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

Python 六级

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	C	B	A	B	B	C	A	B	B	A	C	B	B	B

第 1 题 在Python的面向对象编程中，下列关于“动态绑定（等效于虚函数）”的描述中，错误的是（ ）。

- **A.** 动态绑定用于支持运行时多态。
- **B.** 通过基类变量调用方法时，会根据对象实际类型决定调用版本。
- **C.** 构造函数（ `__init__` ）可以通过动态绑定实现多态以支持灵活初始化。
- **D.** 基类析构函数（ `__del__` ）常保证子类调用其实现，以避免资源泄漏析构函数常声明为虚函数以避免资源泄漏。

第 2 题 执行如下代码，将输出 `钢琴：叮咚叮咚` 和 `吉他：咚咚当当` 而不是两行 `乐器在演奏声音` ，这体现了面向对象编程的（ ）特性。

```python
class Instrument:
    """基类：乐器"""
    def play(self):
        print("乐器在演奏声音")

    def __del__(self):
        pass

class Piano(Instrument):
    """子类：钢琴"""
    def play(self):
        print("钢琴：叮咚叮咚")

class Guitar(Instrument):
    def play(self):
        print("吉他：咚咚当当")

if __name__ == "__main__":
    instruments = [Piano(), Guitar()]

    for inst in instruments:
        inst.play()
```

- **A.** 继承
- **B.** 封装
- **C.** 多态
- **D.** 链接

第 3 题 执行下面代码，将输出（ ）。

```python
class Instrument:
    def play(self):
        print("乐器在演奏声音")

    def __del__(self):
        pass

class Piano(Instrument):
    def play(self):
        print("钢琴：叮咚叮咚")

class Guitar(Instrument):
    def play(self):
        print("吉他：咚咚当当")

if __name__ == "__main__":
    instruments = [Piano(), Guitar()]
    for inst in instruments:
        Instrument.play(inst)
```

- [ ] **A.**

```
钢琴：叮咚叮咚
吉他：咚咚当当
```

- [ ] **B.**

```
乐器在演奏声音
乐器在演奏声音
```

- [ ] **C.** 编译错误
- [ ] **D.** 运行错误

第 4 题 某文本编辑器把用户输入的字符依次压入栈 S。用户依次输入 `A`，`B`，`C`，`D` 后，用户按了两次撤销（每次撤销，弹出栈顶一个字符）。此时栈从栈底到栈顶的内容是：（ ）。

- [ ] **A.** `A B`
- [ ] **B.** `A B C`
- [ ] **C.** `A B D`
- [ ] **D.** `B C`

第 5 题 假设循环队列数组长度为 `N`，其中队空判断条件为：`front == rear`，队满判断条件为：`(rear + 1) % N == front`，出队对应的操作为：`front = (front + 1) % N`，入队对于的操作为：`rear = (rear + 1) % N`。循环队列长度 `N = 6`，初始 `front = 1`，`rear = 1`，执行操作序列为：入队, 入队, 入队, 出队, 入队, 入队，则最终 `(front, rear)` 的值是（ ）。

- [ ] **A.** `(2, 5)`
- [ ] **B.** `(2, 0)`
- [ ] **C.** `(3, 5)`
- [ ] **D.** `(3, 0)`

第 6 题 以下函数 `check()` 用于判断一棵二叉树是否为（ ）。

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

from collections import deque

def check(root):
    if not root:
        return True

    q = deque()
    q.append(root)
    has_null = False

    while q:
        cur = q.popleft()

        if not cur:
            has_null = True
        else:
            if has_null:
                return False
            q.append(cur.left)
            q.append(cur.right)

    return True
```

- [ ] **A.** 满二叉树
- [ ] **B.** 完全二叉树
- [ ] **C.** 二叉搜索树
- [ ] **D.** 平衡二叉树

第 7 题 以下代码实现了二叉树的（ ）。

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def traverse(root):
    if not root:
        return
    traverse(root.left)
    traverse(root.right)
    print(root.val, end=" ")

if __name__ == "__main__":
    root = TreeNode(1)
    root.right = TreeNode(2)
    root.right.left = TreeNode(3)

    print("后序遍历结果: ", end="")
    traverse(root)
```

- [ ] **A.** 前序遍历
- [ ] **B.** 中序遍历

- [ ] **C.** 后序遍历
- [ ] **D.** 层序遍历

第 8 题 下面代码实现了哈夫曼编码，则横线处应填写的代码是（ ）。

```python
class Symbol:
    def __init__(self, ch='', freq=0, code=''):
        self.ch = ch
        self.freq = freq
        self.code = code

class Node:
    def __init__(self, w=0, l=-1, r=-1, sym=-1):
        self.w = w
        self.l = l
        self.r = r
        self.sym = sym

def pop_min_node(nodes, leaf_idx, n, pA, internal_idx, pB):

    if pA[0] < n and (pB[0] >= len(internal_idx) or nodes[leaf_idx[pA[0]]].w <= nodes[internal_idx[pB[0]]].w):
        res = leaf_idx[pA[0]]
        pA[0] += 1
        return res
    else:
        res = internal_idx[pB[0]]
        pB[0] += 1
        return res

def dfs_build_codes(u, nodes, sym_list, path):
    if u == -1:
        return
    if nodes[u].sym != -1:
        sym_list[nodes[u].sym].code = ''.join(path)
        return
    path.append('0')
    dfs_build_codes(nodes[u].l, nodes, sym_list, path)
    path.pop()
    path.append('1')
    dfs_build_codes(nodes[u].r, nodes, sym_list, path)
    path.pop()

def build_huffman_codes(sym_list):
    n = len(sym_list)
    for sym in sym_list:
        sym.code = ''
    if n <= 0:
        return -1
    if n == 1:
        sym_list[0].code = '0'
        return 0
    nodes = []
    leaf_idx = []
    for i in range(n):
        leaf_idx.append(len(nodes))
        nodes.append(Node(sym_list[i].freq, -1, -1, i))
    leaf_idx.sort(key=lambda x: (nodes[x].w, nodes[x].sym))
    internal_idx = []
    pA = [0]
    pB = [0]
    for k in range(1, n):
        x = pop_min_node(nodes, leaf_idx, n, pA, internal_idx, pB)
        y = pop_min_node(nodes, leaf_idx, n, pA, internal_idx, pB)
        z = len(nodes)

        ______________________________
    root = internal_idx[-1] if internal_idx else -1
    path = []
    dfs_build_codes(root, nodes, sym_list, path)
```

```
    return root

if __name__ == "__main__":
    syms = [
        Symbol('A', 5),
        Symbol('B', 9),
        Symbol('C', 12),
        Symbol('D', 13),
        Symbol('E', 16),
        Symbol('F', 45)
    ]
    root = build_huffman_codes(syms)
    print(f"哈夫曼树根节点下标：{root}")
    for sym in syms:
        print(f"字符 '{sym.ch}' (频率 {sym.freq})：编码 {sym.code}")
```

- [ ] **A.**

```
nodes.append(Node(nodes[x].w + nodes[y].w, x, y, -1))
internal_idx.append(z)
```

- [ ] **B.**

```
nodes.append(Node(nodes[x].w + nodes[y].w, x, y, 1))
internal_idx.append(z)
```

- [ ] **C.**

```
nodes.append(Node(nodes[x-1].w + nodes[y].w, x, y, 1))
internal_idx.append(z)
```

- [ ] **D.**

```
nodes.append(Node(nodes[x+1].w + nodes[y].w, x, y, 1))
internal_idx.append(z)
```

第 9 题 以下关于哈夫曼编码的说法，正确的是（ ）。

- [ ] **A.** 哈夫曼编码是定长编码
- [ ] **B.** 哈夫曼编码中，没有任何一个字符的编码是另一个字符编码的前缀
- [ ] **C.** 哈夫曼编码一定唯一
- [ ] **D.** 哈夫曼编码不能用于数据压缩

第 10 题 以下函数实现了二叉排序树（BST）的（ ）操作。

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def op(root, x):
    if not root:
        return TreeNode(x)

    if x < root.val:
        root.left = op(root.left, x)
    else:
        root.right = op(root.right, x)
    return root
```

- [ ] **A.** 查找
- [ ] **B.** 插入
- [ ] **C.** 删除
- [ ] **D.** 遍历

第 11 题 下列代码实现了树的深度优先遍历，则横线处应填入（ ）。

```
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

def dfs1(root):
    if not root:
        return
    temp = []
    temp.append(root)
    while temp:
        node = temp[-1]
        temp.pop()
        print(node.val, end=" ")
        if node.right:
            temp.append(node.right)
        if node.left:
            temp.append(node.left)

def dfs2(root):
    if not root:
        return
    st = []
    st.append(root)
    while st:
        node = st[-1]
        st.pop()
        print(node.val, end=" ")
        if node.right:
            st.append(node.right)
        ________________
```

- [ ] **A.**

```

     if node.left:
          st.append(node.left)
```

- [ ] **B.**

```

     if node.right:
          st.append(node.left)
```

- [ ] **C.**

```

     if node.left:
          st.append(node.right)
```

D.

```
     if node.right:
          st.append(node.right)
```

第 12 题 给定一棵普通二叉树（节点值没有大小规律），下面代码判断是否存在值为 `x` 的结点，则横线处应填入（ ）。

```
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

from collections import deque

def bfs_find(root, x):
    if not root:
        return None

    q = deque()
    q.append(root)

    while q:
        cur = q.popleft()
        if cur.val == x:
            return cur
        ______________________

    return None

```

A.

```
            q.append(cur.left)

```

B.

```
            q.append(cur.right)
```

C.

```
        if cur.left:
            q.append(cur.left)
        if cur.right:
            q.append(cur.right)
```

D.

```
        if cur.right:
            q.append(cur.left)
        if cur.left:
            q.append(cur.right)
```

第 13 题 在二叉排序树（Binary Search Tree, BST）中，假设节点值互不相同。给定如下搜索函数，以下说法 一定正确的是（ ）。

```python
class Node:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def find(root, x):
    while root:
        if root.val == x:
            return True
        if x < root.val:
            root = root.left
        else:
            root = root.right
    return False
```

- [ ] **A.** 最坏情况下，访问结点数是 $O(\log n)$
- [ ] **B.** 最坏情况下，访问结点数是 $O(n)$
- [ ] **C.** 无论如何，访问结点数都不超过树高的一半
- [ ] **D.** 一定比在普通二叉树中搜索快

第 14 题 0/1 背包（每件物品最多选一次）问题通常可用一维动态规划求解，核心代码如下。遍历的方向无所谓，则下面说法正确的是（ ）。

```python
def zero_one_knapsack(items, W):
    dp = [0] * (W + 1)
    for w, v in items:
        for j in range(W, w - 1, -1):
            dp[j] = max(dp[j], dp[j - w] + v)

    return dp[W]
```

- [ ] **A.** 内层 `j` 必须从小到大，否则会漏解
- [ ] **B.** 内层 `j` 必须从大到小，否则同一件物品会被用多次
- [ ] **C.** `j` 从大到小或从小到大都一样
- [ ] **D.** 只要 `dp` 初始为 `0`，方向无所谓

第 15 题 以下关于动态规划的说法中，错误的是（ ）。

- [ ] **A.** 动态规划方法通常能够列出递推公式。
- [ ] **B.** 动态规划方法的时间复杂度通常为状态的个数。
- [ ] **C.** 动态规划方法有递推和递归两种实现形式。
- [ ] **D.** 在使用动态规划思想（即避免重复子问题）的前提下，递推实现与递归实现（记忆化搜索）的时间复杂度通常是相当的。

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 以下代码中，构造函数被调用的次数是1次。

```python
class Test:
    init_count = 0

    def __init__(self):
        Test.init_count += 1
        print("T ", end="")

    def __copy__(self):
        print(" (拷贝构造，不触发__init__) ", end="")
        new_obj = Test.__new__(Test)
        return new_obj

if __name__ == "__main__":
    a = Test()
    import copy
    b = copy.copy(a)

```

第 2 题 面向对象编程中，封装是指将数据和操作数据的方法绑定在一起，并对外隐藏实现细节。

第 3 题 以下代码能够正确统计二叉树中叶子结点的数量。

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def count_leaf(root):
    if not root:
        return 0
    if not root.left and not root.right:
        return 1
    return count_leaf(root.left) + count_leaf(root.right)

if __name__ == "__main__":
    root1 = TreeNode(1)
    root1.left = TreeNode(2)
    root1.right = TreeNode(3)
    root1.left.left = TreeNode(4)
    root1.left.right = TreeNode(5)
    root1.right.right = TreeNode(6)
    print(f"二叉树1的叶子节点数: {count_leaf(root1)}")
    root2 = TreeNode(1)
    print(f"二叉树2的叶子节点数: {count_leaf(root2)}") 1

    root3 = None
    print(f"空树的叶子节点数: {count_leaf(root3)}")
```

第 4 题 广度优先遍历二叉树可用栈来实现。

第 5 题 函数调用管理可用栈来管理。

第 6 题 在二叉排序树（BST）中，若某结点的左子树为空，则该结点一定是整棵树中的最小值结点。

第 7 题 下面的函数能正确判断一棵树是不是二叉排序树（左边的数字要比当前数字小，右边的数字要比当前数字大）。

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def is_bst(root, min_val=float('-inf'), max_val=float('inf')):
    if not root:
        return True
    if root.val <= min_val or root.val >= max_val:
        return False
    return is_bst(root.left, min_val, root.val) and is_bst(root.right, root.val, max_val)

```

第 8 题 格雷编码相邻两个编码之间必须有多位不同，以避免数据传输错误。

第 9 题 小杨在玩一个闯关游戏，从第 1 关走到第 4 关。每一关的体力消耗如下（下标表示关卡编号）：`cost = [ 0, 3, 5, 2, 4 ]`，其中 `cost[i]` 表示到达第 `i` 关需要消耗的体力，`cost[0]=0` 表示在开始状态，体力消耗为 0。小杨每次可以从当前关卡 前进 1 步或 2 步。按照上述规则，从第 1 关到第 4 关所需消耗的最小体力为 7。

第 10 题 假定只有一个根节点的树的深度为1，则一棵有 n 个节点的完全二叉树，则树的深度为 $\lfloor \log_2(n) \rfloor + 1$。

3 编程题（每题 25 分，共 50 分）

3.1 编程题 1

- **试题名称**：路径覆盖
- **时间限制**：3.0 s
- **内存限制**：512.0 MB

3.1.1 题目描述

给定一棵有 n 个结点的有根树 T，结点依次以 $1, 2, \ldots, n$ 编号，根结点编号为 1。方便起见，编号为 i 的结点称为结点 i。

初始时 T 中的结点均为白色。你需要将 T 中的若干个结点染为黑色，使得所有叶子到根的路径上至少有一个黑色结点。将结点 i 染为黑色需要代价 c_i，你需要在满足以上条件的情况下，最小化染色代价之和。

叶子是指 T 中没有子结点的结点。

3.1.2 输入格式

第一行，一个正整数 n，表示结点数量。

第二行，$n-1$ 个正整数 $f_2, f_3, \ldots, f_n$，其中 f_i 表示结点 i 的父结点的编号，保证 $f_i < i$。

第三行，n 个正整数 $c_1, c_2, \ldots, c_n$，其中 c_i 表示将结点 i 染为黑色所需的代价。

3.1.3 输出格式

一行，一个整数，表示在满足所有叶子到根的路径上至少有一个黑色结点的前提下，染色代价之和的最小值。

3.1.4 样例

3.1.4.1 输入样例 1

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

3.1.4.2 输出样例 1

```
2
```

3.1.4.3 输入样例 2

```
7
1 1 2 2 3 3
64 16 15 4 3 2 1
```

3.1.4.4 输出样例 2

```
10
```

3.1.5 数据范围

对于 40 的测试点，保证 $2 \leq n \leq 16$。

对于另外 20 的测试点，保证 $f_i = i - 1$。

对于所有测试点，保证 $2 \leq n \leq 10^5$，$1 \leq c_i \leq 10^9$。

3.1.6 参考程序

```python
getint = lambda: map(int, input().split())
getints = lambda: list(getint())
# 获取节点数
n = int(input())
# 各个节点的根节点
f = [0, 0] + getints()
# 各个节点涂黑的成本
c = [0] + getints()
# cnt[i]表示节点i有多少子节点
cnt = [0] * (n + 1)
# ans[i]表示以节点i为根节点的子树，需要多少成本才能让所有叶子结点到根节点都有涂黑的节点
ans = [0] * (n + 1)

# 统计一个节点有多少子节点
for i in range(2, n + 1):
    cnt[f[i]] += 1
# 遍历每一个节点，因为一个节点的父节点编号肯定这个节点，所以大编号的一定更深
for i in range(n, 0, -1):
    # 如果一个节点没有子节点，那么显然ans[i]=c[i]就是把自己涂黑
    if cnt[i] == 0:
        ans[i] = c[i]
    # 一个节点当前已经找到的答案等于
    ans[i] = min(ans[i], c[i])
    ans[f[i]] += ans[i]
print(ans[1])
```

3.2 编程题 2

- **试题名称**：道具商店
- **时间限制**：30.0 s
- **内存限制**：512.0 MB

3.2.1 题目描述

道具商店里有 n 件道具可供挑选。第 i 件道具可为玩家提升 a_i 点攻击力，需要 c_i 枚金币才能购买，每件道具只能购买一次。现在你有 k 枚金币，请问你最多可以提升多少点攻击力?

3.2.2 输入格式

第一行，两个正整数 n, k，表示道具数量以及你所拥有的金币数量。

接下来 n 行，每行两个正整数 a_i, c_i，表示道具所提升的攻击力点数，以及购买所需的金币数量。

3.2.3 输出格式

输出一行，一个整数，表示最多可以提升的攻击力点数。

3.2.4 样例

3.2.4.1 输入样例 1

```
3 5
99 1
33 2
11 3
```

3.2.4.2 输出样例 1

```
132
```

3.2.4.3 输入样例 2

```
4 100
10 1
20 11
40 33
100 99
```

3.2.4.4 输出样例 2

```
110
```

3.2.5 数据范围

对于 60 的测试点，保证 $1 \leq k \leq 500$，$1 \leq c_i \leq 500$。

对于所有测试点，保证 $1 \leq n \leq 500$，$1 \leq k \leq 10^9$，$1 \leq a_i \leq 500$，$1 \leq c_i \leq 10^9$。

3.2.6 参考程序

```python
getint = lambda: map(int, input().split())
getints = lambda: list(getint())
# 得到物品数量和预算
n, k = getint()
# 这里进行了滚动优化，dp[j]表示攻击力为j的情况下的最小预算
# 最大攻击力为500*500，所以初始化这么大的数组，并且所有值都是1e10表示无穷大预算
f = [0] + [int(1e10)] * 500 * 500
# 能够达到的最大攻击力，用来缩小遍历的范围
s = 0
for i in range(n):
    # 输入当前物品的攻击力和价格
    a, c = getint()
    s += a
    # 计算题解中的转移公式，因为j是从高到低遍历的，f[j-a]和f[j]此时尚未被更新，相当于是f[i-1][j-a]与f[i-1][j]
    for j in range(s, a - 1, -1):
        f[j] = min(f[j], f[j - a] + c)

# 遍历f[i]得到符合预算条件的最高攻击力
ans = 0
for i in range(s + 1):
    if f[i] <= k:
        ans = i
print(ans)
```