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

C++ 五级

2025 年 09 月

1 单选题（每题 2 分，共 30 分）

第 1 题 以下哪种情况使用链表比数组更合适？

- A. 数据量固定且读多写少
- B. 需要频繁在中间或开头插入、删除元素
- C. 需要高效随机访问元素
- D. 存储空间必须连续

第 2 题 函数 `removeElements` 删除单链表中所有结点值等于 `val` 的结点，并返回新的头结点，其中链表头结点为 `head`，则横线处填写（ ）。

```
// 结点结构体
struct Node {
    int val;
    Node* next;

    Node() : val(0), next(nullptr) {}
    Node(int x) : val(x), next(nullptr) {}
    Node(int x, Node *next) : val(x), next(next) {}
};

Node* removeElements(Node* head, int val) {
    Node dummy(0, head);        // 哑结点，统一处理头结点
    Node* cur = &dummy;
    while (cur->next) {
        if (cur->next->val == val) {
            ______________________        // 在此填入代码

        }
        else {
            cur = cur->next;
        }
    }
    return dummy.next;
}
```

- A.

```
Node* del = cur;
cur = del->next;
delete del;
```

- B.

```
Node* del = cur->next;
cur->next = del;
delete del;
```

- [ ] **C.**

```
Node* del = cur->next;
cur->next = del->next;
delete del;
```

- [ ] **D.**

```
Node* del = cur->next;
delete del;
cur->next = del->next;
```

第 3 题 函数 `hasCycle` 采用Floyd快慢指针法判断一个单链表中是否存在环，链表的头节点为 `head` ，即用两个指针在链表上前进： `slow` 每次走 1 步， `fast` 每次走 2 步，若存在环， `fast` 终会追上 `slow` （相遇）；若无环， `fast` 会先到达 nullptr，则横线上应填写（ ）。

```
struct Node {
    int val;
    Node *next;
    Node(int x) : val(x), next(nullptr) {}
};

bool hasCycle(Node *head) {
    if (!head || !head->next)
        return false;
    Node* slow = head;
    Node* fast = head->next;
    while (fast && fast->next) {
        if (slow == fast) return true;
        ______________________        // 在此填入代码
    }
    return false;
}
```

- [ ] **A.**

```
slow = slow->next;
fast = fast->next->next;
```

- [ ] **B.**

```
slow = fast->next;
fast = slow->next->next;
```

- [ ] **C.**

```
slow = slow->next;
fast = slow->next->next;
```

- [ ] **D.**

```
slow = fast->next;
fast = fast->next->next;
```

第 4 题 函数 `isPerfectNumber` 判断一个正整数是否为完全数（该数是否即等于它的真因子之和），则横线上应填写（ ）。一个正整数 `n` 的真因子包括所有小于 `n` 的正因子，如28的真因子为1, 2, 4, 7, 14。

```cpp
bool isPerfectNumber(int n) {
    if(n <= 1) return false;
    int sum = 1;
    for(int i = 2; ______; i++) {
        if(n % i == 0) {
            sum += i;
            if(i != n/i) sum += n/i;
        }
    }
    return sum == n;
}
```

- [ ] **A.** `i <= n`
- [ ] **B.** `i*i <= n`
- [ ] **C.** `i <= n/2`
- [ ] **D.** `i < n`

第 5 题 以下代码计算两个正整数的最大公约数(GCD)，横线上应填写（ ）。

```cpp
int gcd0(int a, int b) {
    if (a < b) {
        swap(a, b);
    }
    while(b != 0) {
        int temp = a % b;
        a = b;
        b = temp;
    }
    return ______;
}
```

- [ ] **A.** `b`
- [ ] **B.** `a`
- [ ] **C.** `temp`
- [ ] **D.** `a * b`

第 6 题 函数 `sieve` 实现埃拉托斯特尼筛法(埃氏筛)，横线处应填入（ ）。

```cpp
vector<bool> sieve(int n) {
    vector<bool> is_prime(n+1, true);
    is_prime[0] = is_prime[1] = false;
    for(int i = 2; i <= n; i++) {
        if(is_prime[i]) {
            for(int j = ______; j <= n; j += i) {
                is_prime[j] = false;
            }
        }
    }
    return is_prime;
}
```

- [ ] **A.** `i`
- [ ] **B.** `i+1`
- [ ] **C.** `i*2`
- [ ] **D.** `i*i`

第 7 题 函数 `linearSieve` 实现线性筛法(欧拉筛)，横线处应填入（ ）。

```cpp
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 p : primes) {
            if(p * i > n) break;
            is_prime[p * i] = false;
            if(________) break;
        }
    }
    return primes;
}
```

- [ ] **A.** `i % p == 0`
- [ ] **B.** `p % i == 0`
- [ ] **C.** `i == p`
- [ ] **D.** `i * p == n`

第 8 题 关于 埃氏筛 和 线性筛 的比较，下列说法错误的是（ ）。

- [ ] **A.** 埃氏筛可能会对同一个合数进行多次标记
- [ ] **B.** 线性筛的理论时间复杂度更优，所以线性筛的速度往往优于埃氏筛
- [ ] **C.** 线性筛保证每个合数只被其最小质因子筛到一次
- [ ] **D.** 对于常见范围（$n \leq 10^7$），埃氏筛因实现简单，常数较小，其速度往往优于线性筛

第 9 题 唯一分解定理描述的是( )。

- [ ] **A.** 每个整数都能表示为任意素数的乘积
- [ ] **B.** 每个大于 1 的整数能唯一分解为素数幂乘积（忽略顺序）
- [ ] **C.** 合数不能分解为素数乘积
- [ ] **D.** 素数只有两个因子：1 和自身

第 10 题 给定一个 `n x n` 的矩阵 `matrix`，矩阵的每一行和每一列都按升序排列。函数 `countLE` 返回矩阵中第 `k` 小的元素，则两处横线上应分别填写（ ）。

```cpp
// 统计矩阵中 <= x 的元素个数：从左下角开始
int countLE(const vector<vector<int>>& matrix, int x) {
    int n = (int)matrix.size();
    int i = n - 1, j = 0, cnt = 0;
    while (i >= 0 && j < n) {
        if (matrix[i][j] <= x) {
            cnt += i + 1;
            ++j;
        }
        else {
            --i;
        }
    }
    return cnt;
}

int kthSmallest(vector<vector<int>>& matrix, int k) {
    int n = (int)matrix.size();
    int lo = matrix[0][0];
    int hi = matrix[n - 1][n - 1];

```

```
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        if (countLE(matrix, mid) >= k) {
            _________________    // 在此处填入代码
        } else {
            _________________    // 在此处填入代码
        }
    }
    return lo;
}
```

- [ ] **A.**

```
hi = mid - 1;
lo = mid + 1;
```

- [ ] **B.**

```
hi = mid;
lo = mid;
```

- [ ] **C.**

```
hi = mid;
lo = mid + 1;
```

- [ ] **D.**

```
hi = mid + 1;
lo = mid;
```

第 11 题 下述C++代码实现了快速排序算法，下面说法错误的是（ ）。

```
int partition(vector<int>& arr, int low, int high) {
    int i = low, j = high;
    int pivot = arr[low];                // 以首元素为基准
    while (i < j) {
        while (i < j && arr[j] >= pivot) j--;   //从右往左查找
        while (i < j && arr[i] <= pivot) i++;   //从左往右查找
        if (i < j) swap(arr[i], arr[j]);
    }
    swap(arr[i], arr[low]);
    return i;
}

void quickSort(vector<int>& arr, int low, int high) {
    if (low >= high) return;
    int p = partition(arr, low, high);
    quickSort(arr, low, p - 1);
    quickSort(arr, p + 1, high);
}
```

- [ ] **A.** 快速排序之所以叫“快速”，是因为它在平均情况下运行速度较快，常数小、就地排序，实践中通常比归并排序更高效。
- [ ] **B.** 在平均情况下，划分的递归层数为 $\log n$，每层中的总循环数为n ，总时间为 $O(n \log n)$。
- [ ] **C.** 在最差情况下，每轮划分操作都将长度为n的数组划分为长度为 0 和 $n-1$ 的两个子数组，此时递归层数达到 n ，每层中的循环数为n，总时间为$O\left(n^2\right)$。
- [ ] **D.** 划分函数 `partition` 中“从右往左查找”与“从左往右查找”的顺序可以交换。

第 12 题 下述C++代码实现了归并排序算法，则横线上应填写（ ）。

```
void merge(vector<int> &nums, int left, int mid, int right) {
```

```
    // 左子数组区间为 [left, mid]，右子数组区间为 [mid+1, right]
    vector<int> tmp(right - left + 1);
    int i = left, j = mid + 1, k = 0;
    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j])
            tmp[k++] = nums[i++];
        else
            tmp[k++] = nums[j++];
    }
    while (i <= mid) {
        tmp[k++] = nums[i++];
    }
    while (________) {    // 在此处填入代码
        tmp[k++] = nums[j++];
    }
    for (k = 0; k < tmp.size(); k++) {
        nums[left + k] = tmp[k];
    }
}

void mergeSort(vector<int> &nums, int left, int right) {
    if (left >= right)
        return;

    int mid = (left + right) / 2;
    mergeSort(nums, left, mid);
    mergeSort(nums, mid + 1, right);
    merge(nums, left, mid, right);
}
```

- ☐ **A.** `i < mid`
- ☐ **B.** `j < right`
- ☐ **C.** `i <= mid`
- ☐ **D.** `j <= right`

第 13 题 假设你是一家电影院的排片经理，只有一个放映厅。你有一个电影列表 `movies`，其中 `movies[i] = [start_i, end_i]` 表示第 `i` 部电影的开始和结束时间。请你找出最多能安排多少部不重叠的电影，则横线上应分别填写的代码为（ ）。

```
int maxMovies(vector<vector<int>>& movies) {
    if (movies.empty()) return 0;

    sort(movies.begin(), movies.end(), [](const vector<int>& a, const vector<int>& b) {
        return ______; // 在此处填入代码
    });

    int count = 1;
    int lastEnd = movies[0][1];

    for (int i = 1; i < movies.size(); i++) {
        if (movies[i][0] >= lastEnd) {
            count++;
            ______ = movies[i][1]; // 在此处填入代码
        }
    }

    return count;
}
```

- ☐ **A.** `a[0] < b[0]` 和 `lastEnd`
- ☐ **B.** `a[1] < b[1]` 和 `lastEnd`

- [ ] **C.** `a[0] < b[0]` 和 `movies[i][0]`
- [ ] **D.** `a[1] < b[1]` 和 `movies[i][0]`

第 14 题 给定一个整数数组 `nums`，下面代码找到一个具有最大和的连续子数组，并返回该最大和。则下面说法错误的是（ ）。

```cpp
int crossSum(vector<int>& nums, int left, int mid, int right) {
    int leftSum = INT_MIN, rightSum = INT_MIN;
    int sum = 0;
    for (int i = mid; i >= left; i--) {
        sum += nums[i];
        leftSum = max(leftSum, sum);
    }
    sum = 0;
    for (int i = mid + 1; i <= right; i++) {
        sum += nums[i];
        rightSum = max(rightSum, sum);
    }
    return leftSum + rightSum;
}

int helper(vector<int>& nums, int left, int right) {
    if (left == right)
        return nums[left];
    int mid = left + (right - left) / 2;
    int leftMax = helper(nums, left, mid);
    int rightMax = helper(nums, mid + 1, right);
    int crossMax = crossSum(nums, left, mid, right);
    return max({leftMax, rightMax, crossMax});
}

int maxSubArray(vector<int>& nums) {
    return helper(nums, 0, nums.size() - 1);
}
```

- [ ] **A.** 上述代码采用分治算法实现
- [ ] **B.** 上述代码采用贪心算法
- [ ] **C.** 上述代码时间复杂度为$O(n \log n)$
- [ ] **D.** 上述代码采用递归方式实现

第 15 题 给定一个由非负整数组成的数组 `digits`，表示一个非负整数的各位数字，其中最高位在数组首位，且 `digits` 不含前导0（除非是0本身）。下面代码对该整数执行 +1 操作，并返回结果数组，则横线上应填写（ ）。

```cpp
vector<int> plusOne(vector<int>& digits) {
    for (int i = (int)digits.size() - 1; i >= 0; --i) {
        if (digits[i] < 9) {
            digits[i] += 1;
            return digits;
        }
        ----------------    // 在此处填入代码
    }
    digits.insert(digits.begin(), 1);
    return digits;
}
```

- [ ] **A.**

```cpp
digits[i] = 0;
```

- [ ] **B.**

```
digits[i] = 9;
```

- [ ] **C.**

```
digits[i] = 1;
```

- [ ] **D.**

```
digits[i] = 10;
```

2　判断题（每题 2 分，共 20 分）

第 1 题 基于下面定义的函数，通过判断 `isDivisibleBy9(n) == isDigitSumDivisibleBy9(n)` 代码可验算如果一个数能被9整除，则它的各位数字之和能被9整除。

```
bool isDivisibleBy9(int n) {
    return n % 9 == 0;
}

bool isDigitSumDivisibleBy9(int n) {
    int sum = 0;
    string numStr = to_string(n);
    for (char c : numStr) {
        sum += (c - '0');
    }
    return sum % 9 == 0;
}
```

第 2 题 假设函数 `gcd()` 能正确求两个正整数的最大公约数，则下面的 `findMusicalPattern(4, 6)` 函数返回2。

```
void findMusicalPattern(int rhythm1, int rhythm2) {
    int commonDivisor = gcd(rhythm1, rhythm2);
    int patternLength = (rhythm1 * rhythm2) / commonDivisor;
    return patternLength;
}
```

第 3 题 下面递归实现的斐波那契数列的时间复杂度为 $O(2^n)$。

```
long long fib_memo(int n, long long memo[]) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo);
    return memo[n];
}

int main() {
    int n = 40;
    long long memo[100];
    fill_n(memo, 100, -1);
    long long result2 = fib_memo(n, memo);
    return 0;
}
```

第 4 题 链表通过更改指针实现高效的结点插入与删除，但结点访问效率低、占用内存较多，且对缓存利用不友好。

第 5 题 二分查找依赖数据的有序性，通过循环逐步缩减一半搜索区间来进行查找，且仅适用于数组或基于数组实现的数据结构。

第 6 题 线性筛关键是“每个合数只会被最小质因子筛到一次”，因此为 $O(n)$。

第 7 题 快速排序和归并排序都是稳定的排序算法。

第 8 题 下面代码采用分治算法求解标准 3 柱汉诺塔问题，时间复杂度为 $O(n \log n)$。

```cpp
void move(vector<int> &src, vector<int> &tar) {
    int pan = src.back();
    src.pop_back();
    tar.push_back(pan);
}

void dfs(int n, vector<int> &src, vector<int> &buf, vector<int> &tar) {
    if (n == 1) {
        move(src, tar);
        return;
    }

    dfs(n - 1, src, tar, buf);
    move(src, tar);
    dfs(n - 1, buf, src, tar);
}

void solveHanota(vector<int> &A, vector<int> &B, vector<int> &C) {
    int n = A.size();
    dfs(n, A, B, C);
}
```

第 9 题 所有递归算法都可以转换为迭代算法。

第 10 题 贪心算法总能得到全局最优解。

3 编程题（每题 25 分，共 50 分）

3.1 编程题 1

- **试题名称**：数字选取
- **时间限制**：1.0 s
- **内存限制**：512.0 MB

3.1.1 题目描述

给定正整数 n，现在有 $1, 2, \ldots, n$ 共计 n 个整数。你需要从这 n 个整数中选取一些整数，使得所选取的整数中任意两个不同的整数均互质（也就是说，这两个整数的最大公因数为 1）。请你最大化所选取整数的数量。

例如，当 $n = 9$ 时，可以选择 $1, 5, 7, 8, 9$ 共计 5 个整数。可以验证不存在数量更多的选取整数的方案。

3.1.2 输入格式

一行，一个正整数 n，表示给定的正整数。

3.1.3 输出格式

一行，一个正整数，表示所选取整数的最大数量。

3.1.4 样例

3.1.4.1 输入样例 1

```
6
```

3.1.4.2 输出样例 1

```
4
```

3.1.4.3 输入样例 2

```
9
```

3.1.4.4 输出样例 2

```
5
```

3.1.5 数据范围

对于 40% 的测试点，保证 $1 \leq n \leq 1000$。

对于所有测试点，保证 $1 \leq n \leq 10^5$。

3.2 编程题 2

- **试题名称**：有趣的数字和
- **时间限制**：1.0 s
- **内存限制**：512.0 MB

3.2.1 题目描述

如果一个正整数的二进制表示包含奇数个 1，那么小 A 就会认为这个正整数是**有趣的**。

例如，7 的二进制表示为 $(111)_2$，包含 1 的个数为 3 个，所以 7 是有趣的。但是 $9=(1001)_2$ 包含 2 个 1，所以 9 不是有趣的。

给定正整数 l,r，请你统计满足 $l\le n\le r$ 的有趣的整数 n 之和。

3.2.2 输入格式

一行，两个正整数 l,r，表示给定的正整数。

3.2.3 输出格式

一行，一个正整数，表示 l,r 之间有趣的整数之和。

3.2.4 样例

3.2.4.1 输入样例 1

```
3 8
```

3.2.4.2 输出样例 1

```
19
```

3.2.4.3 输入样例 2

```
65 36248
```

3.2.4.4 输出样例 2

```
328505490
```

3.2.5 数据范围

对于 40% 的测试点，保证 $1\le l\le r\le 10^4$。

对于另外 30% 的测试点，保证 $l=1$ 并且 $r=2^k-1$，其中 k 是大于 1 的正整数。

对于所有测试点，保证 $1\le l\le r\le 10^9$。

3.2.6 提示

由于本题的数据范围较大，整数类型请使用 `long long` 。