
        heap = [num*(-1) for num in ls]

        heapq.heapify(heap) #build a max heap in linear time
        i = 0
        summ = 0
        while i < 5:
            summ -= heapq.heappop(heap)
            i += 1

        return summ/5
    res = {}
    for name in score_table:
        score_list = score_table[name]
        print(score_list)
        avg = avg_five_largest(score_list)
        res[name] = avg

    return res



def isCustomerWinner(codeList, shoppingCart):
        if not codeList: return 1
        if not shoppingCart: return 0
        i, j = 0, 0
        while i < len(codeList) and j+len(codeList[i]) <= len(shoppingCart):
                match = True
                for k in range(len(codeList[i])):
                        if codeList[i][k] != 'anything' and codeList[i][k] !=␣
 ↪shoppingCart[j+k]:
                                match = False
                                break
                if match:
                        j+=len(codeList[i])
                        i+=1
                else:
                        j+=1
        return i == len(codeList)
```

```python
# print(cluster(3, ["aabba", "aabba","aaacb"]))


# print(nearestCity(6,['c1', 'c2', 'c3', 'c4', 'c5', 'c6'],␣
↪[3,2,1,1,2,3],[3,2,3,5,5,2], 3, ['c4', 'c5', 'c6']))


"""
Let C[i] = number of ways to split S[:i] into primes

If for a j<i such that S[j:i] is a prime, then can split at j to form a prime,␣
↪and have at least C[j] ways to split the rest S[:j] into primes

So, the DP formula is

C[i] = sum(C[j] for j < i such that S[j:i] is a prime) + (1 if S[:i] is a prime)

Notes: if the sets of primes is limited to (0, 100), the range of j can be␣
↪shortened to range(i-2,i) to speed up the DP.
"""


def closestPair(numRobots, X, Y):

    def dist(point1, point2):
        return (point1[0]-point2[0])**2 + (point1[1] - point2[1])**2

    def strip(sorted_x, sorted_y, d, mid_x):
        sy = [point for point in sorted_y if mid_x - d <= point[0] <= mid_x + d]
        for i in range(len(sy)):
            for j in range(i+1, len(sy)):
                p, q = sy[i], sy[j]
                cur_dist = dist(p, q)
                if cur_dist > 0 and d > 0:
                    d = min(d, cur_dist)
                else:
                    d = max(d, cur_dist)
        return d

    def brute(ls):
        if len(ls) == 2:
            return dist(ls[0], ls[1])
        d = float('inf')
        for i in range(len(ls)-1):
            for j in range(i+1, len(ls)):
                cur_d = dist(ls[i], ls[j])
                if cur_d > 0:
                    d = min(d, cur_d)
        return d

    p = list(zip(X,Y))

    px = sorted(p, key = lambda x:x[0])
    py = sorted(p, key = lambda x:x[1])

    def divide(sorted_x, sorted_y):
        if len(sorted_x) <= 3:
            return brute(sorted_x)
        mid = len(sorted_x)//2
        leftX = sorted_x[:mid]
        rightX = sorted_x[mid:]
        mid_x = sorted_x[mid][0]

        leftY = []
        rightY = []
        for point in sorted_x:
            if point[0] < mid_x:
                leftY.append(point)
            else:
                rightY.append(point)

        dis1 = divide(leftX, leftY)
        dis2 = divide(rightX, rightY)
        if dis1 > 0 and dis2 > 0:
            dis = min(dis1, dis2)
        else:
            dis = max(dis1, dis2)

        dis3 = strip(sorted_x, sorted_y, dis, mid_x)

        if dis3 >0 and dis > 0:
            dis = min(dis, dis3)
        else:
            dis = max(dis, dis3)

        return dis

    return divide(px, py)


# distinct productes after removing "rem" itemsSolution
class Solution:
    def findLeastNumOfUniqueInts(self, arr: List[int], k: int) -> int:
            # Get the count of our elements.
        cnts = collections.Counter(arr)
                # Create a heap with our counts.
        heap = [(v, k) for k, v in cnts.items()]
        heapq.heapify(heap)
                # Remove k from the cnts of the elements in our heap, always␣
↪popping the lowest cnts.
        for _ in range(k):
            cnt, val = heapq.heappop(heap)
            cnt -= 1
            if cnt != 0:
                heapq.heappush(heap, (cnt, val))
        # The len of whats left in the heap is our answer.
        return len(heap)
```

[13]:
```
import collections
def optimizeMemoryUsage(foregroundTasks, backgroundTasks, K):
    """
    :type foregroundTasks: List[int]
    :type backgroundTasks: List[int]
    :type K: int
    :rtype: List[List[int]]
    """
    fore = {} # idx: value
    for i, t in enumerate(foregroundTasks):
        fore[i] = t
    back = {}
    for i, t in enumerate(backgroundTasks):
        back[i] = t

    fore = sorted(fore.items(), key = lambda x:x[1])
    back = sorted(back.items(), key = lambda x:x[1])

    def binarySearch(ls, target):
        l, r = 0, len(ls) - 1
        while l + 1 < r:
            m = l + (r-l)//2
            if target > ls[m][1]:
                l = m
            else:
                r = m

        if ls[r][1] > target:
            return ls[l][0],ls[l][1]
        else:
            return ls[r][0],ls[r][1]

    def binarySearchExact(ls, target):
        l, r = 0, len(ls) - 1
        while l + 1 < r:
            m = l + (r-l)//2
            if target == ls[m][1]:
                return ls[m][0]
            elif target > ls[m][1]:
                l = m
            else:
                r = m

        if ls[l][1] == target:
            return ls[l][0]
        elif ls[r][1] == target:
            return ls[r][0]
        else:
            return -1

    res = []
    # exact match
    fore_id = binarySearchExact(fore, K)
    back_id = binarySearchExact(back, K)
#     print(fore_id, back_id)
    if fore_id != -1:
        res.append([fore_id, -1])
    if back_id != -1:
        res.append([-1, back_id])

#     print(fore)
#     print(back)
#     print(res)
    if len(res) > 0:
        # check other exact match
        for f_id, f_val in fore:
            target = K - f_val
            if target > 0:
                b_id = binarySearchExact(back, target)
                if b_id != -1:
                    res.append([f_id, b_id])
            elif target < 0:
                return res
    else:
        cand = collections.defaultdict(list)
        for f_id, f_val in fore:
            target = K - f_val

            if target > 0:
                b_id, b_val = binarySearch(back, target)
#                 print(f_val, target)
                cand[b_val + f_val].append([f_id, b_id])
            elif target < 0:
                break

        cand = sorted(cand.items(), key = lambda x:x[0], reverse = True)
#         print(cand)
        return cand[0][1]

print(optimizeMemoryUsage( [1, 7, 2, 4, 5, 6],  [1, 1, 2], 10))
```

```
[[1, 2]]
```

[ ]:
```
# Treasure Island
def solution(m):# bfs
    if len(m) == 0 or len(m[0]) == 0:
        return -1  # impossible

    matrix = [row[:] for row in m]
    nrow, ncol = len(matrix), len(matrix[0])

    q = deque([((0, 0), 0)])  # ((x, y), step)
    matrix[0][0] = "D"
    while q:
        (x, y), step = q.popleft()

        for dx, dy in [[0, 1], [0, -1], [1, 0], [-1, 0]]:
            if 0 <= x+dx < nrow and 0 <= y+dy < ncol:
                if matrix[x+dx][y+dy] == "X":
                    return step+1
                elif matrix[x+dx][y+dy] == "O":
                    # mark visited
                    matrix[x + dx][y + dy] = "D"
                    q.append(((x+dx, y+dy), step+1))

    return -1

# dfs
def find_treasure(t_map, row, col, curr_steps, min_steps):
    if row >= len(t_map) or row < 0 or col >= len(t_map[0]) or col < 0 or␣
↪t_map[row][col] == 'D' or t_map[row][col] == '#':
        return None, min_steps

    if t_map[row][col] == 'X':
        curr_steps += 1
        if min_steps > curr_steps:
            min_steps = min(curr_steps, min_steps)

        return None, min_steps

    else:
        tmp = t_map[row][col]
        t_map[row][col] = '#'
        curr_steps += 1
        left = find_treasure(t_map, row, col-1, curr_steps, min_steps)
        right = find_treasure(t_map, row, col+1, curr_steps, min_steps)
        up = find_treasure(t_map, row-1, col, curr_steps, min_steps)
        down = find_treasure(t_map, row+1, col, curr_steps, min_steps)

        t_map[row][col] = tmp

        return curr_steps, min(left[1], right[1], up[1], down[1])
```

[ ]:
```
# Treasure Island II

from collections import deque
class solution:
    def shortestPath(self, grid):
        if not grid or not grid[0]:
            return 0

        res = float('inf')
        row, col = len(grid), len(grid[0])
        self.directions = [[0,1],[0,-1],[1,0],[-1,0]]

        for i in range(row):
            for j in range(col):
                if grid[i][j] == 'S':
                    q = deque([[i,j,0]])
                    res = min(self.bfs(q, grid, row, col), res)
        return res


    def bfs(self, q, grid, row, col):
        visited = [[-1 for _ in range(col)]for _ in range(row)]
        while len(q):
            i, j, step = q.popleft()
            visited[i][j] = step
            if grid[i][j] == 'X':
                return step

            for d in self.directions:
                next_i, next_j = i + d[0], j + d[1]
                if next_i >= 0 and next_i < row and next_j >= 0 and next_j <␣
↪col:
                    if grid[next_i][next_j] != 'D' and visited[next_i][next_j]␣
↪== -1:
                        q.append([next_i, next_j, step+1])
        return -1
```

[ ]:
```
# https://aonecode.com/amazon-online-assessment-questions
# https://wdxtub.com/interview/14520850399861.html
```

[59]:
```
# roll dice
import collections

def rollDice(A):
    ans = float('inf')
    ctr = collections.Counter(A)
    for i in range(1, 7):
        tmp = 0
        for j in ctr.keys():
            w = 0
            if i + j == 7:
                w = 2
            elif i == j:
                w = 0
            else:
                w = 1
            tmp += w * ctr[j]
        print(tmp
             )
        ans = min(ans, tmp)
    return ans

print(rollDice([1, 6, 2, 3]))
```

```
4
3
3
5
5
4
```

```
3
```

[75]:
```python
def favGenres(userSongs, songGenres):
    song_genra = {}
    for genra in songGenres:
        songs = songGenres[genra]
        for song in songs:
            song_genra[song] = genra
#     print(song_genra)
    res = collections.defaultdict(list)
    user_genra = collections.defaultdict(list)
    for user in userSongs:

        songs = userSongs[user]
#         print(songs)
        for song in songs:
#             print(song)
            if song in song_genra:
#                 user_genra[user].append([])
                user_genra[user].append(song_genra[song])
        if user_genra:
            cnt = collections.Counter(user_genra[user]) # genra:cnt
            maxx = max(cnt.values())

            for genra in cnt:
                if cnt[genra] == maxx:
                    res[user].append(genra)

        else:
            res[user] = []

    return res
userSongs = {
   "David": ["song1", "song2"],
   "Emma":  ["song3", "song4"]}
songGenres = {   "Rock":    ["song1", "song3"],
   "Dubstep": ["song7"]}
print(favGenres(userSongs, songGenres))
```

```
defaultdict(<class 'list'>, {'David': ['Rock'], 'Emma': ['Rock']})
```

[ ]:
```python
# critical node
# https://leetcode.com/problems/critical-connections-in-a-network/ 1192
def criticalConnections(self, n: int, connections: List[List[int]]) ->␣
 ↪List[List[int]]:


        # node is index, neighbors are in the list
        graph = [[] for i in range(n)]

        # build graph
        for n1, n2 in connections:
            graph[n1].append(n2)
            graph[n2].append(n1)

        # min_discovery_time of nodes at respective indices from start node
        # 1. default to max which is the depth of continuous graph
        lows = [n] * n

        # critical edges
        critical = []

        # args: node, node discovery_time in dfs, parent of this node
        def dfs(node, discovery_time, parent):

            # if the low is not yet discovered for this node
            if lows[node] == n:

                # 2. default it to the depth or discovery time of this node
                lows[node] = discovery_time

                # iterate over neighbors
                for neighbor in graph[node]:

                    # all neighbors except parent
                    if neighbor != parent:

                        expected_discovery_time_of_child = discovery_time + 1
                        actual_discovery_time_of_child = dfs(neighbor,␣
↪expected_discovery_time_of_child, node)

                        # nothing wrong - parent got what was expected => no␣
↪back path
                        # this step is skipped if there is a back path
                        if actual_discovery_time_of_child >=␣
↪expected_discovery_time_of_child:
                            critical.append((node, neighbor))

                        # low will be equal to discovery time of this node or␣
↪discovery time of child
                        # whichever one is minm
                        # if its discovery time of child - then there is a␣
↪backpath
                        lows[node] = min(lows[node],␣
↪actual_discovery_time_of_child)
```

```
            # return low of this node discovered previously or during this call
            return lows[node]

        dfs(n-1, 0, -1)

        return critical
```

[ ]:
```
# shoekeeper sale discount
# returns the final prices of all items after discounts
# https://leetcode.com/discuss/interview-question/algorithms/124783/
↪coursera-online-assessment-min-discount
def finalPrice(prices):
    if len(prices) == 0:
        return 0

    nle_stack = [] # nle = next lesser element
    res = 0

    # we go from right to left
    for i in reversed(range(len(prices))):
        # pop the stack until we find the next lesser element
        while nle_stack and nle_stack[-1] > prices[i]:
            nle_stack.pop()

        # add the discount, if any, and then add the current price to the stack
        res += prices[i] if not nle_stack else prices[i] - nle_stack[-1]
        nle_stack.append(prices[i])
    return res
```

[ ]:
```
# Longest String Without 3 Consecutive Characters
def longestDiverseString(self, a: int, b: int, c: int) -> str:
    t = sorted([[a, 'a'], [b, 'b'], [c, 'c']], reverse=True)
    s = ''

    k = max(0, t[0][0] - 2 * t[1][0])
    print(k)
    for i in range(t[1][0]):
        s += (1 + (i < t[0][0] - t[1][0])) * t[0][1] + t[1][1]
        print(s)
        if i < t[2][0]:
            s += min(2, k) * t[0][1] + t[2][1]
            k -= min(2, k)
    return s + min(2, k) * t[0][1]
```

```
[ ]: # https://leetcode.com/problems/maximum-number-of-non-overlapping-substrings/
      ↪discuss/743223/C%2B%2BJava-Greedy-O(n)
     def maxNumOfSubstrings(self, s):
             """
             :type s: str
             :rtype: List[str]
             """

             def checkSubstr(s, i, l, r):
                 right = r[ord(s[i])-ord('a')]
                 j=i
                 while j<=right:
                     if l[ord(s[j])-ord('a')]<i:
                         return -1
                     right = max(right, r[ord(s[j])-ord('a')])
                     j+=1
                 return right

             l = [float('inf')]*26
             r = [float('-inf')]*26
             res = []
             for i in range(len(s)):
                 l[ord(s[i])-ord('a')] = min(l[ord(s[i])-ord('a')], i)
                 r[ord(s[i])-ord('a')] = max(r[ord(s[i])-ord('a')], i)

             right = float('inf')
             previous_sub = ''
             for i in range(len(s)):
                 if i==l[ord(s[i])-ord('a')]:
                     new_right = checkSubstr(s, i, l, r)
                     if new_right!=-1:
                         if i>right:
                             res.append(previous_sub)
                         right = new_right
                         previous_sub=(s[i:i+(right-i+1)])
             if previous_sub:
                 res.append(previous_sub)
             return res```
```

```
[ ]: # critical connections in a network
     """
     https://leetcode.com/problems/critical-connections-in-a-network/discuss/410345
     /
      ↪Python-(98-Time-100-Memory)-clean-solution-with-explanaions-for-confused-people-like-me
     """
     class Solution:
```

```
    def criticalConnections(self, n: int, connections: List[List[int]]) ->␣
↪List[List[int]]:

                graph = [[] for _ in range(n)] ## vertex i ==> [its neighbors]

        currentRank = 0 ## please note this rank is NOT the num (name) of the␣
↪vertex itself, it is the order of your DFS level

        lowestRank = [i for i in range(n)] ## here lowestRank[i] represents the␣
↪lowest order of vertex that can reach this vertex i

        visited = [False for _ in range(n)] ## common DFS/BFS method to mark␣
↪whether this node is seen before

        ## build graph:
        for connection in connections:
            ## this step is straightforward, build graph as you would normally␣
↪do
            graph[connection[0]].append(connection[1])
            graph[connection[1]].append(connection[0])

        res = []
        prevVertex = -1 ## This -1 a dummy. Does not really matter in the␣
↪beginning.
                ## It will be used in the following DFS because we need to know␣
↪where the current DFS level comes from.
                ## You do not need to setup this parameter, I setup here ONLY␣
↪because it is more clear to see what are passed on in the DFS method.

        currentVertex = 0 ## we start the DFS from vertex num 0 (its rank is␣
↪also 0 of course)
        self._dfs(res, graph, lowestRank, visited, currentRank, prevVertex,␣
↪currentVertex)
        return res

    def _dfs(self, res, graph, lowestRank, visited, currentRank, prevVertex,␣
↪currentVertex):

        visited[currentVertex] = True
        lowestRank[currentVertex] = currentRank

        for nextVertex in graph[currentVertex]:
            if nextVertex == prevVertex:
                continue ## do not include the the incoming path to this vertex␣
↪since this is the possible ONLY bridge (critical connection) that every␣
↪vertex needs.
```

```
            if not visited[nextVertex]:
                self._dfs(res, graph, lowestRank, visited, currentRank + 1,␣
↪currentVertex, nextVertex)
                                # We avoid visiting visited nodes here instead␣
↪of doing it at the beginning of DFS -
                                # the reason is, even that nextVertex may be␣
↪visited before, we still need to update my lowestRank using the visited␣
↪vertex's information.

            lowestRank[currentVertex] = min(lowestRank[currentVertex],␣
↪lowestRank[nextVertex])
                        # take the min of the current vertex's and next␣
↪vertex's ranking
            if lowestRank[nextVertex] >= currentRank + 1: ####### if all the␣
↪neighbors lowest rank is higher than mine + 1, then it means I am one␣
↪connecting critical connection ###
                res.append([currentVertex, nextVertex])
```