2019 CSP-S 真题详细题解

一、单项选择题

1. 若有定义：int a=7; float x=2.5, y=4.7 ； 则表达式 x+a%3*(int)(x+y)%2 的值是：( )

A.0.000000 B.2.750000 C.2.500000 D.3.500000

答案：D

分析：本题考查运算符优先级和类型转换。

- 步骤 1：计算 a%3，即 7%3=1。
- 步骤 2：计算 x+y=2.5+4.7=7.2，强制转换为 int 得 7。
- 步骤 3：计算 1*7=7。
- 步骤 4：计算 7%2=1。

步骤 5：计算 x+1=2.5+1=3.5

2. 下列属于图像文件格式的有（）

A.WMV B.MPEG C.JPEG D.AVI

答案：C

分析：本题考查文件格式分类。

- A.WMV 是视频格式；B.MPEG 是视频格式；C.JPEG 是图像格式；D.AVI 是视频格式。

3. 二进制数 11 1011 1001 0111 和 01 0110 1110 1011 进行逻辑或运算的结果是（）

A. 11 1111 1101 1111 B. 11 1111 1111 1101

C. 10 1111 1111 1111 D. 11 1111 1111 1111

答案：D

分析：本题考查二进制或运算（逐位相或，有 1 则 1）。

两个二进制数逐位或运算后，每一位均为 1，结果为 11 1111 1111 1111。

4. 编译器的功能是（）

A. 将源程序重新组合

B. 将一种语言（通常是高级语言）翻译成另一种语言（通常是低级语言）

C. 将低级语言翻译成高级语言

D. 将一种编程语言翻译成自然语言

答案：B

分析：本题考查编译器的定义。

- 编译器的核心功能是将高级语言翻译为低级语言（如机器码）。

5. 设变量 x 为 float 型且已赋值，则以下语句中能将 x 中的数值保留到小数点后两位，并将第三位四舍五入的是（）

```
A. x=(x*100+0.5)/100.0          B. x=(int)(x*100+0.5)/100.0;
C. x=(x/100+0.5)*100.0          D. x=x*100+0.5/100.0;
```

答案：B

分析：本题考查四舍五入的实现。

- 步骤：x*100 将小数点后两位移到整数部分，+0.5 实现四舍五入，(int)截断小数部分，再/100.0 恢复小数点位置。

6. 由数字 1，1，2，4，8，8 所组成的不同的 4 位数的个数是（）

A.104　　B.102　　C.98　　D.100

答案：B

分析：本题考查排列组合（含重复元素）。

- 分情况讨论：
  - 无重复数字：从 4 个不同数字（1,2,4,8）选 4 个排列，A(4,4)=24。
  - 含两个 1：从 2,4,8 中选 2 个，与两个 1 排列，C(3,2)*4!/(2!)=3*12=36。

    非 1 的不同数字有 2、4、8（共 3 个），需从这 3 个中选 2 个，选法为：C(3,2)（即 2 和 4、2 和 8、4 和 8）。

    每组选法对应 4 个数字：2 个 1+2 个不同数字，因 2 个 1 重复，排列数为：4!/(2!)（分母 2! 是因 2 个 1 相同）。
  - 含两个 8：同理，36 种。
  - 含两个 1 和两个 8：4!/(2!2!)=6。
  - 总计：24+36+36+6=102。

7. 排序的算法很多，若按排序的稳定性和不稳定性分类，则（　）是不稳定排序。

A. 冒泡排序　　B. 直接插入排序　　C. 快速排序　　D. 归并排序

答案：C

分析：本题考查排序算法的稳定性。

- 快速排序在交换元素时可能改变相等元素的相对顺序，是不稳定排序。

8. G 是一个非连通无向图（没有重边和自环），共有 28 条边，则该图至少有（ ）个顶点

A.10 B.9 C.11 D.8

答案：B

分析：本题考查非连通图的顶点数计算。

- 要使顶点最少，需一个连通分量边数最多（接近完全图），其余为孤立点。
- 设连通分量有 n 个顶点，边数最多为 n(n-1)/2，需 n(n-1)/2 ≥28，解得 n≥8（8*7/2=28），加 1 个孤立点，共 9 个？但非连通需至少两个分量，故 8 个顶点的完全图（28 条边）加 3 个孤立点？不对，8 个顶点的完全图有 28 条边，若为非连通，至少需 9 个顶点（8 个成完全图，1 个孤立，共 28 条边）。但选项中 B 为 9，C 为 11

9. 一些数字可以颠倒过来看，例如 0、1、8 颠倒过来看还是本身，6 颠倒过来是 9 ，9 颠倒过来看还 是 6 ，其他数字颠倒过来都不构成数字。类似的，一些多位数也可以颠倒过来看，比如 106 颠倒过来是 901。假设某个城市的车牌只有 5 位数字，每一位都可以取 0 到 9。请问这个城市有多少个车牌倒过来恰 好还是原来的车牌，并且车牌上的 5 位数能被 3 整除？( )

A.40 B.25 C.30 D.20

答案：B

第 1、2 位有(0、1、8、6、9)五个数字，第 3 位有(0、1、8)三个数字，第 4、5 位由第 1、2 位决定。

由于 0,1,8,6,9 模 3 正好余 0,1,2,0,0，为了使得和所以其他位确定则第 3 位自然确定，共 5*5=25 种。

举例：第 1 位选 1,第 5 位必然是 1;第 2 位选 0,第 4 位必然是 0;为能被 3 整除，第 3 位只能选 8。

第 1 位可选 5 个数字，第 2 位也可选 5 个数字。5*5=25 种。

10. 一次期末考试，某班有 15 人数学得满分，有 12 人语文得满分，并且有 4 人语、数都是满分，那么这 个班至少有一门得满分的同学有多少人？( )

A.23 B.21 C.20 D.22

答案：A

分析：本题考查容斥原理。

- 至少一门满分人数 = 数学满分 + 语文满分 - 两科满分 = 15+12-4=23。

11. 设 A 和 B 是两个长为 n 的有序数组，现在需要将 A 和 B 合并成一个排好序的数组，请问任何以元素比较 作为基本运算的归并算法，在最坏情况下至少要做多少次比较？（ ）

A.n^2 B.n log n C.2n D.2n-1

答案：D

分析：本题考查归并排序的比较次数。

- 最坏情况下，两个数组交替元素，需比较 2n-1 次（如 A=1,3,5，B=2,4,6）。

12. 以下哪个结构可以用来存储图（ ）

A. 栈 B. 二叉树 C. 队列 D. 邻接矩阵

答案：D

分析：本题考查图的存储结构。

- 邻接矩阵是图的常用存储结构之一，栈、队列、二叉树均不直接用于存储图。

13. 以下哪些算法不属于贪心算法？（ ）

A.Dijkstra 算法 B.Floyd 算法 C.Prim 算法 D.Kruskal 算法

答案：B

分析：本题考查算法分类。

- Floyd 算法用于求多源最短路径，基于动态规划，不属于贪心算法。

14. 有一个等比数列，共有奇数项，其中第一项和最后一项分别是 2 和 118098 ，中间一项是 486 ，请 问一下哪个数是可能的公比？（ ）

A.5 B.3 C.4 D.2

答案：B

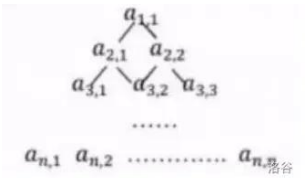
分析：本题考查等比数列性质。

- 设公比为 q，项数为 2k+1，则中间项为第 k+1 项：a1*q^k=486，最后一项 a1*q^(2k)=118098。
- 由(a1*q^k)^2 = a1*a1*q^(2k) → $486^2=2\times118098$ → 236196=236196，成立。
- 代入 2*q^k=486 → q^k=243=3^5，故 q=3，k=5，符合条件。

15. 有正实数构成的数字三角形排列形式如图所示。第一行的数为 $a_{1,1}$，第二行为 $a_{2,1}$ ， $a_{2,2}$ ，第 n 行的数 为 $a_{n,1}$ ，$a_{n,2}$ ，… ，$a_{n,n}$。从 $a_{1,1}$ 开始，每一行的数 $a_{i,j}$ 只有两条边可以分别通向下一行的两个数 $a_{i+1,j}$ 和 $a_{i+1,j+1}$。用动态规划算法找出一条从 $a_{1,1}$ 向下通道 $a_{n,1}$ ，$a_{n,2}$ ，… ，$a_{n,n}$ 中某个数的路径，使得该路径上的数之和最大。

令 C[i][j]是从 a1,1 到 ai,j 的路径上的数的最大和，并且 C[i][0]= C[0][j]=0 ，则 C[i][j]=（ ）。

A.max{C[i-1][j-1],C[i-1][j]}+ ai,j
B.C[i-1][j-1]+C[i-1][j]
C.max{C[i-1][j-1],c[i-1][j]}+1
D.max{C[i][j-1],C[i-1][j]}+ ai,j



答案：A

分析：本题考查动态规划在数字三角形中的应用。

- 到达 ai,j 的路径只能来自上方的 ai-1,j-1 或 ai-1,j，故最大和为两者中的最大值加 ai,j。

二、阅读程序题

```
01  #include <cstdio>
02  using namespace std;
03  int n;
04  int a[100];
05
06  int main() {
07      scanf("%d", &n);
08      for (int i = 1; i <= n; ++i)
09          scanf("%d", &a[i]);
10      int ans = 1;
11      for (int i = 1; i <= n; ++i) {
12          if (i > 1 && a[i] < a[i - 1])
13              ans = i;
14          while (ans < n && a[i] >= a[ans + 1])//  5 4 3 2 1
15              ++ans;                                i     ^
16          printf("%d\n", ans);
17      }
18      return 0;
19  }
```

●判断题

1) （ 1 分）第 16 行输出 ans 时，ans 的值一定大于 i。（ F ）
2) （ 1 分）程序输出的 ans 小于等于 n。（ T ）
3) 若将第 12 行的“<”改为“ ！=” ，程序输出的结果不会改变。（ ？）
4) 当程序执行到第 16 行时，若 ans-i>2 ，则 a[i+1]≦a[i]。（ T ）
5) （ 3 分）若输入的 a 数组是一个严格单调递增的数列，此程序的时间复杂度是（ D ）。

A. O(log n)　　B . O(n^2)　　C. O(nlog n)　　D. O(n)

6) 最坏情况下，此程序的时间复杂度是（ A ）。

A. O(n^2)　　B. O(log n)　　C. O(n)　　D. O(nlog n)

程序含义：找到每个 a[i]之后第一个大于 a[i]的位置(下标 ans+1)。

10

3 2 1 3 2 1 3 2 1 4

9 3 3 9 6 6 9 9 9 10

●判断题

1. 第 16 行输出 ans 时，ans 的值一定大于 i。( )

答案：×
分析：存在 ans 等于 i 的情况。例如，当 n=1 时，i=1，ans 初始为 1，while 循环不执行，输出 ans=1（等于 i）；又如 n=2 且 a=[1,2]，i=1 时，a [1]=1 < a [2]=2，while 循环不执行，ans=1（等于 i）。因此“一定大于 i”不成立。

2. 程序输出的 ans 小于等于 n。( )

答案：√
分析：ans 的初始值为 1，while 循环的条件是“ans < n”，即当 ans 达到 n 时循环停止。因此 ans 的最大值为 n，不可能超过 n，故输出的 ans 一定小于等于 n。

3. 若将第 12 行的“<”改为“！=”，程序输出的结果不会改变。( )

答案：√
分析：原逻辑中，第 12 行仅当 a [i] < a [i-1] 时重置 ans 为 i（因当前元素小于前一个，需重新计算右边界）。改为“!=”后，当 a [i] > a [i-1] 时也会重置 ans 为 i，但此时：

- 原逻辑中 ans 可能小于 i，进入 while 循环后会从 ans 开始扩展，最终会达到 i 并继续扩展；
- 修改后 ans 直接从 i 开始扩展，最终结果与原逻辑一致。
  当 a [i] == a [i-1] 时，两者均不重置 ans，结果也一致。因此输出结果不变。

4. 当程序执行到第 16 行时，若 ans-i>2 ，则 a [i+1]≦a [i]。( )

答案：√
分析：ans-i>2 意味着 ans ≥ i+3。ans 的扩展依赖 while 循环：只有 a [i] ≥ a [ans+1] 时，ans 才会增加。要使 ans 达到 i+3，必须满足：

- ans 从 i 开始，先因 a [i] ≥ a [i+1] 扩展到 i+1；
- 再因 a [i] ≥ a [i+2] 扩展到 i+2；
- 最后因 a [i] ≥ a [i+3] 扩展到 i+3。
  因此 a [i+1] 必然≤a [i]。

●选择题

5) 若输入的 a 数组是一个严格单调递增的数列，此程序的时间复杂度是（ ）。

A. O (log n)　　B . O (n²)　　C. O (nlog n)　　D. O (n)

答案：D

分析：严格单调递增时，a [i] < a [i+1]。对于每个 i：

第 12 行不执行（a [i] > a [i-1]）；
while 循环仅执行 1 次（ans 从 i-1 扩展到 i，因 a [i] < a [i+1] 停止）。
总循环次数为 O (n)，故时间复杂度为 O (n)。

1. 最坏情况下，此程序的时间复杂度是（　　）。

A. O (n^2)　　B. O (log n)　　C. O (n)　　D. O (nlog n)

答案：A

答案：A（O (n^2)）
分析：最坏情况为数组严格单调递减（a [i] > a [i+1]）。此时：

- i=1 时，while 循环执行 n-1 次（ans 从 1 扩展到 n）；
- i=2 时，while 循环执行 n-2 次（ans 从 2 扩展到 n）；
- ...
- 总次数为 1+2+...+(n-1) = n (n-1)/2 = O (n^2)。

```
#include <cstdio>     // 包含标准输入输出库
using namespace std;   // 使用标准命名空间
int n;   // 存储数组元素个数
int a[100];   // 存储输入的数组（索引从 1 开始）

int main() {
    scanf("%d", &n);   // 读取元素个数 n
    for (int i = 1; i <= n; ++i)   // 读取 n 个元素到数组 a 中
        scanf("%d", &a[i]);
    int ans = 1;   // 记录当前 i 对应的右边界
    for (int i = 1; i <= n; ++i) {   // 遍历每个元素
        // 若当前元素小于前一个，重置右边界为当前位置
        if (i > 1 && a[i] < a[i - 1])
            ans = i;
        // 扩展右边界：只要下一个元素不大于当前元素，就继续扩展
        while (ans < n && a[i] >= a[ans + 1])
```

```
            ++ans;
        printf("%d\n", ans);  // 输出当前 i 对应的最大右边界
    }
    return 0;
}
```

2.

```
01  #include <iostream>
02  using namespace std;
03
04  const int maxn = 1000;
05  int n;
06  int fa[maxn], cnt[maxn];
07
08  int getRoot(int v) {
09      if (fa[v] == v) return v;
10      return getRoot(fa[v]);
11  }
12
13  int main() {
14      cin >> n;
15      for (int i = 0; i < n; ++i) {
16          fa[i] = i;
17          cnt[i] = 1;
18      }
19      int ans = 0;
20      for (int i = 0; i < n - 1; ++i) {
21          int a, b, x, y;
22          cin >> a >> b;
23          x = getRoot(a);
24          y = getRoot(b);
25          ans += cnt[x] * cnt[y];
26          fa[x] = y;
27          cnt[y] += cnt[x];
28      }
29      cout << ans << endl;
30      return 0;
31  }
```

●判断题

1) (1 分)输入的 a 和 b 值应在[0 ， n-1]的范围内。(T)

2) (1 分)第 16 行改成“fa[i]=0;”,不影响程序运行结果。(F)

3) 若输入的 a 和 b 值均在[0,n-1]的范围内,则对于任意 0≤i＜n,都有 0≤fa[i]＜n。(T)

4）若输入的 a 和 b 值均在[0,n-1]的范围内,则对于任意≤i＜n,都有 0≤cnt[i] ≤n。(F)

●选择题

5) 当 n 等于 50 时，若 a,b 的值都在[0,49]的范围内，且在第 25 行时总是 x 不等于 y ，那么输出为( )

A. 1276　　B. 1176　　C.1225　　D.1250

6) 此程序的时间复杂度是( )

A. O(n)　　B. O(log n)　　C. O(n^2)　　D.O(n log n)

程序含义：该程序使用并查集处理一棵树的构建过程，并计算一个特定值 ans。程序读取节点数 n，然后读取 n-1 条边（树的结构）。每条边连接两个节点，程序通过并查集维护每个子树的节点个数，并在每次合并分量时，将两个分量大小的乘积加到 ans 中。最终，ans 表示树中每条边对两个子树大小的乘积

●判断题

1. 输入的 a 和 b 值应在 [0 ， n-1] 的范围内。(　　)

答案：√

分析：程序中数组 fa 和 cnt 的索引范围为 0 到 n-1（第 15 行循环 i 从 0 到 n-1）。若 a 或 b 超出此范围，访问 fa[a]或 fa[b]会导致数组越界，程序可能崩溃或产生错误结果。因此输入的 a 和 b 必须在[0, n-1]范围内。

2. 第 16 行改成 “fa [i]=0;”，不影响程序运行结果。(　　)

答案：×

分析：原代码中 fa[i] = i 是并查集的初始化，每个元素初始为自身的根节点，保证初始时每个元素独立成集。若改为 fa[i] = 0，则所有元素初始根节点均为 0（即所有元素一开始就在同一集合中）。后续合并时，x 和 y 会频繁相等（因根相同），导致 ans 的计算逻辑完全改变（无法累加不同集合的大小乘积），最终结果必然不同。

3. 若输入的 a 和 b 值均在 [0,n-1] 的范围内，则对于任意 0≤i < n，都有 0≤fa [i] < n。(　　)

答案：√

分析：fa[i]表示的是 i 的父节点的编号，满足 0≤fa [i] < n。

4. 若输入的 a 和 b 值均在 [0,n-1] 的范围内，则对于任意 0≤i < n，都有 1≤cnt [i] ≤n。(　　)

答案：×

分析：初始化 cnt[i] = 1。合并时只更新根节点的 cnt（cnt[y] += cnt[x]），根节点的 cnt 最小为 1（单节点），最大为 n（所有节点合并）。非根节点的 cnt 保持初始值 1。因此所有 cnt[i] 均在 [1, n]。

cnf[i]是合并之后集合的元素个数，严谨的并查集操作 cnt[i]是不会超过 n 的但是本题没有判断是否重复合并。所以如果先输入(1,2)， (3,4),之后一直不断输入(2,4),最后 cnt[4]会超过 n。

●选择题

5) 当 n 等于 50 时，若 a,b 的值都在 [0,49] 的范围内，且在第 25 行时总是 x 不等于 y ，那么输出为（ ）

A. 1276　　B. 1176　　C.1225　　D.1250

答案：C

分析：x 始终不等于 y 意味着每次合并的是两个不同的集合。程序中 ans 累加的是每次合并的两个集合的大小乘积。对于 n 个元素，将其逐步合并为一个集合的过程中，该乘积和为固定值 n*(n-1)/2（推导：每次合并相当于计算新连接的两个集合中元素的两两组合数，总组合数为所有元素的两两组合数）。当 n=50 时，50*49/2 = 1225。

cnt[i]表示当前集合的元素个数，每次执行都是两个集合合并，我们可尝试举例，比如

令每次都是单个集合合并进入大集合，

ans=1*1+1*2+1*3+.....1*48+1*49=49*(49+1)/2=1225。

6) 此程序的时间复杂度是（）

A. O (n)　　B. O (log n)　　C. O (n^2)　　D.O (n log n)

答案：C

分析：程序的核心是 getRoot 函数（查找根节点）和 n-1 次合并操作。由于 getRoot 未实现路径压缩优化，最坏情况下（如元素成链状），每次查找根节点的时间复杂度为 O(n)。因此 n-1 次合并的总时间复杂度为 O(n^2)。

```
#include <iostream>
using namespace std;

const int maxn = 1000;   // 定义数组最大长度
int n;                   // 元素个数
int fa[maxn];            // 并查集的父节点数组，fa[v]表示 v 的父节点
int cnt[maxn];           // 记录每个根节点所在集合的元素个数

// 查找 v 所在集合的根节点（递归实现）
int getRoot(int v) {
    if (fa[v] == v) return v;   // 若 v 是根节点，直接返回
```

```
    return getRoot(fa[v]);        // 否则递归查找父节点的根
}

int main() {
    cin >> n;   // 输入元素个数
    // 初始化并查集：每个元素自身为根，集合大小为 1
    for (int i = 0; i < n; ++i) {
        fa[i] = i;      // 父节点指向自身
        cnt[i] = 1;     // 初始集合大小为 1
    }
    int ans = 0;   // 累加每次合并的集合大小乘积
    // 进行 n-1 次合并（形成一棵树）
    for (int i = 0; i < n - 1; ++i) {
        int a, b, x, y;
        cin >> a >> b;           // 输入待合并的两个元素
        x = getRoot(a);          // 查找 a 所在集合的根
        y = getRoot(b);          // 查找 b 所在集合的根
        ans += cnt[x] * cnt[y];   // 累加两个集合大小的乘积
        fa[x] = y;                // 将 x 的根合并到 y 的根下
        cnt[y] += cnt[x];       // 更新 y 所在集合的大小
    }
    cout << ans << endl;   // 输出累加的结果
    return 0;
}
```

3. t 是 s 的子序列的意思是：从 s 中删去若干个字符，可以得到 t；特别的，如果 s=t，那么 t 也是 s 的子序列；空串是任何串的子序列。例如“acd”是“abcde”的子序列，“acd”是“acd”的子序列，但“adc”不是“abcde”的子序列。
S[x..y]表示 s[x]…s[y]共 y-x+1 个字符构成的字符串，若 x＞y 则 s[x..y]是空串。t[x..y]同理。

```
01  #include <iostream>
02  #include <string>
03  using namespace std;
04  const int max1 = 202;
05  string s, t;
06  int pre[max1], suf[max1];
07
```

```
08  int main() {
09      cin >> s >> t;
10      int slen = s.length(), tlen = t.length();
11
12      for (int i = 0, j = 0; i < slen; ++i) {
13          if (j < tlen && s[i] == t[j]) ++j;
14          pre[i] = j; // t[0..j-1] 是 s[0..i] 的子序列
15      }
16
17      for (int  i = slen - 1 , j = tlen - 1; i >= 0; --i) {
18          if(j >= 0 && s[i] == t [j]) --j;
19          suf[i]= j; // t[j+1..tlen-1] 是 s[i..slen-1] 的子序列
20      }
21
22      suf[slen] = tlen -1;
23      int ans = 0;
24      for (int i = 0, j = 0, tmp = 0; i <= slen; ++i){
25          while(j <= slen && tmp >= suf[j] + 1) ++j;
26          ans = max(ans, j - i - 1);
27          tmp = pre[i];


28      }
29      cout << ans << endl;
30      return 0;
31  }
```

提示：
t[0..pre[i]-1]是 s[0..i]的子序列；
t[suf[i]+1..tlen-1]是 s[i..slen-1]的子序列

●判断题
1.(1 分)程序输出时，suf 数组满足：对任意 0≤i＜slen ， suf[i] ≤suf[i+1]。(T)
2.(2 分) 当 t 是 s 的子序列时，输出一定不为 0。 ( )
3.(2 分)程序运行到第 23 行时，“j-i-1”一定不小于 0。 (F)
4.(2 分)当 t 时 s 的子序列时，pre 数组和 suf 数组满足：对任意 0≤i＜slen ，pre[i]＞suf[i+1]。( )

●选择题
5.若 tlen=10 ， 输出为 0 ， 则 slen 最小为( )
A. 10 B. 12 C.0 D.1
6.若 tlen=10 ， 输出为 2 ， 则 slen 最小为( )
A. 0 B.10 C.12 D.1

该程序的核心目标是：在字符串 s 中找到一段最长的连续子串，使得删除这段子串后，剩余部分（即 s 中该子串左侧的前缀与右侧的后缀拼接而成的字符串）仍然包含 t 作为子序列。最终输出这段最长可删除子串的长度。

具体逻辑拆解

核心问题转化

要删除 s 中的一段子串 s[i+1..j-1]（i < j），剩余部分为 s[0..i] + s[j..slen-1]。需满足：t 是这个剩余字符串的子序列。

程序通过**拆分匹配任务**实现这一判断：

1. t 的前缀部分（t[0..a-1]）由 s[0..i]匹配；
2. t 的后缀部分（t[b..tlen-1]）由 s[j..slen-1]匹配；
3. 当 a > b 时，前缀和后缀可拼接覆盖整个 t（即 t 是剩余字符串的子序列）。

abcdefsfdsg

acefg

pre 数组的作用

pre[i]表示：在 s[0..i]（s 从开头到第 i 个字符的前缀）中，能匹配到的 t 的**最长前缀长度（这个长度就是上面提到的 a，因此可以理解为字符串 t 的前缀的右边界）**。

1. 例如，若 pre[5] = 3，说明 s[0..5]包含 t[0..2]作为子序列（匹配了 t 的前 3 个字符，值 3 就是**字符串 t 的前缀的右边界**）。
2. 计算方式：通过双指针遍历 s 和 t，当 s[i]与 t[j]匹配时，j 递增，最终 pre[i]记录 j 的值 。

suf 数组的作用

suf[i]表示：在 s[i..slen-1]（s 从第 i 个字符到结尾的后缀）中，能匹配到的 t 的**最短后缀起始索引（就是上面提到的 b，因此可以理解为字符串 t 的后缀的左边界）**。

1. 例如，若 suf[4] = 2，说明 s[4..slen-1]包含 t[3..tlen-1]作为子序列（从 t 的第 3 个字符开始匹配，题目用 suf[j]+1 表示**字符串 t 的后缀的左边界**）。
2. 计算方式：从 s 末尾反向遍历，当 s[i]与 t[j]匹配时，j 递减，最终 suf[i]记录 j 的值。

寻找最长可删除子串

程序通过双指针 i 和 j 遍历所有可能的删除区间：

1. i 表示删除区间的左边界（s[0..i]为左侧剩余部分）；
2. j 表示删除区间的右边界（s[j..slen-1]为右侧剩余部分）；

   （左边界） （右边界） ，对应代码：tmp >= suf[j] + 1

3. 当 pre[i] > suf[j]时（即左侧匹配的前缀长度覆盖了右侧匹配的后缀起始位置），s[i+1..j-1]是可删除子串，其长度为 j - i - 1；
4. 循环中不断更新 i 和 j，记录最大的可删除长度 ans。

举例说明

示例 1: s = "abcde", t = "ace"

计算 pre 数组：pre[i] 表示从 s 的开头到位置 i 的子序列中，能匹配 t 的前缀的最大长度

i=0: s[0]='a'匹配 t[0]='a', j=1 → pre[0]=1 （对应代码：j++, pre[i] = j）
i=1: s[1]='b'不匹配 t[1]='c', j=1 → pre[1]=1
i=2: s[2]='c'匹配 t[1]='c', j=2 → pre[2]=2
i=3: s[3]='d'不匹配 t[2]='e', j=2 → pre[3]=2
i=4: s[4]='e'匹配 t[2]='e', j=3 → pre[4]=3

pre = [1, 1, 2, 2, 3]

计算 suf 数组：
suf[i] 表示从 s 的位置 i 到末尾的子序列中，能匹配 t 的后缀时，匹配的 t 的索引（从后往前匹配）。

初始化 i=4, j=2
s = "abcde", t = "ace"
i j

i=4: s[4]='e'匹配 t[2]='e', j=1(左移一个位置,相当于 j-1) → suf[4]=1 (对应代码：j--,suf[i]= j)
i=3: s[3]='d'不匹配 t[1]='c', j=1 → suf[3]=1
i=2: s[2]='c'匹配 t[1]='c', j=0 → suf[2]=0
i=1: s[1]='b'不匹配 t[0]='a', j=0 → suf[1]=0
i=0: s[0]='a'匹配 t[0]='a', j=-1 → suf[0]=-1

设置 suf[5]=2 (tlen-1)
suf = [-1, 0, 0, 1, 1]，且 suf[5]=2

s = "abcde", t = "ace"
pre = [1, 1, 2, 2, 3]
suf = [-1, 0, 0, 1, 1]

abcde ace

示例 2: s = "abacaba", t = "aba"

计算 pre 数组：
i=0: s[0]='a'匹配 t[0]='a', j=1 → pre[0]=1
i=1: s[1]='b'匹配 t[1]='b', j=2 → pre[1]=2
i=2: s[2]='a'匹配 t[2]='a', j=3 → pre[2]=3
i=3: s[3]='c'不匹配, j=3 → pre[3]=3
i=4: s[4]='a'不匹配, j=3 → pre[4]=3
i=5: s[5]='b'不匹配, j=3 → pre[5]=3
i=6: s[6]='a'不匹配, j=3 → pre[6]=3

pre = [1, 2, 3, 3, 3, 3, 3]

计算 suf 数组：
初始化 i=6, j=2

i=6: s[6]='a'匹配 t[2]='a', j=1 → suf[6]=1
i=5: s[5]='b'匹配 t[1]='b', j=0 → suf[5]=0
i=4: s[4]='a'匹配 t[0]='a', j=-1 → suf[4]=-1
i=3: s[3]='c'不匹配, j=-1 → suf[3]=-1
i=2: s[2]='a'不匹配, j=-1 → suf[2]=-1
i=1: s[1]='b'不匹配, j=-1 → suf[1]=-1
i=0: s[0]='a'不匹配, j=-1 → suf[0]=-1
设置 suf[7]=2 (tlen-1)
suf = [-1, -1, -1, -1, -1, 0, 1]，且 suf[7]=2

最终 ans=4。

通过这种拆分匹配 + 双指针遍历的方式，程序高效地找到最长可删除子串，时间复杂度为 O(len(s) + len(t))，适用于中小型字符串的处理。

●判断题

1. 程序输出时，suf 数组满足：对任意 $0 \le i < slen$ ，suf [i] ≤suf [i+1]。( )

答案：√
分析：suf 数组的计算逻辑是从 s 的末尾向开头遍历（i 从 slen-1 到 0）。对于 i 和 i+1（i < i+1）：

- 若 s [i] 与 t [suf [i+1]] 匹配，则 suf [i] = suf [i+1] - 1（小于 suf [i+1]）；
- 若不匹配，则 suf [i] = suf [i+1]（等于 suf [i+1]）。
  因此，始终有 suf [i] ≤ suf [i+1]。

2. 当 t 是 s 的子序列时，输出一定不为 0。 ( )

答案：×
分析：当 s = t 时，t 是 s 的子序列，但此时 s 中没有可删除的字符（删除任何字符都会导致剩余部分无法包含 t）。程序会计算出最长可删除子串长度为 0，故输出为 0。因此“输出一定不为 0”不成立。

3. 程序运行到第 23 行时，“j-i-1”一定不小于 0。 ( )

答案：×
分析：第 23 行是 ans 的初始化（ans=0），后续循环中 j-i-1 可能为负数。例如，初始时 i=0、j=0，此时 j-i-1 = -1，但 ans 会取 max(0, -1) = 0。因此“一定不小于 0”不成立。

4. 当 t 是 s 的子序列时，pre 数组和 suf 数组满足：对任意 $0 \le i < slen$ ，pre [i] > suf [i+1]。( )

答案：×
分析：当 t 是 s 的子序列时，存在某些 i 使得 pre[i] ≤ suf[i+1]。例如，设 s="abc"，t="abc"：

- pre[1]（s [0..1] 匹配 t 的前缀长度）为 1（匹配 "a"）；
- suf[2]（s [2..2] 匹配 t 的后缀起始位置）为 1（t [2] 由 s [2] 匹配，故 suf[2]=1）。
  此时 pre[1] = 1，suf[2] = 1，显然 pre[i] > suf[i+1]不成立。

●选择题

5) 若 tlen=10 ， 输出为 0 ， 则 slen 最小为 ( )

A. 10　　B. 12　　C.0　　D.1

答案：D

分析：若 t 不是 S 子串(或==s)输出都为 0，但为保证程序执行，最少应输入一个字符。

比如 s="a",t="bbbbbbbbbb"

6） 若 tlen=10 ， 输出为 2 ， 则 slen 最小为 ( )

A. 0　　B.10　　C.12　　D.1

答案：C

分析：输出为 2 表示存在长度为 2 的可删除子串，删除后剩余部分可包含 t 作为子序列。因此 s 需包含 t 的全部 10 个字符，加上 2 个可删除字符，故 slen 最小为 10 + 2 = 12。

```
#include <iostream>
#include <string>
using namespace std;
const int max1 = 202;   // 定义最大长度
string s, t;                  // 输入的两个字符串
int pre[max1], suf[max1];   // pre 记录前缀匹配长度，suf 记录后缀匹配位置

int main() {
    cin >> s >> t;
    int slen = s.length(), tlen = t.length();

    // 计算 pre 数组：pre[i]表示 t[0..pre[i]-1]是 s[0..i]的子序列
    for (int i = 0, j = 0; i < slen; ++i) {
        if (j < tlen && s[i] == t[j]) ++j;   // 匹配 t 的下一个字符
        pre[i] = j;   // 记录当前匹配到的 t 的前缀长度
    }

    // 计算 suf 数组：suf[i]表示 t[suf[i]+1..tlen-1]是 s[i..slen-1]的子序列
    for (int i = slen - 1, j = tlen - 1; i >= 0; --i) {
```

```
        if (j >= 0 && s[i] == t[j]) --j;   // 匹配 t 的前一个字符
        suf[i] = j;   // 记录当前匹配到的 t 的后缀起始位置
    }

    suf[slen] = tlen - 1;   // 处理 s 为空串的情况（s[slen..slen-1]是空串）
    int ans = 0;   // 存储最长可删除子串的长度

    // 寻找最长可删除子串：s[i+1..j-1]，需满足 t 可由 s[0..i]和 s[j..slen-1]的子序列拼接而成
    for (int i = 0, j = 0, tmp = 0; i <= slen; ++i) {
        // 扩展 j，使得 t 的前缀（匹配到 tmp）和后缀（从 suf[j]+1 开始）能覆盖 t
        while (j <= slen && tmp >= suf[j] + 1) ++j;
        ans = max(ans, j - i - 1);   // 更新最长可删除长度
        tmp = pre[i];   // 即前一个 i 的 pre 值,更新当前前缀匹配长度
    }

    cout << ans << endl;   // 输出结果
    return 0;
}
```

三、完善程序题

1.（匠人的自我修养）一个匠人决定要学习 n 个新技术，要想成功学习一个新技术，他不仅要拥有一定的 经验值，而且还必须要先学会若干个相关的技术。学会一个新技术之后，他的经验值会增加一个对应的值。给定每个技术的学习条件和习得后获得的经验值，给定他已有的经验值，请问他最多能学会多少个新技术。

输入第一行有两个数，分别为新技术个数 n（$1 \leqslant n \leqslant 10^3$），以及已有经验值（$\leqslant 10^7$）。

接下来 n 行。第 i 行的两个整数，分别表示学习第 i 个技术所需的最低经验值（$\leqslant 10^7$），以及学会第 i 个技 术后可获得的经验值（$\leqslant 10^4$ ）。

接下来 n 行。第 i 行的第一个数 m_i（$0 \leqslant m_i < n$），表示第 i 个技术的相关技术数量。紧跟着 m 个两两不同的数，表示第 i 个技术的相关技术编号，输出最多能学会的新技术个数。

下面的程序以 O（ n^2 ）的时间复杂完成这个问题，试补全程序。

```
01  #include<cstdio>
02  using namespace std;
03  const int maxn = 1001;
04
05  int n;
06  int cnt[maxn];
07  int child [maxn][maxn];
08  int unlock[maxn];
```

```
09  int threshold[maxn], bonus[maxn];
10  int points;
11  bool find(){
12      int target = -1;
13      for (int i = 1; i <= n; ++i)
14          if(① && ②){
15              target = i;
16              break;
17      }
18      if(target == -1)
19          return false;
20      unlock[target] = -1;
21      ③
22      for (int i = 0; i < cnt[target]; ++i)
23          ④
24      return true;
25  }
26
27  int main(){
28      scanf("%d%d", &n, &points);
29      for (int i = 1; i <= n; ++i){
30          cnt[i] = 0;
31          scanf("%d%d", &threshold[i], &bonus[i]);
32      }
33      for (int i = 1; i <= n; ++i){
34          int m;
35          scanf("%d", &m);
36          ⑤
37          for (int j = 0; j < m; ++j){
38              int fa;
39              scanf("%d", &fa);
40              child[fa][cnt[fa]] = i;
41              ++cnt[fa];
42          }
43      }
44
45      int ans = 0;
46      while(find())
47          ++ans;
48
49      printf("%d\n", ans);
50      return 0;
51  }
```

程序通过模拟“匠人学习技术”的过程，计算在给定初始经验值和技术依赖关系的条件下，最多能学会的新技术数量。具体流程：

1. 初始化每个技术的前置条件数量（unlock[i] = m）和依赖关系（child 数组）。
2. 反复调用 find()函数寻找可学习的技术（前置条件满足且经验值达标）。

3. 学会技术后，更新经验值并减少依赖该技术的子技术的前置条件数量。
4. 直到没有可学习的技术，输出学会的技术总数。

```
#include<cstdio>
using namespace std;
const int maxn = 1001;

int n;                          // 新技术的总个数
int cnt[maxn];                  // cnt[fa]表示以 fa 为前置技术的子技术数量（即 child[fa]的长度）
int child[maxn][maxn];          // child[fa][i]表示第 i 个依赖 fa 作为前置技术的子技术
int unlock[maxn];               // unlock[i]表示技术 i 还需多少个前置技术未被学会（0 表示前置条件满足）
int threshold[maxn];            // threshold[i]表示学习技术 i 所需的最低经验值
int bonus[maxn];                // bonus[i]表示学会技术 i 后获得的经验值
int points;                     // 当前拥有的经验值

// 寻找一个当前可学习的技术，若找到则处理并返回 true，否则返回 false
bool find(){
    int target = -1;      // 存储找到的可学习技术的编号
    // 遍历所有技术，寻找可学习的目标
    for (int i = 1; i <= n; ++i)
        // 条件 1：前置条件已满足（unlock[i] == 0）且未被学会（未被标记为-1）
        // 条件 2：当前经验值满足学习要求（points >= threshold[i]）
            //① && ②
        if(unlock[i] == 0 && points >= threshold[i]){
            target = i;
            break;
        }

    if(target == -1)      // 没有可学习的技术
        return false;

    unlock[target] = -1;   // 标记技术 target 已被学会
    points += bonus[target];    //③ 学会后增加经验值

    // 遍历所有依赖 target 作为前置技术的子技术，更新它们的前置条件状态
    for (int i = 0; i < cnt[target]; ++i){
        unlock[child[target][i]] -= 1;   // ④子技术的未满足前置数量减 1
    }
    return true;
}

int main(){
    scanf("%d%d", &n, &points);   // 读取技术数量和初始经验值

    // 读取每个技术的学习门槛和学会后的经验值奖励
    for (int i = 1; i <= n; ++i){
        cnt[i] = 0;   // 初始化子技术数量为 0
        scanf("%d%d", &threshold[i], &bonus[i]);
        // threshold[i]表示学习技术 i 所需的最低经验值
            // bonus[i]表示学会技术 i 后获得的经验值
    }
```

```
    // 读取每个技术的前置技术，并构建依赖关系
    for (int i = 1; i <= n; ++i){
        int m;
        scanf("%d", &m);   // 技术 i 的前置技术数量
        unlock[i] = m;      //  ⑤初始化 unlock[i]为前置技术数量（需满足 m 个前置）

        // 遍历每个前置技术 fa，将 i 添加到 fa 的子技术列表中
        for (int j = 0; j < m; ++j){
            int fa;
            scanf("%d", &fa);
            child[fa][cnt[fa]] = i;   // 记录 fa 的子技术 i
            ++cnt[fa];   // fa 的子技术数量加 1
        }
    }

    int ans = 0;   // 记录最多可学会的技术数量
    // 循环寻找可学习的技术，每学会一个就累加计数
    while(find())
        ++ans;

    printf("%d\n", ans);   // 输出结果
    return 0;
}
```

1. ①处应填（）

A.unlock [i] < =0　　B.unlock [i] > =0　　C.unlock [i]==0　　D.unlock [i]==-1

答案：C

分析: unlock[i]用于记录技术 i 的前置技术未学会的数量。初始时 unlock[i]等于前置技术总数 m，每学会一个前置技术，unlock[i]减 1。当 unlock[i]==0 时，说明所有前置技术都已学会（前置条件满足）。find()函数需要寻找“可学习”的技术，首先需满足前置条件，因此①处应为 unlock[i]==0。

2. ②处应填（）

A.threshold [i] > points　　B.threshold [i] > =points

C.points > threshold [i]　　D.points > =threshold [i]

答案：D

分析: 学习技术 i 的第二个条件是当前经验值足够。threshold[i]是学习技术 i 所需的最低经验值，因此需满足 points >= threshold[i]（当前经验值不低于最低要求），故②处应为此条件。

3. ③处应填（）

A.target = -1　　B.--cnt [target]

C. bonus [target]　　D.points+= bonus [target]

答案：D

分析：学会技术 target 后，匠人会获得对应的经验值 bonus[target]，因此需要更新当前经验值 points，使其增加 bonus[target]。其他选项中，A（重置 target）、B（减少计数）、C（语法错误）均不符合逻辑，故③处正确答案为 D

4. ④处应填（）

A. cnt [child [target][i]]-=1　　B. cnt [child [target][i]]=0

C. unlock [child [target][i]]-=1　　D. unlock [child [target][i]] =0

答案：C

分析:child[target][i]表示以 target 为前置技术的子技术(即必须先学会 target 才能学习的技术)。当 target 被学会后，这些子技术的“未学会前置技术数量”应减少 1，因此需将其 unlock 值减 1。选项 C 符合此逻辑。

5. ⑤处应填（）

A. unlock [i] = cnt [i]　　B. unlock [i] =m

C. unlock [i] = 0　　D. unlock [i] =-1

答案：B

分析：m 是技术 i 的前置技术总数。unlock[i]的初始值应等于前置技术总数（表示有 m 个前置技术未学会），后续每学会一个前置技术，unlock[i]减 1。因此读入 m 后，需将 unlock[i]初始化为 m，选项 B 正确。

2.（取石子）Alice 和 Bob 两个人在玩取石子游戏，他们制定了 n 条取石子的规则，第 i 条规则为：如果剩 余的石子个数大于等于 a[i]且大于等于 b[i] ，那么她们可以取走 b[i]个石子。他们轮流取石子。如果轮到某个人取石子，而她们无法按照任何规则取走石子，那么他就输了，一开始石子有 m 个。请问先取石子的人是否有必胜的方法？
输入第一行有两个正整数，分别为规则个数 n(1≤n≤64),以及石子个数 m(≤10^7）。接下来 n 行。第 i 行有两个正整数 a[i]和 b[i]。(1≤a[i]≤10^7,1b[i]≤ 64)
如果先取石子的人必胜，那么输出“Win”，否则输出 “Loss”。
提示：
可以使用动态规划解决这个问题。由于 b[i]不超过，所以可以使用位无符号整数去压缩必要的状态。

Status 是胜负状态的二进制压缩，trans 是状态转移的二进制压缩。
试补全程序。
代码说明:
“~”表示二进制补码运算符，它将每个二进制位的 0 变成 1、1 变为 0 ；
而“^”表示二进制异或运算符，它将两个参与运算的数重的每个对应的二进制位一一进行比较，若两个二进制位相同，则运算结果的对应二进制位为 0 ，反之为 1。
Ull 标识符表示它前面的数字是 unsigned long long 类型。

```
01  #include <cstdio>
02  #include<algorithm>
03  using namespace std;
04  const int maxn = 64;
05  int n, m;
06  int a[maxn], b[maxn];
07  unsigned long long status, trans;
08  bool win;
09  int main(){
10      scanf("%d%d", &n, &m);
11      for (int i = 0; i < n; ++i)
12          scanf("%d%d", &a[i], &b[i]);
13      for(int i = 0; i < n; ++i)
14          for(int j = i + 1; j < n; ++j)
15              if (a[i] > a[j]){
16                  swap(a[i], a[j]);
17                  swap(b[i], b[j]);
18              }
19      status = ①;
20      trans = 0;
21      for(int i = 1, j = 0; i <= m; ++i){
22          while (j < n && ②){
23              ③;
24              ++j;
25          }
26          win = ④;
27          ⑤;
28      }
29
30      puts(win ? "Win" : "Loss");
31
32      return 0;
33  }
```

1)①处应填(　　)
A.0　　B . ~0ull　　C. ~0ull^1　　D.　1
2 ）处应填(　　)
A.a[j]< i　　B.a[j] ==i　　C.a[j] !=i　　D.a[j] >i
3)③处应填(　　)
A.trans　|= 1ull <<(b[j] - 1)　　B.status |=1ull << (b[j]- 1)

C. status +=1ull << (b[j]-1)　　D. trans +=1ull<< (b[j]-1)

4)④处应填(　　)

A. ~status | trans　　B. status & trans

B. status | trans　　D. ~status & trans

5 ）⑤处应填(　　)

A. trans = status | trans ^win　　B. status = trans >> 1^win

C. trans = status ^trans |win　　D. status =status <<1^win

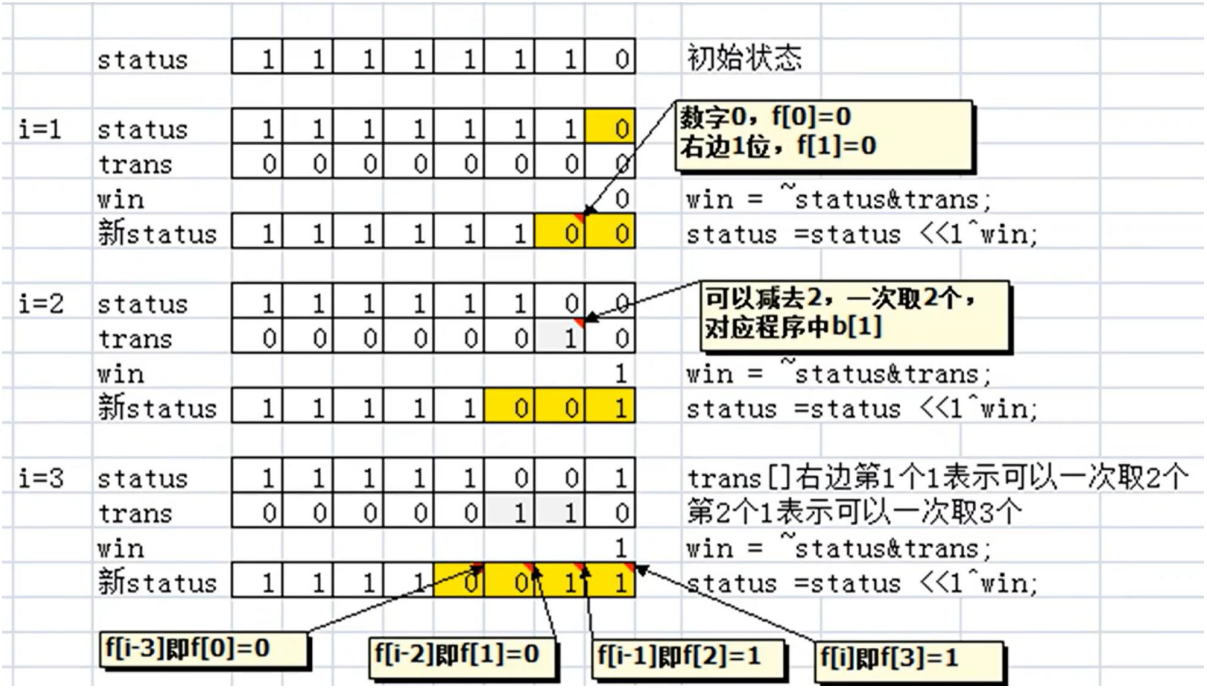
该程序用于判断取石子游戏中先取者是否有必胜策略。核心逻辑基于动态规划，通过位运算压缩状态以高效处理：

1. **状态定义**：dp[i]表示剩余 i 个石子时当前玩家是否必胜（true 为胜，false 为败）。
2. **转移规则**：dp[i] = true 当且仅当存在一种取法 b[j]（满足 i >= a[j]且 i >= b[j]），使得 dp[i - b[j]] = false（即对方必败）。
3. **优化**：利用 b[j] <= 64 的特性，用 unsigned long long（64 位）压缩存储最近 64 个状态（status）和可用取法（trans），通过位运算快速计算当前状态，时间复杂度优化为 O(m + n log n)（排序 n 条规则，遍历 m 个石子数）。

涉及知识点

1. **动态规划（DP）**：通过子问题（i - b[j]的状态）推导当前问题（i 的状态）。
2. **位运算压缩**：用二进制位存储状态集合，通过|（标记可用取法）、~（取反得必败状态）、&（判断交集）等操作高效计算。
3. **排序优化**：对规则按 a[j]排序，使遍历石子数时可线性加入可用规则，减少重复判断。
4. **游戏理论**：基于“必胜态（存在必败子状态）”和“必败态（所有子状态均为必胜）”的博弈逻辑。

	f[i]，表示i个石子按照规则取石子，先手胜f[i]=1,先手负f[i]=0							
	先手下，可以决定拿走哪些石子，第1次取走石子后的剩下石子的所有若干状态中							
	只要存在一个状态必负，则当前的f[i]=1							
	n=2	m=7	a[1]=2	b[1]=2	a[2]=3	b[2]=3		
游戏模拟	0个石子		先手负				f[0]=0	
	1个石子		先手负				f[1]=0	
	2个石子		先手赢		第一步取走2个		f[2]=1	从f[0]=0转移过来
	3个石子		先手赢		第一步取走2个或3个		f[3]=1	从f[0]=0、f[1]=0转移过来
	4个石子		先手赢		第一步取走3个		f[4]=1	从f[1]=0转移过来
	5个石子		先手负		第一步无论取2或3		f[5]=0	从f[2]=1、f[3]=1转移过来
	6个石子		先手负		第一步无论取2或3		f[6]=0	从f[3]=1、f[4]=1转移过来
	7个石子		先手赢		第一步取走2个		f[7]=1	从f[5]=0转移过来



1. ①处应填（）

A.0　　B . ~0ull　　C. ~0ull^1　　D. 1

答案：C

分析：初始状态 status 为 0（表示 0 个石子时无法取，输）。

2. ②处应填（）

A.a [j]< i　　B.a [j] ==i　　C.a [j] !=i　　D.a [j] >i

答案：B

分析：当 a [j] < i 时，规则 j 适用于当前石子数 i。

3. ③处应填（）

A.trans |= 1ull <<(b [j] - 1)　　B.status |=1ull << (b [j]- 1)

C.status +=1ull <<(b [j]-1)　　D.trans +=1ull<< (b [j]-1)

答案：A? C

分析：trans 用于记录当前石子数 i 下可用的取法（b[j]），用二进制位压缩存储：第 k 位（从 0 开始）为 1 表示 b[j] = k+1 是可用取法。对于符合条件的规则 j，需将其 b[j]对应的位设置为 1，故用 trans |= 1ull << (b[j] - 1)（左移 b[j]-1 位实现位标记）。

4. ④处应填（）

A.~status | trans　　B.status & trans

B. status | trans　　D.~status & trans

答案：D

分析：当前石子数 i 的胜负状态（win）取决于：是否存在一种取法 b[j]，使得取后剩余石子数 i - b[j] 的状态为必败（即对方必败）。status 存储了之前的胜负状态（每一位表示对应石子数的必胜状态），~status 则表示必败状态；trans 存储可用取法。两者的交集（~status & trans）若不为 0，说明存在有效取法，当前玩家必胜，故 win 为此条件。

5. ⑤处应填（）

A. trans = status | trans ^win B. status = trans >> 1^win

C. trans = status ^trans |win D. status =status <<1^win

答案：D

分析：status 需更新以包含当前石子数 i 的状态（win），同时保留最近 64 个状态（因 b[j] ≤ 64，后续计算只需用到最近 64 个状态）。左移 status 一位（移除最旧状态），再将 win 作为最低位加入（^ win 等价于拼接），实现状态更新。