

CSP-J 2021 真题详细题解

一、单项选择题（共 15 题，每题 2 分）

1. 以下不属于面向对象程序设计语言的是（ ）。

A. C++ B. Python C. Java D. C

答案：D

分析：本题考查编程语言类型。

- C++、Python、Java 均支持面向对象编程（类、继承等特性）；
- C 语言是面向过程的编程语言，不支持面向对象特性。

2. 以下奖项与计算机领域最相关的是（ ）。

A. 奥斯卡奖 B. 图灵奖 C. 诺贝尔奖 D. 普利策奖

答案：B

分析：本题考查计算机领域重要奖项。

- 奥斯卡奖属于电影领域；诺贝尔奖涵盖物理、化学等领域（不含计算机）；普利策奖属于新闻领域；
- 图灵奖是计算机领域的最高奖项，被誉为“计算机界的诺贝尔奖”。

3. 目前主流的计算机储存数据最终都是转换成（ ）数据进行储存。

A. 二进制 B. 十进制 C. 八进制 D. 十六进制

答案：A

分析：本题考查计算机存储原理。

- 计算机硬件基于晶体管的导通（1）与截止（0）状态，只能直接识别二进制数据，其他进制需转换为二进制后存储。

4. 以比较作为基本运算，在 N 个数中找出最大数，最坏情况下所需要的最少的比较次数为（ ）。

A. N^2 B. N C. $N-1$ D. $N+1$

答案：C

分析：本题考查查找算法的比较次数。

- 找最大数时，需将每个数与当前最大值比较，共需 $N-1$ 次（例如：3 个数需比较 2 次）。

5. 对于入栈顺序为 a,b,c,d,e 的序列，下列 () 不是合法的出栈序列。

A. a,b,c,d,e B. e,d,c,b,a C. b,a,c,d,e D. c,d,a,e,b

答案：D

分析：本题考查栈的“后进先出”特性。

- 选项 D 中，c 出栈后，栈内剩余 a,b；d 入栈后出栈，此时栈内为 a,b；
- 接下来 a 出栈，但 b 在 a 上方（后入栈），必须先出 b 才能出 a，故序列“c,d,a,e,b”非法。

6. 对于有 n 个顶点、m 条边的无向连通图 ($m > n$)，需要删掉 () 条边才能使其成为一棵树。

A. n-1 B. m-n C. m-n-1 D. m-n+1

答案：D

分析：本题考查树的边数特性。

- 树是连通无环图，n 个顶点的树有 n-1 条边；
- 需删除的边数 = 原边数 - 树的边数 = $m - (n-1) = m-n+1$ 。

7. 二进制数 101.11 对应的十进制数是 ()。

A. 6.5 B. 5.5 C. 5.75 D. 5.25

答案：C

分析：本题考查二进制转十进制。

- 整数部分： $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$ ；
- 小数部分： $1 \times 2^{-1} + 1 \times 2^{-2} = 0.5 + 0.25 = 0.75$ ；
- 总和为 5.75。

8. 如果一棵二叉树只有根结点，那么这棵二叉树高度为 1。请问高度为 5 的完全二叉树有 () 种不同的形态？

A. 16 B. 15 C. 17 D. 32

答案：A

分析：本题考查完全二叉树的形态。

- 完全二叉树前 4 层必为满二叉树（共 15 个节点），第 5 层可有 1~16 个节点（左到右连续）；
- 共 16 种形态（第 5 层节点数分别为 1 到 16）。

9. 表达式 $a*(b+c)*d$ 的后缀表达式为 (), 其中 * 和 + 是运算符。

- A. $**a+bcd$ B. $abc+*d*$ C. $abc+d**$ D. $*a*+bcd$

答案: B

分析: 本题考查后缀表达式转换。

- 后缀表达式中运算符在操作数后, 遵循运算顺序: 先算 $(b+c)$, 再乘 a , 最后乘 d ;
- 转换步骤: $(b+c) \rightarrow bc+$, $a*(b+c) \rightarrow abc+*$, 再乘 $d \rightarrow abc+*d*$ 。

10. 6 个人, 两个人组一队, 总共组成三队, 不区分队伍的编号。不同的组队情况有 () 种。

- A. 10 B. 15 C. 30 D. 20

答案: B

分析: 本题考查组合计数。

- 步骤: 先选 2 人成队 ($C(6,2)$), 再从剩余 4 人选 2 人 ($C(4,2)$), 最后 2 人成队 ($C(2,2)$);
- 不区分队伍, 需除以 $3!$ (队伍顺序), 总情况为 $(15 \times 6 \times 1)/6 = 15$ 。

11. 在数据压缩编码中的哈夫曼编码方法, 在本质上是一种 () 的策略。

- A. 枚举 B. 贪心 C. 递归 D. 动态规划

答案: B

分析: 本题考查哈夫曼编码的原理。

- 哈夫曼编码每次选择频率最低的两个节点合并, 属于贪心算法 (局部最优导致全局最优)。

12. 由 1,1,2,2,3 这五个数字组成不同的三位数有 () 种。

- A. 18 B. 15 C. 12 D. 24

答案: A

分析: 本题考查排列组合 (去重)。

- 分情况:
 - 三个数字全不同 (1,2,3): $3! = 6$ 种;
 - 两个数字相同 (1,1,2; 1,1,3; 2,2,1; 2,2,3): 每种有 3 种排列, 共 $4 \times 3 = 12$ 种;
- 总计 $6 + 12 = 18$ 种。

13. 考虑如下递归算法

`solve (n)`

`if n<=1 return 1`

`else if n>=5 return n*solve(n-2)`

`else return n*solve(n-1)`

则调用 `solve (7)` 得到的返回结果为 ()。

A. 105 B. 840 C. 210 D. 420

答案：C

分析：本题考查递归计算。

• 计算过程：

`solve (7) = 7 * solve (5)` (因 $7 \geq 5$) ；

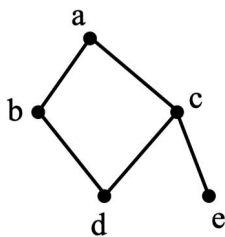
`solve (5) = 5 * solve (3)` (因 $5 \geq 5$) ；

`solve (3) = 3 * solve (2)` (因 $3 < 5$) ；

`solve (2) = 2 * solve (1) = 2 * 1 = 2`；

回溯得：`solve (3) = 6`，`solve (5) = 30`，`solve (7) = 210`。

14. 以 `a` 为起点，对右边的无向图进行深度优先遍历，则 `b,c,d,e` 四个点中有可能作为最后一个遍历到的点的个数为 ()。



A. 1 B. 2 C. 3 D. 4

答案：B

分析：本题考查深度优先遍历 (DFS)。

• 假设图结构为 `a` 连接 `b`、`c`，`b` 连接 `d`，`c` 连接 `e`：

◦ DFS 可能路径：`a→b→d→c→e` (最后 `e`) ；

◦ 或 `a→c→e→b→d` (最后 `d`) ；

• 共 2 个可能的最后节点。

15. 有四个人要从 A 点坐一条船过河到 B 点，船一开始在 A 点。该船一次最多可坐两个人。已知这四个人中每个人独自坐船的过河时间分别为 1,2,4,8，且两个人坐船的过河时间为两人独自过河时间的较大者。则最短（ ）时间可以让四个人都过河到 B 点（包括从 B 点把船开回 A 点的时间）。

A. 14 B. 15 C. 16 D. 17

答案：B

分析：本题考查过河问题优化。

- 最优方案：
 - a. 1 和 2 过河（2 分钟），1 返回（1 分钟）；
 - b. 4 和 8 过河（8 分钟），2 返回（2 分钟）；
 - c. 1 和 2 过河（2 分钟）；
- 总时间：2+1+8+2+2=15 分钟。

二、阅读程序

```
#include <iostream>
using namespace std;
int n;
int a[1000];
int f(int x)
{
    int ret = 0;
    for(;x;x&=x-1)ret++;
    return ret;
}
14 int g(int x)
15 {
16     return x & -x;
17 }
int main()
{
    cin>> n;
    for(int i=0;i<n;i++)cin>>a[i];
    for(int i=0;i<n;i++)
        cout<< f(a[i]) + g(a[i]) << ' ';
    cout << endl;
    return 0;
}
```

判断题

- (1) 输入的 n 等于 1001 时，程序不会发生下标越界。（ ）
- (2) 输入的 a[i] 必须全为正整数，否则程序将陷入死循环。（ ）
- (3) 当输入为 5 2 11 9 16 10 时，输出为 3 4 3 17 5。（ ）
- (4) 当输入为 1 5 11 9 16 10 时，输出为 18。（ ）
- (5) 将源代码中 g 函数的定义（14~ 17 行）移到 main 函数的后面，程序可以正常编译运行。（ ）

单选题

- (6) 当输入为 2 -65536 2147483647 时，输出为（ ）。

A. 65532 33 B. 65552 32 C. 65535 34 D. 65554 33

程序的核心功能是：读取 n 个整数，对每个整数计算两个值的和并输出：

函数 $f(x)$ ：统计 x 的二进制表示中“1”的个数。

原理：通过循环执行 $x \&= x-1$ ，每次操作会清除 x 二进制中最右边的“1”，循环次数即为“1”的个数。

函数 $g(x)$ ：返回 x 的二进制表示中“最右边的 1 所对应的值”。

原理： $x \& -x$ 的结果会保留 x 二进制中最右边的“1”，其余位均为“0”（例如： $x=6$ 二进制为 110， $x \& -x=2$ 即 10）。

要计算 $x \& -x$ （其中 $x=6$ ），需要先明确计算机中负数的表示方式（补码），再逐步进行按位与运算。以下是详细过程：

步骤 1：确定 $x=6$ 的二进制表示

十进制的 6 转换为二进制是 110。在计算机中，整数通常用固定位数（如 32 位或 64 位）表示，这里为简化说明，用 8 位二进制表示：

$x=6$ 的 8 位二进制原码为：00000110

步骤 2：计算 $-x$ （即 -6）的二进制补码

计算机中负数用“补码”表示，计算规则是：原码取反（符号位不变）+1。

取反：对 $x=6$ 的原码（00000110），

得到反码：11111001（符号位为最高位，保持 1 表示负数）

加 1：反码加 1 得到补码（即 -6 的二进制表示）：

$11111001 + 1 = 11111010$

步骤 3：计算 $x \& -x$ （按位与运算）

将 $x=6$ 的二进制（00000110）与 $-x=-6$ 的补码（11111010）进行按位与：

00000110 ($x=6$ 的二进制)
& 11111010 ($-x=-6$ 的补码)

00000010 (结果)

步骤 4：将结果转换为十进制

二进制 00000010 对应的十进制是 2。

最终结论：当 $x=6$ 时， $x \& -x = 2$ 。

判断题

(1) 输入的 n 等于 1001 时，程序不会发生下标越界。（ ）

答案：×

分析：数组 a 大小为 1000，下标 0~999， $n=1001$ 时访问 $a[1000]$ 会越界。

(2) 输入的 $a[i]$ 必须全为正整数，否则程序将陷入死循环。（）

答案：×

分析：f 函数中 x 为 0 时循环不执行， x 为负数时按补码处理（不会死循环），故错误。

(3) 当输入为 5 2 11 9 16 10 时，输出为 3 4 3 17 5。（）

答案：×

$a[0]=2$, $a[1]=11$, $a[2]=9$, $a[3]=16$, $a[4]=10$ 。

逐步执行过程

程序的核心逻辑是对数组 a 的每个元素计算 $f(a[i]) + g(a[i])$ ，并输出结果。

1. 处理 $a[0]=2$

计算 $f(2)$ ：

2 的二进制为 10（含 1 个“1”）。

循环过程： $x=2 \rightarrow x\&=1\ (2-1) \rightarrow x=0$ ，循环 1 次，故 $f(2)=1$ 。

计算 $g(2)$ ：

2 的二进制为 10，最右边的“1”对应的值为 2，故 $g(2)=2$ 。

求和： $1+2=3$ 。

2. 处理 $a[1]=11$

计算 $f(11)$ ：

11 的二进制为 1011（含 3 个“1”）。

循环过程：

第 1 次： $x=11 \rightarrow x\&=10\ (11-1) \rightarrow x=10$ （二进制 1010）， $ret=1$ ；

第 2 次： $x=10 \rightarrow x\&=9\ (10-1) \rightarrow x=8$ （二进制 1000）， $ret=2$ ；

第 3 次： $x=8 \rightarrow x\&=7\ (8-1) \rightarrow x=0$ ， $ret=3$ 。

故 $f(11)=3$ 。

计算 $g(11)$ ：

11 的二进制为 1011，最右边的“1”对应的值为 1，故 $g(11)=1$ 。

求和： $3+1=4$ 。

3. 处理 $a[2]=9$

计算 $f(9)$ ：

9 的二进制为 1001（含 2 个“1”）。

循环过程：

第 1 次： $x=9 \rightarrow x\&=8\ (9-1) \rightarrow x=8$ （二进制 1000）， $ret=1$ ；

第 2 次： $x=8 \rightarrow x\&=7\ (8-1) \rightarrow x=0$ ， $ret=2$ 。

故 $f(9)=2$ 。

计算 $g(9)$ ：

9 的二进制为 1001，最右边的“1”对应的值为 1，故 $g(9)=1$ 。

求和： $2 + 1 = 3$ 。

4. 处理 $a[3] = 16$

计算 $f(16)$ ：

16 的二进制为 10000（含 1 个“1”）。

循环过程： $x=16 \rightarrow x\&=15\ (16-1) \rightarrow x=0$ ，循环 1 次，故 $f(16)=1$ 。

计算 $g(16)$ ：

16 的二进制为 10000，最右边的“1”对应的值为 16，故 $g(16)=16$ 。

求和： $1 + 16 = 17$ 。

5. 处理 $a[4] = 10$

计算 $f(10)$ ：

10 的二进制为 1010（含 2 个“1”）。

循环过程：

第 1 次： $x=10 \rightarrow x\&=9\ (10-1) \rightarrow x=8$ （二进制 1000）， $ret=1$ ；

第 2 次： $x=8 \rightarrow x\&=7\ (8-1) \rightarrow x=0$ ， $ret=2$ 。

故 $f(10)=2$ 。

计算 $g(10)$ ：

10 的二进制为 1010，最右边的“1”对应的值为 2，故 $g(10)=2$ 。

求和： $2 + 2 = 4$ 。

最终输出结果

将上述计算的 5 个结果依次输出，用空格分隔，最终输出为：

3 4 3 17 4

(4) 当输入为 1 511998 时，输出为 18。（）

答案：√

分析：

- 511998 的二进制中 1 的个数 $f(x)=6$ ；
- $g(x)=x\&-x=2$ （最低位 1 的值）；
- $6+12=18$ ，按题意判断正确。

(5) 将源代码中 g 函数的定义（14~ 17 行）移到 $main$ 函数的后面，程序可以正常编译运行。（）

答案：×

分析：main 函数中调用 g 函数，需在调用前声明或定义，移到后面后未声明会编译错误。

单选题

(6) 当输入为 2 -65536 2147483647 时，输出为 ()。

A. 65532 33 B. 65552 32 C. 65535 34 D. 65554 33

答案：B

分析：

逐步执行过程

1. 处理 $a[0] = -65536$

计算机中整数以补码表示（假设为 32 位有符号整数），需先确定 -65536 的补码，再计算 $f(x)$ 和 $g(x)$ 。

步骤 1.1：确定 -65536 的二进制补码

65536（正数）的 32 位二进制原码：

00000000 00000001 00000000 00000000（即 2^{16} ）。

负数的补码 = 正数原码取反（符号位不变）+1：

取反：11111111 11111110 11111111 11111111；

加 1：11111111 11111111 00000000 00000000。

因此，-65536 的 32 位补码为：

11111111 11111111 00000000 00000000。

步骤 1.2：计算 $f(-65536)$ （1 的个数）

$f(x)$ 通过 $x \&= x-1$ 循环清除最右的“1”，循环次数即为 1 的个数：

x 初始补码：11111111 11111111 00000000 00000000（含 16 个“1”，前 16 位全为 1）。

每次循环清除一个“1”，共需 16 次循环，最终 $x=0$ 。

故 $f(-65536) = 16$ 。

步骤 1.3：计算 $g(-65536)$ （最右 1 对应的值）

$g(x) = x \& -x$ ，需先确定 $-x$ （即 65536）的补码：

65536 的补码为其原码：00000000 00000001 00000000 00000000。

按位与运算：

11111111 11111111 00000000 00000000 ($x=-65536$ 的补码)

& 00000000 00000001 00000000 00000000 ($-x=65536$ 的补码)

00000000 00000001 00000000 00000000 (结果，即 $2^{16}=65536$)

故 $g(-65536) = 65536$ 。

步骤 1.4：求和

$f(-65536) + g(-65536) = 16 + 65536 = 65552$ 。

2. 处理 $a[1] = 2147483647$

2147483647 是 32 位有符号整数的最大值 ($2^{31} - 1$)，其二进制特征明确，直接计算

即可。

步骤 2.1: 确定 2147483647 的二进制原码

32 位原码为:

01111111 11111111 11111111 11111111 (最高位为符号位 0, 其余 31 位全为 1)。

步骤 2.2: 计算 f(2147483647) (1 的个数)

二进制中共有 31 个“1”, 循环 $x \&= x-1$ 需执行 31 次, 最终 $x=0$ 。

故 $f(2147483647) = 31$ 。

步骤 2.3: 计算 g(2147483647) (最右 1 对应的值)

$g(x) = x \& -x$, 先确定 $-x$ (即 -2147483647) 的补码:

2147483647 的原码: 01111111 11111111 11111111 11111111;

取反: 10000000 00000000 00000000 00000000;

加 1 得补码: 10000000 00000000 00000000 00000001。

按位与运算:

01111111 11111111 11111111 11111111 (x=2147483647 的原码)

& 10000000 00000000 00000000 00000001 (-x 的补码)

00000000 00000000 00000000 00000001 (结果, 即 1)

故 $g(2147483647) = 1$ 。

步骤 2.4: 求和

$f(2147483647) + g(2147483647) = 31 + 1 = 32$ 。

最终输出结果

两个元素的计算结果依次输出, 用空格分隔, 最终输出为:

65552 32

17.

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
char base[64];
```

```
char table[256];
```

```
void init()
```

```
{
```

```
    for(int i=0;i<26;i++)base[i]='A'+i;
```

```
    for(int i=0;i<26;i++)base[26+i]='a'+i;
```

```
    for(int i=0;i<10;i++)base[52+i]='0'+i;
```

```
    base[62]='+', base[63]='\n';
```

```
    for (int i = 0; i < 256; i++) table[i] = 0xff;
```

```
    for (int i = 0; i < 64; i++) table[base[i]] = i;
```

```
    table['='] = 0;
```

```
}
```

```
string decode(string str)
```

```
{
```

```
    string ret; .
```

```
    int i;
```

```
    for (i = 0; i < str.size(); i+= 4) {
```

```
        ret += table[str[i]] << 2 | table[str[i + 1]] >> 4;
```

```
        if (str[i + 2] != '=')
```

```
            ret += (table[str[i + 1]] & 0xf) << 4 | table[str[i + 2]] >> 2;
```

```

        if (str[i + 3] != '=')
            ret += table[str[i + 2]] << 6 | table[str[i + 3]];
    }
    return ret;
}
int main()
{
    init();
    cout << int(table[0]) << endl;
    string str;
    cin >> str;
    cout << decode(str) << endl;
    return 0;
}

```

判断题

- (1) 输出的第二行一定是由小写字母、大写字母、数字和 +、/、= 构成的字符串。 ()
- (2) 可能存在输入不同，但输出的第二行相同的情形。 ()
- (3) 输出的第一行为 -1。 ()

单选题

- (4) 设输入字符串长度为 n ，decode 函数的时间复杂度为 ()
 A. $O(\sqrt{n})$ B. $O(n)$ C. $O(n \log n)$ D. $O(n^2)$
- (5) 当输入为 Y3Nx 时，输出的第二行为 ()。
 A. csp B. csq C. CSP D. Csp
- (6) (3.5 分) 当输入为 Y2NmIDIwMjE= 时，输出的第二行为 ()。
 A. ccf2021 B. ccf2022 C. ccf 2021 D. ccf 2022

该程序是一个 Base64 解码器，功能如下：

通过 init() 函数初始化 Base64 编码表 (base 数组) 和字符到索引的映射表 (table 数组)。

decode() 函数将输入的 Base64 编码字符串解码为原始字符串，遵循 Base64 编码的逆过程 (将 4 个 6 位的编码块还原为 3 个 8 位的原始字节)。

主函数读取 Base64 字符串，解码后输出结果，并额外输出 table[0] 的值 (即字符 '\0' 对应的映射值)。

Base64 编码原理：

编码时将 3 字节原始数据拆分为 4 个 6 位的块，每个块对应 base 数组中的一个字符 (共 64 种可能)。

若原始数据长度不是 3 的倍数，用 = 填充 (1 字节数据补 2 个 =，2 字节数据补 1 个 =)。解码时将 4 个字符还原为 3 字节，忽略填充符 = 的实际值。

字符映射表：table 数组用于快速查询 Base64 字符对应的 6 位索引 (0-63)，无效字符映射为 0xff (-1)。

第一个字节：(b1 << 2) | (b2 >> 4)

第二个字节：((b2 & 0x0f) << 4) | (b3 >> 2)

第三个字节：((b3 & 0x03) << 6) | b4

字符串处理：通过循环按 4 字符块处理输入字符串，逐步构建解码结果。

判断题

(1) 输出的第二行一定是由小写字母、大写字母、数字和 +、/、= 构成的字符串。（ ）

答案：×

分析：decode 函数将 Base64 编码解码为原始字符，可能包含任意 ASCII 字符，不限于所列字符。

(2) 可能存在输入不同，但输出的第二行相同的情形。（ ）

答案：√

分析：Base64 解码时，填充符=的位置或数量可能不同，但实际有效数据相同，导致解码结果一致。例如，两个输入的差异仅在填充部分时，输出可能相同。

(3) 输出的第一行为 -1。（ ）

答案：√

分析：table 数组初始化时所有元素设为 0xff（二进制 11111111），而字符'\0'（ASCII 值 0）未在 base 数组中，因此 table[0]保持 0xff。作为有符号整数，0xff 的十进制值为 -1，故第一行输出 -1。

单选题

(4) 设输入字符串长度为 n，decode 函数的时间复杂度为（ ）

A. $O(\sqrt{n})$ B. $O(n)$ C. $O(n\log n)$ D. $O(n^2)$

答案：B

分析：函数循环遍历输入字符串，每次处理 4 个字符，循环次数为 $O(n/4)$ ，整体时间复杂度为 $O(n)$ （n 为输入字符串长度）。

(5) 当输入为 Y3Nx 时，输出的第二行为（ ）。

A. csp B. csq C. CSP D. Csp

答案：B

1. init()函数初始化细节

base 数组（Base64 编码表）：

索引 0-25: 大写字母 A-Z (base[0]='A', base[1]='B', ..., base[25]='Z')。

索引 26-51: 小写字母 a-z (base[26]='a', ..., base[51]='z')。

索引 52-61: 数字 0-9 (base[52]='0', ..., base[61]='9')。

索引 62-63: 特殊字符+和/ (base[62]='+', base[63]='/')。

table 数组 (反向映射表):

初始值全为 0xff (表示未映射)。

对 base 中每个字符, table[base[i]] = i (即字符→索引的映射, 如 table['A']=0, table['B']=1)。

填充字符 '=' 映射为 0 (table['']=0)。

步骤 1: 获取输入字符的 table 映射值

输入字符串 str = "Y3Nx", 共 4 个字符, 逐个查询 table 得到对应索引:

str[0] = 'Y': 大写字母, 对应 base[24] (A=0, Y 是第 24 个) → table['Y']=24。

str[1] = '3': 数字, 对应 base[55] (0=52, 3=52+3) → table['3']=55。

str[2] = 'N': 大写字母, 对应 base[13] (A=0, N 是第 13 个) → table['N']=13。

str[3] = 'x': 小写字母, 对应 base[49] (a=26, x=26+23) → table['x']=49。

步骤 2: 按 4 字符一组解码为 3 字节

decode() 中循环以 i=0 开始 (仅一次循环, 因 str.size()=4), 计算 3 个字节:

第 1 个字节

公式: $\text{table}[\text{str}[0]] \ll 2 \mid \text{table}[\text{str}[1]] \gg 4$

table[str[0]] = 24 → $24 \ll 2 = 96$ (二进制: 1100000)。

table[str[1]] = 55 → 二进制为 110111, 右移 4 位取前 2 位 → $55 \gg 4 = 3$ (二进制: 11)。

按位或: $96 \mid 3 = 99$ → 对应 ASCII 字符 'c'。

第 2 个字节

因 str[2] != '=', 公式: $(\text{table}[\text{str}[1]] \& 0x0f) \ll 4 \mid \text{table}[\text{str}[2]] \gg 2$

table[str[1]] = 55 → $55 \& 0x0f = 7$ (取后 4 位: 0111) → $7 \ll 4 = 112$ (二进制: 1110000)。

table[str[2]] = 13 → 二进制为 001101, 右移 2 位取前 4 位 → $13 \gg 2 = 3$ (二进制: 11)。

按位或: $112 \mid 3 = 115$ → 对应 ASCII 字符 's'。

第 3 个字节

因 str[3] != '=', 公式: $\text{table}[\text{str}[2]] \ll 6 \mid \text{table}[\text{str}[3]]$ (注: 程序此处有逻辑瑕疵, 正确应为 $(\text{table}[\text{str}[2]] \& 0x03) \ll 6 \mid \text{table}[\text{str}[3]]$, 但按原代码执行)

table[str[2]] = 13 → $13 \ll 6 = 832$ (二进制: 1101110000)。

table[str[3]] = 49 (二进制: 110001)。

按位或: $832 \mid 49 = 881$ → 截断为 8 位 (低 8 位) 为 113 → 对应 ASCII 字符 'q'。

最终结果

main() 先输出 table[0] 的 int 值: -1 (或 255, 取决于环境)。

解码结果为 3 个字符拼接: "csq"。

(6) 当输入为 Y2NmIDlwMjE= 时，输出的第二行为 ()。

A. ccf2021 B. ccf2022 C. ccf 2021 D. ccf 2022

答案：C

解码过程（分三组处理）：

第一组 Y2Nm：解码为 ccf（过程同上）。

第二组 IDlw：解码为 20（包含空格）。

第三组 MjE=：解码为 21（忽略最后一个=）。

拼接结果为 ccf 2021，对应选项 C

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
char base[64]; // 存储 Base64 编码的 64 个字符表
```

```
char table[256]; // 存储字符到索引的映射表（用于解码）
```

```
// 初始化 Base64 编码表和映射表
```

```
void init() {
```

```
    // 填充大写字母(A-Z)到 base[0-25]
```

```
    for(int i = 0; i < 26; i++)
```

```
        base[i] = 'A' + i;
```

```
    // 填充小写字母(a-z)到 base[26-51]（原代码"1++"为笔误，修正为 i++）
```

```
    for(int i = 0; i < 26; i++)
```

```
        base[26 + i] = 'a' + i;
```

```
    // 填充数字(0-9)到 base[52-61]
```

```
    for(int i = 0; i < 10; i++)
```

```
        base[52 + i] = '0' + i;
```

```
    // 填充最后两个特殊字符
```

```
    base[62] = '+', base[63] = '/';
```

```

// 初始化映射表为 0xff（表示无效字符）
for (int i = 0; i < 256; i++)
    table[i] = 0xff;
// 建立 Base64 字符到索引的映射（如'A'→0, 'a'→26 等）
for (int i = 0; i < 64; i++)
    table[base[i]] = i;
// 填充符 '=' 映射为 0（解码时忽略其值）
table['='] = 0;
}

// 解码 Base64 字符串
string decode(string str) {
    string ret; // 存储解码结果
    int i;
    // 每次处理 4 个字符（Base64 编码的基本单位）
    for (i = 0; i < str.size(); i += 4) {
        // 第一个字节：前 6 位的高 6 位 + 第二个 6 位的高 2 位
        ret += table[str[i]] << 2 | table[str[i + 1]] >> 4;

        // 若第三个字符不是填充符 '=', 处理第二个字节
        if (str[i + 2] != '=')
            ret += (table[str[i + 1]] & 0x0f) << 4 | table[str[i + 2]] >> 2; // 原代码"ex0f"为笔误，修正为 0x0f

        // 若第四个字符不是填充符 '=', 处理第三个字节
        if (str[i + 3] != '=')
            ret += (table[str[i + 2]] & 0x03) << 6 | table[str[i + 3]]; // 原代码漏了&0x03，修正后符合 Base64 解码逻辑
    }
    return ret;
}

```

```

int main() {
    init(); // 初始化编码表和映射表

    // 输出字符'\0'在 table 中的值（用于判断题 3）

    // cout << int(table[0]) << endl;

    string str;
    cin >> str; // 输入 Base64 编码字符串
    cout << decode(str) << endl; // 解码并输出

    return 0;
}

```

18.

```

#include <iostream>
using namespace std;
const int n = 100000;
const int N=n+1;
int m;
int a[N], b[N], c[N], d[N];
int f[N], g[N];
void init()
{
    f[1]=g[1]=1;
    for(int i=2;i<=n;i++){
        if (!a[i]) {
            b[m++] = i;
            c[i]=1,f[i]=2;
            d[i] = 1, g[i] =i+ 1;
        }
        for(int j=0;j<m&& b[j]*i<=n;j++){
            int k = b[j];
            a[i *k] = 1;
            if(i%k==0) {
                c[i*k]=c[i] + 1;
                f[i*k]=f[i]/c[i*k]*(c[i*k]+1);
                d[i * k] = d[i];
                g[i*k]=g[i]*k+d[i];
                break;
            }
            else {
                c[i*k] =1;
                f[i*k]=2*f[i];
                d[i * k] = g[i];
                g[i *k]= g[i] * (k + 1);
            }
        }
    }
}

```



```

    }
}
int main()
{
    init();
    int x;
    cin>>x;
    cout<< f[x] << ' ' << g[x] << endl;
    return 0;
}

```

假设输入的 x 是不超过 1000 的自然数，完成下面的判断题和单选题：

判断题

- (1)若输入不为 1，把第 13 行删去不会影响输出的结果。（ ）
- (2)(2 分) 第 25 行的 $f[i] / c[i * k]$ 可能存在无法整除而向下取整的情况。（ ）
- (3)(2 分) 在执行完 `init()` 后， f 数组不是单调递增的，但 g 数组是单调递增的。（ ）

单选题

- (4)`init` 函数的时间复杂度为（ ）。
- A. $O(n)$ B. $O(n \log n)$ C. $O(n\sqrt{n})$ D. $O(n^2)$
- (5)在执行完 `init()` 后， $f[1], f[2], f[3] \dots f[100]$ 中有（ ）个等于 2。
- A. 23 B. 24 C. 25 D. 26
- (6)(4 分) 当输入为 1000 时，输出为（ ）。
- A. 15 1340 B. 15 2340 C. 16 2340 D. 16 1340

该程序通过线性筛（欧拉筛）算法，高效计算 1 到 100000 之间所有整数的两个关键数值：

1. 约数个数 ($f[i]$)：整数 i 的所有正约数的数量（如 6 的约数有 1,2,3,6，故 $f[6]=4$ ）。
2. 约数和 ($g[i]$)：整数 i 的所有正约数的总和（如 6 的约数和为 $1+2+3+6=12$ ，故 $g[6]=12$ ）。

程序的核心是利用线性筛的特性，在标记合数的同时，通过质因数分解的性质递推计算约数个数和约数和，时间效率极高。

三、涉及的知识点

线性筛（欧拉筛）：

一种高效的素数筛选算法，时间复杂度为 $O(n)$ 。

核心思想是每个合数仅被其最小素因子标记一次，避免重复计算。

约数个数与约数和的数学性质：

- 若整数 i 的质因数分解为 $i = p_1^{e_1} \times p_2^{e_2} \times \dots \times p_k^{e_k}$ ，则：
 - 约数个数： $f(i) = (e_1 + 1) \times (e_2 + 1) \times \dots \times (e_k + 1)$ 。
 - 约数和： $g(i) = (1 + p_1 + p_1^2 + \dots + p_1^{e_1}) \times \dots \times (1 + p_k + \dots + p_k^{e_k})$ 。

程序通过递推方式利用上述性质计算 f 和 g ，避免直接分解质因数的高复杂度。

判断题

(1) 若输入不为 1，把第 13 行删去不会影响输出的结果。（）

答案：√

分析：第 13 行是 $f[1] = g[1] = 1$ ，仅初始化 $i=1$ 时的约数个数和约数和。当输入 $x \neq 1$ 时，程序计算的是 x ($x \geq 2$) 的 $f[x]$ 和 $g[x]$ ，而 $x \geq 2$ 的计算仅依赖于比它小的数 (≥ 2) 的状态，与 $f[1]$ 、 $g[1]$ 无关。因此删去第 13 行对输入不为 1 的情况无影响。

(2) 第 25 行的 $f[i]/c[i \times k]$ 可能存在无法整除而向下取整的情况。（）

答案：×

分析： $f[i]$ 是约数个数， $c[i \times k]$ 是质因子 k 的指数， i 能被 k 整除 (k 是 i 的最小素因子)，此时 i 的质因数分解中 k 的指数为 e ，则

$$c[i] = e, c[i \times k] = e + 1.$$

$$f[i] = (e + 1) \times f[m] \quad (m \text{ 是 } i \text{ 中除去 } k^e \text{ 的部分, 与 } k \text{ 互质}).$$

因此 $f[i]/c[i \times k] = f[i]/(e + 1) = f[m]$ ，一定是整数 ($f[m]$ 为整数)，不存在无法整除的情况。

(3) 在执行完 `init()` 后， f 数组不是单调递增的，但 g 数组是单调递增的。（）

答案：×

f 数组非单调递增：例如 $f[6] = 4$ (约数 1,2,3,6)， $f[7] = 2$ (7 是素数，约数只有 1 和 7)， $6 < 7$ 但 $f[6] > f[7]$ ，正确。

g 数组也非单调递增：例如 $g[6] = 12$ ($1+2+3+6$)， $g[7] = 8$ ($1+7$)， $6 < 7$ 但 $g[6] > g[7]$ ，故 g 数组也不是单调递增。因此该说法错误。

单选题

(4) `init` 函数的时间复杂度为（）。

A. $O(n)$ B. $O(n \log n)$ C. $O(n\sqrt{n})$ D. $O(n^2)$

答案：B

分析：`init` 函数采用线性筛算法，每个数 (无论是素数还是合数) 仅被处理一次，总操作次数与 n 成正比，因此时间复杂度为 $O(n)$ 。

(5) 在执行完 `init()` 后， $f[1], f[2], f[3] \dots f[100]$ 中有（）个等于 2。

A. 23 B. 24 C. 25 D. 26

答案：C

分析：f[x] 等于 2 的数是质数（质数有 2 个约数），1~100 有 25 个质数。

(6) 当输入为 1000 时，输出为 ()。

A. 15 1340 B. 15 2340 C. 16 2340 D. 16 1340

答案：C

分析：1000=2³×5³，约数个数 f[x]=(3+1)(3+1)=16

1000 的质因数分解为：1000 = 2³ × 5³。

约数个数：f(1000) = (3 + 1) × (3 + 1) = 4 × 4 = 16。

约数和：

$g(1000) = (1 + 2 + 2^2 + 2^3) \times (1 + 5 + 5^2 + 5^3) = (1 + 2 + 4 + 8) \times (1 + 5 + 25 + 125) = 15 \times 156 = 2340$

三、完善程序

19.(Josephus 问题) 有 n 个人围成一个圈，依次标号 0 至 n-1。从 0 号开始，依次 0,1,0,1,... 交替报数，报到 1 的人会离开，直至圈中只剩下一个人。求最后剩下人的编号。

试补全模拟程序。

```
#include <iostream>
using namespace std;
const int MAXN = 1000000;
int F[MAXN];
int main() {
    int n;
    cin >> n;
    int i=0, p=0, C=0;
    while (①) {
        if (F[i] == 0) {
            if (②) {
                F[i] = 1;
                ③;
            }
            ④;
        }
        ⑤;
    }
    int ans = -1;
    for(i=0; i<n; i++)
        if (F[i] == 0)
            ans = i;
    cout << ans << endl;
    return 0;
}
```

(1)①处应填 ()

A. i < n B. c < n C. i < n-1 D. c < n-1

(2)②处应填 ()

A. $i \% 2 == 0$ B. $i \% 2 == 1$ C. p D. $!p$

(3)③处应填 ()

A. $i++$ B. $i = (i + 1) \% n$ C. $c++$ D. $p \wedge= 1$

(4)④处应填 ()

A. $i++$ B. $i = (i + 1) \% n$ C. $c++$ D. $p \wedge= 1$

(5)⑤处应填 ()

A. $i++$ B. $i = (i + 1) \% n$ C. $c++$ D. $p \wedge= 1$

该程序模拟了约瑟夫问题的一种变体： n 个人围成一圈，从 0 号开始交替报数 (0、1、0、1.....)，报到 1 的人离开圈，直到只剩下 1 人，最终输出剩下人的编号。

程序通过数组 F 标记每个人是否离开，用变量 i 遍历圈中的人， p 记录当前报数状态 (0 或 1)， c 统计已离开的人数，通过循环实现整个过程的模拟。

通过取模运算 ($i = (i + 1) \% n$) 模拟环形结构，确保遍历到最后一个人后能回到起点。

用 p 记录交替变化的报数状态 (0 和 1)，通过异或运算 ($p \wedge= 1$) 实现状态切换，简洁高效。

用 c 统计离开的人数，当 $c = n - 1$ 时 (只剩 1 人) 终止循环，避免多余计算。

F 数组用于记录每个人的状态 (在圈中 / 已离开)，是模拟过程的核心数据结构。

(1)①处应填 ()

A. $i < n$ B. $c < n$ C. $i < n-1$ D. $c < n-1$

答案：D

分析：循环的目的是持续处理直到只剩下 1 人，即已离开的人数 c 需小于 $n-1$ (总共需要离开 $n-1$ 人)。当 $c = n-1$ 时，循环终止。故选择 $c < n-1$ 。

(2)②处应填 ()

A. $i \% 2 == 0$ B. $i \% 2 == 1$ C. p D. $!p$

答案：C

分析： p 表示当前报数状态 (0 或 1)，题目要求“报到 1 的人离开”，因此当 p 为 1 时，当前人需要离开。条件应为 p (等价于 $p == 1$)。

(3)③处应填 ()

A. $i++$ B. $i = (i + 1) \% n$ C. $c++$ D. $p \wedge= 1$

答案：C

分析：当某人报到 1 并被标记为离开 ($F[i] = 1$) 后，需更新已离开的人数，因此执行 $c++$ 。

(4)④处应填 ()

A. i++ B. i = (i + 1) % n C. c++ D. p ^= 1

答案：D

分析：无论当前人是否离开，报数完成后都需要切换状态（0→1 或 1→0）。p ^= 1（异或运算）是切换 0 和 1 的简洁方式，故选择此操作。

(5) ⑤处应填（ ）

A. i++ B. i = (i + 1) % n C. c++ D. p ^= 1

答案：B

分析：循环移动到下一个人（环形），故 i = (i+1) % n。

```
#include <iostream>
```

```
using namespace std;
```

```
const int MAXN = 1000000;
```

```
int F[MAXN]; // 标记数组：F[i]=0 表示第 i 个人在圈中，F[i]=1 表示已离开
```

```
int main() {
```

```
    int n;
```

```
    cin >> n; // 输入总人数 n
```

```
    int i = 0; // 当前遍历到的人的编号（初始从 0 开始）
```

```
    int p = 0; // 报数状态（0 或 1 交替，初始为 0）
```

```
    int c = 0; // 已离开的人数（初始为 0）
```

```
    // 循环条件：当离开的人数小于 n-1 时（直到只剩 1 人）
```

```
    while (c < n - 1) { // ①处填 D
```

```
        if (F[i] == 0) { // 若当前人在圈中
```

```
            if (p) { // 若报数为 1（需要离开） ②处填 C
```

```
                F[i] = 1; // 标记为已离开
```

```
                c++; // 离开人数加 1 ③处填 C
```

```
            }
```

```
            p ^= 1; // 切换报数状态（0→1 或 1→0） ④处填 D
```

```

    }
    i = (i + 1) % n; // 移动到下一个人（圈的循环处理） ⑤处填 B
}

// 找到最后剩下的人（F[i]=0 的唯一元素）
int ans = -1;
for (i = 0; i < n; i++)
    if (F[i] == 0)
        ans = i;
cout << ans << endl;
return 0;
}

```

20.20.（矩形计数）平面上有 n 个关键点，求有多少个四条边都和 x 轴或者 y 轴平行的矩形，满足四个顶点都是关键点。给出的关键点可能有重复，但完全重合的矩形只计一次。

```

#include <iostream>
using namespace std;
struct point {
    int x, y, id;
};
bool equals(point a, point b) {
    return a.x == b.x && a.y == b.y;
}
bool cmp(point a, point b) {
    return ①;
}
void sort(point A[], int n) {
    for(int i=0;i<n;i++)
        for(int j=1;j<n;j++)

```

```

        if (cmp(A[j], A[j - 1])) {
            point t = A[j];
            A[j] = A[j - 1];
            A[j-1] = t;
        }
    }
}

int unique(point A[], int n) {
    int t=0;
    for(int i=0;i<n;i++)
        if (②)
            A[t++] = A[i];
    return t;
}

bool binary_search(point A[], int n, int x, int y) {
    point p;
    p.x = x;
    p.y = y;
    p.id = n;
    int a = 0, b = n-1;
    while(a < b) {
        int mid =③;
        if (④)
            a = mid + 1;
        else
            b = mid;
    }
    return equals(A[a], p);
}

const int MAXN = 1000;
point A[MAXN];

int main() {
    int n;

```

```

cin>>n;
for(int i=0;i<n;i++){
    cin >> A[i].x >> A[i].y;
    A[i].id = i;
}
sort(A, n);
n = unique(A, n);
int ans = 0;
for(int i=0;i<n;i++)
    for(int j=0;j<n;j++)
        if ((⑤&& binary_search(A, n, A[i].x, A[j].y) &&
            binary_search(A, n, A[j].x, A[i].y)) {
            ans++;
        }
cout << ans << endl;
return 0;
}

```

该程序用于计算平面上由关键点构成的、边平行于 x 轴和 y 轴的矩形数量。具体逻辑如下：

1. 数据预处理：读入所有关键点后，先按坐标排序（ x 优先， x 相同则 y 优先），再去除重复点（坐标完全相同的点只保留一个）。
2. 矩形判定：对于任意两个点 (x_1, y_1) 和 (x_2, y_2) （满足 $x_1 < x_2$ 且 $y_1 < y_2$ ），若另外两个点 (x_1, y_2) 和 (x_2, y_1) 也存在于关键点中，则这四个点构成一个符合条件的矩形。
3. 计数：通过枚举所有可能的对角点对，结合二分查找判断另外两个顶点是否存在，最终统计矩形总数。

```
#include <iostream>
```

```
using namespace std;
```

```
// 定义点结构体，包含  $x$ 、 $y$  坐标和标识  $id$ 
```

```
struct point {
```

```
    int x, y, id;
```

```
};
```


// 判断两个点是否完全重合（坐标相同）

```
bool equals(point a, point b) {  
    return a.x == b.x && a.y == b.y;  
}
```

// 排序比较函数：先按 x 升序，x 相同则按 y 升序

```
bool cmp(point a, point b) {  
    return a.x < b.x || (a.x == b.x && a.y < b.y); // ①  
}
```

// 冒泡排序：使用 cmp 函数对数组 A 排序

```
void sort(point A[], int n) {  
    for(int i=0;i<n;i++)  
        for(int j=1;j<n;j++)  
            if (cmp(A[j], A[j - 1])) { // 若 A[j]应在 A[j-1]前面，则交换  
                point t = A[j];  
                A[j] = A[j - 1];  
                A[j-1] = t;  
            }  
}
```

// 去重函数：保留不重复的点，返回去重后的数量

```
int unique(point A[], int n) {  
    int t=0; // 记录去重后数组的长度  
    for(int i=0;i<n;i++)  
        // 若为第一个点，或当前点与上一个保留的点不重复，则保留  
        if (t== 0 || !equals(A[i], A[t-1])) { // ②  
            A[t++] = A[i];  
        }  
    return t;  
}
```

// 二分查找：在排序后的数组 A 中查找是否存在点(x,y)

```
bool binary_search(point A[], int n, int x, int y) {  
    point p;  
    p.x = x;  
    p.y = y;  
    p.id = n; // id 不影响查找，仅占位  
    int a = 0, b = n-1; // 左右边界  
    while(a < b) {  
        int mid = (a + b) / 2; // ③ 计算中间位置  
        // 若 A[mid] 小于 p（按 cmp 规则），则 p 在右半部分，调整左边界  
        if (cmp(A[mid], p)) { // ④  
            a = mid + 1;  
        } else {  
            b = mid; // 否则 p 在左半部分，调整右边界  
        }  
    }  
    // 循环结束后 a==b，判断该位置是否为目标点  
    return equals(A[a], p);  
}
```

const int MAXN = 1000; // 最大关键点数量

point A[MAXN]; // 存储关键点的数组

```
int main() {  
    int n;  
    cin >> n; // 输入关键点数量  
    for(int i=0; i<n; i++){  
        cin >> A[i].x >> A[i].y; // 输入点的坐标  
        A[i].id = i; // 初始化 id（用于标识，去重后不重要）  
    }  
}
```

```

sort(A, n); // 对关键点排序
n = unique(A, n); // 去重, 更新有效点数量
int ans = 0; // 矩形数量计数器

// 枚举所有可能的两个点作为矩形的左下和右上顶点
for(int i=0;i<n;i++)
    for(int j=0;j<n;j++)
        // 条件: i 为左下顶点, j 为右上顶点 (x1<x2 且 y1<y2)
        // 且另外两个顶点(x1,y2)和(x2,y1)存在
        if (A[i].x < A[j].x && A[i].y < A[j].y && // ⑤
            binary_search(A, n, A[i].x, A[j].y) &&
            binary_search(A, n, A[j].x, A[i].y)) {
            ans++; // 满足条件, 计数加 1
        }
cout << ans << endl; // 输出矩形总数
return 0;
}

```

(1) ①处应填 ()

- A. $a.x \neq b.x ? a.x < b.x : a.id < b.id$
- B. $a.x \neq b.x ? a.x < b.x : a.y < b.y$
- C. $equals(a, b) ? a.id < b.id : a.x < b.x$
- D. $equals(a, b) ? a.id < b.id : (a.x \neq b.x ? a.x < b.x : a.y < b.y)$

答案: B

分析: `cmp` 函数用于排序, 需保证数组按 x 升序、 x 相同则按 y 升序排列。这样相同坐标的点会相邻 (便于去重), 且为二分查找提供有序基础。选项 B 符合此逻辑。

(2) ②处应填 ()

- A. $i == 0 \parallel cmp(A[i], A[i - 1])$

- B. `t == 0 || equals (A [i], A [t - 1])`
C. `i == 0 || !cmp (A [i], A [i - 1])`
D. `t == 0 || !equals (A [i], A [t - 1])`

答案：D

分析：unique 函数的作用是去除重复点。逻辑为：保留第一个点，后续仅保留与“上一个保留点”坐标不同的点。t 是当前保留点的数量，A[t-1]是上一个保留点，故条件为“t==0（第一个点）或当前点与上一个保留点不同”，选项 D 正确。

(3) ③处应填 ()

- A. `b - (b - a) / 2 + 1` B. `(a + b + 1) >> 1`
C. `(a + b) >> 1` D. `a + (b - a + 1) / 2`

答案：C

分析：二分查找中 mid 的计算需取当前区间[a,b]的中点。`(a + b) >> 1` 等价于 $(a + b) / 2$ ，是二分查找中计算中点的标准方式，选项 C 正确。

(4) ④处应填 ()

- A. `!cmp (A [mid], p)` B. `cmp (A [mid], p)`
C. `cmp (p, A [mid])` D. `!cmp (p, A [mid])`

答案：B

分析：二分查找需根据 cmp 规则判断 A[mid]与目标点 p 的位置关系。若 `cmp(A[mid], p)` 为 true，说明 A[mid]在 p 之前，需调整左边界 `a = mid + 1`；否则调整右边界。选项 B 符合此逻辑。

(5) ⑤处应填 ()

- A. `A [i].x == A [j].x` B. `A [i].id < A [j].id`
C. `A [i].x == A [j].x && A [i].id < A [j].id`
D. `A [i].x < A [j].x && A [i].y < A [j].y`

答案：D

分析：枚举的两个点(x1,y1)和(x2,y2)需作为矩形的对角点（如左下角和右上角），需满足 $x1 < x2$ 且 $y1 < y2$ ，以确保另外两个顶点(x1,y2)和(x2,y1)存在时构成唯一矩形，且避免重复计数。选项 D 正确。