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

Python 五级

2025 年 06 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	D	D	A	C	D	C	D	C	D	D	A	A	D

第 1 题 有关下列Python代码的说法，错误的是(　　)。

```
#自定义函数SORTED()仿Python的sorted()功能
def SORTED(for_in_data, fx = None, Reverse = False):
    lst = list(for_in_data)
    if fx == None:
        lst.sort(reverse = Reverse)
    else:
        lst.sort(key = lambda x:fx(x), reverse = Reverse)
    return lst
```

- [ ] **A.** 执行 `SORTED({1:1,2:4,3:9})` 不会报错
- [ ] **B.** 执行 `SORTED(range(10))` 不会报错
- [ ] **C.** 执行 `SORTED([1,20,3], Reverse = True)[::-1]` 不会报错
- [ ] **D.** 执行 `SORTED([1,None,'123'])` 不会报错

第 2 题 下列Python代码用于判断一个正整数是否是质数（素数），相关说法中正确的是(　　)。

```
def is_prime(N):
    if N <= 1:
        return False  # 处理所有非正整数
    if N == 2 or N == 3 or N == 5:
        return True
    if N % 2 == 0 or N % 3 == 0 or N % 5 == 0:
        return False
    i = 7
    step = 4
    finish_number = int(N ** 0.5) + 1

    while i <= finish_number:
        if N % i == 0:
            return False
        i += step
        step = 6 - step
    return True

#列出1-N之间所有质数
N = int(input())
print([n for n in range(1,N+1) if is_prime(n)])

```

- [ ] **A.** 代码存在错误，比如5是质数，但因为 `5 % 5` 余数是0返回了False
- [ ] **B.** `finish_number` 的值应该是 `N // 2` ，当前写法将导致错误
- [ ] **C.** 当前while循环正确的前提是：所有大于3的质数都符合 `6k±1` 形式
- [ ] **D.** while循环修改如下，其执行效果与执行时间相同。

```
    for i in range(2, finish_number):
        if N % i == 0:
            return False
    return True
```

第 3 题 下列Python代码用于求解两个正整数的最大公约数，相关说法中错误的是( )。

```
def gcd0(big, small):
    if big < small:
        big, small = small, big
    if big % small == 0:
        return small
    return gcd0(small, big % small)

def gcd1(big,small):
    if big < small:
        big, small = small, big
    for i in range(small, 0, -1):
        if big % i == 0 and small % i == 0:
            return i

print(gcd0(48,24))
print(gcd1(36,24))
```

- [ ] **A.** gcd0()函数的时间复杂度为 `O(logN)`
- [ ] **B.** gcd1()函数的时间复杂度为 `O(N)`
- [ ] **C.** 一般说来， `gcd0()` 的效率高于 `gcd1()`
- [ ] **D.** `gcd1()` 中的代码 `range(small, 0, -1)` 应该修改为 `range(small, 1, -1)`

第 4 题 在Python中，可以用字典模拟单向或双向链表的实现。下面的代码模拟单向链表，和链表对象相比，有关其缺点的说法，错误的是( )。

```
node1 = {'data': 1, 'next': None}
node2 = {'data': 2, 'next': None}
node1['next'] = node2  #node1的下一个节点是node2
```

- [ ] **A.** 类型安全差，易出错。比如： `node1["NEXT"] = node2` 不会报错，导致逻辑错误
- [ ] **B.** 内存开销大。字典需要保存键名称以及哈希表
- [ ] **C.** 无法封装方法，如 `insert()` 插入函数较为常用，但其代码需要分散在外部
- [ ] **D.** 重复存储，难以保证一致性。如在 `node1['next'] = node2` 代码中， `node1['next']` 的值为 `node2` ，而 `node2` 自身也将保存一份

第 5 题 基于上题代码，模拟单向链表实现插入新节点函数insertNode()的代码如下，横线处应填入的代码是( )。

```
def insertNode(linkNode, newNodeData):
    newNode = ____________________________
    ____________________________
```

- [ ] **A.**

```
    {'data': newNodeData, 'next': linkNode['next']}
    linkNode['next'] = newNode
```

- [ ] **B.**

```
    {'data': newNodeData, 'next': None}
    linkNode['next'] = newNode
```

- [ ] **C.**

```
    {'data': newNodeData, 'next': newNode}
    linkNode['next'] = newNode
```

- [ ] **D.**

```
    {'data': newNodeData, 'next': linkNode['next']}
    linkNode = newNode
```

第 6 题 下面的Python代码用于实现双向链表。is_empty()函数用于判断链表是否为空，如为空返回True，否则False。横线处不能填入的代码是(　)。

```
class double_link:
    class Node:
        def __init__(self, data):
            self.data = data
            self.prev = None  # 指向前一个节点
            self.next = None  # 指向下一个节点

    def __init__(self):
        self.head = None
        self.tail = None
        self._size = 0

    def is_empty(self):
        return ___________
```

- [ ] **A.** `self.tail is None`
- [ ] **B.** `self._size == 0`
- [ ] **C.** `self == None`
- [ ] **D.** `self.head is None`

第 7 题 基于上题代码正确的前提下，填入相应代码完善append()函数，用于在尾部增加新节点(　)。

```
def append(self, data):
    new_node = self.Node(data)

    if self.is_empty():
       self.head = new_node
       self.tail = new_node
    else:
       ____________________
       ____________________
       ____________________
    self._size += 1
    return new_node
```

☐ **A.**

```
self.tail.next = new_node
```

☐ **B.**

```
new_node.prev = self.tail
self.tail.next = new_node
```

☐ **C.**

```
self.tail = new_node
new_node.prev = self.tail
self.tail.next = new_node
```

☐ **D.**

```
new_node.prev = self.tail
self.tail.next = new_node
self.tail = new_node
```

第 8 题 下面的Python代码，用于求一系列数据中的最大值。有关其算法说法错误的是（ ）。

```
def find_max(nums):
    if not nums:
        raise ValueError("输入数组不能为空")

    def _find_max(left, right):
        if left == right:
            return nums[left]
        mid = (left + right) // 2
        return max(
            _find_max(left, mid),
            _find_max(mid + 1, right)
        )

    return _find_max(0, len(nums) - 1)

print(find_max([3,39,1,31,1,2,3,12,2]))
```

☐ **A.** 该算法采用分治算法

☐ **B.** 该算法是递归实现

☐ **C.** 该算法采用贪心算法

☐ **D.** 该算法不是递推算法

第 9 题 基于上题的find_max()实现，下面的说法错误的是（　　）。

- [ ] **A.** `find_max()` 适用于 `str`
- [ ] **B.** `find_max()` 适用于 `list`
- [ ] **C.** `find_max()` 适用于 `tuple`
- [ ] **D.** `find_max()` 适用于 `dict`

第 10 题 下面的Python代码，用于求一系列数据中的最大值。有关其算法说法错误的是（　）。

```python
def find_max(nums):
    if not nums:
        raise ValueError("输入数组不能为空")
    if len(nums) == 1:
        return nums[0]
    mid = len(nums) // 2
    return max(
        find_max(nums[:mid]),
        find_max(nums[mid:])
    )

print(find_max([1,2,2,11,21,1,2,3]))
```

- [ ] **A.** 本题的 `find_max()` 函数采用分治算法
- [ ] **B.** 和上上题的 `find_max()` 函数相比，本题 `find_max()` 运行效率相对较低，因为需要分配额外的内存空间，用以存储nums的切片结果
- [ ] **C.** 和上上题的 `find_max()` 函数相比，本题 `find_max()` 的空间复杂度与之相同，不需要额外的存储资源
- [ ] **D.** 本题的 `find_max()` 的时间复杂度为 `O(n log n)`

第 11 题 下面的Python代码，用于求一系列数据中的最大值。有关其算法说法错误的是（　）。

```python
def find_max(nums):
    if not nums:
        raise ValueError("输入数组不能为空")
    max_value = nums[0]
    for i in nums:
        if max_value < i:
            max_value = i
    return max_value

print(find_max([1,2,2,11,21,1,2,3]))
```

- [ ] **A.** 本题 `find_max()` 函数的实现是递推（迭代）算法
- [ ] **B.** 本题 `find_max()` 函数的时间复杂度为 `O(n)`
- [ ] **C.** 和前面题的 `find_max()` 相比，因为没有递归，所以也就没有栈的创建和销毁开销，因此不会有与递归相关的栈溢出错误
- [ ] **D.** 本题的 `find_max()` 函数支持 `dict` 类型，因为 `dict` 也支持 `for-in` 循环。

第 12 题 下面的代码用于列出求1到N之间的所有质数，错误的说法是（ ）。

```python
def is_prime(N):
    if N <= 1:return False
    finish_number = int(N ** 0.5) + 1

    for i in range(2, finish_number):
        if N % i == 0:
            return False
    return True

#列出1-N之间所有质数
N = int(input())
print([n for n in range(1,N+1) if is_prime(n)])
```

- [ ] **A.** 埃氏筛算法相对于上面的代码效率更高
- [ ] **B.** 线性筛算法相对于上面的代码效率更高
- [ ] **C.** 上面的代码，有很多重复计算，因为不是判断单个数是否为质数，故而导致筛选出连续数中质数的效率不高
- [ ] **D.** 相对而言，埃氏筛算法比上面代码以及线性筛算法效率都高

第 13 题 硬币找零，要求找给客户最少的硬币。第一行输入硬币规格且空格间隔，单位为角，规格假设都小于10角，且一定有1角规格。硬币规格不一定是标准的货币系统，可能出现4角、2角等规格。第二行输入找零金额，约定必须为1角的整数倍。输出为每种规格及其数量，按规格从大到小输出，如果某种规格不必要，则输出为0。下面是其实现代码，相关说法正确的是（　）。

```python
coins = sorted(list(map(int, input().split())), reverse=True)
amount = int(input())
result = {coin: 0 for coin in coins}
for coin in coins:
    num = amount // coin
    result[coin] = num
    amount -= num * coin
    if amount == 0:
        break

for coin in coins:
    print(f"{coin}角需要{result[coin]}枚")
```

- [ ] **A.** 上述代码采用贪心算法实现
- [ ] **B.** 上述代码总能找到本题目要求的最优解
- [ ] **C.** 上述代码采用枚举算法实现
- [ ] **D.** 上述代码采用分治算法

第 14 题 下面Python代码用于在升序lst（list类型）中查找目标值target最后一次出现的位置。相关说法，正确的是（）。

```python
def binary_search(lst, target):
    if len(lst) == 0:
        return None
    low, high = 0, len(lst)-1
    while low < high:
        mid = (low + high + 1) // 2  # 向上取整
        if lst[mid] <= target:
            low = mid
        else:
            high = mid - 1
    return low if lst[low] == target else None
```

- [ ] **A.** 当lst中存在重复的target时，该函数总能返回最后一个target的位置，即便lst全由相同元素组成
- [ ] **B.** 当target小于lst中所有元素时，该函数会返回0
- [ ] **C.** 循环条件改为 `while low <= high` 程序执行效果相同，且能提高准确性
- [ ] **D.** 将代码中 `(low + high + 1) // 2` 修改为 `(low + high) // 2` 效果相同

第 15 题 有关下面Python代码的说法，错误的是（　）。

```python
def sqrt_binary(n, epsilon=1e-10):
    if n < 0:
        raise ValueError("输入必须为非负整数")
    if n == 0 or n == 1:
        return n

    # 阶段1:
    low, high = 1, n
    k = 0
    while low <= high:
        mid = (low + high) // 2
        mid_sq = mid * mid
        if mid_sq == n:
            return mid
        elif mid_sq < n:
            k = mid
            low = mid + 1
        else:
            high = mid - 1

    next_k = k + 1
    next_sq = next_k * next_k

    if next_sq == n:
        return next_k

    # 阶段2:
    low_float, high_float = float(k), float(k + 1)

    while high_float - low_float >= epsilon:
        mid = (low_float + high_float) / 2
        mid_sq = mid * mid

        if mid_sq < n:
            low_float = mid
        else:
            high_float = mid

    # 浮点数阶段后的最终结果
    result = (low_float + high_float) / 2

    check_int = int(result + 0.5)
    if check_int * check_int == n:
        return check_int

    return result

#测试示例
print(sqrt_binary(16))   # 输出: 4
print(sqrt_binary(17))   # 输出: ≈4.123
print(sqrt_binary(10**14)) # 输出: 10000000
```

- [ ] **A.** “阶段1”的目标是寻找正整数n可能的正完全平方根

☐ **B.** “阶段2”的目标是如果正整数n没有正完全平方根，则在可能产生完全平方根附近寻找带小数点的平方根

☐ **C.** 代码 `check_int = int(result + 0.5)` 是检查因浮点误差是否为正完全平方根

☐ **D.** 对巨大数求平方根，本算法同样适用。

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 下面Python代码是用欧几里得算法（辗转相除法）求两个大于0的正整数的最大公约数，a大于b还是小于b都适用。（ ）

```
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
```

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()` 函数能求相应两数的最小公倍数。(　　)

```
def lcm(a, b):
    return a * b // gcd(a, b)
```

第 3 题 在下面的Python代码中，for-in每次循环都将执行 `n ** 0.5` ，并取整后加上1。(　　)

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

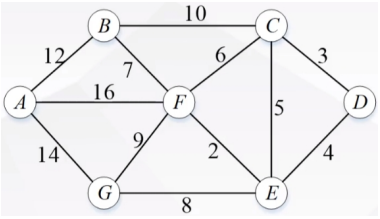
第 4 题 下面的Python代码用于输出每个数对应的质因数列表，输出形如：`{5: [5], 6: [2, 3], 7: [7], 8: [2, 2, 2]}` 。(　　)

```
n, m = map(int,input().split())
if n > m:
    n, m = m, n
prime_factor = {} #保存每个数的质因数
for i in range(n, m + 1):
    j, k = 2, i
    while k != 1:
        if k % j == 0:
            prime_factor[i] = prime_factor.get(i,[]) + j
            k //= j
            continue
        j += 1
print(prime_factor)
```

第 5 题 下面Python代码执行后输出是15。（ ）

```python
def func(n):
    if n <= 0:
        return 0
    n -= 1
    return n + func(n)
print(func(5))
```

第 6 题 求解下图中A点到D点最短路径，其中A到B之间的12可以理解为距离。求解这样的问题，常用Dijkstra算法，其思路是通过逐步选择当前距离起点最近的节点来求解非负权重图（如距离不能为负值）单源最短路径的算法。从该算法的描述可以看出，Dijkstra算法是贪心算法。（ ）



第 7 题 下面的Python代码用于归并排序（merge sort）,已测试执行正确。假如整体交换 `merge()` 和 `merge_sort()` 两个函数的先后位置，则程序执行将触发异常。（ ）

```python
def merge(left, right):
    result, i, j = [], 0, 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_arr = merge_sort(arr[:mid])
    right_arr = merge_sort(arr[mid:])
    print("HERE")
    return merge(left_arr, right_arr)

#测试代码
arr = [38, 27, 43, 3, 9, 82, 10]
sorted_arr = merge_sort(arr)
print(sorted_arr)
```

第 8 题 基于上一题的Python代码。上题代码在执行时，将输出一次 `HERE` 字符串，因为merge_sort()递归调用在 `print("HERE")` 之前，因此merge()函数仅被调用一次。（ ）

第 9 题 基于上上题的Python代码。代码片段 `left_arr = merge_sort(arr[:mid])` 和 `right_arr = merge_sort(arr[mid:])` 因为切片，将产生的新的list，对于大容量list的排序，将需要大量额外存储空间，可以优化为就地（implace）排序。（ ）

第 10 题 下面的Python代码实现如果对一维list【形如：[32,12,32,13,42,1],而不是[(1,3),(3,1),(321,321),(32,13)]】进行快速排序，该排序是稳定排序。（ ）

```python
def qSort(arr):
    if len(arr) <= 1:
```

```
        return arr
    pivot = arr[len(arr) // 2]
    left, middle, right  = [], [], []
    for num in arr:
        if num < pivot:
            left.append(num)
        elif num > pivot:
            right.append(num)
        else:
            middle.append(num)

    return qSort(left) + middle + qSort(right)
```

3　编程题（每题 25 分，共 50 分）

3.1　编程题 1

- **试题名称**：奖品兑换
- **时间限制**：3.0 s
- **内存限制**：512.0 MB

3.1.1　题目描述

班主任给上课专心听讲、认真完成作业的同学们分别发放了若干张课堂优秀券和作业优秀券。同学们可以使用这两种券找班主任兑换奖品。具体来说，可以使用 a 张课堂优秀券和 b 张作业优秀券兑换一份奖品，或者使用 b 张课堂优秀券和 a 张作业优秀券兑换一份奖品。

现在小 A 有 n 张课堂优秀券和 m 张作业优秀券，他最多能兑换多少份奖品呢?

3.1.2　输入格式

第一行，两个正整数 n, m，分别表示小 A 持有的课堂优秀券和作业优秀券的数量。

第二行，两个正整数 a, b，表示兑换一份奖品所需的两种券的数量。

3.1.3　输出格式

输出共一行，一个整数，表示最多能兑换的奖品份数。

3.1.4　样例

3.1.4.1　输入样例 1

```
8 8
2 1
```

3.1.4.2　输出样例 1

```
5
```

3.1.4.3　输入样例 2

```
314159 2653589
27 1828
```

3.1.4.4 输出样例 2

```
1599
```

3.1.5 数据范围

对于 60% 的测试点，保证 $1 \leq a, b \leq 100$，$1 \leq n, m \leq 500$。

对于所有测试点，保证 $1 \leq a, b \leq 10^4$，$1 \leq n, m \leq 10^9$。

3.1.6 参考程序

```python

# 接受小A持有的券的数量
n, m = map(int, input().split())
# 兑换奖品所需的券的数量
a, b = map(int, input().split())

# 因为兑换礼品其实两种券可以交换，所以顺序不是很重要，直接令需求较少的一种券作为n或者a即可
n, m = min(n, m), max(n, m)
a, b = min(a, b), max(a, b)

def check(v):
    """
    检测手头的券的数量能否兑换v份礼品
    :param v: 目标礼品数目
    :return 是或否
    """
    # 兑换v份礼品需要的两种券的数量，尽可能用手头较多的券来应对需求量较多的券。
    x, y = a * v, b * v
    # 如果应对不过来，再想办法拿手头数量较少的券应对需求较多的券
    # 相当于有v-t份礼品是a张手头较少的券，b张手头较多的券兑换的；剩下的t份奖品是a张手头较多的券和b张手头较少的券兑换的。
    if y >= m and b > a:
        t = (y - m + (b - a) - 1) // (b - a)
        y -= t * (b - a)
        x += t * (b - a)
    return x <= n and y <= m

l, r = 0, int(1e9)
while l < r:
    mid = (l + r + 1) // 2
    if check(mid):
        l = mid
    else:
        r = mid - 1
print(r)
```

3.2 编程题 2

- **试题名称**：最大公因数
- **时间限制**：3.0 s
- **内存限制**：512.0 MB

3.2.1 题目描述

对于两个正整数 a, b，它们的最大公因数记为 $\gcd(a, b)$。对于 $k \geq 3$ 个正整数 $c_1, c_2, \ldots, c_k$，它们的最大公因数为：

$$\gcd(c_1, c_2, \ldots, c_k) = \gcd(\gcd(c_1, c_2, \ldots, c_{k-1}), c_k)$$

给定 n 个正整数 $a_1, a_2, \ldots, a_n$ 以及 q 组询问。对于第 i（$1 \leq i \leq q$）组询问，请求出 $a_1 + i, a_2 + i, \ldots, a_n + i$ 的最大公因数，也即 $\gcd(a_1 + i, a_2 + i, \ldots, a_n + i)$。

3.2.2 输入格式

第一行，两个正整数 n, q，分别表示给定正整数的数量，以及询问组数。

第二行，n 个正整数 $a_1, a_2, \ldots, a_n$。

3.2.3 输出格式

输出共 q 行，第 i 行包含一个正整数，表示 $a_1 + i, a_2 + i, \ldots, a_n + i$ 的最大公因数。

3.2.4 样例

3.2.4.1 输入样例 1

```
5 3
6 9 12 18 30
```

3.2.4.2 输出样例 1

```
1
1
3
```

3.2.4.3 输入样例 2

```
3 5
31 47 59
```

3.2.4.4 输出样例 2

```
4
1
2
1
4
```

3.2.5 数据范围

对于 60% 的测试点，保证 $1 \leq n \leq 10^3$，$1 \leq q \leq 10$。

对于所有测试点，保证 $1 \leq n \leq 10^5$，$1 \leq q \leq 10^5$，$1 \leq a_i \leq 1000$。

3.2.6 参考程序

```python
# 得到n和q
n, q = map(int, input().split())
# 对n个数进行排序
a = sorted(list(map(int, input().split())))

# 欧几里得算法求a和b的最大公约数
def gcd(a, b):
    if a == 0 or b == 0:
        return a + b
    return gcd(b, a % b)

g = 0
for i in range(1, n):
    g = gcd(g, a[i] - a[i - 1])
```

```
for i in range(q):
    print(gcd(g, a[0] + i + 1))
```