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

Python　五级

2023 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	B	C	B	C	A	D	A	B	A	C	B	D	B	D	B

第 1 题 通讯卫星在通信网络系统中主要起到（）的作用。

- ☐ **A.** 信息过滤
- ☐ **B.** 信号中继
- ☐ **C.** 避免攻击
- ☐ **D.** 数据加密

第 2 题 小杨想编写一个判断任意输入的整数N是否为素数的程序，下面哪个方法不合适？（ ）

- ☐ **A.** 埃氏筛法
- ☐ **B.** 线性筛法
- ☐ **C.** 二分答案
- ☐ **D.** 枚举法

第 3 题

内排序有不同的类别，下面哪种排序算法和冒泡排序是同一类？（ ）

- ☐ **A.** 希尔排序
- ☐ **B.** 快速排序
- ☐ **C.** 堆排序
- ☐ **D.** 插入排序

第 4 题 下面Python代码用于求斐波那契数列，该数列第1、2项为1，以后各项均是前两项之和。下面有关说法错误的是(　　)。

```python
def fiboA(N):
    if N == 1 or N == 2:
        return 1
    return fiboA(N - 1) + fiboA(N - 2)

def fiboB(N):
    if N == 1 or N == 2:
        return 1

    last2, last1 = 1, 1
    for i in range(2,N):
        nowVal = last1 + last2
        last2, last1 = last1, nowVal
    return nowVal
```

- [ ] **A.** `fiboA( )` 用递归方式，`fiboB()` 循环方式
- [ ] **B.** `fiboA( )` 更加符合斐波那契数列的数学定义，直观易于理解，而 `fiboB()` 需要将数学定义转换为计算机程序实现
- [ ] **C.** `fiboA( )` 不仅仅更加符合数学定义，直观易于理解，且因代码量较少执行效率更高
- [ ] **D.** `fiboB( )` 虽然代码量有所增加，但其执行效率更高

第 5 题 下面Python代码以递归方式实现合并排序，并假设 `merge(left,right)` 函数能对有序（同样排序规则）的 `left` 和 `right` 排序。横线处应填上代码是( )。

```python
def mergeSort(listData):
    if len(listData) <= 1:
        return listData

    Middle = len(listData) // 2
    Left, Right = _____

    return merge(Left, Right)
```

- [ ] **A.** `mergeSort(listData[:Middle]), mergeSort(listData[Middle:])`
- [ ] **B.** `mergeSort(listData[:Middle-1]), mergeSort(listData[Middle+1:])`
- [ ] **C.** `mergeSort(listData[:Middle]), mergeSort(listData[Middle+1:])`
- [ ] **D.** `mergeSort(listData[:Middle-1]), mergeSort(listData[Middle:])`

第 6 题 阅读下面的Python代码，执行后其输出是( )。

```
stepCount = 0
def countIt(Fx):
    def wrapper(*args,**kwargs):
        rtn = Fx(*args,**kwargs)
        global stepCount
        stepCount += 1
        print(stepCount,end="->")
        return rtn
    return wrapper

@countIt
def fracA(N):
    rtn = 1
    for i in range(1,N+1):
        rtn *= i
    return rtn;

@countIt
def fracB(N):
    if N == 1:
        return 1
    return N*fracB(N-1)

print(fracA(5),end="")
print("<===>",end="")
print(fracB(5))
```

- [ ] **A.** `1->120<===>2->120`
- [ ] **B.** `1->120<===>1->120`
- [ ] **C.** `1->120<===>1->2->3->4->5->120`
- [ ] **D.** `1->120<===>2->3->4->5->6->120`

第 7 题 下面的Python用于对 `lstA` 排序，使得偶数在前奇数在后，横线处不应填入(　　)。

```
def isEven(N):
    return N % 2 == 0

lstA = list(range(1,100))
lstA.sort(_____________________________)
```

- [ ] **A.** `key = not isEven`
- [ ] **B.** `key = lambda x: isEven(x), reverse = True`
- [ ] **C.** `key = isEven, reverse = True`
- [ ] **D.** `key = lambda x: not isEven(x)`

第 8 题 下面的Python代码用于排序 `sA` 字符串中每个字符出现的次数（字频），`sA` 字符串可能很长，此处仅为示例。排序要求是按字频降序，如果字频相同则按字符的ASCII升序，横线处应填入代码是（　）。

```
sA = "Simple is better than complex"
charCount = {} #每个字符对应数量
for c in sA:
    charCount[c] = charCount.get(c, 0) + 1
print(sorted(_____________________________))
```

- [ ] **A.** `charCount, key = lambda x:(-x[1],x[0])`
- [ ] **B.** `charCount.items(), key = lambda x:(-x[1],ord(x[0]))`
- [ ] **C.** `charCount.items(), key = lambda x:(x[1],-ord(x[0]))`
- [ ] **D.** 触发异常，不能对字典进行排序。

第 9 题 有关下面Python代码正确的是（ ）。

```
isEven = lambda x: x % 2 == 0
def isOdd(N):
    return N % 2 == 1
print(type(isEven) ==  type(isOdd), isEven(10),isOdd(10))
```

- ☐ **A.** `True True False`
- ☐ **B.** `False True False`
- ☐ **C.** `False False True`
- ☐ **D.** 触发异常

第 10 题 下面的Python代码实现对 `list` 的快速排序，有关说法，错误的是（ ）。

```
def qSort(lst):
    if len(lst) < 2:
        return lst

    pivot = lst[0]
    less = [i for i in lst[1:] if i <= pivot]
    greater = [i for i in lst[1:] if i > pivot]

    return _____________________________
```

- ☐ **A.** `qSort(less) + qSort(greater) + [pivot]`
- ☐ **B.** `[pivot] + qSort(less) + qSort(greater)`
- ☐ **C.** `qSort(less) + [pivot] + qSort(greater)`
- ☐ **D.** `qSort(less) + pivot + qSort(greater)`

第 11 题 下面Python代码中的 `isPrimeA()` 和 `isPrimeB()` 都用于判断参数N是否素数，有关其时间复杂度的正确说法是（ ）。

```
def isPrimeA(N):
    if N < 2:return False

    for i in range(2,N // 2 + 1):
        if N % i == 0:return False
    return True

def isPrimeB(N):
    if N < 2:return False

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

- ☐ **A.** `isPrimeA( )` 的最坏时间复杂度是$O(\frac{N}{2})$，`isPrimeB( )` 的最坏时间复杂度是$O(\log N)$，isPrimeA()优于isPrimeB()
- ☐ **B.** `isPrimeA()` 的最坏时间复杂度是$O(\frac{N}{2})$，`isPrimeB( )` 的最坏时间复杂度是$O(N^{\frac{1}{2}})$，isPrimeB()绝大多数情况下优于isPrimeA()
- ☐ **C.** `isPrimeA()` 的最坏时间复杂度是$O(N^{\frac{1}{2}})$，`isPrimeB( )` 的最坏时间复杂度是$O(N)$，`isPrimeA( )` 优于 `isPrimeB( )`
- ☐ **D.** `isPrimeA()` 的最坏时间复杂度是$O(\log N)$，`isPrimeB( )` 的最坏时间复杂度是$O(N)$，isPrimeA()优于 `isPrimeB( )`

第 12 题 下面Python代码用于有序 `list` 的二分查找，有关说法错误的是（ ）。

```
def bSearch(lst,Val):

    def _binarySearch(lst, Low, High, Target):
        if Low > High:
            return -1
        Mid = (Low + High) // 2  #求序列中间位置
        if Target == lst[Mid]:
            return Mid
        elif Target < lst[Mid]:  #如目标值小于中间元素，在左半部分查找
            return _binarySearch(lst, Low, Mid - 1, Target)
        else:  #如目标值大于中间元素，在右半部分查找
            return _binarySearch(lst, Mid + 1, High, Target)

    return _binarySearch(lst,0,len(lst),Val)
```

- ☐ **A.** 代码采用二分法实现有序list的查找
- ☐ **B.** 代码采用分治算法实现有序list的查找
- ☐ **C.** 代码采用递归方式实现有序list的查找
- ☐ **D.** 代码采用动态规划算法实现有序list的查找

第 13 题 在上题的算法中，其时间复杂度是（ ）。

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

第 14 题 下面的Python代码用于实现每个字符后紧跟随字符及其出现次数，并对紧跟随字符排序，即出现次数最多排在前，形如：`{'中': [('文', 1), ('国', 2), ('华', 2)]}`，此处 `S` 仅是部分字符，可能很多，横线处应填入代码是（ ）。

```
S = """中国华夏中华中华人民共和国中国人中文"""

dictAfter = {}
for i,hz in enumerate(S[1:]):
    tmpDict = dictAfter.get(S[i], {})
    tmpDict[hz] = tmpDict.get(hz, 0) + 1
    dictAfter[S[i]] = tmpDict
dictAfter = { _____ for x in dictAfter.items()}
print(dictAfter)
```

- ☐ **A.** `x[0]:x[1].items().sort(key = lambda x:x[1], reverse = True)`
- ☐ **B.** `x[0]:x[1].items().sort(key = lambda x:x[0], reverse = True)`
- ☐ **C.** `x[0]:sorted(x[1].items(), key = lambda x:-x[0])`
- ☐ **D.** `x[0]:sorted(x[1].items(), key = lambda x:-x[1])`

第 15 题 有关下面Python代码的说法正确的是（ ）。

```
class Node:
    def __init__(self, Val, Prv=None, Nxt = None)
        self.Value = Val
        self.Prev = Prv
        self.Next = Nxt

firstNode = Node(10)
firstNode.Next = Node(100,firstNode)
firstNode.Next.Next = Node(111,firstNode.Next)
```

- [ ] **A.** 上述代码构成单向链表
- [ ] **B.** 上述代码构成双向链表
- [ ] **C.** 上述代码构成循环链表
- [ ] **D.** 上述代码构成指针链表

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 小杨想写一个程序来算出正整数N有多少个因数，经过思考他写出了一个重复没有超过N/2次的循环就能够算出来了。（ ）

第 2 题 同样的整数序列分别保存在单链表和双向链中，这两种链表上的简单冒泡排序的复杂度相同。（ ）

第 3 题 归并排序的时间复杂度是$O(N \log N)$。（ ）

第 4 题 在Python中，当对 `list` 类型进行 `in` 运算查找元素是否存在时，其查找通常采用二分法。（ ）

第 5 题 以下Python代码能以递归方式实现斐波那契数列，该数列第1、2项为1，以后各项均是前两项之和。（ ）

```
def Fibo(N):
    if N == 1 or N == 2:
        return 1
    else:
        m = fiboA(N - 1 )
        n = fiboB(N - 2)
        return m + n
```

第 6 题 贪心算法可以达到局部最优，但可能不是全局最优解。（ ）

第 7 题 如果自定已class已经定义了 `__lt__()` 魔术方法，则自动支持内置函数 `sorted()` 。（ ）

第 8 题 插入排序有时比快速排序时间复杂度更低。（ ）

第 9 题 下面的Python代码能实现十进制正整数N转换为八进制并输出。（ ）

```
N = int(input())
rst = "" #保存转换结果
while N != 0:
    rst += str(N % 8)
    N //= 8
print(rst)
```

第 10 题 Python代码 `print(sorted(list(range(10)), key = lambda x:x % 5))` 执行后将输出 `[0, 5, 1, 6, 2, 7, 3, 8, 4, 9]` 。（ ）

3　编程题（每题 25 分，共 50 分）

3.1　编程题 1

- **试题名称**：小杨的幸运数
- **时间限制**：5.0 s
- **内存限制**：128.0 MB

3.1.1　问题描述

小杨认为，所有大于等于 a 的完全平方数都是他的超级幸运数。

小杨还认为，所有超级幸运数的倍数都是他的幸运数。自然地，小杨的所有超级幸运数也都是幸运数。

对于一个非幸运数，小杨规定，可以将它一直 $+1$，直到它变成一个幸运数。我们把这个过程叫做幸运化。例如，如果 $a=4$，那么 4 是最小的幸运数，而 1 不是，但我们可以连续对 1 做 3 次 $+1$ 操作，使其变为 4，所以我们可以说，1 幸运化后的结果是 4。

现在，小样给出 N 个数，请你首先判断它们是不是幸运数；接着，对于非幸运数，请你将它们幸运化。

3.1.2　输入描述

第一行 2 个正整数 a, N。

接下来 N 行，每行一个正整数 x，表示需要判断（幸运化）的数。

3.1.3　输出描述

输出 N 行，对于每个给定的 x，如果它是幸运数，请输出 `lucky`，否则请输出将其幸运化后的结果。

3.1.4　特别提醒

在常规程序中，输入、输出时提供提示是好习惯。但在本场考试中，由于系统限定，请不要在输入、输出中附带任何提示信息。

3.1.5　样例输入 1

```
2 4
1
4
5
9
```

3.1.6　样例输出 1

```
4
lucky
8
lucky
```

3.1.7 样例解释 1

1 虽然是完全平方数，但它小于 a，因此它并不是超级幸运数，也不是幸运数。将其进行 3 次 $+1$ 操作后，最终得到幸运数 4。

4 是幸运数，因此直接输出 `lucky`。

5 不是幸运数，将其进行 3 次 $+1$ 操作后，最终得到幸运数 8。

9 是幸运数，因此直接输出 `lucky`。

3.1.8 样例输入 2

```
16 11
1
2
4
8
16
32
64
128
256
512
1024
```

3.1.9 样例输出 2

```
16
16
16
16
lucky
lucky
lucky
lucky
lucky
lucky
lucky
```

3.1.10 数据规模

对于 30% 的测试点，保证 $a, x \leq 100$，$N \leq 100$。

对于 60% 的测试点，保证 $a, x \leq 10^6$。

对于所有测试点，保证 $a \leq 1,000,001$；保证 $N \leq 2 \times 10^5$；保证 $1 \leq x \leq 1,000,001$。

3.1.11 参考程序

```python
a, n = input().split()
a, n = int(a), int(n)

max_lucky = 1001 ** 2 + 10

is_lucky = [False for i in range(max_lucky + 1)]
next_lucky = [-1 for i in range(max_lucky + 1)]
```

```python

for i in range(1, max_lucky):
    if i >= a and int(i ** 0.5) ** 2 == i:
        is_lucky[i] = True
        for j in range(i + i, max_lucky, i):
            is_lucky[j] = True

for i in range(max_lucky - 1, 0, -1):
    if is_lucky[i]:
        next_lucky[i] = i
    else:
        next_lucky[i] = next_lucky[i + 1]

for i in range(n):
    x = int(input())
    assert 1 <= x <= 10 ** 6 + 1
    if next_lucky[x] == x:
        print("lucky")
    else:
        print(next_lucky[x])
```

3.2 编程题 2

- **试题名称**：烹饪问题
- **时间限制**：1.0 s
- **内存限制**：128.0 MB

3.2.1 问题描述

有 N 种食材，编号从 0 至 $N-1$，其中第 i 种食材的美味度为 a_i。

不同食材之间的组合可能产生奇妙的化学反应。具体来说，如果两种食材的美味度分别为 x 和 y，那么它们的契合度为 x and y。

其中，and 运算为按位与运算，需要先将两个运算数转换为二进制，然后在高位补足 0，再逐位进行与运算。例如，12 与 6 的二进制表示分别为 `1100` 和 `0110`，将它们逐位进行与运算，得到 `0100`，转换为十进制得到 4，因此 $12 \text{ and } 6 = 4$。**在 C++ 或 Python 中，可以直接使用 `&` 运算符表示与运算。**

现在，请你找到契合度最高的两种食材，并输出它们的契合度。

3.2.2 输入描述

第一行一个整数 N，表示食材的种数。

接下来一行 N 个用空格隔开的整数，依次为 $a_0, \ldots, a_{N-1}$，表示各种食材的美味度。

3.2.3 输出描述

输出一行一个整数，表示最高的契合度。

3.2.4 特别提醒

在常规程序中，输入、输出时提供提示是好习惯。但在本场考试中，由于系统限定，请不要在输入、输出中附带任何提示信息。

3.2.5 样例输入 1

```
3
1 2 3
```

3.2.6 样例输出 1

```
2
```

3.2.7 样例解释 1

可以编号为 $1, 2$ 的食材之间的契合度为 2 and $3 = 2$，是所有食材两两之间最高的契合度。

3.2.8 样例输入 2

```
5
5 6 2 10 13
```

3.2.9 样例输出 2

```
8
```

3.2.10 样例解释 1

可以编号为 $3, 4$ 的食材之间的契合度为 10 and $13 = 8$，是所有食材两两之间最高的契合度。

3.2.11 数据规模

对于 40% 的测试点，保证 $N \leq 1,000$；

对于所有测试点，保证 $N \leq 10^6$，$0 \leq a_i \leq 2,147,483,647$。

3.2.12 参考程序

```python
n = int(input())
a = list(map(int, input().split()))
a = [0] + a


def sort_(l, r, k):
    while l <= r:
        while (l <= r) and (a[l] >> k & 1):
            l += 1
        while (l <= r) and (not (a[r] >> k & 1)):
            r -= 1
        if l <= r:
            a[l], a[r] = a[r], a[l]
            l += 1
            r -= 1
    return r


```

```
ans = 0
for i in range(31, -1, -1):
    j = sort_(1, n, i)
    if j >= 2:
        ans |= 1 << i
        n = j

print(ans)
```