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

Python 五级

2026 年 03 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	D	C	B	A	C	C	B	A	D	A	B	C	C	B	B

第 1 题 关于 Python 实现的单链表、双链表和循环链表，下列说法正确的是（ ）。

- **A.** 在 Python 实现的单链表中，若已知任意结点对象的引用，则可以在 $O(1)$ 时间内删除该结点。
- **B.** Python 实现的循环链表中一定不存在值为 `None` 的引用属性。
- **C.** 在 Python 实现的循环双链表中，尾结点对象的 `next` 属性值一定为 `None`。
- **D.** 在 Python 实现的带头结点的循环单链表中，判定链表是否为空只需判断头结点对象的 `next` 属性是否引用头结点自身（即 `head.next is head`）。

第 2 题 双向循环链表中要在结点 `p` 之前插入新结点 `s`（均非空），以下操作正确的是（ ）。

- **A.**

```
s.next=p
p.prev=s
q.next=s
s.prev=q
```

- **B.**

```
s.prev=p
s.next=p.next
p.next.prev=s
p.next=s
```

- **C.**

```
s.next=p
s.prev=p.prev
p.prev.next=s
p.prev=s
```

- **D.**

```
s.next=p
s.prev=nullptr
p.prev=s
```

第 3 题 下面函数删除单向链表中 `val == x` 的节点，并且使用哑结点统一对头结点和中间节点的删除操作。横线处应填（ ）。

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

def eraseAll(head, x):
    dummy = Node(0)
    dummy.next = head
    cur = dummy

    while cur.next:
        if cur.next.val == x:
            ____________________  # 填空处
        else:
            cur = cur.next

    return dummy.next
```

- [ ] **A.**

```
cur = cur.next
```

- [ ] **B.**

```
cur.next = cur.next.next
```

- [ ] **C.**

```
 cur.next = None
```

- [ ] **D.**

```
cur.next = nullptr
```

第 4 题 对如下代码实现的欧几里得算法（辗转相除法），调用 `gcd(48, 18)` 得到的调用序列为（ ）。

```python
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

- [ ] **A.**

```
gcd(48,18) -> gcd(18,12) -> gcd(12,6) -> gcd(6,0)
```

- [ ] **B.**

```
gcd(48,18) -> gcd(30,18) -> gcd(12,18)
```

- [ ] **C.**

```
gcd(48,18) -> gcd(18,30) -> gcd(30,6)
```

- [ ] **D.**

```
gcd(48,18) -> gcd(12,18) -> gcd(6,12)
```

第 5 题 下面代码实现了欧拉（线性）筛，横线处应填写（ ）。

```python
def euler_sieve_for(n):
    if n < 2:
        return []
    is_composite = [False] * (n + 1)
    primes = []

    for i in range(2, n + 1):
        if not is_composite[i]:
            primes.append(i)

        ---------------------------
            p = primes[j]
            if i * p > n:
                break
            is_composite[i * p] = True
            if i % p == 0:
                break
    return primes

```

- [ ] **A.**

```python
for j in range(len(primes+1)):
```

- [ ] **B.**

```python
for j in range(len(primes)+1):
```

- [ ] **C.**

```python
for j in range(len(primes)):
```

- [ ] **D.**

```python
for j in range(len(primes)-1):
```

第 6 题 埃氏筛中将内层循环从 `j = i*i` 开始而不是 `j = 2*i` 的主要原因是（ ）。

```python
def eratosthenes_sieve_for(n):
    if n < 2:
        return []
    is_composite = [False] * (n + 1)
    primes = []

    for i in range(2, n + 1):
        if is_composite[i]:
            continue
        primes.append(i)

        # 用for循环模拟C++的 j = i*i; j <=n; j +=i
        for j in range(i * i, n + 1, i):
            is_composite[j] = True

    return primes
```

- [ ] **A.** 因为 `2*i` 一定不是合数
- [ ] **B.** `i*i` 一定是质数
- [ ] **C.** 小于 `i*i` 的 `i` 的倍数已被更小质因子筛过
- [ ] **D.** 这样可以把时间复杂度降为 $O(n)$

第 7 题 下面程序的运行结果为（ ）。

```python
def check(n, a, k, dist):
    cnt = 1
    last = a[0]

    for i in range(1, n):
        if a[i] - last >= dist:
            cnt += 1
            last = a[i]

    return cnt >= k

def solve(n, a, k):
    a.sort()

    l = 0
    r = a[-1] - a[0]

    while l < r:
        mid = (l + r + 1) // 2
        if check(n, a, k, mid):
            l = mid
        else:
            r = mid - 1

    return l

if __name__ == "__main__":
    a = [1, 2, 8, 4, 9]
    n = 5
    k = 3
    result = solve(n, a, k)
    print(result)
```

- [ ] **A.** 2
- [ ] **B.** 3
- [ ] **C.** 4
- [ ] **D.** 5

第 8 题 在升序数组中查找第一个大于等于 `x` 的位置，下面循环中横线应填（ ）。

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

if __name__ == "__main__":
    a1 = [1, 3, 5, 7, 9]
    x1 = 5
    print(f"数组{a1}中第一个≥{x1}的位置: {lowerBound(a1, x1)}")
```

- [ ] **A.**

```python
r = mid
```

- [ ] **B.**

```
r = mid - 1
```

- [ ] **C.**

```
l = mid
```

- [ ] **D.**

```
l = mid + 1
```

第 9 题 关于递归函数调用，下列说法错误的是（ ）。

- [ ] **A.** 递归调用层次过深时，可能会耗尽栈空间导致栈溢出
- [ ] **B.** 尾递归函数可以通过编译器优化来避免栈溢出
- [ ] **C.** 所有递归函数都可以通过循环结构来改写，从而避免栈溢出
- [ ] **D.** 栈溢出发生时，程序会抛出异常并可以继续执行后续代码

第 10 题 给定 `n` 根木头，第 `i` 根长度为 `a[i]`。要切成不少于 `m` 段等长木段，求最大可能长度，则横线上应填写（ ）。

```python
def check(a, m, x):
    cnt = 0
    for length in a:
        if x == 0:
            return True
        cnt += length
        if cnt >= m:
            return True
    return cnt >= m

def main():
    import sys
    input = sys.stdin.read().split()
    idx = 0
    n = int(input[idx])
    idx += 1
    m = int(input[idx])
    idx += 1

    a = []
    mx = 0
    for _ in range(n):
        num = int(input[idx])
        idx += 1
        a.append(num)
        mx = max(mx, num)

    l = 1
    r = mx
    ans = 0

    while l <= r:
        mid = l + (r - l) // 2
        if check(a, m, mid):
            ans = mid
            ---------------
        else:
            ---------------

    print(ans)

if __name__ == "__main__":
    main()
```

- [ ] **A.**

```
l = mid + 1 和 r = mid - 1
```

- [ ] **B.**

```
r = mid - 1 和 l = mid + 1
```

- [ ] **C.**

```
l = mid + 1 和 r = mid
```

- [ ] **D.**

```
l = mid - 1 和 r = mid + 1
```

第 11 题 下面代码用分治求“最大连续子段和”，其时间复杂度为（ ）。

```python
import sys

def solve(a, l, r):
    if l == r:
        return a[l]

    mid = l + (r - l) // 2

    left = solve(a, l, mid)
    right = solve(a, mid + 1, r)

    sum_val = 0
    lmax = -sys.maxsize - 1
    for i in range(mid, l - 1, -1):
        sum_val += a[i]
        lmax = max(lmax, sum_val)

    sum_val = 0
    rmax = -sys.maxsize - 1
    for i in range(mid + 1, r + 1):
        sum_val += a[i]
        rmax = max(rmax, sum_val)

    return max(left, right, lmax + rmax)

if __name__ == "__main__":
    a1 = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
    print(solve(a1, 0, len(a1)-1))

    a2 = [-5, -3, -1, -4]
    print(solve(a2, 0, len(a2)-1))

    a3 = [10]
    print(solve(a3, 0, 0))
```

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

第 12 题 游戏大赛决赛，两组选手分别按得分从小到大排好队，现在要把他们合并成一个有序排行榜。

A组：`A = {12, 35, 67, 89}`，B组：`B = {20, 45, 55, 78}`，下面是归并合并函数的核心循环，横线处应填入（ ）。

```python
A = [12, 35, 67, 89]
B = [20, 45, 55, 78]

i = 0
j = 0
result = []

while i < len(A) and j < len(B):
    ____________________
        result.append(A[i])
        i += 1
    else:
        result.append(B[j])
        j += 1

while i < len(A):
    result.append(A[i])
    i += 1

while j < len(B):
    result.append(B[j])
    j += 1


print("合并后的有序排行榜：", result)
```

- [ ] **A.** `if A[i+1] <= B[j]:`
- [ ] **B.** `if A[i] <= B[j+1]:`
- [ ] **C.** `if A[i] <= B[j]:`
- [ ] **D.** `if i < j:`

第 13 题 有 n 位同学的成绩已经从小到大排好序，现在对它执行下面这段以第一个元素为 `pivot` 的快速排序，请问此次排序的时间复杂度是（ ）。

```python
def quicksort(a, l, r):
    if l >= r:
        return
    pivot = a[l]
    i, j = l, r

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

    a[l], a[i] = a[i], a[l]

    quicksort(a, l, i - 1)
    quicksort(a, i + 1, r)

if __name__ == "__main__":
    scores = [60, 70, 80, 90, 100]
    print("排序前: ", scores)
    quicksort(scores, 0, len(scores)-1)
    print("排序后: ", scores)  # 输出: [60, 70, 80, 90, 100]
    def quicksort_with_log(a, l, r, depth=0):
        if l >= r:
            return
        print(f"递归深度{depth}, 处理区间[{l},{r}], 数组: {a[l:r+1]}")
        pivot = a[l]
        i, j = l, r
        while i < j:
            while i < j and a[j] >= pivot: j -= 1
            while i < j and a[i] <= pivot: i += 1
            if i < j: a[i], a[j] = a[j], a[i]
        a[l], a[i] = a[i], a[l]
        quicksort_with_log(a, l, i-1, depth+1)
        quicksort_with_log(a, i+1, r, depth+1)

    scores2 = [60,70,80,90,100]
    print("\n递归过程: ")
    quicksort_with_log(scores2, 0, 4)
```

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

第 14 题 下面关于排序算法的描述中，不正确的是(　)。

- ☐ **A.** 冒泡排序和插入排序都是稳定的排序算法
- ☐ **B.** 快速排序和归并排序都是不稳定的排序算法
- ☐ **C.** 冒泡排序和插入排序最好时间复杂度均为 $O(n)$
- ☐ **D.** 归并排序在最好、最坏和平均三种情况的时间复杂度均为 $O(n \log n)$

第 15 题 下面代码实现两个整数除法，其中被除数为一个“大整数”，用字符串表示，除数是一个小整数，用 `int` 表示，则横线处应该填写（ ）。

```python
def big_integer_division():
    s, b = input().split()
    b = int(b)

    a = [int(c) for c in s]

    c = []
    rem = 0

    for i in range(len(a)):
        rem = rem * 10 + a[i]
        q = rem // b
        c.append(q)
        ______________

    pos = 0
    while pos < len(c) - 1 and c[pos] == 0:
        pos += 1

    for i in range(pos, len(c)):
        print(c[i], end='')
    print()

    print(rem)

if __name__ == "__main__":
    big_integer_division()
```

- [ ] **A.**

```python
rem =rem / b
```

- [ ] **B.**

```python
rem = rem % b
```

- [ ] **C.**

```python
rem = b
```

- [ ] **D.**

```python
rem = q
```

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 有一个存储了 n 个整数的线性表，分别用 Python 列表（数组）和自定义单链表两种方式实现。在已知元素下标（或结点对象引用）的前提下，Python 列表的随机访问操作时间复杂度为 $O(1)$；而在 Python 实现的单链表中，已知某结点对象的引用时，在该结点之后插入一个新结点的操作时间复杂度也为 $O(1)$。

第 2 题 若数组 `a` 已按升序排列，则下面代码可以正确实现“在 a 中查找第一个大于等于 x 的元素的位置”。

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

if __name__ == "__main__":
    a1 = [1, 3, 5, 7, 9]
    x1 = 5
    print(f"数组{a1}中第一个≥{x1}的位置: {lowerBound(a1, x1)}")

    x2 = 6
    print(f"数组{a1}中第一个≥{x2}的位置: {lowerBound(a1, x2)}")

    x3 = 10
    print(f"数组{a1}中第一个≥{x3}的位置: {lowerBound(a1, x3)}")

    x4 = 0
    print(f"数组{a1}中第一个≥{x4}的位置: {lowerBound(a1, x4)}")
```

第 3 题 快速排序只要每次都选取中间元素作为枢轴，就一定是稳定排序。

第 4 题 若某算法满足递推式：

```
T(n) = 2T(n/2) + O(n)
```

则其时间复杂度为 $O(n \log n)$。

第 5 题 在一个数组中，如果两个元素 `a[i]` 和 `a[j]` 满足 `i < j` 且 `a[i] > a[j]`，则 `a[i]` 和 `a[j]` 是一个逆序对。
下面代码可以正确统计数组 `a` 区间 `[l,r]` 内的逆序对总数。

```python
cnt = 0

def merge_count(a, l, m, r):

    global cnt
    i = l
    j = m + 1

    while i <= m and j <= r:
        if a[i] <= a[j]:
            i += 1
        else:
            cnt += (m - i + 1)
            j += 1

if __name__ == "__main__":
    a = [2, 4, 1, 3]
    merge_count(a, 0, 1, 3)
    print(f"跨区间逆序对数量: {cnt}")
```

第 6 题 唯一分解定理保证：若一个数未被任何不超过其平方根的质数筛去，则它一定是质数。

第 7 题 假设数组 a 的值域范围是 D，以下程序的时间复杂度是 $O(n \log n + n \log D)$。

```python
def check(n, a, k, dist):
    cnt = 1
    last = a[0]

    for i in range(1, n):
        if a[i] - last >= dist:
            cnt += 1
            last = a[i]

    return cnt >= k

def solve(n, a, k):
    a_sorted = a.copy()
    a_sorted.sort()

    l = 0
    r = a_sorted[-1] - a_sorted[0]

    while l < r:
        mid = (l + r + 1) // 2
        if check(n, a_sorted, k, mid):
            l = mid
        else:
            r = mid - 1

    return l

if __name__ == "__main__":
    a = [1, 2, 8, 4, 9]
    n = 5
    k = 3
    result = solve(n, a, k)
    print(result)
```

第 8 题 若一个问题满足最优子结构性质，则一定可以用贪心算法得到最优解。（ ）

第 9 题 线性筛相比埃氏筛的核心改进在于：埃氏筛中一个合数可能被多个质数重复标记，线性筛通过"每个合数只被其最大质因子筛去"的策略，保证每个合数恰好被标记一次，从而实现 $O(n)$ 的时间复杂度。

第 10 题 任何递归程序都可以改写为等价的非递归程序，但改写后的非递归程序一定需要显式地使用栈来模拟递归调用过程。

3 编程题（每题 25 分，共 50 分）

3.1 编程题 1

- **试题名称**：有限不循环小数
- **时间限制**：1.0 s
- **内存限制**：512.0 MB

3.1.1 题目描述

若 $\frac{1}{a}$ 可化为一个有限的，不循环的小数，则称 a 为**终止数**。

请你求出在 L 到 R 中终止数的数量。

3.1.2 输入格式

输入一行，包含两个整数 L, R。

3.1.3 输出格式

输出一行，包含一个整数，表示 L 到 R 中终止数的数量。

3.1.4 样例

3.1.4.1 输入样例

```
2 11
```

3.1.4.2 输出样例

```
5
```

3.1.5 样例解释

在 $[2, 11]$ 终止数有 2、4、5、8、10。

3.1.6 数据范围

保证 $1 \leq L \leq R \leq 10^6$。

3.1.7 参考程序

```python
# L, R输入
l, r = map(int, input().split())
# 最终的答案，初始化为0
ans = 0
for i in range(l, r + 1):
    t = i
    # 不断除以2
    while t > 0 and t % 2 == 0:
        t //= 2
    # 不断除以5
    while t > 0 and t % 5 == 0:
        t //= 5
    # 如果不含其他因子，则这个数能够表示为不循环的小数
    if t == 1:
        ans += 1
print(ans)
```

3.2 编程题 2

- **试题名称**：找数
- **时间限制**：1.0 s
- **内存限制**：512.0 MB

3.2.1 题目描述

给定一个包含 n 个互不相同的正整数的数组 A 与一个包含 m 个互不相同的正整数的数组 B，请你帮忙计算有多少数在数组 A 与 数组 B 中均出现。

3.2.2 输入格式

第一行包含两个整数 n, m。

第二行包含 n 个正整数 $a_1, a_2, \cdots, a_n$ 表示数组 A。

第二行包含 m 个正整数 $b_1, b_2, \cdots, b_m$ 表示数组 B。

3.2.3 输出格式

输出一个整数，表示在数组 A 与 数组 B 中均出现的数的个数。

3.2.4 样例

3.2.4.1 输入样例

```
3 5
4 2 3
3 1 5 4 6
```

3.2.4.2 输出样例

```
2
```

3.2.5 样例解释

样例 1 中，4、3 在数组 A 与 B 中均出现。

3.2.6 数据范围

对于 40% 的数据，保证 $1 \leq n, m \leq 1000$。

对于 100% 的数据，保证 $1 \leq n, m \leq 10^5$，$1 \leq a_i, b_i \leq 10^9$。

3.2.7 参考程序

```python
# 接收a,b数组的大小以及数组内容
n, m = map(int, input().split())
a = list(map(int, input().split()))
b = list(map(int, input().split()))

a.sort()
ans = 0
# 遍历b中的每一个数
for x in b:
    l, r = 0, n - 1
    # 二分查找a中是否包含x
    while l < r:
        mid = (l + r) // 2
        # 如果a[mid]<x，则x一定在mid+1之后
        if a[mid] < x:
            l = mid + 1
        else:
            r = mid
    if a[l] == x:
        ans += 1
print(ans)
```