CCF 编程能力等级认证
Grade Examination of Software Programming

Python 五级

2025 年 12 月

1 单选题（每题 2 分，共 30 分）

题号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
答案	C	B	C	D	D	B	A	B	C	C	A	A	A	A	B

第 1 题 对如下定义的循环单链表，横线处填写（ ）。

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def create_list(value):
    head = Node(value)
    head.next = head
    return head

def insert_tail(head, value):
    p = head
    while p.next != head:
        p = p.next
    node = Node(value)
    node.next = head
    p.next = node

def print_list(head):
    ______________________________
    while True:
        print(p.data, end=" ")
        p = p.next
        if p == head:
            break
    print()
```

☐ **A.**

```
if head is None:
       return
   p.next = head
```

☐ **B.**

```
if head is None:
       return
   p = head.next
```

C.

```
if head is None:
        return
    p = head
```

D.

```
if head.next is None:
        return
    p = head
```

第 2 题 区块链技术是比特币的基础。在区块链中，每个区块指向前一个区块，构成链式列表，新区块只能接在链尾，不允许在中间插入或删除。下面代码实现插入区块添加函数，则横线处填写（ ）。

```
class Block:
    def __init__(self, idx, data, prev_block):
        self.index = idx
        self.data = data
        self.prev = prev_block
class Blockchain:
    def __init__(self):
        self.tail = None

    def init(self):
        genesis_block = Block(0, "Genesis Block", None)
        self.tail = genesis_block

    def add_block(self, data):
        ____________________________________

    def clear(self):
        cur = self.tail
        while cur is not None:
            prev_block = cur.prev
            cur.prev = None
            cur = prev_block
        self.tail = None

    def print_chain(self):
        cur = self.tail
        chain = []
        while cur is not None:
            chain.append(f"Block {cur.index}: {cur.data}")
            cur = cur.prev
        for block_info in reversed(chain):
            print(block_info)

```

A.

```

new_block = Block(self.tail.index, data, self.data)

```

B.

```

new_block = Block(self.tail.index + 1, data, self.tail)

        self.tail = new_block
```

☐ **C.**

```
new_block = Block(self.tail.index, data+1, self.data)

        self.tail = new_block
```

☐ **D.**

```
new_block = Block(self.tail.index, data, self.tail)

        self.tail.data = new_block
```

第 3 题 下面关于单链表和双链表的描述中，正确的是（ ）。

```
class DNode:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

def delete_dnode(node):
    if node.prev:
        node.prev.next = node.next
    if node.next:
        node.next.prev = node.prev
    node.prev = None
    node.next = None

class SNode:
    def __init__(self, data):
        self.data = data
        self.next = None

def delete_snode(head, node):
    if head is None or node is None:
        return
    prev = head
    while prev.next != node:
        prev = prev.next
    prev.next = node.next
    node.next = None

```

☐ **A.** 双链表删除指定节点是$O(1)$，单链表是$O(1)$

☐ **B.** 双链表删除指定节点是$O(n)$，单链表是$O(1)$

☐ **C.** 双链表删除指定节点是$O(1)$，单链表是$O(n)$

☐ **D.** 双链表删除指定节点是$O(n)$，单链表是$O(n)$

第 4 题 假设我们有两个数 $a = 38$ 和 $b = 14$，它们对模 m 同余，即$a \equiv b \pmod{m}$。以下哪个值不可能是 m?

☐ **A.** 3

☐ **B.** 4

☐ **C.** 6

☐ **D.** 9

第 5 题 下面代码实现了欧几里得算法，下面有关说法，错误的是（ ）。

```python
def gcd1(a: int, b: int) -> int:
    return a if b == 0 else gcd1(b, a % b)

def gcd2(a: int, b: int) -> int:
    while b != 0:
        temp = b
        b = a % b
        a = temp
    return a
```

- [ ] **A.** `gcd1()` 实现为递归方式。
- [ ] **B.** `gcd2()` 实现为迭代方式。
- [ ] **C.** 当 a 较大时，`gcd1()` 实现会多次调用自身，需要较多额外的辅助空间。
- [ ] **D.** 当 a 较大时，`gcd1()` 的实现比 `gcd2()` 执行效率更高。

第 6 题 唯一分解定理描述的内容是（ ）。

- [ ] **A.** 任何正整数都可以表示为两个素数的和。
- [ ] **B.** 任何大于1的合数都可以唯一分解为有限个质数的乘积。
- [ ] **C.** 两个正整数的最大公约数总是等于它们的最小公倍数除以它们的乘积。
- [ ] **D.** 所有素数都是奇数。

第 7 题 下述代码实现素数表的线性筛法，筛选出所有小于等于 n 的素数，则横线上应填的代码是( )。

```python
def linear_sieve(n):
    if n < 2:
        return []
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False
    primes = []

    for i in range(2, n + 1):
        if is_prime[i]:
            primes.append(i)
        for j in range(len(primes)):
            p = primes[j]
            if i * p > n:
                break
            ____________________
            if i % p == 0:
                break

    return primes
```

- [ ] **A.** `is_prime[i * p] = False`
- [ ] **B.** `is_prime[i] = False`
- [ ] **C.** `is_prime[i * p] = True`
- [ ] **D.** `is_prime[i + p] = False`

第 8 题 下列关于排序的说法，正确的是( )。

- [ ] **A.** 快速排序是稳定排序
- [ ] **B.** 归并排序通常是稳定的
- [ ] **C.** 插入排序是不稳定排序

☐ **D.** 冒泡排序不是原地排序

第 9 题 下面代码实现了归并排序。下述关于归并排序的说法中，不正确的是（ ）。

```python
def merge(arr, temp, l, mid, r):
    i = l
    j = mid + 1
    k = l

    while i <= mid and j <= r:
        if arr[i] <= arr[j]:
            temp[k] = arr[i]
            i += 1
        else:
            temp[k] = arr[j]
            j += 1
        k += 1

    while i <= mid:
        temp[k] = arr[i]
        i += 1
        k += 1

    while j <= r:
        temp[k] = arr[j]
        j += 1
        k += 1

    for p in range(l, r + 1):
        arr[p] = temp[p]

def merge_sort(arr, temp, l, r):
    if l >= r:
        return
    mid = l + (r - l) // 2

    merge_sort(arr, temp, l, mid)
    merge_sort(arr, temp, mid + 1, r)
    merge(arr, temp, l, mid, r)
def merge_sort_wrapper(arr):
    if not arr:
        return []
    temp = [0] * len(arr)
    merge_sort(arr, temp, 0, len(arr) - 1)
    return arr
```

☐ **A.** 归并排序的的平均复杂度是 $O(n \log n)$。

☐ **B.** 归并排序需要 $O(n)$ 的额外空间。

☐ **C.** 归并排序在最坏情况的时间复杂度是 $O(n^2)$。

☐ **D.** 归并排序适合大规模数据。

第 10 题 下述python代码实现了快速排序算法，最差情况时间复杂度是（ ）。

```python
def partition(arr, low, high):
    i = low
    j = high
    pivot = arr[low]

    while i < j:
        while i < j and arr[j] >= pivot:
            j -= 1
        while i < j and arr[i] <= pivot:
            i += 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]

    arr[i], arr[low] = arr[low], arr[i]
    return i

def quick_sort(arr, low, high):
    if low >= high:
        return

    p = partition(arr, low, high)
    quick_sort(arr, low, p - 1)
    quick_sort(arr, p + 1, high)

def quick_sort_wrapper(arr):
    if not arr:
        return []
    quick_sort(arr, 0, len(arr) - 1)
    return arr
```

- [ ] **A.** $O(n)$
- [ ] **B.** $O(\log n)$
- [ ] **C.** $O(n^2)$
- [ ] **D.** $O(n \log n)$

第 11 题 下面代码尝试在有序数组中查找第一个大于等于 `x` 的元素位置。如果没有大于等于 `x` 的元素，返回 `arr.size()`。以下说法正确的是（ ）。

```python
def lower_bound(arr, x):
    l = 0
    r = len(arr)
    while l < r:
        mid = l + (r - l) // 2
        if arr[mid] >= x:
            r = mid
        else:
            l = mid + 1
    return l

```

- [ ] **A.** 上述代码逻辑正确
- [ ] **B.** 上述代码逻辑错误，`while` 循环条件应该用 `l <= r`
- [ ] **C.** 上述代码逻辑错误，`mid` 计算错误
- [ ] **D.** 上述代码逻辑错误，边界条件不对

第 12 题 小杨要把一根长度为 `L` 的木头切成 `K` 段，使得每段长度小于等于 `x`。已知每切一刀只能把一段木头分成两段，他用二分法找到满足条件的最小 `x`（`x` 为正整数），则横线处应填写（ ）。

```python
def check(L, K, x):
    cuts = (L - 1) // x
    return cuts <= K

def binary_cut(L, K):
    l = 1
    r = L
    while l < r:
        ____________________
    return l
if __name__ == "__main__":
    L = 10
    K = 2
    result = binary_cut(L, K)
    print(result)
```

☐ **A.**

```python
mid = l + (r - l) // 2
if check(L, K, mid):
   r = mid
else:
   l = mid + 1
```

☐ **B.**

```python
mid = l + (r) // 2
if check(L, K, mid+1):
   r = mid
else:
   l = mid + 1
```

☐ **C.**

```python
mid = l + (r - l) // 2
if check(L+1, K, mid):
   r = mid
else:
   l = mid + 1

```

☐ **D.**

```python
mid = l + (r - l) // 2
if check(L+1, K, mid):
   r = mid
else:
   l = mid
```

第 13 题 根据下面代码，以下说法正确的是（ ）。

```python

def factorial1(n):
    if n <= 1:
        return 1
    return n * factorial1(n - 1)

def factorial2(n):
    result = 1
    while n > 1:
        result *= n
        n -= 1
    return result
```

- [ ] **A.** 上面两种实现方式的时间复杂度相同，都为$O(n)$
- [ ] **B.** 上面两种实现方式空间复杂度相同，都为$O(1)$
- [ ] **C.** 函数 `factorial1()` 的时间复杂度为 $O(2^n)$，函数 `factorial2()` 的时间复杂度为 $O(n)$
- [ ] **D.** 上面两种实现方式空间复杂度相同，都为$O(n)$

第 14 题 给定有 n 个任务，每个任务有截止时间和利润，每个任务耗时 1 个时间单位、必须在截止时间前完成，且每个时间槽最多做 1 个任务。为了在规定时间内获得最大利润，可以采用贪心策略，即按利润从高到低排序，尽量安排，则横线处应填写（ ）。

```
class Task:
    def __init__(self, deadline, profit):
        self.deadline = deadline
        self.profit = profit

def sort_by_profit(tasks):
    tasks.sort(key=lambda x: x.profit, reverse=True)

def max_profit(tasks):
    sort_by_profit(tasks)
    max_time = 0
    for task in tasks:
        if task.deadline > max_time:
            max_time = task.deadline

    slot = [False] * (max_time + 1)
    total_profit = 0

    for task in tasks:
        t = task.deadline
        while t >= 1:
            if not slot[t]:
                ____________________________
            t -= 1

    return total_profit
```

- [ ] **A.**

```
slot[t] = True
total_profit += task.profit
break
```

- [ ] **B.**

```
slot[t] = True
total_profit += task.profit
```

- [ ] **C.**

```
slot[t] = False
total_profit += task.profit
break
```

- [ ] **D.**

```
slot[t] = True
total_profit -= task.profit
```

第 15 题 下面代码实现了对两个数组表示的正整数的高精度加法（数组低位在前），则横线上应填写（ ）。

```python
def add(a, b):
    c = []
    carry = 0

    i = 0
    while i < len(a) or i < len(b):
        if i < len(a):
            carry += a[i]
        if i < len(b):
            carry += b[i]

        --------------------

        i += 1

    if carry:
        c.append(carry)

    return c
```

☐ **A.**

```python
c.append(carry % 10)
```

☐ **B.**

```python
c.append(carry % 10)
carry = carry // 10
```

☐ **C.**

```python
carry = carry // 10
```

☐ **D.**

```python
c.append(carry // 10)
carry = carry % 10
```

2 判断题（每题 2 分，共 20 分）

题号	1	2	3	4	5	6	7	8	9	10
答案	×	√	√	×	√	√	√	×	√	×

第 1 题 数组和链表都是线性表。链表的优点是插入删除不需要移动元素，并且能随机查找。

第 2 题 假设函数 `gcd()` 函数能正确求两个正整数的最大公约数，则下面的 `lcm(a, b)` 函数能正确找到两个正整数 `a` 和 `b` 的最小公倍数。

```python
import math

def lcm(a, b):
    if a == 0 or b == 0:
        return 0
    return abs(a) // math.gcd(abs(a), abs(b)) * abs(b)
```

第 3 题 在 Python 实现的单链表中，已知引用 `p` 指向要删除的节点（该节点不是尾节点），若想在 $O(1)$ 时间复杂度内删除这个节点，可行的做法是：用 `p` 的后继节点 `p.next` 的值覆盖 `p` 自身的值，再用 `p` 的后继节点的 `next` 引用覆盖 `p` 的 `next` 引用，最后断开 `p` 的后继节点的引用以辅助垃圾回收（）

第 4 题 在求解所有不大于 n 的素数时，线性筛法（欧拉筛）都应当优先于埃氏筛法使用，因为线性筛法的时间复杂度为 $O(n)$，低于埃氏筛法的 $O(n \log \log n)$。

第 5 题 二分查找仅适用于有序数据。若输入数据无序，当仅进行一次查找时，为了使用二分而排序通常不划算。

第 6 题 通过在数组的第一个、最中间和最后一个这3个数据中选择中间值作为枢轴（比较基准），快速排序算法可降低落入最坏情况的概率。

第 7 题 贪心算法在每一步都做出当前看来最优的局部选择，并且一旦做出选择就不再回溯；而分治算法将问题分解为若干子问题分别求解，再将子问题的解合并得到原问题的解。

第 8 题 以下 `fib` 函数计算第 `n` 项斐波那契数（ `fib(0)=0` , `fib(1)=1` ），其时间复杂度为 $O(n)$。

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

第 9 题 递归函数一定要有终止条件，否则可能会造成栈溢出。

第 10 题 使用贪心算法解决问题时，通过对每一步求局部最优解，最终一定能找到全局最优解。

3 编程题（每题 25 分，共 50 分）

3.1 编程题 1

- **试题名称**：数字移动
- **时间限制**：3.0 s
- **内存限制**：512.0 MB

3.1.1 题目描述

小 A 有一个包含 N 个正整数的序列 $A = \{A_1, A_2, \ldots, A_N\}$，序列 A 恰好包含 $\frac{N}{2}$ 对不同的正整数。形式化地，对于任意 $1 \le i \le N$，存在**唯一一个** j 满足 $1 \le j \le N, i \ne j, A_i = A_j$。

小 A 希望每对相同的数字在序列中相邻，为了实现这一目的，小 A 每次操作会选择任意 i $(1 \le i \le N)$，将当前序列的第 i 个数字移动到任意位置，并花费对应数字的体力。

例如，假设序列 $A = \{1, 2, 1, 3, 2, 3\}$，小 A 可以选择 $i = 2$，将 $A_2 = 2$ 移动到 $A_3 = 1$ 的后面，此时序列变为 $\{1, 1, 2, 3, 2, 3\}$，耗费 2 点体力。小 A 也可以选择 $i = 3$，将 $A_3 = 1$ 移动到 $A_2 = 2$ 的前面，此时序列变为 $\{1, 1, 2, 3, 2, 3\}$，花费 1 点体力。

小 A 可以执行任意次操作，但他希望自己每次花费的体力尽可能小。小 A 希望你能帮他计算出一个**最小**的 x，使得他能够在每次花费的体力均不超过 x 的情况下令每对相同的数字在序列中相邻。

3.1.2 输入格式

第一行一个正整数 N，代表序列长度，保证 N 为偶数。

第二行包含 N 个正整数 $A_1, A_2, \ldots, A_N$，代表序列 A。且对于任意 $1 \le i \le N$，存在唯一一个 j 满足 $1 \le j \le N, i \ne j, A_i = A_j$。

数据保证小 A 至少需要执行一次操作。

3.1.3 输出格式

输出一行，代表满足要求的 x 的最小值。

3.1.4 样例

3.1.4.1 输入样例

```
6
1 2 1 3 2 3
```

3.1.4.2 输出样例

```
2
```

3.1.5 数据范围

对于40%的测试点，保证 $1 \leq N, A_i \leq 100$。

对于所有测试点，保证 $1 \leq N, A_i \leq 10^5$。

3.1.6 参考程序

```python
# 输入n和a数组
n = int(input())
a = []
while len(a) < n:
    a.extend(map(int, input().split()))
# 答案的上下界
left, right = 0, 1000000
ans = 1000000

# 二分搜索
while left <= right:
    # 考虑mid的力气能否完成任务
    mid = (left + right) // 2
    # 筛选出所有大于mid的数字
    b = [x for x in a if x > mid]
    # 先假设能够完成任务
    possible = True

    for i in range(0, len(b), 2):
        # 如果两个相同且大于mid的数字，中间还有其它的大于mid的数字，则本任务无法完成
        # 以(1 2 1 3 2 3)为例，两个3之间存在一个2。假设最大只能挪动1，则不论如何两个3之间的2不会被挪动，最终无法完成任务
        if b[i] != b[i+1]:
            possible = False
            break
    # 如果能够成功，则尝试更小的mid
    if possible:
        ans = mid
        right = mid - 1
    else:
        left = mid + 1

print(ans)
```

3.2 编程题 2

- **试题名称**：相等序列
- **时间限制**：3.0 s
- **内存限制**：512.0 MB

3.2.1 题目描述

小 A 有一个包含 N 个正整数的序列 $A = \{A_1, A_2, \ldots, A_N\}$。小 A 每次可以花费 1 个金币执行以下任意一种操作：

- 选择序列中一个正整数 A_i（$1 \le i \le N$），将 A_i 变为 $A_i \times P$，P 为任意质数；
- 选择序列中一个正整数 A_i（$1 \le i \le N$），将 A_i 变为 $\frac{A_i}{P}$，P 为任意质数，要求 A_i 能整除 P。

小 A 想请你帮他计算出令序列中所有整数都相同，最少需要花费多少金币。

3.2.2 输入格式

第一行一个正整数 N，含义如题面所示。

第二行包含 N 个正整数 $A_1, A_2, \ldots, A_N$，代表序列 A。

3.2.3 输出格式

输出一行，代表最少需要花费的金币数量。

3.2.4 样例

3.2.4.1 输入样例

```
5
10 6 35 105 42
```

3.2.4.2 输出样例

```
8
```

3.2.5 数据范围

对于60%的测试点，保证 $1 \le N, A_i \le 100$。

对于所有测试点，保证 $1 \le N, A_i \le 10^5$。

3.2.6 参考程序

```python
tokens = []
def get_val():
    while not tokens:
        tokens.extend(input().split())
    return int(tokens.pop(0))
# 一共有多少数
n = get_val()
# key: 质数
# value: 一个列表，num[2][3]表示有多少数质因数分解后可以包含2^3
# 因为一个数最大就到10^5，相当于是10w，所以列表最多包含20个数肯定够用，即2^20
num = {}

for _ in range(n):
    # 统计一下各个输入数字有多少质因数
    x = get_val()
    d = 2
    # 质因数必定小于根号x
    while d * d <= x:
        # 如果找到了一个数是x的因数
        if x % d == 0:
            cnt = 0
            # 那就一直除以这个数，并且更新x的质因数数目
            # 这样一来剩下能够整除x的必定是质数
            while x % d == 0:
                x //= d
                cnt += 1
            if d not in num: num[d] = [0] * 20
            # 数字x包含d^cnt，更新状态
            num[d][cnt] += 1
        d += 1
    # 如果x是质数，则x的值为1，也更新状态
    if x > 1:
        if x not in num: num[x] = [0] * 20
        num[x][1] += 1

ans = 0
for p in num:
    counts = num[p]
    # 计算有多少数字不包含这个质因数
    counts[0] = n - sum(counts)

    # 寻找中位数指数
    # 也就是寻找一个d^n，使得一半的数字包含超过n次方的d，另一半不包含
    # 如果让所有的数都统一到d^(n+1)那么将有超过一半的数字需要多乘以一个d，小于一半的数字可以少除以一个d，成本一定比现在大；反之亦然
    pos = 0
    median_exponent = 0
    for j in range(20):
        pos += counts[j]
        if pos * 2 >= n:
            median_exponent = j
            break

    # 计算代价
    for j in range(20):
        if counts[j]:
            ans += counts[j] * abs(j - median_exponent)

print(ans)
```