转盘print数时间

In [8]:

```python
def get_print_time(s):
    ans = 0
    for pre, nxt in zip('A'+s, s):
        delta = abs(ord(pre)-ord(nxt))
        ans += min(delta, 26-delta)
    return ans
```

stars and bars

In [9]:

```python
def stars_bars(s, startindex, endindex):
    res = []
    for i in range(len(startindex)):
        start = startindex[i]-1
        end = endindex[i]-1
        first_bar, end_bar, count_bar = -1, 0, 0    # first_bar 一开始赋值-1，若
        curr = start

        first_bar, end_bar, count_bar = solution3_helper(s, curr, end, first_

        if end_bar:
            res.append(end_bar-first_bar-count_bar)
        else:
            res.append(0)
    return res

def solution2_helper(s, curr, end, first_bar, end_bar, count_bar):
    while curr <= end:
        if first_bar==-1 and s[curr] == '|':
            first_bar = curr
        elif first_bar!=-1 and s[curr] == '|':
            end_bar = curr
            count_bar += 1
        curr += 1
    return first_bar, end_bar, count_bar
```

Inversion

In [10]:

```python
def maxInversions(arr):
    largerThanRightCount = [0] * len(arr)
    seen = set()
    for i in range(len(arr)):
        if arr[i] in seen:
            largerThanRightCount[i] = 0
        else:
            seen.add(arr[i])
            get_count(i, arr, largerThanRightCount)

    result = 0
    seen = set()
    for i in range(len(arr)):
        if arr[i] in seen:
            continue
        else:
            seen.add(arr[i])
            result = get_res(i, arr, largerThanRightCount, result)
    return result
```

```
def get_count(i, arr, largerThanRightCount):
    for j in range(i+1, len(arr)):
        if arr[i] > arr[j]:
            largerThanRightCount[i] += 1


def get_res(i, arr, largerThanRightCount, res):
    for j in range(i + 1, len(arr)):
        if arr[i] > arr[j]:
            res += largerThanRightCount[j]
    return res
```

字典顺序最小最大substring in k size

In [12]:

```
def min_max_substing(s, k):
    curr = s[:k]
    mincurr = curr
    maxcurr = curr
    for i in range(k,len(s)):
        curr = curr[1:k]+s[i]
        if (curr < mincurr):
            mincurr = curr
        elif (curr > maxcurr):
            maxcurr = curr
    print(mincurr, maxcurr)
```

In [27]:

```
def min_k_substring(s, k):
    left, right, count = 0, 0, 0
    res = s
    while left <= len(s)-k:
        if s[left] == '0':
            left += 1
            while left > right:
                right += 1
            continue

        # s[left] == '1':
        while count < k and right < len(s):
            if s[right] == '1':    # (1) 为了后续一致性，第一次寻找时，指导left和rig
                count += 1
            right += 1             # (2) 每次判断后right加1

        # count == k or right == len(s)
        if count == k:
            curr_res = s[left:right]
            print(curr_res)
            if len(curr_res)<len(res):
                res = curr_res
            elif len(curr_res) == len(res) and curr_res < res: # 分两次判断，只
                res = curr_res
            left += 1      # left位置已经判断过是1，需要右移
            count -= 1     # left位置原本是1，右移后count-1
                           # 由于(2)，right-1位置是1，right位置还没有判断过，所以ri
        else:
            break
    return res
```

Super stack

In [31]:

```
class CustomStack:
    def __init__(self, maxSize: int):
        self.stack = []
        self.inc = []
        self.maxsize = maxSize



    def push(self, x: int) -> None:
        if len(self.stack) < self.maxsize:
            self.stack.append(x)
            self.inc.append(0)

    def pop(self) -> int:
        if not self.stack:
            return -1
        if len(self.inc)>1:
            self.inc[-2] += self.inc[-1]
        return self.stack.pop(-1)+self.inc.pop(-1)


    def increment(self, k: int, val: int) -> None:
        if self.inc:                          # 如果self.inc是空的，self.inc[-1]会报错
            self.inc[min(k-1, len(self.inc)-1)] += val  # 注意表示前k个数的index
```

ancestral names

In [32]:

```
def roman_to_int(s:str):
    rom_to_int_map = {"I": 1, "V": 5, "X": 10, "L": 50, "C": 100, "D": 500, "
    sub_map = {'IV': 4, 'IX':9, 'XL': 40, 'XC': 90, 'CD':400, 'CM': 900}
    summation = 0
    idx = 0

    while idx < len(s):
        if s[idx:idx+2] in sub_map:
            summation += sub_map.get(s[idx:idx+2])
            idx += 2
        else:
            summation += rom_to_int_map.get(s[idx])
            idx += 1
    return summation

def sort_roman(names):
    name_array = []
    for name in names:
        n, num = name.split()
        num = roman_to_int(num)
        name_array.append((n, num, name))
    name_array.sort(key=lambda x: [x[0], x[1]])
    return list(map(lambda x:x[2], name_array))
```

String Chain

In [34]:

```
class Solution:
    def longestStrChain(self, words):
        dp = {}
        for w in sorted(words, key=len):
            dp[w] = max(dp.get(w[:i] + w[i + 1:], 0) + 1 for i in range(len(w
        return max(dp.values())
```

permeation in string

In [35]:
```
from collections import Counter
class Solution:
    def checkInclusion(self, s1: str, s2: str) -> bool:
        l1 = [0]*26
        for s in s1:
            l1[ord(s)-ord('a')] += 1

        l2 = [0]*26
        left, right = 0, len(s1)-1
        for s in s2[left:right+1]:
            l2[ord(s)-ord('a')] += 1

        while right < len(s2):
            if l1 == l2:
                return True
            l2[ord(s2[left])-ord('a')] -= 1
            left += 1
            right += 1
            if right <= len(s2)-1:
                l2[ord(s2[right])-ord('a')] += 1
        return False
```

Count binary strings

In [36]:
```
class Solution:
    def countBinarySubstrings(self, s: str) -> int:
        group = [1]
        for i in range(1, len(s)):
            if s[i] != s[i-1]:
                group.append(1)
            else:
                group[-1] += 1
        ans = 0
        for i in range(1, len(group)):
            ans += min(group[i-1], group[i])

        return ans
```

The Number of Weak Characters in the Game

In [38]:
```
class Solution:
    def numberOfWeakCharacters(self, properties) -> int:
        properties.sort(key = lambda x: (-x[0], x[1]))
        ans = 0
        max_d = 0

        for _, defense in properties:
            if defense < max_d:
                ans += 1
            else:
                max_d = defense

        return ans
```

LRU cache

In [39]:
```
from collections import OrderedDict
```

```
class LRUCache:

    def __init__(self, capacity: int):
        self.dict = OrderedDict()
        self.capacity = capacity

    def get(self, key: int) -> int:
        if key in self.dict:
            self.dict.move_to_end(key)
            return self.dict[key]
        return -1

    def put(self, key: int, value: int) -> None:
        if key in self.dict:
            self.dict.move_to_end(key)
        self.dict[key] = value
        if len(self.dict) > self.capacity:
            self.dict.popitem(last=False)
```

In [42]:

```
class DLinkedNode():
    def __init__(self):
        self.key = 0
        self.value = 0
        self.prev = None
        self.next = None

class LRUCache():
    def _add_node(self, node):
        node.prev = self.head
        node.next = self.head.next

        self.head.next.prev = node
        self.head.next = node

    def _remove_node(self, node):
        prev = node.prev
        new = node.next

        prev.next = new
        new.prev = prev

    def _move_to_head(self, node):
        self._remove_node(node)
        self._add_node(node)

    def _pop_tail(self):
        res = self.tail.prev
        self._remove_node(res)
        return res

    def __init__(self, capacity):
        self.cache = {}
        self.size = 0
        self.capacity = capacity
        self.head, self.tail = DLinkedNode(), DLinkedNode()

        self.head.next = self.tail
        self.tail.prev = self.head


    def get(self, key):
        node = self.cache.get(key, None)
        if not node:
            return -1
        self._move_to_head(node)

        return node.value

    def put(self, key, value):
        node = self.cache.get(key)

        if not node:
            newNode = DLinkedNode()
            newNode.key = key
            newNode.value = value

            self.cache[key] = newNode
            self._add_node(newNode)

            self.size += 1

            if self.size > self.capacity:
                tail = self._pop_tail()
                del self.cache[tail.key]
                self.size -= 1
        else:
            node.value = value
            self._move_to_head(node)
```

reaching points

In [43]:

```
class Solution:
    def reachingPoints(self, sx: int, sy: int, tx: int, ty: int) -> bool:
        while tx >= sx and ty >= sy:
            if tx == ty:
                break
            elif tx > ty:
                if ty > sy:
                    tx %= ty
                else:
                    return (tx - sx) % ty == 0
            else:
                if tx > sx:
                    ty %= tx
                else:
                    return (ty - sy) % tx == 0

        return tx == sx and ty == sy
```