SCP-S 2023 讲评

disangan233
中国人民大学
上海洛谷网络科技有限公司

洛谷
www.luogu.com.cn

选择题

1. 以下物品可以携带进 CSP 第二轮测试考场的是（ D ）

A. 带有计算器功能的直尺
B. Um_nik 签名限量款背包
C. 印有完整 Splay 代码的文化衫 T 恤
D. 一大份绿鸟牌烤鸡柳

选择题

2.在 CSP 第二轮测试中，小 A 发现小 B 正在抄袭自己的代码，若小 A 未及时举报小 B 的行为，且比赛结束 后查出两人代码雷同，则小 A 与小 B 分别受到的惩罚禁赛时间为（ A ）

发现后没有举报，两人都禁赛三年

选择题

3.「流程结构」是编程中用于控制程序执行流程的一种方式，它包括顺序结构、分支结构和循环结构。在一些诗歌作品中，也有对「流程结构」的体现。下列诗歌片段中体现循环结构的是？（B）

语文题

B. 只要我还能够行走，只要我还能够张望，只要我还能够呼吸，就一直走向前方

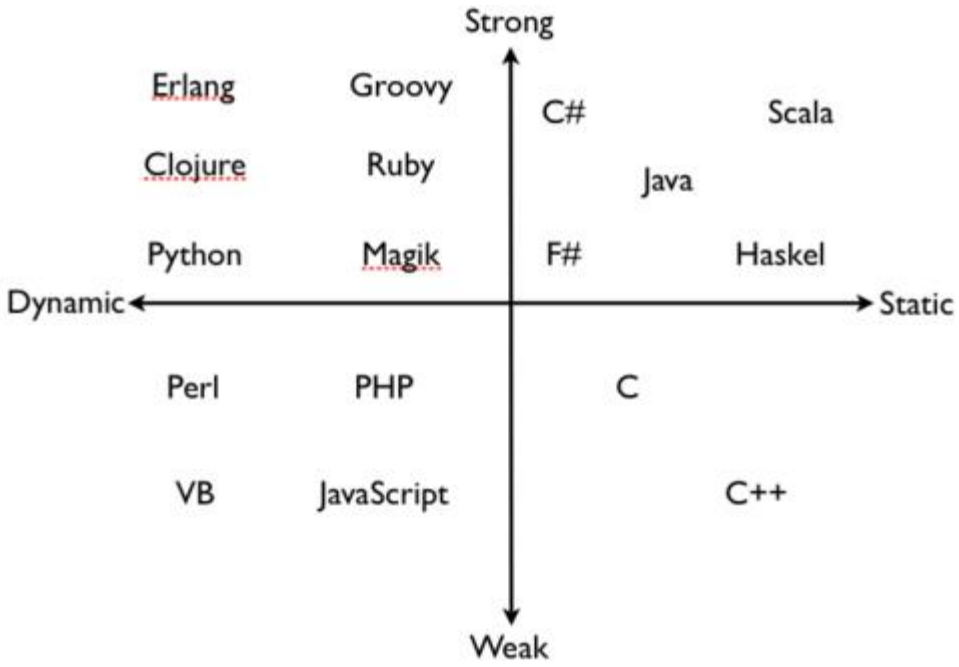
「只要」：`for/while`

选择题

4. 以下四种编程语言中，属于弱类型编程语言的是（D）

C++ 是弱类型编程语言

弱类型编程语言才可以强制类型转换



选择题

5. 现有一段长度为 3 分钟，采样率为 44.1 千赫兹、位深为 16 比特的 wav 文件的双通道立体声音频，该音频文件占用的存储空间大小约是（ B ）

$$3 \times 60 \times 44{,}100 \times 16 \div 8 \times 2 = 31{,}752{,}000\ \text{B} \approx 30.3\ \text{MiB}$$

注意要乘上双声道的 2

选择题

6. 在 vim 编辑器中，若当前位于 normal 模式，你需要将光标移动到这一行的末尾并进入 insert 模式。依次按下以下按键无法实现该目标的是（ A ）

$ a 和 A 是一样的，$ 移到最后一个字符，a 右移一位进入 insert 模式

A 是直接移动到这一行的末尾并进入 insert 模式

o 是移动到下一行开头，而 Backspace 是后退一格并进入 insert 模式

所以三种都是可以的

n 是进行下一次搜索

w q 会写入然后退出

选择题

7. 在下列四组时空限制中，书写正确且允许使用的内存最多的一组是（D）

A,C,D 书写正确，而 $8\ \text{Mbit} = 1\ \text{MiB}$，$1\ \text{MiB} = 1024\ \text{KiB}$

A. $10\ s$，$256\ 000\ \text{KiB}$

B. $4\ 000\ kg$，$10^{24}\ \text{bit}$

C. $3\ 000\ 000\ \mu s$，$1024\ \text{Mbit}$

D. $2\ 000\ ms$，$256\ \text{MiB}$

选择题

8. 假设某算法的计算时间表示为递推关系式 $T(1)=O(1)$，$T(n)=T(\lceil\frac{\sqrt{2}}{2+\sqrt{2}}n\rceil)+O(\lg n)$，则算法的时间复杂度为（ D ）

本质上是一个等比数列求和

$$\left(\frac{\sqrt{2}}{2+\sqrt{2}}\right)^{\lg n}\sim\frac{1}{2^{\lg n}}=\frac{1}{n}$$

所以等比数列一共有 $\lg n$ 项，时间复杂度为

$$\sum\lg nk^{i}=\lg n^{\lg n}k^{\lg^{2}n}=\lg^{2}n$$

选择题

9. 给定矩阵 $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$。我们定义一次操作为：选择同行或同列的两个元素，另其中一个加一，另一个减一。下列矩阵中不能通过 A 矩阵进行若干次操作得到的是？（D）

D. $\begin{pmatrix} -1 & 5 \\ 6 & 1 \end{pmatrix}$

操作不能改变四个元素的和，D 选项四个元素的和不等于 A 四个元素的和

选择题

10. 「平衡三进制」是一种特殊的计数体系，与标准三进制相比，其中用于计数的符码为 $-1, 0, 1$，为书写方便，通常使用字母 Z 代替 -1。例如，因为 $1 \times 3^2 + (-1) \times 3^1 + 0 \times 3^0 = 6$，所以十进制的 6 用平衡三进制表示为 $1Z0$。则十进制的 -11 可用平衡三进制表示为（ D ）

$$-11 = -9 - 3 + 1$$

所以十进制的 -11 可用平衡三进制表示为 $ZZ1$

选择题

11. 竞赛图也叫有向完全图，每对顶点之间都有一条边相连的有向图称为竞赛图。5 个点的无标号竞赛图数量是（D）

无标号竞赛图数量：0,1,1,2,4,12

没有任何技巧，能实现的技巧都是非常困难的

拿代码跑的很抽象的题，只能自行枚举

选择题

12. 依次抛出四个六面骰子，按照抛出顺序将骰子上的数值记为 a,b,c,d。则 $a<b,b>c,c<d$ 的概率为（A）

考虑枚举 b,c，方案数为

$$\sum_{i=2}^{6}(i-1)\sum_{j<i}(6-j)=1\times5+2\times9+3\times12+4\times14+5\times15$$

$$=190$$

所以概率为

$$\frac{190}{6^4}=\frac{95}{648}$$

选择题

13. 有 6 堆石子，每堆石子分别有 $1, 4, 7, 1, 5, 4$ 个。Alice 和 Bob 两人轮流操作，轮到 Alice 时需要选择一堆 石子并拿走其中的任意正奇数个，轮到 Bob 时需要选择一堆石子并拿走其中的任意正偶数个，最先无法操作的人判输。下列说法正确的是（ A ）

Alice 可以直接拿走 $1,5,7$，并且可以把 4 变成奇数

而 Bob 拿走偶数之后奇数依然是奇数，且有两个 1 无法拿走

那么无论 Bob 是先手还是后手，Alice 都必胜

```cpp
uint32_t Trans(uint32_t x) {
  x ^= x << 13;
  x ^= x >> 17;
  x ^= x << 5;
  return x;
}
```

选择题

```cpp
void Cycle(uint32_t x) {
  uint32_t old = x, cnt = 0;
  do {
    x = Trans(x);
    ++cnt;
  } while (x != old);
}
```

洛谷

14. 阅读以下 C++ 代码片段，其中 uint32_t 表示无符号 32 位整数，在大多数环境中等同于 unsigned int。下列说法有误的一项是？（A）

A. 若将任意 uint32_t 值代入 Trans 函数，得到的返回值一定和输入参数不同

B. 若两 uint32_t 变量 x 与 y 的值不同，则 Trans(x) 与 Trans(y) 的值一定不同

C. 若将任意 uint32_t 值代入 Cycle 函数，则函数执行时 ++cnt; 一行被执行到的次数一定为奇数

D. 若将任意 uint32_t 值代入 Cycle 函数，一定不会出现无限循环

Trans 是 xor shift 算法，其循环节是 $2^{32}-1$

最重要的是 Trans(0)=0，于是显然选 A

选择题

15. 阅读以下 C++ 代码:

运行程序后输出为？（A）

这里的 enum 依然是 4

union 取最大所以是 sizeof(u)=4

```cpp
#include <cstdint>
#include <iostream>

union U {
  bool fl0, fl1, fl2, fl3, fl4, fl5, fl6, fl7;
  uint16_t us0, us1;
  int16_t ss0, ss1;
  enum : int32_t {
    ZERO,
    ONE,
    THREE = 3,
    TEN = (uint64_t)10ull,
    COPY = TEN,
  } e;
} u;

signed main() {
  std::cout << sizeof(u) << '\n';
  return 0;
}
```

阅读程序 1

已经告诉了是一种排序算法

可以发现，每次会二分值域中点 mid，将 $> mid$ 的数放在左边，$\leq mid$ 的数放在右边，继承下标后再次递归排序

所以是从大到小排序

```cpp
#include<iostream>
using namespace std;
int a[100005], b[100005], n, m;
void very_quick_sort(int l, int r, int p, int q){
    if(l >= r || p > q){    // ①
        return;
    }
    int mid = (l + r) / 2;
    int p0 = p - 1;
    int q0 = q + 1;
    for(int i = p;i <= q;i ++){
        if(a[i]  > mid) b[++ p0] = a[i];
            else        b[-- q0] = a[i];
    }
    for(int i = p;i <= q;i ++)
        a[i] = b[i];
    very_quick_sort(mid + 1, r, p, p0);
    very_quick_sort(l, mid, q0, q);
}
int main(){
    cin >> n >> m;
    for(int i = 1;i <= n;i ++)
        cin >> a[i];
    very_quick_sort(1, m, 1, n);
        // ②
    for(int i = 1;i <= n;i ++)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

阅读程序 1 判断题

1. 上述代码实现了一种排序算法，可以将 a 数组按照从小到大的顺序排序。（F）

是从大到小的顺序

```
#include<iostream>
using namespace std;
int a[100005], b[100005], n, m;
void very_quick_sort(int l, int r, int p, int q){
    if(l >= r || p > q){    // ①
        return;
    }
    int mid = (l + r) / 2;
    int p0 = p - 1;
    int q0 = q + 1;
    for(int i = p;i <= q;i ++){
        if(a[i]  > mid) b[++ p0] = a[i];
            else        b[-- q0] = a[i];
    }
    for(int i = p;i <= q;i ++)
        a[i] = b[i];
    very_quick_sort(mid + 1, r, p, p0);
    very_quick_sort(l, mid, q0, q);
}
int main(){
    cin >> n >> m;
    for(int i = 1;i <= n;i ++)
        cin >> a[i];
    very_quick_sort(1, m, 1, n);
        // ②
    for(int i = 1;i <= n;i ++)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

阅读程序 1 判断题

2. 如果在程序开始之前向 b 数组里写入数据（保证不会发生数组越界），则上述代码的输出不会发生变化。（T）

调用的 b 会先在比较的时候赋值，也只会再次访问相同范围的下标，所以是否有初值不影响

```cpp
#include<iostream>
using namespace std;
int a[100005], b[100005], n, m;
void very_quick_sort(int l, int r, int p, int q){
    if(l >= r || p > q){    // ①
        return;
    }
    int mid = (l + r) / 2;
    int p0 = p - 1;
    int q0 = q + 1;
    for(int i = p;i <= q;i ++){
        if(a[i]  > mid) b[++ p0] = a[i];
            else        b[-- q0] = a[i];
    }
    for(int i = p;i <= q;i ++)
        a[i] = b[i];
    very_quick_sort(mid + 1, r, p, p0);
    very_quick_sort(l, mid, q0, q);
}
int main(){
    cin >> n >> m;
    for(int i = 1;i <= n;i ++)
        cin >> a[i];
    very_quick_sort(1, m, 1, n);
        // ②
    for(int i = 1;i <= n;i ++)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

阅读程序 1 判断题

3. 若 $n = m$，存在某种数据构造方式，使得上述代码运行的时间复杂度为 $O(n^2)$，这是因为算法本身是对快速排序的改进，但是这种改进不能避免由于对数组的划分不够均等而在极端数据下导致复杂度发生退化。（F）

每一次递归，$[l, r]$ 的长度会减半，所以最坏条件下都是 $O(n \log m)$ 的。

```
#include<iostream>
using namespace std;
int a[100005], b[100005], n, m;
void very_quick_sort(int l, int r, int p, int q){
    if(l >= r || p > q){    // ①
        return;
    }
    int mid = (l + r) / 2;
    int p0 = p - 1;
    int q0 = q + 1;
    for(int i = p;i <= q;i ++){
        if(a[i]  > mid) b[++ p0] = a[i];
            else        b[-- q0] = a[i];
    }
    for(int i = p;i <= q;i ++)
        a[i] = b[i];
    very_quick_sort(mid + 1, r, p, p0);
    very_quick_sort(l, mid, q0, q);
}
int main(){
    cin >> n >> m;
    for(int i = 1;i <= n;i ++)
        cin >> a[i];
    very_quick_sort(1, m, 1, n);
        // ②
    for(int i = 1;i <= n;i ++)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

阅读程序 1 判断题

4. 如果将①处的 l>=r 条件删除（同时删除 || 使得程序能正常编译运行，下同），程序的时间复杂度不会发生变化；而将 p>q 条件删除，程序在某些数据下的运行效率将会明显降低。（F）

因为是在二分值域，所以数组下标不会有影响，但是值域的 return 条件有

```
#include<iostream>
using namespace std;
int a[100005], b[100005], n, m;
void very_quick_sort(int l, int r, int p, int q){
    if(l >= r || p > q){    // ①
        return;
    }
    int mid = (l + r) / 2;
    int p0 = p - 1;
    int q0 = q + 1;
    for(int i = p;i <= q;i ++){
        if(a[i]  > mid) b[++ p0] = a[i];
            else        b[-- q0] = a[i];
    }
    for(int i = p;i <= q;i ++)
        a[i] = b[i];
    very_quick_sort(mid + 1, r, p, p0);
    very_quick_sort(l, mid, q0, q);
}
int main(){
    cin >> n >> m;
    for(int i = 1;i <= n;i ++)
        cin >> a[i];
    very_quick_sort(1, m, 1, n);
        // ②
    for(int i = 1;i <= n;i ++)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

阅读程序 1 选择题

5. 不认为 n, m 同阶，即可能出现 n 远大于 m 或 m 远大于 n 的情况。则该程序的最坏时间复杂度为？（D）

每一次递归，$[l, r]$ 的长度会减半，所以最坏条件下复杂度为 $O(n \log m)$。

```cpp
#include<iostream>
using namespace std;
int a[100005], b[100005], n, m;
void very_quick_sort(int l, int r, int p, int q){
    if(l >= r || p > q){    // ①
        return;
    }
    int mid = (l + r) / 2;
    int p0 = p - 1;
    int q0 = q + 1;
    for(int i = p;i <= q;i ++){
        if(a[i]  > mid) b[++ p0] = a[i];
            else        b[-- q0] = a[i];
    }
    for(int i = p;i <= q;i ++)
        a[i] = b[i];
    very_quick_sort(mid + 1, r, p, p0);
    very_quick_sort(l, mid, q0, q);
}
int main(){
    cin >> n >> m;
    for(int i = 1;i <= n;i ++)
        cin >> a[i];
    very_quick_sort(1, m, 1, n);
        // ②
    for(int i = 1;i <= n;i ++)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

阅读程序 1 选择题

6. 若输入数据为 10 10 10 4 5 2 2 3 1 5 8 3，那么程序执行到②位置时，b 数组内的值为？（C）

此时 b 数组内跟 a 数组一样，已经是完全排好序后的结果了

```
#include<iostream>
using namespace std;
int a[100005], b[100005], n, m;
void very_quick_sort(int l, int r, int p, int q){
    if(l >= r || p > q){    // ①
        return;
    }
    int mid = (l + r) / 2;
    int p0 = p - 1;
    int q0 = q + 1;
    for(int i = p;i <= q;i ++){
        if(a[i]  > mid) b[++ p0] = a[i];
            else        b[-- q0] = a[i];
    }
    for(int i = p;i <= q;i ++)
        a[i] = b[i];
    very_quick_sort(mid + 1, r, p, p0);
    very_quick_sort(l, mid, q0, q);
}
int main(){
    cin >> n >> m;
    for(int i = 1;i <= n;i ++)
        cin >> a[i];
    very_quick_sort(1, m, 1, n);
        // ②
    for(int i = 1;i <= n;i ++)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

阅读程序 2

```
void dfs(int u, int fa){
    d[u] = d[fa] + 1;
    f[u] = fa;

    for(auto &v : e[u]) if(v != fa){
        dfs(v, u);
    }
}
bool cmp(int a, int b){
    return d[a] < d[b];
}
int main(){
    cin >> n >> m >> k;
    for(int i = 2;i <= n;i ++){
        int u, v;
        cin >> u >> v;
        e[u].push_back(v);
        e[v].push_back(u);
    }
    dfs(1, 0);                          // ①
    for(int i = 1;i <= m;i ++){
        int x, w;
        cin >> x >> w;
        w1[x] = (w1[x] + w) % mod;
        w2[x] = (w2[x] + w) % mod;
    }
    for(int i = 1;i <= n;i ++)
        id[i] = i;
    sort(id + 1, id + 1 + n, cmp);
    for(int i = 1;i <= k;i ++){
        for(int j = n;j >= 1;j --){
            int x = id[j];
            for(auto &y : e[x]) if(y != f[x]){
                w1[y] = (w1[y] + w1[x]) % mod;
            }
            w1[x] = 0;
        }
        for(int x = 1;x <= n;x ++)
            w1[x] = (w1[x] - w0[x] + mod) % mod,
            w0[x] = 0;
        for(int j = 1;j <= n;j ++){ // ②
            int x = id[j];
            if(f[x]){
                w1[f[x]] = (w1[f[x]] + w2[x]) % mod;
                w2[f[x]] = (w2[f[x]] + w2[x]) % mod;
                w0[x] = (w0[x] + w2[x]) % mod;
                w2[x] = 0;
            }
        }
    }
    for(int i = 1;i <= n;i ++)
        cout << w1[i] << " ";
    return 0;
}
```

阅读程序 2

在完整阅读完程序之后，可以发现：

实质上就是对所有距离恰好为 k 的点加上 x，一共 m 组操作

代码中 w1[x] 对所有子节点更新，

w2[x] 对父节点更新，而 w0[x] 的作用是可以处理实际距离为 k，防止走回来

```
    for(int i = 1;i <= k;i ++){
        for(int j = n;j >= 1;j --){
            int x = id[j];
            for(auto &y : e[x]) if(y != f[x]){
                w1[y] = (w1[y] + w1[x]) % mod;
            }
            w1[x] = 0;
        }
        for(int x = 1;x <= n;x ++)
            w1[x] = (w1[x] - w0[x] + mod) % mod,
            w0[x] = 0;
        for(int j = 1;j <= n;j ++){ // ②
            int x = id[j];
            if(f[x]){
                w1[f[x]] = (w1[f[x]] + w2[x]) % mod;
                w2[f[x]] = (w2[f[x]] + w2[x]) % mod;
                w0[x] = (w0[x] + w2[x]) % mod;
                w2[x] = 0;
            }
        }
    }
    for(int i = 1;i <= n;i ++)
        cout << w1[i] << " ";
    return 0;
}
```

阅读程序 2 判断题

1. 如果更改 ① 处 `dfs(1,0)` 为 `dfs(n,0)`，则输出结果可能有变化。（F）

不会有变化

因为有 w0[x] 的存在，根节点并没有影响

依然计算出深度然后排序

```
int main(){
    cin >> n >> m >> k;
    for(int i = 2;i <= n;i ++){
        int u, v;
        cin >> u >> v;
        e[u].push_back(v);
        e[v].push_back(u);
    }
    dfs(1, 0);                          // ①
    for(int i = 1;i <= m;i ++){
        int x, w;
        cin >> x >> w;
        w1[x] = (w1[x] + w) % mod;
        w2[x] = (w2[x] + w) % mod;
    }
```

阅读程序 2 判断题

2. 如果 $k = n$，那么输出结果均为 0。（T）

实质上就是对所有距离恰好为 k 的点加上 x，一共 m 组操作

不存在恰好为 n 的点，所以所有点的权值都为 0

```
for(int i = 1;i <= k;i ++){
    for(int j = n;j >= 1;j --){
        int x = id[j];
        for(auto &y : e[x]) if(y != f[x]){
            w1[y] = (w1[y] + w1[x]) % mod;
        }
        w1[x] = 0;
    }
    for(int x = 1;x <= n;x ++)
        w1[x] = (w1[x] - w0[x] + mod) % mod,
        w0[x] = 0;
    for(int j = 1;j <= n;j ++){ // ②
        int x = id[j];
        if(f[x]){
            w1[f[x]] = (w1[f[x]] + w2[x]) % mod;
            w2[f[x]] = (w2[f[x]] + w2[x]) % mod;
            w0[x] = (w0[x] + w2[x]) % mod;
            w2[x] = 0;
        }
    }
}
```

阅读程序 2 判断题

3. 如果更改 ② 处 `for(int j=1;j<=n;j++)` 为 `for(int j=n;j>=1;j--)`，那么对于任意合法的输入数据，更改前后程序的输出均相同。（F）

排序好的顺序是按照深度从小到大更新父节点，这样才能保证只更新一次

如果按照深度从大到小，那么一个点将对该点到根节点链上的所有点都更新

```
bool cmp(int a, int b){
    return d[a] < d[b];
}
```

```
for(int j = 1;j <= n;j ++){ // ②
    int x = id[j];
    if(f[x]){
        w1[f[x]] = (w1[f[x]] + w2[x]) % mod;
        w2[f[x]] = (w2[f[x]] + w2[x]) % mod;
        w0[x] = (w0[x] + w2[x]) % mod;
        w2[x] = 0;
    }
}
```

阅读程序 2 单选题

4. 该程序的时间复杂度为？（B）

复杂度瓶颈在于循环枚举 k 次

每一次都是对所有点 $O(n)$ 更新

而初始处理所有修改是 $O(m)$ 的

于是总复杂度为 $O(nk+m)$

```
for(int i = 1;i <= k;i ++){
    for(int j = n;j >= 1;j --){
        int x = id[j];
        for(auto &y : e[x]) if(y != f[x]){
            w1[y] = (w1[y] + w1[x]) % mod;
        }
        w1[x] = 0;
    }
    for(int x = 1;x <= n;x ++)
        w1[x] = (w1[x] - w0[x] + mod) % mod,
        w0[x] = 0;
    for(int j = 1;j <= n;j ++){ // ②
        int x = id[j];
        if(f[x]){
            w1[f[x]] = (w1[f[x]] + w2[x]) % mod;
            w2[f[x]] = (w2[f[x]] + w2[x]) % mod;
            w0[x] = (w0[x] + w2[x]) % mod;
            w2[x] = 0;
        }
    }
}
```

阅读程序 2 单选题

5. 如果输入数据为：

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

则程序的输出应该是：（A）

有 $m=2, k=1$，1 距离为 1 的点是 2，3 距离为 1 的点是 2 4 5

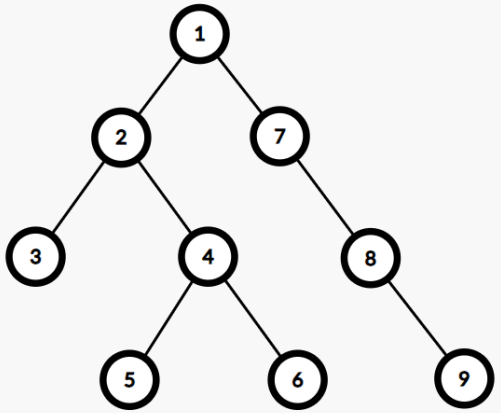
所以输出为 0 7 0 2 2

阅读程序 2 单选题

6. 如果输入数据为：

```
9 9 2
1 2
1 7
2 3
2 4
7 8
4 5
4 6
8 9
```

```
1 1
2 10
3 100
4 1000
5 10000
6 100000
7 1000000
8 10000000
9 100000000
```



那么输出数据应该为：（A）

画图算一下，每个点距离为 2 的点是哪几个就好

阅读程序 3

首先先看两个 build 函数

发现是用左儿子右兄弟表示法

实际上是一颗线段树，但是没有右儿子

son[x] 表示线段树的左儿子节点

nex[x] 表示不存在的右儿子节点的左儿子节点

因为只有 build1 才会新建节点

实际的树结构仍是二叉树，且与没有右儿子节点的线段树等价，叶子节点也要建出

然后是带有懒标记的

```
namespace CirnoTree{
    const int SIZ = 4e6 + 3;

    int build1(int l, int r);
    int build2(int l, int r);

    int siz = 0;
    int lft[SIZ], rgt[SIZ];
    int nex[SIZ], son[SIZ];
    i64 sum[SIZ];
    i64 below_tag[SIZ];
    i64 right_tag[SIZ];

    int build1(int l, int r){
        int u = ++ siz;
        lft[u] = l;
        rgt[u] = r;
        son[u] = l == r ? 0 : build2(l, r);
        return u;
    }
    int build2(int l, int r){
        int mid = (l + r) / 2;
        int p = build1(l, mid);
        nex[p] = l == r ? 0 : build2(mid + 1, r);
        return p;
    }
```

阅读程序 3

```
void modify(int t, int pos, int w){
    if(rgt[t] <= pos){
        below_tag[t] += w;
        sum[t] += 1ll * w * (rgt[t] - lft[t] + 1);
    } else {
        int l = max(lft[t],   1);
        int r = min(rgt[t], pos);
        update(t);
        sum[t] += 1ll * (r - l + 1) * w;
        for(int p = son[t];p != 0 && lft[p] <= pos;p = nex[p])
            update(p), modify(p, pos, w);
    }
}
```

update 是标记下传

所以先来看 modify，是一个前缀的区间加 w

于是如果包含在 [1,pos] 内，直接进行修改

否则标记下传后，递归修改

p=nex[p] 是可以循环很多次的

因为右兄弟的链长实际上是线段树的高，为 $\log n$

相当于每一次 lrt[t] <= pos < rgt[t] 的点只有一个，其余的会直接修改

跟 bit 用 $O(\log n)$ 个极大区间来合并信息是一样的

所以最后总修改次数也是 $O(\log n)$ 次的

阅读程序 3

query 的查询方法跟修改是一致的，都是合并区间信息

初始化时，发现是 modify $[1, i]$ 区间加 x，那么读入的应该是一个差分数组

```
    i64 query(int t, int pos){
        if(rgt[t] <= pos){
            return sum[t];
        } else {
            update(t);
            i64 ans = 0;
            for(int p = son[t];p != 0 && lft[p] <= pos;p = nex[p])
                update(p), ans += query(p, pos);
            return ans;
        }
    }
};
int main(){
    int n, m, root;
    cin >> n >> m;
    root = CirnoTree :: build1(1, n);
    for(int i = 1, x;i <= n;i ++){
        cin >> x;
        CirnoTree :: modify(root, i, x);
    }
```

阅读程序 3

提到了修改和查询都是对一个前缀

所以差分一下就可以得到代码里实现的

区间加和区间求和

那么就可以做题了

```
for(int i = 1, op;i <= m;i ++){
    cin >> op;
    if(op == 1){
        int l, r, w;
        cin >> l >> r >> w;
        CirnoTree :: modify(root, r, w);
        CirnoTree :: modify(root, l - 1, -w);
    } else {
        int l, r;
        cin >> l >> r;
        i64 w1 = CirnoTree :: query(root, r);
        i64 w2 = CirnoTree :: query(root, l - 1);
        cout << w1 - w2 << endl;
    }
}
```

阅读程序 3 判断题

1. 该程序的功能为，读入一个数组 $[a_1, a_2, \cdots, a_n]$，并执行 m 次操作：

- $1\ l\ r\ w$：将 $a_l, a_{l+1}, \cdots, a_r$ 加上 w
- $2\ l\ r$：计算 $a_l + a_{l+1} + \cdots + a_r$ 的值并输出

（F）

操作没错，但是读入的应该是差分后的数组（或者代码忘了差分）

阅读程序 3 判断题

2. 如果将主函数里的程序片段 `root = CirnoTree :: build1(1, n)` 修改成 `root = CirnoTree :: build2(1, 3 * n)`，则程序针对任意合法输入数据的输出结果不会发生任何改变。（T）

只是相当于多建了很多没用的点而已，输出结果不变

阅读程序 3 判断题

3. 如果在 build 操作以后，执行函数片段 CirnoTree :: query(root, -114514)，程序会因为发生数组越界而非正常退出。（F）

```
i64 query(int t, int pos){
    if(rgt[t] <= pos){
        return sum[t];
    } else {
        update(t);
        i64 ans = 0;
        for(int p = son[t];p != 0 && lft[p] <= pos;p = nex[p])
            update(p), ans += query(p, pos);
        return ans;
    }
}
```

有 p!=0，所以不会越界

阅读程序 3 单选题

4. 当读入的 n 充分大时，CirnoTree 的大小（即在主程序执行完 build 操作后变量 siz 的值）大约为？（D）

是没有右端点的线段树

线段树有 n 个叶子节点，n 个非叶子节点，所以是 $1.5n$

阅读程序 3 单选题

5. 对于该数据结构，调用一次 build 函数的最好时间复杂度和调用一次 modify 或 query 的最坏时间复杂度为（ A ）

共 $1.5n$ 个节点，所以 build 是 $O(n)$ 的

前面提到 modify/query 只会合并最多 $O(\log n)$ 个区间，且标记下传是 $O(1)$ 的，所以时间复杂度最坏为 $O(\log n)$

阅读程序 3 单选题

6. 对于如下输入数据，在程序主要片段运行结束、执行到 return 0 之前，below_tag[8] 和 right_tag[8] 的值分别是（ A ）

```
8 3
0 0 0 0 0 0 0 0
1 1 8 3
1 2 6 2
2 5 8
```

进行了 [1,8] 的前缀加 3，[1,1] 的前缀加 -2，[1,6] 的前缀加 2

第 8 个点代表了区间 [5,6]，于是两次区间修改后应该有 below_tag[8]=2，right_tag[8]=0

询问相当于询问前缀 [1,4] 和前缀 [1,8]，而 [5,6] 并没有被访问，也不会有标记下传，所以答案为 2 0

完善程序 1

给定序列 a_n，求其所有子区间异或和的和，其中 $n \le 10^5$，$0 \le a_i \le 10^9$。

思路：我们对每一位独立计算，对右端点扫描线，并用异或前缀和辅助统计。

所有子区间异或和的和，等价于异或前缀和两两异或和的和

拆位，扫一遍用桶记录 0 和 1 的个数，与当前数当前位不同的个数乘上位的大小，就是当前数贡献的答案

完善程序 1

1. ① 处应当填入（ B ）

维护异或前缀和，所以是异或上前一个 a[i]

A. a[i-1]^=a[i]

B. a[i]^=a[i-1]

C. a[i-1]+=a[i]

D. a[i]+=a[i-1]

```
int n, a[N], cnt[2]; i64 ans;
int main () {
    cin >> n;
    for (int i = 1; i <= n; i ++) cin >> a[i], ①;
    for (int bit = ②; ③; bit --) {
        cnt[0] = cnt[1] = 0;
        for (int i = 0; i <= n; i ++) {
            cnt[④] ++;
            ans += ⑤;
        }
    } cout << ans;
}
```

完善程序 1

2. ② 处应当填入（B）

数据范围是 $10^9 < 2^{30}$

所以一共有 30 位，开到 bit=29

A. n-1

B. 29

C. n

D. n+1

```
int n, a[N], cnt[2]; i64 ans;
int main () {
    cin >> n;
    for (int i = 1; i <= n; i ++) cin >> a[i], ①;
    for (int bit = ②; ③; bit --) {
        cnt[0] = cnt[1] = 0;
        for (int i = 0; i <= n; i ++) {
            cnt[④] ++;
            ans += ⑤;
        }
    } cout << ans;
}
```

完善程序 1

3. ③ 处应当填入（ D ）

从 29 取到 0，共 30 位

bit>=0 等价于 ~bit

A. bit

B. bit>=n

C. bit-1

D. ~bit

```
int n, a[N], cnt[2]; i64 ans;
int main () {
    cin >> n;
    for (int i = 1; i <= n; i ++) cin >> a[i], ①;
    for (int bit = ②; ③; bit --) {
        cnt[0] = cnt[1] = 0;
        for (int i = 0; i <= n; i ++) {
            cnt[④] ++;
            ans += ⑤;
        }
    } cout << ans;
}
```

完善程序 1

4. ④ 处应当填入（A）

统计当前位应该是 A

选项 B,C 返回的不是 0/1

顺序从高到低或者从低到高无所谓

A. (a[i] >> bit) & 1

B. a[i] & (1 << bit)

C. a[i] & (1ll << bit)

D. (a[i] >> bit) ^ 1

```
int n, a[N], cnt[2]; i64 ans;
int main () {
    cin >> n;
    for (int i = 1; i <= n; i ++) cin >> a[i], ①;
    for (int bit = ②; ③; bit --) {
        cnt[0] = cnt[1] = 0;
        for (int i = 0; i <= n; i ++) {
            cnt[④] ++;
            ans += ⑤;
        }
    } cout << ans;
}
```

完善程序 1

5. ⑤ 处应当填入（B）

当前位的大小是 1 << bit

计算的桶应该是 (a[i] >> bit) & 1 ^ 1

记得 long long

A. cnt[(a[i] >> bit) & 1] << i)

B. 1ll * cnt[(a[i] >> bit) & 1 ^ 1] << bit

C. 1ll * cnt[(a[i] >> bit) & 1] << bit

D. cnt[(a[i] >> bit) & 1 ^ 1] << i

```
for (int bit = ②; ③; bit --) {
    cnt[0] = cnt[1] = 0;
    for (int i = 0; i <= n; i ++) {
        cnt[④] ++;
        ans += ⑤;
    }
} cout << ans;
```

完善程序 2

给定序列 $\{a_n\}(1 \le n \le 100, a_i \in \{0,1,2\})$，描述 n 个格子的状态。若 a_i 为 0，则表示覆不覆盖均可；若为 1，则表示一定不能覆盖；若为 2，则表示一定要被覆盖。

询问依次进行 $m(1 \le m \le 10^9)$ 次操作，每次选定一个区间 $[l,r]$ 的格子将其覆盖，使得满足 $\{a_n\}$ 数组的所有限制条件，有多少种方案？

两种方案不同，当且仅当存在某次操作覆盖的区间不同。如果格子可以被覆盖，则可以被覆盖多次。

考虑容斥原理，使用动态规划求出每种权值的贡献次数。

钦定 $a_i = 2$ 的格子变为 0/1，2 必须被覆盖相当于可能被覆盖减去一定不被覆盖。令 $f_{i,j,k}$ 代表 i 个 2 被钦定为 1，前 j 格有 k 种可能的覆盖区间的染色方案数，每一次枚举一个不含有 1 的连续段进行转移

代码中压缩第一维空间，$f_{0,j,k}$ 代表不含 $(-1)^n$，$f_{1,j,k}$ 代表含 $(-1)^n$

完善程序 2

1. ① 处应当填入（B）

送分的快速幂

A. r = 1ll * r * r % MOD

B. a = 1ll * a * a % MOD

C. r = 1ll * r * a % MOD

D. a = 1ll * r * a % MOD

```
int power(int a, int b){
    int r = 1;
    while(b){
        if(b & 1) r = 1ll * r * a % MOD;
        b >>= 1,  ①;
    }
    return r;
}
```

完善程序 2

2. ② 处应当填入（A）

代码没有处理边界

于是给两边设为 1 即可

A. A[0] = 1, A[n + 1] = 1

B. A[0] = 1, A[n + 1] = 2

C. A[0] = 2, A[n + 1] = 1

D. A[0] = 2, A[n + 1] = 2

```
int main(){
    int n, m;
    cin >> n >> m;
    int h = n * (n + 1) / 2;
    for(int i = 1;i <= n;i ++)
        cin >> A[i];
    ②;
    F[0][0][0] = 1;
```

完善程序 2

3. ③ 处应当填入（ C ）

c 代表了第 i 个是否被钦定

所以是 A[i]==2

```
F[0][0][0] = 1;
for(int i = 1;i <= n + 1;i ++) if(A[i] == 1 || A[i] == 2){
    for(int j = i - 1;j >= 0;j --){
        int c = ③;
        int g = ④;
        for(int k = h;k >= g;-- k){
            F[0][i][k] = (F[0][i][k] + F[0 ^ c][j][k - g]) % MOD;
            F[1][i][k] = (F[1][i][k] + F[1 ^ c][j][k - g]) % MOD;
        }
        if(A[j] == 1) break;
    }
}
int ans = 0;
for(int i = 0;i <= h;i ++){
    ans = (ans + 1ll * F[0][n + 1][i]  * power(i, m)) % MOD;
    ans = (ans + 1ll * ⑤ * power(i, m)) % MOD;
}
cout << ans << endl;
```

完善程序 2

4. ④ 处应当填入（ B ）

$(j,i]$ 中全是 0，共 $\frac{(i-j)(i-j+1)}{2}$ 种可能的区间

转移时由于区间直接被钦定

所以是枚举初始的可能 k

由 $f_{i,j,k-g}$ 转移到 $f_{i,j,k}$

```
F[0][0][0] = 1;
for(int i = 1;i <= n + 1;i ++) if(A[i] == 1 || A[i] == 2){
    for(int j = i - 1;j >= 0;j --){
        int c = ③;
        int g = ④;
        for(int k = h;k >= g;-- k){
            F[0][i][k] = (F[0][i][k] + F[0 ^ c][j][k - g]) % MOD;
            F[1][i][k] = (F[1][i][k] + F[1 ^ c][j][k - g]) % MOD;
        }
        if(A[j] == 1) break;
    }
}
int ans = 0;
for(int i = 0;i <= h;i ++){
    ans = (ans + 1ll * F[0][n + 1][i]  * power(i, m)) % MOD;
    ans = (ans + 1ll * ⑤ * power(i, m)) % MOD;
}
cout << ans << endl;
```

完善程序 2

5. ⑤ 处应当填入（ C ）

```
int ans = 0;
for(int i = 0;i <= h;i ++){
    ans = (ans + 1ll * F[0][n + 1][i]  * power(i, m)) % MOD;
    ans = (ans + 1ll * ⑤ * power(i, m)) % MOD;
}
cout << ans << endl;
return 0;
```

现在已经得到了 $f_{i,j,k}$，考虑如何计算答案

选取 m 种区间的方案数其实就是 k^m，于是可得

$$ans = \sum_{i=0}^{n} (-1)^i \sum_{j=0}^{\frac{n(n+1)}{2}} f_{i,n+1,j} \times j^m$$

压缩空间已经去掉了第一重循环，直接乘上 -1 即可，所以填入部分是 `(MOD - F[1][n + 1][i])`