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

C++ 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	C	C	D	A	C	D	D	B	C	D	A	D	A	D	A

第 1 题 与数组相比，链表在（ ）操作上通常具有更高的效率。

- ☐ **A.** 随机访问元素
- ☐ **B.** 查找指定元素
- ☐ **C.** 在已知位置插入或删除节点
- ☐ **D.** 遍历所有元素

第 2 题 下面C++代码实现双向链表。函数 `is_empty()` 判断链表是否为空，如链表为空返回 `true`，否则返回 `false`。横线处**不能**填写（ ）。

```
// 节点结构体
struct Node {
    int data;
    Node* prev;
    Node* next;
};

// 双向链表结构体
struct DoubleLink {
    Node* head;
    Node* tail;
    int size;

    DoubleLink() {
        head = nullptr;
        tail = nullptr;
        size = 0;
    }

    ~DoubleLink() {
        Node* curr = head;
        while (curr) {
            Node* next = curr->next;
            delete curr;
            curr = next;
        }
    }

    // 判断链表是否为空
    bool is_empty() const {
        ____________________
    }
```

```
};
```

- [ ] **A.** `return head == nullptr;`
- [ ] **B.** `return tail == nullptr;`
- [ ] **C.** `return head.data == 0;`
- [ ] **D.** `return size == 0;`

第 3 题 基于上题代码正确的前提下，填入相应代码完善 `append()`，用于在双向链表尾部增加新节点，横线上应填写（ ）。

```
void append(int data) {
    Node* newNode = new Node{data, nullptr, nullptr};

    if (is_empty()) {
        head = tail = newNode;
    } else {
        ____________________
    }
    ++size;
}
```

- [ ] **A.**

```
tail->next = newNode;
```

- [ ] **B.**

```
newNode->prev = tail;
tail = newNode;
```

- [ ] **C.**

```
tail = newNode;
newNode->prev = tail;
tail->next = newNode;
```

- [ ] **D.**

```
tail->next = newNode;
newNode->prev = tail;
tail = newNode;
```

第 4 题 下列C++代码用循环链表解决约瑟夫问题，即假设 `n` 个人围成一圈，从第一个人开始数，每次数到第 `k` 个的人就出圈，输出最后留下的那个人的编号。横线上应填写（ ）。

```
struct Node {
    int data;
    Node* next;
};

Node* createCircularList(int n) {
    Node* head = new Node{1, nullptr};
    Node* prev = head;
    for (int i = 2; i <= n; ++i) {
        Node* node = new Node{i, nullptr};
        prev->next = node;
```

```
        prev = node;
    }
    prev->next = head;
    return head;
}

int fingLastSurvival(int n, int k) {
    Node* head = createCircularList(n);
    Node* p = head;
    Node* prev = nullptr;

    while (p->next != p) {
        for (int count = 1; count < k; ++count) {
            prev = p;
            p = p->next;
        }
        ____________________
    }

    cout << "最后留下的人编号是: " << p->data << endl;
    delete p;

    return 0;
}
```

A.

```
prev->next = p->next;
delete p;
p = prev->next;
```

B.

```
delete p;
prev->next = p->next;
p = prev->next;
```

C.

```
delete p;
p = prev->next;
prev->next = p->next;
```

D.

```
prev->next = p->next;
p = prev->next;
delete p;
```

第 5 题 下列C++代码判断一个正整数是否是质数，说法正确的是( )。

```
bool is_prime(int n) {
    if (n <= 1)
        return false;
    if (n == 2 || n == 3 || n == 5)
        return true;
    if (n % 2 == 0 || n % 3 == 0 || n % 5 == 0)
        return false;

    int i = 7;
    int step = 4;
    int finish_number = sqrt(n) + 1;
```

```
    while (i <= finish_number) {
        if (n % i == 0)
            return false;
        i += step;
        step = 6 - step;
    }
    return true;
}
```

- [ ] **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，其执行效果和执行时间相同。

```
for (int i = 2; i < finish_number; i++) {
    if (n % i == 0)
        return false;
}
return true;
```

第 6 题 下列C++代码用两种方式求解两个正整数的最大公约数，说法错误的是( )。

```
int gcd0(int big, int small) {
    if (big < small) {
        swap(big, small);
    }
    if (big % small == 0) {
        return small;
    }
    return gcd0(small, big % small);
}

int gcd1(int big, int small) {
    if (big < small) {
        swap(big, small);
    }
    for (int i = small; i >= 1; --i) {
        if (big % i == 0 && small % i == 0)
            return i;
    }
    return 1;
}
```

- [ ] **A.** `gcd0()` 函数的时间复杂度为$O(\log n)$
- [ ] **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 `for (int i = small; i >= 1; --i)` 应该修改为 `for (int i = small; i > 1; --i)`

第 7 题 下面的代码用于判断整数n是否是质数，错误的说法是（ ）。

```cpp
bool is_prime(int n) {
    if (n <= 1) return false;

    int finish_number = static_cast<int>(sqrt(n)) + 1;
    for (int i = 2; i < finish_number; ++i) {
        if (n % i == 0)
            return false;
    }
    return true;
}
```

- [ ] **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 8 题 唯一分解定理描述了关于正整数的什么性质?

- [ ] **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 9 题 下面的C++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（ ）。

```cpp
int find_max_recursive(const vector<int>& nums, int left, int right) {
    if (left == right)
        return nums[left];

    int mid = left + (right - left) / 2;
    int left_max = find_max_recursive(nums, left, mid);
    int right_max = find_max_recursive(nums, mid + 1, right);

    return max(left_max, right_max);
}

int find_max(const vector<int>& nums) {
    if (nums.empty()) {
        throw invalid_argument("输入数组不能为空");
    }
    return find_max_recursive(nums, 0, nums.size() - 1);
}
```

- [ ] **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 10 题 下面的C++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（ ）。

```cpp
int find_max(const vector<int>& nums) {
    if (nums.empty()) {
        throw invalid_argument("输入数组不能为空");
    }

    int max_value = nums[0];
    for (int num : nums) {
        if (num > max_value) {
            max_value = num;
        }
    }
    return max_value;
}
```

- [ ] **A.** 本题 `find_max()` 函数采用的是迭代算法
- [ ] **B.** 本题 `find_max()` 函数的时间复杂度为$O(n)$
- [ ] **C.** 和上一题的 `find_max()` 相比，因为没有递归，所以没有栈的创建和销毁开销
- [ ] **D.** 本题 `find_max()` 函数和上一题的 `find_max()` 空间复杂度相同

第 11 题 下面的 C++ 代码用于在升序数组 `lst` 中查找目标值 `target` 最后一次出现的位置。相关说法，正确的是（ ）。

```cpp
int binary_search_last_occurrence(const vector<int>& lst, int target) {
    if (lst.empty()) return -1;

    int low = 0, high = lst.size() - 1;

    while (low < high) {
        int mid = (low + high + 1) / 2;
        if (lst[mid] <= target) {
            low = mid;
        } else {
            high = mid - 1;
        }
    }

    if (lst[low] == target)
        return low;
    else
        return -1;
}
```

- [ ] **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 12 题 有关下面C++代码的说法，错误的是（ ）。

```cpp
double sqrt_binary(long long n, double epsilon = 1e-10) {
    if (n < 0) {
        throw invalid_argument("输入必须为非负整数");
    }

    if (n == 0 || n == 1) return n;

```

```
    // 阶段 1
    long long low = 1, high = n;
    long long k = 0;

    while (low <= high) {
        long long mid = (low + high) / 2;
        long long mid_sq = mid * mid;

        if (mid_sq == n) {
            return mid;
        } else if (mid_sq < n) {
            k = mid;
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    long long next_k = k + 1;
    if (next_k * next_k == n) {
        return next_k;
    }

    // 阶段 2
    double low_d = (double)k;
    double high_d = (double)(k + 1);
    double mid;

    while (high_d - low_d >= epsilon) {
        mid = (low_d + high_d) / 2;
        double mid_sq = mid * mid;

        if (mid_sq < n) {
            low_d = mid;
        } else {
            high_d = mid;
        }
    }

    double result = (low_d + high_d) / 2;
    long long check_int = (long long)(result + 0.5);
    if (check_int * check_int == n) {
        return check_int;
    }

    return result;
}
```

- [ ] **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 = (long long)(result + 0.5)` 是检查因浮点误差是否为正完全平方根
- [ ] **D.** 阶段2的二分法中 `high_d - low_d >= epsilon` 不能用于浮点数比较，会进入死循环

第 13 题 13.硬币找零问题中要求找给客户最少的硬币。 `coins` 存储可用硬币规格，单位为角，假设规格都小于10角，且一定有1角规格。 `amount` 为要找零的金额，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（ ）。

```
const int MAX_COINS = 10;
int result[MAX_COINS] = {0};  // 假设最多10种面额

```

```cpp
int find_coins(const vector<int>& coins, int amount) {
    sort(coins.begin(), coins.end(), greater<int>());

    int n = coins.size();

    for (int i = 0; i < n; ++i) {
        int coin = coins[i];
        int num = amount / coin;
        result[i] = num;
        amount -= num * coin;
        if (amount == 0) break;
    }

    cout << "找零方案如下: " << endl;
    for (int i = 0; i < n; ++i) {
        cout << sorted_coins[i] << "角需要" << result[i] << "枚" << endl;
    }

    return 0;
}
```

- [ ] **A.** 上述代码采用贪心算法实现
- [ ] **B.** 针对本题具体要求，上述代码总能找到最优解
- [ ] **C.** 上述代码采用枚举算法
- [ ] **D.** 上述代码采用分治算法

第 14 题 关于下述C++代码的快速排序算法，说法错误的是（ ）。

```cpp
int randomPartition(std::vector<int>& arr, int low, int high) {
    int random = low + rand() % (high - low + 1);
    std::swap(arr[random], arr[high]);

    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = randomPartition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

- [ ] **A.** 在 `randomPartition` 函数中，变量 `i` 的作用是记录大于基准值的元素的边界
- [ ] **B.** `randomPartition` 函数随机选择基准值，可以避免输入数据特定模式导致的最坏情况下时间复杂度$O(n^2)$
- [ ] **C.** 快速排序平均时间复杂度是$O(n \log n)$

☐ **D.** 快速排序是稳定排序算法

第 15 题 小杨编写了一个如下的高精度除法函数，则横线上应填写的代码为（ ）。

```
const int MAXN = 1005; // 最大位数
struct BigInt {
    int d[MAXN]; // 存储数字，d[0]是个位，d[1]是十位，...
    int len;     // 数字长度

    BigInt() {
        memset(d, 0, sizeof(d));
        len = 0;
    }
};

// 比较两个高精度数的大小
int compare(BigInt a, BigInt b) {
    if(a.len != b.len) return a.len > b.len ? 1 : -1;
    for(int i = a.len - 1; i >= 0; i--) {
        if(a.d[i] != b.d[i]) return a.d[i] > b.d[i] ? 1 : -1;
    }
    return 0;
}

// 高精度减法
BigInt sub(BigInt a, BigInt b) {
    BigInt c;
    for(int i = 0; i < a.len; i++) {
        c.d[i] += a.d[i] - b.d[i];
        if(c.d[i] < 0) {
            c.d[i] += 10;
            c.d[i+1]--;
        }
    }
    c.len = a.len;
    while(c.len > 1 && c.d[c.len-1] == 0) c.len--;
    return c;
}

// 高精度除法（a/b，返回商和余数）
pair<BigInt, BigInt> div(BigInt a, BigInt b) {
    BigInt q, r; // q是商，r是余数

    if(compare(a, b) < 0) { // 如果a<b，商为0，余数为a
        q.len = 1;
        q.d[0] = 0;
        r = a;
        return make_pair(q, r);
    }

    // 初始化余数r为a的前b.len位
    r.len = b.len;
    for(int i = a.len - 1; i >= a.len - b.len; i--) {
        r.d[i - (a.len - b.len)] = a.d[i];
    }

    // 逐位计算商
    for(int i = a.len - b.len; i >= 0; i--) {
        // 把下一位加入余数
        if(r.len > 1 || r.d[0] != 0) {
            for(int j = r.len; j > 0; j--) {
                r.d[j] = r.d[j-1];
            }
            ____________________
        } else {
```

```
            r.d[0] = a.d[i];
            r.len = 1;
        }

        // 计算当前位的商
        while(compare(r, b) >= 0) {
            r = sub(r, b);
            q.d[i]++;
        }
    }

    // 确定商的长度
    q.len = a.len - b.len + 1;
    while(q.len > 1 && q.d[q.len-1] == 0) q.len--;

    // 处理余数前导零
    while(r.len > 1 && r.d[r.len-1] == 0) r.len--;

    return make_pair(q, r);
}
```

☐ **A.**

```
r.d[0] = a.d[i];
r.len++;
```

☐ **B.**

```
r.d[i] = a.d[i];
r.len++;
```

☐ **C.**

```
r.d[i] = a.d[i];
r.len = 1;
```

☐ **D.**

```
r.d[0] = a.d[i];
r.len = 1;
```

2　判断题（每题 2 分，共 20 分）

第 1 题 下面C++代码是用欧几里得算法（辗转相除法）求两个正整数的最大公约数， `a` 大于 `b` 还是小于 `b` 都适用。

```
int gcd(int a, int b) {
    while (b) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    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。

```cpp
int lcm(int a, int b) {
    return a * b / gcd(a, b);
}
```

第 3 题 下面的C++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]}`。

```cpp
int main() {
    int n, m;
    cin >> n >> m;
    if (n > m) swap(n, m);

    map<int, vector<int>> prime_factor;

    for (int i = n; i <= m; ++i) {
        int j = 2, k = i;
        while (k != 1) {
            if (k % j == 0) {
                prime_factor[i] = prime_factor[i] + j;
                k /= j;
            } else {
                ++j;
            }
        }
    }

    for (auto& p : prime_factor) {
        cout << p.first << ": ";
        for (int v : p.second)
            cout << v << " ";
        cout << endl;
    }

    return 0;
}
```

第 4 题 下面的C++代码实现归并排序。代码在执行时，将输出一次 HERE 字符串，因为merge()函数仅被调用一次。

```cpp
void merge(std::vector<int>& arr, int left, int mid, int right) {
    std::vector<int> temp(right - left + 1);

    int i = left;
    int j = mid + 1;
    int k = 0;

    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }

    while (i <= mid) {
        temp[k++] = arr[i++];
    }

    while (j <= right) {
        temp[k++] = arr[j++];
    }

    for (int p = 0; p < k; ++p) {
        arr[left + p] = temp[p];
```

```
    }
}

void mergeSort(std::vector<int>& arr, int left, int right) {
    if (left >= right) {
        return;
    }

    int mid = left + (right - left) / 2;

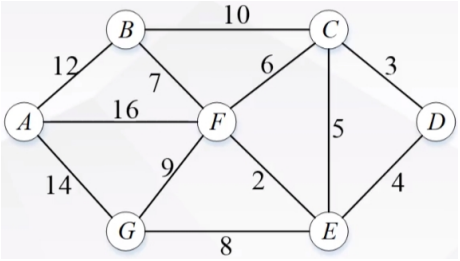
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);

    std::cout << "HERE";
    merge(arr, left, mid, right);
}
```

第 5 题 归并排序的最好、最坏和平均时间复杂度均为$O(n \log n)$。

第 6 题 查字典这个小学生必备技能，可以把字典视为一个已排序的数组。假设小杨要查找一个音首字母为 `g` 的单词，他首先翻到字典约一半的页数，发现该页的首字母是 `m`，由于字母表中 `g` 位于 `m` 之前，所以排除字典后半部分，查找范围缩小到前半部分；不断重复上述步骤，直至找到首字母为 `g` 的页码。这种查字典的一系列操作可看作二分查找。

第 7 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 8 题 分治算法将原问题可以分解成规模更小的子问题，使得求解问题的难度降低。但由于分治算法需要将问题进行分解，并且需要将多个子问题的解合并为原问题的解，所以分治算法的效率通常比直接求解原问题的效率低。

第 9 题 函数 `puzzle` 定义如下，则调用 `puzzle(7)` 程序会无限递归。

```
int puzzle(int n) {
    if (n == 1) return 1;
    if (n % 2 == 0) return puzzle(n / 2);
    return puzzle(3 * n + 1);
}
```

第 10 题 如下为线性筛法，用于高效生成素数表，其核心思想是每个合数只被它的最小质因数筛掉一次，时间复杂度为$O(n)$。

```
vector<int> linearSieve(int n) {
    vector<bool> is_prime(n + 1, true);
    vector<int> primes;

    for (int i = 2; i <= n; ++i) {
        if (is_prime[i]) {
            primes.push_back(i);
        }

        for (int j = 0; j < primes.size() && i * primes[j] <= n; ++j) {
```

```
            is_prime[i * primes[j]] = false;
            if (i % primes[j] == 0) {
                break;
            }
        }
    }
    return primes;
}
```

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**：1.0 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 $1 \leq a, b \leq 10^4$，$1 \leq n, m \leq 10^9$。

3.2 编程题 2

- **试题名称**：最大公因数
- **时间限制**：1.0 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 $1 \leq n \leq 10^5$，$1 \leq q \leq 10^5$，$1 \leq a_i \leq 1000$。