2021 CSP-S 真题详细题解

一、单项选择题

1. 在 Linux 系统终端中，用于列出当前目录下所含的文件和子目录的命令为（ ）。

A. ls　　B. cd　　C. cp　　D. all

答案：A

分析：本题考查 Linux 基本命令。

- ls 命令用于列出当前目录下的文件和子目录；
- cd 命令用于切换目录；
- cp 命令用于复制文件或目录；
- all 不是有效的 Linux 命令。

因此，正确答案为 A。

2. 二进制数 00101010_2和 00010110_2的和为（）。

A. 00111100_2 B. 01000000_2 C. 00111100_2 D. 01000010_2

答案：B

分析：本题考查二进制加法运算。

二进制加法规则为“逢二进一”，计算过程如下：

```
  00101010
+ 00010110
----------
  01000000
```

两个二进制数相加的结果为 01000000_2，故正确答案为 B。

3. 在程序运行过程中，如果递归调用的层数过多，可能会由于（ ）引发错误。

A. 系统分配的栈空间溢出　　B. 系统分配的队列空间溢出

C. 系统分配的链表空间溢出　　D. 系统分配的堆空间溢出

答案：A

分析：本题考查递归的内存机制。

递归调用时，每次调用的上下文（如参数、返回地址等）会被压入栈中，栈的空间是有限的。当递归层数过多时，栈空间会被耗尽，导致栈溢出错误。而队列、链表、堆与递归调用的层数无关，因此正确答案为 A。

4. 以下排序方法中，（ ）是不稳定的。

A. 插入排序　　B. 冒泡排序　　C. 堆排序　　D. 归并排序

答案：C

分析：本题考查排序算法的稳定性。

- 稳定排序是指排序后，相等元素的相对位置保持不变；
- 插入排序、冒泡排序、归并排序均为稳定排序；
- 堆排序在交换元素时可能改变相等元素的相对位置，属于不稳定排序。

因此，正确答案为 C。

5. 以比较为基本运算，对于 2n 个数，同时找到最大值和最小值，最坏情况下需要的最小的比较次数为（ ）。

A. 4n−2　　B. 3n+1　　C. 3n−2　　D. 2n+1

答案：C

分析：本题考查查找最值的最优比较次数。

采用配对比较的方法：

将 2n 个数分成 n 对，每对比较一次，共 n 次比较，得到每对的局部最大值和局部最小值。
从 n 个局部最大值中找到全局最大值，需要 n-1 次比较。
从 n 个局部最小值中找到全局最小值，需要 n-1 次比较。
总比较次数为：n + (n-1) + (n-1) = 3n-2。

示例验证

n=1（2 个数）：只需比较一次，3×1-2=1，正确。

n=2（4 个数）：例如数字 1,2,3,4：
比较 1 和 2（1 次），比较 3 和 4（1 次），共 2 次。
从局部最大值（2 和 4）中找全局最大值，比较 2 和 4（1 次）。
从局部最小值（1 和 3）中找全局最小值，比较 1 和 3（1 次）。
总比较次数：2+1+1=4，3×2-2=4，正确。

6. 现有一个地址区间为 0~ 10 的哈希表，对于出现冲突情况，会往后找第一个空的地址存储（到 10 冲突了就从 0 开始往后），现在要依次存储 (0,1,2,3,4,5,6,7)，哈希函数为 h (x)=x² mod11。请问 7 存储在哈希表哪个地址中（ ）。

A. 5　　B. 6　　C. 7　　D. 8

答案：C

分析：

1. **计算每个数字的初始哈希值**：
   - $h(0) = 0^2 \mod 11 = 0$
   - $h(1) = 1^2 \mod 11 = 1$
   - $h(2) = 2^2 \mod 11 = 4$
   - $h(3) = 3^2 \mod 11 = 9$
   - $h(4) = 4^2 \mod 11 = 16 \mod 11 = 5$
   - $h(5) = 5^2 \mod 11 = 25 \mod 11 = 3$
   - $h(6) = 6^2 \mod 11 = 36 \mod 11 = 3$
   - $h(7) = 7^2 \mod 11 = 49 \mod 11 = 5$

2. **模拟插入过程（线性探测）**：
   - 初始化哈希表：`[_, _, _, _, _, _, _, _, _, _, _]`（11 个空位）。
   - **插入 0**：$h(0) = 0$ → 地址 0 空 → 存入 `0` → `[0, _, _, _, _, _, _, _, _, _, _]`。
   - **插入 1**：$h(1) = 1$ → 地址 1 空 → 存入 `1` → `[0, 1, _, _, _, _, _, _, _, _, _]`。
   - **插入 2**：$h(2) = 4$ → 地址 4 空 → 存入 `2` → `[0, 1, _, _, 2, _, _, _, _, _, _]`。
   - **插入 3**：$h(3) = 9$ → 地址 9 空 → 存入 `3` → `[0, 1, _, _, 2, _, _, _, _, 3, _]`。
   - **插入 4**：$h(4) = 5$ → 地址 5 空 → 存入 `4` → `[0, 1, _, _, 2, 4, _, _, _, 3, _]`。
   - **插入 5**：$h(5) = 3$ → 地址 3 空 → 存入 `5` → `[0, 1, _, 5, 2, 4, _, _, _, 3, _]`。
   - **插入 6**：$h(6) = 3$ → 地址 3 被占（5），探测地址 4（被占），地址 5（被占），地址 6（空）→ 存入 `6` → `[0, 1, _, 5, 2, 4, 6, _, _, 3, _]`。
   - **插入 7**：$h(7) = 5$ → 地址 5 被占（4），探测地址 6（被占），地址 7（空）→ 存入 `7` → `[0, 1, _, 5, 2, 4, 6, 7, _, 3, _]`。

3. **结论**：
   数字 7 存储在地址 **7**，对应选项 **C. 7**。

7. G 是一个非连通简单无向图（没有自环和重边），共有 36 条边，则该图至少有（ ）个点。

A. 8　　B. 9　　C. 10　　D. 11

答案：C

分析：本题考查非连通图的顶点数与边数关系。

非连通图至少包含两个连通分量，要使顶点数最少，需一个分量为完全图（边数最多），另一个分量为孤立点。设完全图有 k 个顶点，边数为 k(k-1)/2，令 k(k-1)/2≥36，解得 k≥9（9 个顶点的完全图有 36 条边），加上 1 个孤立点，总顶点数为 10。因此，正确答案为 C。

8. 令根结点的高度为 1，则一棵含有 2021 个结点的二叉树的高度至少为（ ）。

A. 10　　B. 11　　C. 12　　D. 2021

答案：B

分析：本题考查二叉树的最小高度。

高度为 h 的满二叉树最多有 2^h-1 个结点。当 h=11 时，$2^{11}-1=2047\geq2021$，因此最小高度为 11，正确答案为 B。

9. 前序遍历和中序遍历相同的二叉树为且仅为（ ）。

A. 只有 1 个点的二叉树

B. 根结点没有左子树的二叉树

C. 非叶子结点只有左子树的二叉树

D. 非叶子结点只有右子树的二叉树

答案：D

分析：本题考查二叉树的遍历特性。

- 前序遍历顺序：根→左→右；
- 中序遍历顺序：左→根→右；
- 若非叶子结点只有右子树（左子树为空），则前序和中序遍历均为“根→右”，两者相同。

因此，正确答案为 D。

10. 定义一种字符串操作为交换相邻两个字符。将 DACFEB 变为 ABCDEF 最少需要 ( ) 次上述操作。

A. 7　　B. 8　　C. 9　　D. 6

答案：A 计算逆序对

步骤 1：明确目标位置

目标字符串“ABCDEF”中每个字符的目标位置（索引 0 到 5）为：
A(0)、B(1)、C(2)、D(3)、E(4)、F(5)。

步骤 2：转换原字符串为目标位置序列

原字符串“DACFEB”中每个字符对应的目标位置序列为：
D (3)、A (0)、C (2)、F (5)、E (4)、B (1)，即序列为：`[3, 0, 2, 5, 4, 1]`。

步骤 3：计算逆序对数量

逆序对是指序列中“位置靠前的元素大于位置靠后的元素”的组合。对序列`[3, 0, 2, 5, 4, 1]`计算逆序对：

- 元素 3（索引 0）：与 0（1）、2（2）、1（5）构成逆序对 → 3 个
- 元素 0（索引 1）：无逆序对 → 0 个
- 元素 2（索引 2）：与 1（5）构成逆序对 → 1 个
- 元素 5（索引 3）：与 4（4）、1（5）构成逆序对 → 2 个
- 元素 4（索引 4）：与 1（5）构成逆序对 → 1 个
- 元素 1（索引 5）：无逆序对 → 0 个

步骤 4：总和逆序对数量

总逆序对数量为：3 + 0 + 1 + 2 + 1 = 7。

因此，最少需要 7 次操作。

11. 有如下递归代码

solve (t, n):

if t=1 return 1

else return 5*solve (t-1,n) mod n

则 solve (23,23) 的结果为（ ）。

A. 1　　B. 7　　C. 12　　D. 22

答案：A

分析：本题考查模运算与递归。

根据费马小定理，若 n 为质数，a 与 n 互质，则 $a^{n-1}\equiv 1 \bmod n$。23 是质数，5 与 23 互质，故 $5^{22}\equiv 1 \bmod 23$。solve (23,23) 即 $5^{22} \bmod 23=1$，因此正确答案为 A。

12. 斐波那契数列的定义为：$F_1=1$，$F_2=1$，$F_n=F_{n-1}+F_{n-2}(n\geq 3)$。现在用如下程序来计算斐波那契数列的第 n 项，其时间复杂度为（ ）。

F (n):

if n<=2 return 1

else return F (n-1) + F (n-2)

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

答案：C

分析：本题考查递归算法的时间复杂度。

该递归程序存在大量重复计算，递归树的深度为 n，每一层的节点数约为 2^n，因此时间复杂度为 $O(2^n)$，正确答案为 C。

13. 有 8 个苹果从左到右排成一排，你要从中挑选至少一个苹果，并且不能同时挑选相邻的两个苹果，一共有（ ）种方案。

A. 36　　B. 48　　C. 54　　D. 64

答案：C

1. **问题本质：** 这是一个组合问题，等价于在 8 个位置中选择一个子集，要求子集中任意两个元素不相邻，且至少选一个苹果。
2. **数学模型：** 设 $f(n)$ 表示 n 个苹果的方案数（包括不选任何苹果的情况）。根据动态规划：
   - $f(0) = 1$（不选任何苹果）
   - $f(1) = 2$（不选或选第 1 个苹果）
   - 对于 $n \geq 2$:
     - 若第 n 个苹果不选，方案数为 $f(n-1)$
     - 若第 n 个苹果选，则第 $n-1$ 个苹果不能选，方案数为 $f(n-2)$
   - 递推公式：

$$f(n) = f(n-1) + f(n-2)$$

14. 设一个三位数 n=abc, a,b,c 均为 1~9 之间的整数，若以 a、b、c 作为三角形的三条边可以构成等腰三角形（包括等边），则这样的 n 有（ ）个。

A. 81　　B. 120　　C. 165　　D. 216

答案：C

分析：本题考查等腰三角形的组合计数。

解题步骤

分类情况：
情况 1：a=b 且 2a>c
情况 2：a=c 且 2a>b
情况 3：b=c 且 2b>a

计算每种情况的数量：

对于情况 1：a=b，a 从 1 到 9，对于每个 a，c 需满足 $1 \leq c \leq \min(9, 2a-1)$。

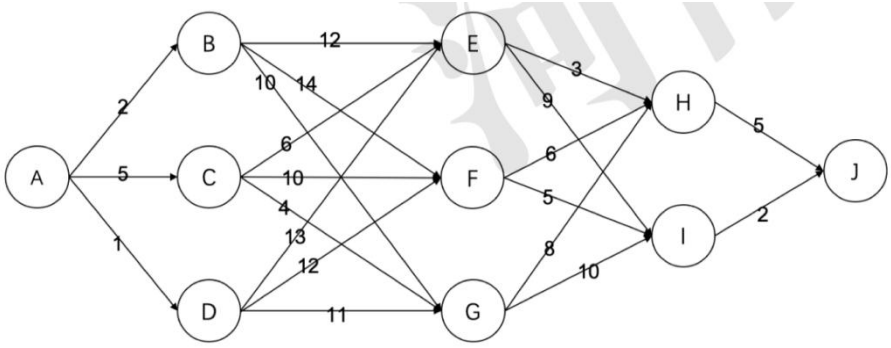
a=1：c=1（1 个）
a=2：c=1,2,3（3 个）
a=3：c=1,2,3,4,5（5 个）
a=4：c=1,2,3,4,5,6,7（7 个）
a=5：c=1,2,3,4,5,6,7,8,9（9 个）
a=6：c=1 到 9（9 个）
a=7：c=1 到 9（9 个）
a=8：c=1 到 9（9 个）
a=9：c=1 到 9（9 个）
总和：1+3+5+7+9+9+9+9+9=61

情况 2：a=c 且 2a>b，计算方式与情况 1 相同，因此也有 61 个三元组。
情况 3：b=c 且 2b>a，计算方式与情况 1 相同，因此也有 61 个三元组。

初步总和：61+61+61=183
处理重复计数：
等边三角形（a=b=c）在所有三种情况中都被计数，因此每个等边三角形被计数了 3 次。
等边三角形的数量：a 从 1 到 9，共 9 个。
因此，需要减去额外计数的 2 次 per 等边三角形，即减去 2×9=18。
最终唯一三元组数量：183−18=165

15. 有如下的有向图，节点为 A,B,⋯ ,J，其中每条边的长度都标在图中。则节点 A 到节点 J 的最短路径长度为（ ）。

A. 16　　B. 19　　C. 20　　D. 22



答案：B

分析：本题考查最短路径计算。

通过 Dijkstra 算法计算，A→C→F→J 的路径长度为 10+5+4=19，为最短路径，因此正确答案为 B。

二、阅读程序

1. （本题共 12 分）

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std ; .
```

```
4 const double r = acos(0.5);
5 int a1, b1, c1, d1;
6 int a2, b2, C2, d2;
7 inline int sq(const int x) { return x * x; }
8 inline int cu(const int x) { return x*x*x;}
9 int main()
10 {
11  cout.flags(ios::fixed);
12  cout.precision(4);
13  cin>>a1>>b1>>c1>>d1;
14  cin>>a2>>b2>>c2>>d2;
15  int t=sq(a1-a2)+sq(b1-b2)+sq(c1-c2);
16  if (t <= sq(d2 - d1)) cout << cu(min(d1, d2)) *r*4;
17  else if (t >= sq(d2 + d1)) cout << 0;
18  else {
19    double x = d1 - (sq(d1) - sq(d2) + t) / sqrt(t) / 2;
20    double y = d2 - (sq(d2) - sq(d1) + t) / sqrt(t) / 2;
21    cout << (x*x*(3*d1-x)+y*y*(3*d2-y))*r;
22   }
23  cout << endl;
24  return 0;
25 }
```

假设输入的所有数的绝对值都不超过 1000 ，完成下面的判断题和单选题：

●判断题

(1)将第 15 行中 t 的类型声明从 int 改为 double，不会影响程序运行的结果。(　　)

(2)将第 19、20 行中的/sqrt(t)/2 替换为/2/sqrt(t),不会影响程序运行的结果。(　　)

(3)将第 21 行中的 x*x 改成 sq(x)、y*y 改成 sq(y)，不会影响程序运行的结果。(　　)

(4)(2 分)当输入为 0 0 0 1 1 0 0 1 时，输出为 1.3090 (　　)

●单选题

(5)当输入为 1 1 1 1 1 1 1 2 时，输出为（ ）。

A. 3.1416　　B. 6.2832　　C. 4.7124　　D. 4.1888

(6)(2.5 分)这段代码的含义为（ ）。

A. 求圆的面积并　　B. 求球的体积并

C. 求球的体积交　　D. 求椭球的体积并

假设输入的所有数的绝对值都不超过 1000，完成下面的判断题和单选题:

该程序用于计算两个球体的体积交，输入为两个球的三维球心坐标和半径，输出根据两球的位置关系（外离、内含、相交）计算并输出体积交的结果（保留 4 位小数）。

内含：体积交为较小球的体积；
相离：体积交为 0；
相交：体积交为两个球缺体积之和。

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

```
// 定义常量 r = acos(0.5) = π/3（用于体积公式计算）
const double r = acos(0.5);

// 两个球的球心坐标（a1,b1,c1）、（a2,b2,c2）和半径 d1、d2
int a1, b1, c1, d1;
int a2, b2, c2, d2;

// 内联函数：计算整数的平方（返回 int）
inline int sq(const int x) { return x * x; }

// 内联函数：计算整数的立方（返回 int）
inline int cu(const int x) { return x*x*x; }

int main()
{
    // 设置输出格式为固定小数位，精度 4 位
    cout.flags(ios::fixed);
    cout.precision(4);

    // 读取两个球的参数：球心坐标和半径
    cin >> a1 >> b1 >> c1 >> d1;
    cin >> a2 >> b2 >> c2 >> d2;

    // 计算两球心距离的平方（三维空间：(x1-x2)2+(y1-y2)2+(z1-z2)2）
    int t = sq(a1-a2) + sq(b1-b2) + sq(c1-c2);  //  （1）答案：改为 double 不影响结果

    // 情况 1：两球内含（距离 ≤ |半径差|），体积交为较小球的体积
    if (t <= sq(d2 - d1))
        cout << cu(min(d1, d2)) * r * 4;  // 4*r = 4π/3，即球体积公式(4/3)πR3

    // 情况 2：两球外离（距离 ≥ 半径和），体积交为 0
    else if (t >= sq(d2 + d1))
        cout << 0;

    // 情况 3：两球相交，计算相交部分体积（球缺体积和）
    else {
        // 计算两球缺的高（（2）答案：/sqrt(t)/2 与/2/sqrt(t)等价）
```

```
            double x = d1 - (sq(d1) - sq(d2) + t) / sqrt(t) / 2;
            double y = d2 - (sq(d2) - sq(d1) + t) / sqrt(t) / 2;

            // 球缺体积公式：πh2(3R-h)/3，整体乘以 r=π/3 得最终结果
            // （3）答案：x*x 不能改为 sq(x)，因 x 是 double 会被截断
            cout << (x*x*(3*d1 - x) + y*y*(3*d2 - y)) * r;
        }
        cout << endl;
        return 0;
}
```

●判断题

(1) 将第 15 行中 t 的类型声明从 int 改为 double，不会影响程序运行的结果。( )

答案：√
分析：t 是两点间距离的平方，int 和 double 均可表示，且后续计算中 t 会被转换为 double 参与运算，因此类型修改不影响结果。

(2) 将第 19、20 行中的 /sqrt (t)/2 替换为 / 2/sqrt (t)，不会影响程序运行的结果。( )

答案：×

分析：因为前的变量都是 int 类型，先除以 2 会做整数除 2 处理。

(3) 将第 21 行中的 *xx 改成 sq (x)*、yy 改成 sq (y)，不会影响程序运行的结果。( )

答案：×

分析：sq 函数参数为 int，而 x、y 是 double 类型，传递时会发生截断（如 x=2.5 会转为 2），导致 sq(x)与 x*x 结果不同。

(4) (2 分) 当输入为 0 0 0 1 1 0 0 1 时，输出为 1.3090 ( )

答案：√

分析：输入对应两个球，球心距离的平方 t=1，两球半径均为 1。由于 $t \le (d2-d1)^2=0$ 不成立，$t \ge (d2+d1)^2=4$ 也不成立，进入 else 分支。计算得 x=0.5，y=0.5，代入公式得 $(0.5^2\times(3\times1-0.5)+0.5^2\times(3\times1-0.5))\times\pi/3\approx1.3090$，故正确。

●单选题

(5) 当输入为 1 1 1 1 1 1 1 2 时，输出为（ ）。

A. 3.1416 B. 6.2832 C. 4.7124 D. 4.1888

答案：D

分析：两球心重合（t=0），半径分别为 1 和 2。由于 t≤(2-1)²=1，输出为较小球的体积：(4/3)π×1³≈4.1888，故正确答案为 D。

(6) (2.5 分) 这段代码的含义为（ ）。

A. 求圆的面积并　　B. 求球的体积并

C. 求球的体积交　　D. 求椭球的体积并

答案：C

分析：代码中 r=acos (0.5)=π/3，涉及球心坐标（a,b,c）和半径 d，计算逻辑符合两球体积交的公式，因此正确答案为 C。

2. （本题共 13.5 分）

17.（本题共 13.5 分）

```
1 #include <algorithm>
2 #include <iostream>
3 using namespace std;
4 int n, a[1005];
5 struct Node
6 {
7    int h,j,m,W;
8    Node(const int_ h, const int_ j, const int_ m,  const int_ w):
9      h(_ h), j(_ _j), m(_ m), W(. w)
10     {}
11 Node operator+(const Node &o) const
12  {
13   return Node(
14        max(h, W + o.h),
15        max(max(j, o.j), m + o.h),
16        max(m + O.w, o.m),
17        w + o.w);
18   }
19  };
20  Node solve1(int h, int m)
21 {
22   if(h> m)
23     return Node(-1, -1, -1, -1);
24   if(h==m)
25     return Node(max(a[h], 0), max(a[h], 0), max(a[h], 0), a[h]);
26   int j =(h+ m)>>1;
27     return solve1(h, j) + solve1(j + 1, m);
28  }
29  int solve2(int h, int m)
30   {
31    if(h> m)
32      return -1;
```

```
33    if(h==m)
34      return max(a[h], 0);
35    int j=(h+m)>>1;
36    int wh=0,wm=0;
37    int wht=0, wmt=0;
38    for(int i=j;i>=h;i--){
39      wht += a[i];
40      wh = max(wh, wht);
41    }
42   for(int i=j+1;i<=m;i++){
43     wmt += a[i];
44     wm = max(wm, wmt);
45   }
46    return max(max(solve2(h, j), solve2(j + 1, m)), wh + wm);
47}
48int main()
49{
50  cin>>n;
51   for(int i=1;i<=n;i++)cin>>a[i];
52   cout << solve1(1, n).j << endl;
53   cout << solve2(1, n) << endl;
54   return 0;
55 }
```

假设输入的所有数的绝对值都不超过 1000 ，完成下面的判断题和单选题：

●判断题

(1)程序总是会正常执行并输出两行两个相等的数。（　　）

(2)第 23 行与第 32 行分别有可能执行两次及以上。（　　）

(3)当输入为 5 -10 11 -9 5 -7 时，输出的第二行为 7。（　　）

●单选题

(4)solve1(1, n) 的时间复杂度为（ ）。

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

(5)solve2(1, n) 的时间复杂度为（ ）。

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

(6)当输入为 10 -3 2 10 0 -8 9 -4 -5 9 4 时，输出的第一行为（ ）。

A. 13　　B. 17　　C. 24　　D. 12

假设输入的所有数的绝对值都不超过 1000，完成下面的判断题和单选题:

该程序通过两种分治算法计算**数组的最大子段和**（即连续子数组的最大和）：

- `solve1` 利用结构体 `Node` 存储区间的关键信息（最大前缀和、最大后缀和、区间总和、最大子段和），通过重载 `+` 运算符合并子区间结果。
- `solve2` 是经典的分治实现，直接递归计算左右区间的最大子段和，再计算跨越中点的最大子段和，取三者的最大值。

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

```
int n, a[1005];   // n 为数组长度，a 存储输入的数组

// 结构体用于 solve1 的分治合并，存储区间的关键信息
struct Node
{
    int h;   // 区间前缀最大和（从左端点开始的最大子段和）
    int j;   // 区间最大子段和（本题核心结果）
    int m;   // 区间后缀最大和（以右端点结束的最大子段和）
    int W;   // 区间总和
    // 构造函数
    Node(const int _h, const int _j, const int _m, const int _w) :
        h(_h), j(_j), m(_m), W(_w)
    {}
    // 重载+运算符，用于合并两个相邻区间的 Node 信息
    Node operator+(const Node &o) const
    {
        return Node(
            max(h, W + o.h),   // h：前缀最大和（从左端点开始的最大子段和）；
            max(max(j, o.j), m + o.h),   //j：区间最大子段和（最终结果）；
            max(m + o.W, o.m),   // m：后缀最大和（以右端点结束的最大子段和）；
            W + o.W   // 新总和：左右总和相加
        );
    }
};

// 分治计算区间[h,m]的 Node 信息（用于求最大子段和）
Node solve1(int h, int m)//返回的是 Node 类型
{
    if (h > m)
        return Node(-1, -1, -1, -1);
    if (h == m)   // 单元素区间
    {
        int val = max(a[h], 0);   // 单个元素的最大子段和（非负）
        return Node(val, val, val, a[h]);   // 前缀、后缀、最大子段和均为 val，总和为 a[h]
    }
    int j = (h + m) >> 1;   // 中点（右移 1 位等价于除以 2 取整）
    // 递归合并左右区间（4）答案：时间复杂度 O(n)
    return solve1(h, j) + solve1(j + 1, m);
}


// 分治计算区间[h,m]的最大子段和（经典实现）
int solve2(int h, int m)//返回的是整数类型
{
    if (h > m)   // 空区间，返回无效值（2）答案：最多执行 1 次
        return -1;
    if (h == m)   // 单元素区间，返回非负值
        return max(a[h], 0);
    int j = (h + m) >> 1;   // 中点
    // 计算左区间后缀最大和（从 j 向左累加的最大值）
    int wh = 0, wht = 0;
```

```
    for (int i = j; i >= h; i--)
    {
        wht += a[i];
        wh = max(wh, wht);
    }
    // 计算右区间前缀最大和（从 j+1 向右累加的最大值）
    int wm = 0, wmt = 0;
    for (int i = j + 1; i <= m; i++)
    {
        wmt += a[i];
        wm = max(wm, wmt);
    }
    // 最大子段和为左区间最大、右区间最大、跨区间最大（wh+wm）的最大值（5）答案：时间复杂度 O(n log n)
    return max(max(solve2(h, j), solve2(j + 1, m)), wh + wm);
}

int main()
{
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> a[i];
    // 输出 solve1 的最大子段和（j 成员）
        cout << solve1(1, n).j << endl;
    // 输出 solve2 的最大子段和
    cout << solve2(1, n) << endl;
    return 0;
}
```

下面通过一个具体例子详细说明程序的逻辑。以输入数组 a = [1, -2, 3, 4, -5]（n=5）为例，分析 solve1 和 solve2 如何计算最大子段和（结果应为 3+4=7）。

一、数组与分治思路

数组下标为 1~5，值为：a[1]=1，a[2]=-2，a[3]=3，a[4]=4，a[5]=-5。

分治核心思路：将区间 [h, m] 从中间 j=(h+m)/2 分为左区间 [h, j] 和右区间 [j+1, m]，递归求解后合并结果。

二、solve2 函数的执行过程（直观分治）

solve2(h, m) 直接返回区间 [h, m] 的最大子段和，步骤为：

若区间为空（h>m），返回 -1（无效值）；

若区间为单个元素（h=m），返回 max(a[h], 0)（非负的单个元素）；

否则分左、右区间递归，再计算跨区间最大和（左区间后缀最大和 + 右区间前缀最大和），最终返回三者的最大值。

步骤拆解（以 solve2(1,5) 为例）

初始调用：solve2(1,5)，中点 j=(1+5)/2=3，分左区间 [1,3] 和右区间 [4,5]。

a = [1, -2, 3, 4, -5]

左区间 [1,3] 计算：

中点 j=(1+3)/2=2，分 [1,2] 和 [3,3]。
[3,3] 是单元素：max(a[3],0)=3。
[1,2] 计算：
中点 j=1，分 [1,1] 和 [2,2]。
[1,1]：max(1,0)=1；[2,2]：max(-2,0)=0。

跨区间和：左后缀（a[1]=1）+ 右前缀（a[2]=-2 的最大累加为 0）→ 1+0=1。
[1,2] 结果：max(1, 0, 1)=1。
跨区间和（[1,2] 后缀 + [3,3] 前缀）：
左后缀（[1,2] 从 j=2 向左累加：-2→1-2=-1，最大为 1）。
右前缀（[3,3] 从 j+1=3 向右累加：3，最大为 3）。
跨区间和：1+3=4。
[1,3] 结果：max(1, 3, 4)=4。

右区间 [4,5] 计算：

中点 j=(4+5)/2=4，分 [4,4] 和 [5,5]。
[4,4]：max(4,0)=4；[5,5]：max(-5,0)=0。
跨区间和：左后缀（a[4]=4）+ 右前缀（a[5]=-5 的最大累加为 0）→ 4+0=4。
[4,5] 结果：max(4, 0, 4)=4。

合并 [1,5] 结果：

跨区间和（[1,3] 后缀 + [4,5] 前缀）：
左后缀（[1,3] 从 j=3 向左累加：3→-2+3=1→1-2+3=2，最大为 3）。
右前缀（[4,5] 从 j+1=4 向右累加：4→4-5=-1，最大为 4）。
跨区间和：3+4=7。
最终结果：max(4, 4, 7)=7。

三、solve1 函数的执行过程（结构体合并）

solve1(h, m) 返回 Node 结构体，包含区间的 4 个关键值：

h：前缀最大和（从左端点开始的最大子段和）；
j：区间最大子段和（最终结果）；
m：后缀最大和（以右端点结束的最大子段和）；
W：区间总和。

通过重载 operator+ 合并两个 Node，核心是利用左右区间的 h/m/W 快速计算新区间的 h/m/j/W。
步骤拆解（以 solve1(1,5) 为例）

单个元素的 Node：

如 [3,3]（a[3]=3）：
h=3（前缀最大和），j=3（最大子段和），m=3（后缀最大和），W=3（总和）。

合并两个相邻区间的 Node：
若左区间为 L，右区间为 R，则合并后的 Node 计算规则：

新 h：max(L.h, L.W + R.h)（左前缀 或 左总和 + 右前缀）；
新 j：max(L.j, R.j, L.m + R.h)（左最大、右最大、跨区间最大）；
新 m：max(L.m + R.W, R.m)（左后缀 + 右总和 或 右后缀）；

新 W：L.W + R.W（总和相加）。

逐步合并过程：

合并 [1,1] 和 [2,2]：

[1,1]：L=(1,1,1,1)；[2,2]：R=(0,0,0,-2)（因 a[2]=-2，非负处理后为 0）。
新 h：max(1, 1 + 0)=1；新 j：max(1, 0, 1 + 0)=1；
新 m：max(1 + (-2), 0)=max(-1,0)=0；新 W：1 + (-2)=-1；
结果：(1,1,0,-1)（对应区间 [1,2]）。

合并 [1,2] 和 [3,3]：

[1,2]：L=(1,1,0,-1)；[3,3]：R=(3,3,3,3)。
新 h：max(1, -1 + 3)=2；新 j：max(1, 3, 0 + 3)=3；
新 m：max(0 + 3, 3)=3；新 W：-1 + 3=2；
结果：(2,3,3,2)（对应区间 [1,3]）。

合并 [4,4] 和 [5,5]：

[4,4]：L=(4,4,4,4)；[5,5]：R=(0,0,0,-5)。
新 h：max(4, 4 + 0)=4；新 j：max(4, 0, 4 + 0)=4；
新 m：max(4 + (-5), 0)=max(-1,0)=0；新 W：4 + (-5)=-1；
结果：(4,4,0,-1)（对应区间 [4,5]）。

合并 [1,3] 和 [4,5]：

[1,3]：L=(2,3,3,2)；[4,5]：R=(4,4,0,-1)。
新 h：max(2, 2 + 4)=6；新 j：max(3, 4, 3 + 4)=7；
新 m：max(3 + (-1), 0)=2；新 W：2 + (-1)=1；
结果：(6,7,2,1)（对应区间 [1,5]）。

最终结果：
solve1(1,5).j=7，与 solve2 结果一致。

●判断题

(1) 程序总是会正常执行并输出两行两个相等的数。( )

答案：√

solve1 和 solve2 均用于计算最大子段和：solve1 的 j 成员存储最大子段和，solve2 直接返回最大子段和，逻辑上结果必然相等。输入合法时（n ≥ 1）可正常执行。

(2) 第 23 行与第 32 行分别有可能执行两次及以上。( )

答案：×

分析：两个边界值，只会执行 1 次。

(3) 当输入为 5 -10 11 -9 5 -7 时，输出的第二行为 7。( )

答案：×

分析：

- 输入实际为 n=5，数组 a = [-10, 11, -9, 5, -7]。
- 最大子段和为 11（子数组 [11]），而非 7。因此第二行（solve2 结果）应为 11，故该题错误。

●单选题

(4) solve1 (1, n) 的时间复杂度为（ ）。

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

solve1 (1, n) 的时间复杂度为（B）
分析：solve1 采用分治，每次将问题分为 2 个规模为 n/2 的子问题，合并操作（operator+）为常数时间 O(1)。
递归式：T(n) = 2T(n/2) + O(1)，根据主定理，时间复杂度为 O(n)。

(5) solve2 (1, n) 的时间复杂度为（ ）。

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

solve2 (1, n) 的时间复杂度为（C）。
solve2 同样分治，但合并时需遍历左右区间计算跨越中点的最大和，耗时 O(n)。
递归式：T(n) = 2T(n/2) + O(n)，根据主定理，时间复杂度为 O(n log n)。

(6) 当输入为 10 -3 2 10 0 -8 9 -4 -5 9 4 时，输出的第一行为（ ）。

A. 13　　B. 17　　C. 24　　D. 12

答案：B

分析：
输入为 n=10，数组 a = [-3, 2, 10, 0, -8, 9, -4, -5, 9, 4]。
计算最大子段和：子数组 [2, 10, 0, -8, 9, -4, -5, 9, 4] 的和为 2+10+0-8+9-4-5+9+4=17，为最大值。
故 solve1(1, n).j 的结果为 17。

18.（本题共 14.5 分）

```
01 #include <iostream>
02 #include <string>
03 using namespace std;
04
05 char base[64];
06 char table[256];
07
08 void init()
```

```
09 {
10    for (int i = 0; i < 26; i++) base[i] = 'A' + i;
11    for (int i = 0; i < 26; i++) base[26 + i] = 'a' + i;
12    for (int i = 0; i < 10; i++) base[52 + i] = '0' + i;
13    base[62] = '+', base[63] = '/';
14
15    for (int i = 0; i < 256; i++) table[i] = 0xff;
16    for (int i = 0; i < 64; i++) table[base[i]] = i;
17    table['='] = 0;
18 }
19
20 string encode(string str)
21 {
22    string ret;
23    int i;
24    for (i = 0; i + 3 <= str.size(); i += 3) {
25        ret += base[str[i] >> 2];
26        ret += base[(str[i] & 0x03) << 4 | str[i + 1] >> 4];
27        ret += base[(str[i + 1] & 0x0f) << 2 | str[i + 2] >> 6];
28        ret += base[str[i + 2] & 0x3f];
29    }
30    if (i < str.size()) {
31        ret += base[str[i] >> 2];
32        if (i + 1 == str.size()) {
33            ret += base[(str[i] & 0x03) << 4];
34           ret += "==";
35        }
36        else {
37           ret += base[(str[i] & 0x03) << 4 | str[i + 1] >> 4];
38           ret += base[(str[i + 1] & 0x0f) << 2];
39           ret += "=";
40        }
41    }
42    return ret;
43 }
44
45 string decode(string str)
46 {
47    string ret;
48    int i;
49    for (i = 0; i < str.size(); i += 4) {
50        ret += table[str[i]] << 2 | table[str[i + 1]] >> 4;
51         if (str[i + 2] != '=')
52            ret += (table[str[i + 1]] & 0x0f) << 4 | table[str[i +2]] >> 2;
53         if (str[i + 3] != '=')
54            ret += table[str[i + 2]] << 6 | table[str[i + 3]];
55     }
56    return ret;
57 }
58
59 int main()
```

```
60 {
61     init();
62     cout << int(table[0]) << endl;
63 
64     int opt;
65     string str;
66     cin >> opt >> str;
67     cout << (opt ? decode(str) : encode(str)) << endl;
68     return 0;
69 }
```

假设输入总是合法的（一个整数和一个不含空白字符的字符串，用空格隔开），完成下面的判断题和单选题：

●判断题

(1)程序总是先输出一行一个整数，再输出一行一个字符串。（　　）

(2)对于任意不含空白字符的字符串 str1，先执行程序输入 0 str1，得到输出的第二行记为 str2；再执行程序输入 1 str2，输出的第二行必为 str1。（　　）

(3)当输入为 1 SGVsbG93b3JsZA==时，输出的第二行为 HelloWorld。（　　）

●单选题

(4)设输入字符串长度为 n，encode 函数的时间复杂度为（ ）。

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

(5)输出的第一行为（ ）。

A. 0xff　　B. 255　　C. 0xFF　　D. -1

(6)(4 分) 当输入为 0 CSP2021csp 时，输出的第二行为（ ）。

A. Q1NQMjAyMWNzcAv=　　B. Q1NQMjAyMGNzcA==

C. Q1NQMjAyMGNzcAv=　　D. Q1NQMjAyMWNzcA==

假设输入总是合法的（一个整数和一个不含空白字符的字符串，用空格隔开），完成下面的判断题和单选题：

该程序实现了 Base64 编码与解码功能：

Base64 是一种基于 64 个可打印字符的编码方式，用于将二进制数据转换为文本格式，常用于网络传输。

程序通过 init 初始化编码表（base）和解码表（table），encode 函数将输入字符串编码为 Base64 格式，decode 函数将 Base64 字符串解码为原始字符串。

- init 函数初始化 Base64 编码表（64 个字符）和解码表（字符到索引的映射）。
- encode 函数将输入字符串编码为 Base64 格式（3 字节→4 字符，不足 3 字节用=填充）。
- decode 函数将 Base64 格式字符串解码为原始字符串（4 字符→3 字节，忽略=填充）。
- 主函数根据输入的操作类型（0 编码 / 1 解码）输出对应结果，且固定先输出 table[0] 的整数形式。

核心知识点

1. **Base64 编码原理**：将二进制数据按 3 字节（24 位）分组，拆分为 4 个 6 位块，每个块对应编码表中的一个字符；不足 3 字节时用=填充（1 字节补 2 个=，2 字节补 1 个=）。
2. **编码表与解码表**：编码表包含 64 个可打印字符（A-Z、a-z、0-9、+、/），解码表用于将字符映射回原始 6 位索引。
3. **位运算**：通过移位（>>、<<）和按位操作（&、|）实现字节与 6 位块的转换。

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

char base[64];      // Base64 编码表：映射 6 位数值到字符
char table[256];    // Base64 解码表：映射字符到 6 位数值

// 初始化编码表和解码表
void init()
{
    // 填充编码表：A-Z(0-25)、a-z(26-51)、0-9(52-61)、+(62)、/(63)
    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] = '/';

    // 初始化解码表为 0xff
    //table['A']=0，table['a']=26，table['0']=52，table['+']=62，table['x']=0xff（x 不在 base 中）。
    for (int i = 0; i < 256; i++) table[i] = 0xff;
    // 为编码表中的字符赋值对应的 6 位数值
    for (int i = 0; i < 64; i++) table[base[i]] = i;
    // '='用于填充，解码时视为 0
    table['='] = 0;
}
```

```
// Base64 编码：将输入字符串转换为 Base64 编码
string encode(string str)
{
    string ret;
    int i;
    // 处理 3 字节一组的部分
    for (i = 0; i + 3 <= str.size(); i += 3) {
        // 第 1 个 6 位：第 1 字节右移 2 位
        ret += base[str[i] >> 2]; //如果输入 ABC ,此时 ret="Q"
        // 第 2 个 6 位：第 1 字节低 2 位左移 4 位 | 第 2 字节高 4 位
        ret += base[(str[i] & 0x03) << 4 | str[i + 1] >> 4];
        // 第 3 个 6 位：第 2 字节低 4 位左移 2 位 | 第 3 字节高 2 位
        ret += base[(str[i + 1] & 0x0f) << 2 | str[i + 2] >> 6];
        // 第 4 个 6 位：第 3 字节低 6 位
        ret += base[str[i + 2] & 0x3f];
    }
```

假设 str= "ABC"（3 个字节，正好满足循环条件 i + 3 <= str.size()），逐行拆解 encode 函数中 3 字节一组的处理流程)

步骤 1：明确原始数据的二进制

首先，"ABC" 中每个字符的 ASCII 值及二进制（8 位）如下：

str[0] = 'A'：ASCII 值为 65 → 二进制 01000001

str[1] = 'B'：ASCII 值为 66 → 二进制 01000010

str[2] = 'C'：ASCII 值为 67 → 二进制 01000011

这 3 个字节共 24 位，需要拆分成 4 个 6 位组（24÷6=4），每个 6 位组对应 base 表中的一个字符。

步骤 2：循环内逐行代码解析（对应 "ABC" 的处理）

循环条件：i=0 时，i+3=3，str.size()=3，满足 i+3 <= str.size()，进入循环处理这 3 个字节。

第 1 行：ret += base[str[i] >> 2];（计算第 1 个 6 位组）

作用：提取第 1 个字节（'A'）的高 6 位，作为第 1 个 6 位组。

位运算拆解：

str[i]是 'A' 的二进制 01000001，>> 2 表示右移 2 位（丢弃低 2 位）：

01000001 >> 2 = 010000（二进制，共 6 位）→ 十进制值为 16。

映射到 base 表：base[16]对应 'A'+16='Q'（因为 base [0]='A'，base [1]='B'……base [16]='Q'）。

结果：ret 现在为"Q"。

第 2 行：ret += base[(str[i] & 0x03) << 4 | str[i + 1] >> 4];（计算第 2 个 6 位组）

作用：组合第 1 个字节的低 2 位和第 2 个字节的高 4 位，得到第 2 个 6 位组。

分步拆解：

str[i] & 0x03：0x03 是十六进制，二进制为 00000011（掩码），用来提取第 1 个字节的低 2 位：
'A'的二进制 01000001 & 00000011 = 00000001（低 2 位是 01）。
<< 4：将提取的低 2 位左移 4 位，补 4 个 0，变成高 2 位：
00000001 << 4 = 00010000（二进制，此时占 6 位中的高 2 位）。
str[i+1] >> 4：提取第 2 个字节（'B'）的高 4 位（右移 4 位丢弃低 4 位）：
'B' 的二进制 01000010 >> 4 = 0100（二进制，占 4 位）。
|（或运算）组合前两步结果：
00010000 | 00000100 = 00010100（二进制，共 6 位）→ 十进制值为 20。
映射到 base 表：base[20]对应 'A'+20='U'。
结果：ret 现在为"QU"。

第 3 行：ret += base[(str[i + 1] & 0x0f) << 2 | str[i + 2] >> 6];（计算第 3 个 6 位组）

作用：组合第 2 个字节的低 4 位和第 3 个字节的高 2 位，得到第 3 个 6 位组。

分步拆解：

str[i+1] & 0x0f：0x0f 是十六进制，二进制为 00001111（掩码），提取第 2 个字节的低 4 位：
'B' 的二进制 01000010 & 00001111 = 00000010（低 4 位是 0010）。
<< 2：将提取的低 4 位左移 2 位，补 2 个 0，变成高 4 位：
00000010 << 2 = 00001000（二进制）。
str[i+2] >> 6：提取第 3 个字节（'C'）的高 2 位（右移 6 位丢弃低 6 位）：
'C' 的二进制 01000011 >> 6 = 01（二进制，占 2 位）。
|（或运算）组合前两步结果：
00001000 | 00000001 = 00001001（二进制，共 6 位）→ 十进制值为 9。
映射到 base 表：base[9]对应 'A'+9='J'。
结果：ret 现在为"QUJ"。

第 4 行：ret += base[str[i + 2] & 0x3f];（计算第 4 个 6 位组）

作用：提取第 3 个字节的低 6 位，作为第 4 个 6 位组。
位运算拆解：
0x3f 是十六进制，二进制为 00111111（掩码），用来提取低 6 位：
'C' 的二进制 01000011 & 00111111 = 00000011（二进制，共 6 位）→ 十进制值为 3。
映射到 base 表：base[3]对应 'A'+3='D'。
结果：ret 现在为"QUJD"。

总结：3 字节组的完整流程
对 "ABC" 的处理结束后，循环执行 i += 3（i 变为 3），此时 i + 3 = 6 > str.size()=3，退出循环。最终得到的编码结果为"QUJD"，这正是 "ABC" 的标准 Base64 编码。

// 处理剩余不足 3 字节的部分

```
    if (i < str.size()) {
        ret += base[str[i] >> 2];   // 第 1 个 6 位
        if (i + 1 == str.size()) {   // 剩余 1 字节：补两个=
            ret += base[(str[i] & 0x03) << 4];   // 第 2 个 6 位（后 4 位补 0）
            ret += "==";
        }
```

如果剩余的是”A”, str[i]是 'A' 的二进制 01000001，>>2（右移 2 位，丢弃低 2 位）：
01000001 >> 2 = 010000（二进制，6 位）→ 十进制 16。
base[16]对应 'A'+16='Q' → ret 变为"Q"。

str[i] & 0x03：0x03 是二进制 00000011（掩码），提取 'A' 的低 2 位：
01000001 & 00000011 = 00000001（低 2 位是 01）。
<<4：将这 2 位左移 4 位，右侧补 4 个 0（凑 6 位）：
00000001 << 4 = 00010000（二进制，6 位）→ 十进制 16。
base[16]='Q' → ret 变为"QQ"。

补两个 "="：
因为 1 字节仅能拆出 2 个 6 位组（共 12 位），而 Base64 要求每 3 字节对应 4 个字符，不足部分用=填充（1 字节需补 2 个=）→ ret 最终为"QQ=="。

```
        else {   // 剩余 2 字节：补一个=   如果剩余的是”AB”,
            // 第 2 个 6 位：第 1 字节低 2 位左移 4 位 | 第 2 字节高 4 位
            ret += base[(str[i] & 0x03) << 4 | str[i + 1] >> 4];
            // 第 3 个 6 位：第 2 字节低 4 位左移 2 位（后 2 位补 0）
            ret += base[(str[i + 1] & 0x0f) << 2];
            ret += "=";
        }
    }
    return ret;
}
```

str[i] & 0x03：提取 'A' 的低 2 位（同前）：01000001 & 00000011 = 00000001（01）。
<<4：左移 4 位 → 00010000（二进制）。
str[i+1] >>4：提取 'B' 的高 4 位（右移 4 位，丢弃低 4 位）：
'B' 的二进制 01000010 >>4 = 0100（二进制，4 位）→ 00000100（补成 6 位）。
|（或运算）组合：00010000 | 00000100 = 00010100（二进制）→ 十进制 20。
base[20]='U' → ret 变为"QU"。

计算第 3 个 6 位组：

str[i+1] & 0x0f：0x0f 是二进制 00001111（掩码），提取 'B' 的低 4 位：
01000010 & 00001111 = 00000010（低 4 位是 0010）。
<<2：左移 2 位，右侧补 2 个 0（凑 6 位）：
00000010 << 2 = 00001000（二进制）→ 十进制 8。
base[8]='I' → ret 变为"QUI"。

补一个 "="：
2 字节可拆出 3 个 6 位组（共 18 位），需补 1 个=凑 4 个字符 → ret 最终为"QUI="。

```
// Base64 解码：将 Base64 编码字符串转换为原始字符串
string decode(string str)
{
    string ret;//解码 "QUJD" 假设输入 "QUJD"，  输出为“ABC”
    int i;
    // 4 字符一组处理
    for (i = 0; i < str.size(); i += 4) {
        // 第 1 字节：第 1 个 6 位左移 2 位 | 第 2 个 6 位高 4 位
        ret += table[str[i]] << 2 | table[str[i + 1]] >> 4;
        // 若第 3 个字符不是=，处理第 2 字节
        if (str[i + 2] != '=')
            ret += (table[str[i + 1]] & 0x0f) << 4 | table[str[i + 2]] >> 2;
        // 若第 4 个字符不是=，处理第 3 字节
        if (str[i + 3] != '=')
            ret += table[str[i + 2]] << 6 | table[str[i + 3]];
    }
    return ret;
}
```

例子 1：解码 "QUJD"（还原为 "ABC"）
编码串 str = "QUJD"，长度 4，循环 i=0（每次处理 4 字符，i += 4 后循环结束）。

步骤 1：解析 4 个字符的 6 位数值
str[0] = 'Q' → table['Q'] = 16 → 二进制 010000（6 位）；
str[1] = 'U' → table['U'] = 20 → 二进制 010100（6 位）；
str[2] = 'J' → table['J'] = 9 → 二进制 001001（6 位）；
str[3] = 'D' → table['D'] = 3 → 二进制 000011（6 位）。

步骤 2：第 1 行代码
ret += table[str[i]] << 2 | table[str[i + 1]] >> 4;

拆解：
table[str[0]] << 2：第 1 个 6 位组左移 2 位
010000 << 2 = 01000000（二进制，8 位）；
table[str[1]] >> 4：第 2 个 6 位组右移 4 位：
010100 >> 4 = 01（二进制，2 位）→ 补成 8 位为 00000001；
|（或运算）组合：01000000 | 00000001 = 01000001（二进制）→ 十进制 65 → 字符 'A'。
结果：ret 变为"A"。
步骤 3：第 2 行代码（生成第 2 个原始字节）
if (str[i + 2] != '=') ret += (table[str[i + 1]] & 0x0f) << 4 | table[str[i + 2]] >> 2;

拆解：
(table[str[1]] & 0x0f) << 4：
0x0f 是二进制 00001111（掩码），提取第 2 个 6 位组的低 4 位：
010100 & 00001111 = 00000100（4 位）；
左移 4 位：00000100 << 4 = 01000000（8 位）；
table[str[2]] >> 2：第 3 个 6 位组右移 2 位：
001001 >> 2 = 0010（4 位）→ 补成 8 位为 00000010；
|组合：01000000 | 00000010 = 01000010（二进制）→ 十进制 66 → 字符 'B'。
结果：ret 变为"AB"。

步骤 4：第 3 行代码（生成第 3 个原始字节，因 str[3] != '='）
if (str[i + 3] != '=') ret += table[str[i + 2]] << 6 | table[str[i + 3]];

拆解：
table[str[2]] << 6：第 3 个 6 位组左移 6 位：
001001 << 6 = 001001000000（12 位）→ 截取低 8 位为 01000000；
table[str[3]]：第 4 个 6 位组（直接作为 8 位的低 6 位）：000011 → 补成 8 位为 00000011；
|组合：01000000 | 00000011 = 01000011（二进制）→ 十进制 67 → 字符 'C'。
结果：ret 变为"ABC"。

```
int main()
{
    init();
    // 输出 table[0]的整数形式（5）答案：-1
```

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


    int opt;
    string str;
    cin >> opt >> str;
    // 根据 opt 选择编码（0）或解码（1）（1）答案：总是先输出整数再输出字符串
    cout << (opt ? decode(str) : encode(str)) << endl;
    return 0;
}
```

编码过程（encode 函数）

encode 函数将输入字符串转换为 Base64 编码，核心是将 3 字节原始数据拆分为 4 个 6 位组，再映射到 base 表中的字符。若原始数据长度不是 3 的倍数，用=填充。

示例 1：编码"ABC"（长度 3，无填充）

"ABC"的 ASCII 值为：'A'=65（01000001）、'B'=66（01000010）、'C'=67（01000011）。

编码步骤：

分组：3 字节为一组，共 24 位：
01000001 01000010 01000011（二进制）
拆分 6 位组：24 位拆分为 4 个 6 位：
第 1 组：010000（前 6 位）
第 2 组：010100（中间 6 位）
第 3 组：001001（中间 6 位）
第 4 组：000011（后 6 位）
映射到 base 表：
第 1 组 010000=16 → base[16]='Q'
第 2 组 010100=20 → base[20]='U'
第 3 组 001001=9 → base[9]='J'
第 4 组 000011=3 → base[3]='D'
结果："QUJD"（即"ABC"的 Base64 编码）。

示例 2：编码"AB"（长度 2，需填充 1 个=）
"AB"的 ASCII 值：'A'=65（01000001）、'B'=66（01000010）。

编码步骤：

分组：2 字节共 16 位，补 8 位 0（凑 24 位）：
01000001 01000010 00000000（补 0 后）
拆分 6 位组：
第 1 组：010000（16）→ 'Q'
第 2 组：010100（20）→ 'U'
第 3 组：001000（8）→ 'I'
第 4 组：000000（0，但因原始数据不足 3 字节，用=填充）

结果："QUI="（填充 1 个=）。

示例 3：编码"A"（长度 1，需填充 2 个=）
"A"的 ASCII 值：'A'=65（01000001）。

编码步骤：

分组：1 字节共 8 位，补 16 位 0（凑 24 位）：
01000001 00000000 00000000（补 0 后）
拆分 6 位组：
第 1 组：010000（16）→ 'Q'
第 2 组：010000（16）→ 'Q'
第 3、4 组：因原始数据不足，用=填充
结果："QQ=="（填充 2 个=）。

●判断题

(1) 程序总是先输出一行一个整数，再输出一行一个字符串。( )

答案：×

分析：main 函数中先执行 cout << int (table [0]) << endl，再根据输入输出编码或解码结果，因此总是先输出整数，再输出字符串，但解码得到的字符串可以含有换行字符，所以不一定一行。填×。

(2) 对于任意不含空白字符的字符串 str1，先执行程序输入 0 str1，得到输出的第二行记为 str2；再执行程序输入 1 str2，输出的第二行必为 str1。( )

答案：√

分析：先 encode 再 decode,当然是一样的。填√。

(3) 当输入为 1 SGVsbG93b3JsZA== 时，输出的第二行为 HelloWorld。( )

答案：×

分析："SGVsbG93b3JsZA==" 是 "Helloworld"，是小写的 w。

```
60  string decode(string str)
61  {
62      string ret;
63      int i;
64      for (i = 0; i < str.size(); i += 4) {
65          ret += table[str[i]] << 2 | table[str[i + 1]] >> 4;
66          if (str[i + 2] != '=')
67              ret += (table[str[i + 1]] & 0x0f) << 4 | table[str[i + 2]] >> 2;
68          if (str[i + 3] != '=')
69              ret += table[str[i + 2]] << 6 | table[str[i + 3]];
70      }
71      return ret;
72  }
```

循环 1：i=0，处理字符 str[0]='S'、str[1]='G'、str[2]='V'、str[3]='s'

解析 4 个字符：

table['S']=18 → 二进制 010010；
table['G']=6 → 二进制 000110；
table['V']=21 → 二进制 010101；
table['s']=44 → 二进制 101100。

生成第 1 个原始字节（代码第 1 行）：
ret += table['S'] << 2 | table['G'] >> 4

18 << 2：010010 左移 2 位 → 01001000（十进制 72）；
6 >> 4：000110 右移 4 位 → 00000000（取高 2 位，结果 0）；
或运算：72 | 0 = 72 → 对应字符'H'（ASCII 72）。ret 变为"H"。

生成第 2 个原始字节（代码第 2 行，str[2]!='='）：
ret += (table['G'] & 0x0f) << 4 | table['V'] >> 2
(6 & 0x0f) << 4：000110 取低 4 位（00110）左移 4 位 → 01100000（96）；
21 >> 2：010101 右移 2 位 → 0101（5）；
或运算：96 | 5 = 101 → 对应字符'e'（ASCII 101）。
ret 变为"He"。

生成第 3 个原始字节（代码第 3 行，str[3]!='='）：
ret += table['V'] << 6 | table['s']

21 << 6：010101 左移 6 位 → 取低 8 位为 01000000（64）；
table['s']=44 → 二进制 101100（44）；
或运算：64 | 44 = 108 → 对应字符'l'（ASCII 108）。
ret 变为"Hel"。

解码完成后，ret 的值为"HelloWorld"，故程序第二行输出 HelloWorld。

●单选题

(4) 设输入字符串长度为 n，encode 函数的时间复杂度为（ ）。

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

B

分析：encode 函数通过循环处理输入字符串，每次迭代处理 3 个字符（循环次数为 O (n/3)），内部操作均为常数时间（位运算和字符串拼接），整体时间复杂度为 O (n)。

(5) 输出的第一行为（ ）。

A. 0xff　　B. 255　　C. 0xFF　　D. -1

答案：D

分析：char 类型在大多数编译器中默认是有符号的（signed char），其取值范围为-128 到 127。程序中的赋值语句 char x = 0xff;将十六进制值 0xff（即二进制的 11111111）赋给变量 x。由于 char 是有符号的，二进制 11111111 被解释为补码表示的负数，即-1。

当执行 cout << (int)x;时，将 x 强制转换为 int 类型。由于 x 的值为-1，转换后仍然是-1，因此输出为-1。故正确答案为 D。

```
26  string encode(string str)
27  {
28      string ret;
29      int i;
30      for (i = 0; i + 3 <= str.size(); i += 3) {
31          ret += base[str[i] >> 2];
32          ret += base[(str[i] & 0x03) << 4 | str[i + 1] >> 4];
33          ret += base[(str[i + 1] & 0x0f) << 2 | str[i + 2] >> 6];
34          ret += base[str[i + 2] & 0x3f];
35      }
36      if (i < str.size()) {
37          ret += base[str[i] >> 2];
38          if (i + 1 == str.size()) {
39              ret += base[(str[i] & 0x03) << 4];
40              ret += "==";
41          }
42          else {
43              ret += base[(str[i] & 0x03) << 4 | str[i + 1] >> 4];
44              ret += base[(str[i + 1] & 0x0f) << 2];
45              ret += "=";
46          }
47      }
48      return ret;
49  }
```

(6) (4 分) 当输入为 0 CSP2021csp 时，输出的第二行为（ ）。

A. Q1NQMjAyMWNzcAv=　　B. Q1NQMjAyMGNzcA==

C. Q1NQMjAyMGNzcAv=　　D. Q1NQMjAyMWNzcA==

答案：D

循环 i=0,3,6（每次处理 3 字节，步长 3）：
第 1 组：i=0（字符 0-2：'C','S','P'）
3 字节二进制拼接：01000011 01010011 01010000（共 24 位），拆分为 4 个 6 位组：

第 1 个 6 位组（代码 str[i] >> 2）：
取 'C' 的高 6 位：01000011 >> 2 = 010000（十进制 16）→ base[16]='Q'。

第 2 个 6 位组（代码(str[i]&0x03)<<4 | str[i+1]>>4）：
'C' 的低 2 位：01000011 & 0x03 = 11（十进制 3）；
左移 4 位：3 << 4 = 110000（48）；
'S' 的高 4 位：01010011 >> 4 = 0101（5）；
组合：48 | 5 = 53 → base[53]='1'（52 是 '0'，53 是 '1'）。

第 3 个 6 位组（代码(str[i+1]&0x0f)<<2 | str[i+2]>>6）：
'S' 的低 4 位：01010011 & 0x0f = 0011（3）；
左移 2 位：3 << 2 = 001100（12）；
'P' 的高 2 位：01010000 >> 6 = 01（1）；
组合：12 | 1 = 13 → base[13]='N'（A+13=N）。

第 4 个 6 位组（代码 str[i+2] & 0x3f）：
取 'P' 的低 6 位：01010000 & 0x3f = 010000（16）→ base[16]='Q'。

第 1 组编码结果："Q1NQ"。

拼接 3 组编码和剩余部分："Q1NQ" + "MjAy" + "MWNz" + "cA==" = "Q1NQMjAyMWNzcA==" 。

三、完善程序

19.(魔法数字)小 H 的魔法数字是 4。给定 n，他希望用若干个 4 进行若干次加法、减法和整除运算得到 n。但由于小 H 计算能力有限，计算过程中只能出现不超过 M = 10000 的正整数。求至少可能用到多少个 4。本题共 12 分
例如，当 n=2 时，有 2= (4+4)/4,用到了 3 个 4，是最优方案。
试补全程序。

```
01 #include <iostream>
02 #include <cstdlib>
03 #include <climits>
04
05 using namespace std;
06
07 const int M = 10000;
08 bool Vis[M + 1];
09 int F[M + 1];
10
11 void update(int &x, int y) {
12     if (y < x)
13         x = y;
14 }
15
16 int main() {
17     int n;
18     cin >> n;
```

```
19    for (int i = 0; i <= M; i++)
20        F[i] = INT_MAX;
21    ①;
22     int r = 0;
23    while (②) {
24        r++;
25        int x = 0;
26        for (int i = 1; i <= M; i++)
27           if (③)
28               x = i;
29        Vis[x] = 1;
30        for (int i = 1; i <= M; i++)
31           if (④) {
32               int t = F[i] + F[x];
33               if (i + x <= M)
34                   update(F[i + x], t);
35               if (i != x)
36                   update(F[abs(i - x)], t);
37               if (i % x == 0)
38                   update(F[i / x], t);
39               if (x % i == 0)
40                   update(F[x / i], t);
41           }
42    }
43    cout << F[n] << endl;
44    return 0;
45 }
```

(1)①处应填（ ）

A. F[4] = 0　　B. F[1] = 4　　C. F[1] = 2　　D.F[4] = 1

(2)②处应填（ ）

A. !Vis[n]　　B. r < n　　C. F[M] == INT_MAX　　D. F[n] == INT_MAX

(3)③处应填（ ）

A. F[i] == r　　B. !Vis[i] && F[i] == r

C. F[i] < F[x]　　D. !Vis[i] && F[i] < F[x]

(4)④处应填（ ）

A. F[i] < F[x]　　B. F[i]<=r　　C. Vis[i]　　D. i <= x

该程序计算从数字 4 开始，通过加法、减法、乘法和除法运算。程序的目标是找到生成给定数字 n（输入）所需的最小操作数。操作数定义为使用数字的次数，例如直接使用数字 4 算一次操作。程序采用类似 BFS 的方式进行搜索。

核心知识点

1. **动态规划与 BFS 结合：**F[x]存储最少 4 的个数（动态规划状态），Vis 数组标记已处理数字（BFS 的层级扩展控制），确保按 4 的个数递增扩展，保证解的最优性。
2. **状态转移：**通过加法（i+x）、减法（|i-x|）、整除（i/x 和 x/i）四种运算，从已有状态（已处理的 i 和当前 x）转移到新状态，更新新状态的最少 4 的个数。
3. **边界控制：**限制中间结果≤10000 且为正整数，避免无效状态。

```
#include <iostream>
#include <cstdlib>
#include <climits>   // 用于 INT_MAX，表示最大整数值

using namespace std;

const int M = 10000;   // 定义最大数字范围，即处理 0 到 10000 的数字
bool Vis[M + 1];       // 标记数组，Vis[i]表示数字 i 是否已被处理过
int F[M + 1];          // F[i]数组存储生成数字 i 的最小操作数。

// 更新函数：如果 y 小于 x，则将 x 更新为 y（用于维护最小操作数）
void update(int &x, int y) {
    if (y < x)
        x = y;
}

int main() {
    int n;
    cin >> n;   // 输入目标数字 n

    // 初始化 F 数组，将所有数字的操作数设为最大值（表示初始时不可达）
    for (int i = 0; i <= M; i++)
        F[i] = INT_MAX;

    F[4] = 1;   // 1 D 初始状态：数字 4 的操作数为 1（因为直接使用数字 4 本身）

    int r = 0;   // 循环计数器，记录 while 循环次数

    // 主循环：持续处理直到目标数字 n 被访问（即 Vis[n]为 true）
    while (!Vis[n]) { // 2 A
        r++;   // 循环次数增加

        int x = 0;   // 用于存储当前未访问的具有最小操作数的数字（初始 0，但 F[0]为 INT_MAX）

        // 遍历所有数字，找到未访问且操作数最小的数字 x
        for (int i = 1; i <= M; i++)
            if (!Vis[i] && F[i] < F[x]) // 3 D：如果 i 未访问且操作数小于当前 x 的操作数
                x = i;   // 更新 x 为 i

        Vis[x] = 1;   // 标记数字 x 为已访问

        // 遍历所有数字，对于每个已访问的数字 i，尝试与 x 进行四则运算来更新其他数字
        for (int i = 1; i <= M; i++)//
            if (Vis[i]) { // 4 C：如果数字 i 已被访问
                int t = F[i] + F[x];   // 新操作数：组合 i 和 x 所需操作数之和

                // 加法运算：i + x，如果结果不超过 M，则更新 F[i+x]
                if (i + x <= M)
                    update(F[i + x], t);//有点类似 DP 的状态转移，那个次数少就更新

                // 减法运算：|i - x|，如果 i 不等于 x，则更新 F[|i-x|]
                if (i != x)
                    update(F[abs(i - x)], t);
```

```
                // 除法运算：i / x，如果 i 能被 x 整除（即 i % x == 0），则更新 F[i/x]
                if (i % x == 0)
                    update(F[i / x], t);

                // 除法运算：x / i，如果 x 能被 i 整除（即 x % i == 0），则更新 F[x/i]
                if (x % i == 0)
                    update(F[x / i], t);
            }
        }

        cout << F[n] << endl;   // 输出得到数字 n 的最小操作数
        return 0;
}
```

(1) ①处应填（ ）

A. F [4] = 0　　B. F [1] = 4　　C. F [1] = 2　　D. F [4] = 1

答案：D

分析：程序需初始化基础状态 ——1 个 4 可以直接得到数字 4，因此 F[4]应设为 1。

(2) ②处应填（ ）

A. !Vis [n]　　B. r < n　　C. F [M] == INT_MAX　　D. F [n] == INT_MAX

答案：A

分析：如果 n 没有被访问过

(3) ③处应填（ ）

A. F [i] == r　　B. !Vis [i] && F [i] == r　　C. F [i] < F [x]　　D. !Vis [i] && F [i] < F [x]

答案：D

分析：此处需找到当前层级（r 个 4）中未处理的数字 x（用于扩展新状态）。

- F[i] == r 确保 i 是当前层级（需要 r 个 4）；
- !Vis[i]确保 i 未被处理（避免重复扩展）。
- A 选项缺少!Vis[i]，可能重复处理；
- C、D 选项与层级和未处理标记无关，错误；
- B 选项!Vis[i] && F[i] == r 正确。

(4) ④处应填（ ）

A. F [i] < F [x]　　B. F [i]<=r　　C. Vis [i]　　D. i <= x

答案：C

分析：此处需遍历所有已处理的数字 i（已确定最少 4 的个数且层级≤当前 r），与 x 组合运算扩展新状态。

- Vis[i]标记 i 已处理（确保其最少 4 的个数已确定）；
- A 选项 F[i] < F[x]错误（无需比较大小，只需已处理）；
- B 选项 F[i] <= r 错误（Vis[i]已隐含层级≤r）；
- D 选项 i <= x 错误（与大小无关）；
- C 选项 Vis[i]正确，符合已处理状态的要求。

1. (RMQ 区间最值问题)

20.（ RMQ 区间最值问题 ）给定序列，a_0，…，a_{n-1}，m 次询问每次询问给定 l,r,求 $\max\{a_l, \ldots, a_r\}$。为了解决该问题，有一个算法叫 the Method of Four Russians，其时间复杂度为 O(n+m)，步骤如下：
• 建立 Cartesian（笛卡尔）树，将问题转化为树上的 LCA（最近公共祖先）问题。
• 对于 LCA 问题，可以考虑其 Euler 序（即按照 DFS 过程，经过所有点，环游回根的序列），即求 Euler 序列上两点间一个新的 RMQ 问题。
• 注意新的问题为 ±1 RMQ，即相邻两点的深度差一定为 1。
下面解决这个 ±1 RMQ 问题，“序列”指 Euler 序列：
• 设 t 为 Euler 序列长度。取 $b=\lceil \log_2 t/2 \rceil$ 将序列每 b 个分为一大块，使用 ST 表（倍增表）处理大块间的 RMQ 问题，复杂度 O(t/b logt)=O(n)。
• （重点）对于一个块内的 RMQ 问题，也需要 O(1) 的算法。由于差分数组 2b−1 种，可以预处理出所有情况下的最值位置，预处理复杂度 O(b2 b)，不超过 O(n)。
• 最终，对于一个查询，可以转化为中间整的大块的 RMQ 问题，以及两端块内的 RMQ 问题。
试补全程序。

```
001 #include <iostream>
002 #include <cmath>
003
004 using namespace std;
005
006 const int MAXN = 100000, MAXT = MAXN << 1;
007 const int MAXL = 18, MAXB = 9, MAXC = MAXT / MAXB;
008
009 struct node {
010     int val;
011     int dep, dfn, end;
012     node *son[2]; // son[0], son[1] 分别表示左右儿子
013 } T[MAXN];
014
015 int n, t, b, c, Log2[MAXC + 1];
016 int Pos[(1 << (MAXB - 1)) + 5], Dif[MAXC + 1];
017 node *root, *A[MAXT], *Min[MAXL][MAXC];
018
019 void build() { // 建立 Cartesian 树
020     static node *S[MAXN + 1];
```

```
021    int top = 0;
022    for (int i = 0; i < n; i++) {
023        node *p = &T[i];
024        while (top && S[top]->val < p->val)
025           ①;
026        if (top)
027            ②;
028        S[++top] = p;
029    }
030    root = S[1];
031 }
032
033 void DFS(node *p) { // 构建 Euler 序列
034    A[p->dfn = t++] = p;
035    for (int i = 0; i < 2; i++)
036        if (p->son[i]) {
037            p->son[i]->dep = p->dep + 1;
038            DFS(p->son[i]);
039            A[t++] = p;
040        }
041    p->end = t - 1;
042 }
043
044 node *min(node *x, node *y) {
045     return ③ ? x : y;
046 }
047
048 void ST_init() {
049    b = (int)(ceil(log2(t) / 2));
050    c = t / b;
051    Log2[1] = 0;
052    for (int i = 2; i <= c; i++)
053        Log2[i] = Log2[i >> 1] + 1;
054    for (int i = 0; i < c; i++) {
055        Min[0][i] = A[i * b];
056        for (int j = 1; j < b; j++)
057            Min[0][i] = min(Min[0][i], A[i * b + j]);
058    }
059    for (int i = 1, l = 2; l <= c; i++, l <<= 1)
060        for (int j = 0; j + l <= c; j++)
061            Min[i][j] = min(Min[i - 1][j], Min[i - 1][j + (l >>
1)]);
062 }
063
064 void small_init() { // 块内预处理
065    for (int i = 0; i <= c; i++)
066        for (int j = 1; j < b && i * b + j < t; j++)
067           if (④)
068               Dif[i] |= 1 << (j - 1);
069    for (int S = 0; S < (1 << (b - 1)); S++) {
070        int mx = 0, v = 0;
```

```
071         for (int i = 1; i < b; i++) {
072             ⑤;
073            if (v < mx) {
074                 mx = v;
075                 Pos[S] = i;
076             }
077         }
078     }
079 }
080
081 node *ST_query(int l, int r) {
082     int g = Log2[r - l + 1];
083     return min(Min[g][l], Min[g][r - (1 << g) + 1]);
084 }
085
086 node *small_query(int l, int r) { // 块内查询
087      int p = l / b;
088     int S = ⑥;
089     return A[l + Pos[S]];
090 }
091
092 node *query(int l, int r) {
093      if (l > r)
094         return query(r, l);
095      int pl = l / b, pr = r / b;
096     if (pl == pr) {
097         return small_query(l, r);
098     } else {
099         node *s = min(small_query(l, pl * b + b - 1),small_query(pr * b, r));
100          if (pl + 1 <= pr - 1)
101            s = min(s, ST_query(pl + 1, pr - 1));
102         return s;
103     }
104 }
105
106 int main() {
107     int m;
108     cin >> n >> m;
109     for (int i = 0; i < n; i++)
110         cin >> T[i].val;
111     build();
112     DFS(root);
113     ST_init();
114     small_init();
115     while (m--) {
116         int l, r;
117         cin >> l >> r;
118         cout << query(T[l].dfn, T[r].dfn)->val << endl;
119     }
120     return 0;
121 }
```

(1)①处应填（ ）
A. p->son[0] = S[top--]　　B. p->son[1] = S[top--]
C. S[top--]->son[0] = p　　D. S[top--]->son[1] = p

(2)②处应填（ ）
A. p->son[0] = S[top]　　B. p->son[1] = S[top]
C. S[top]->son[0] = p　　D. S[top]->son[1] = p

(3)③处应填（ ）
A. x->dep < y->dep　　B. x < y
C. x->dep > y->dep　　D. x->val < y->val

(4)④处应填（ ）
A. A[i * b + j - 1] == A[i * b + j]->son[0]
B. A[i * b + j]->val < A[i * b + j - 1]->val
C. A[i * b + j] == A[i * b + j - 1]->son[1]
D. A[i * b + j]->dep < A[i * b + j - 1]->dep

(5)⑤处应填（ ）
A. v += (S >> i & 1) ? -1 : 1
B. v += (S >> i & 1) ? 1 : -1
C. v += (S >> (i - 1) & 1) ? 1 : -1
D. v += (S >> (i - 1) & 1) ? -1 : 1

(6)⑥处应填（ ）
A. (Dif[p] >> (r - p * b)) & ((1 << (r - 1)) - 1)
B. Dif[p]
C. (Dif[p] >> (1 - p * b)) & ((1 << (r - 1)) - 1)
D. (Dif[p] >> ((p + 1) * b - r)) & ((1 << (r - 1 + 1)) - 1)

该程序通过 **the Method of Four Russians** 算法解决区间最值查询（RMQ）问题，核心流程为：

1. **建立笛卡尔树**：将原序列转化为二叉树，其中每个节点的值大于左右子节点，原序列的区间最值对应树中两节点的最近公共祖先（LCA）。
2. **生成 Euler 序列**：通过 DFS 遍历笛卡尔树，记录节点的进出顺序，将 LCA 问题转化为 Euler 序列上的 RMQ 问题（±1 RMQ，相邻元素深度差为 1）。
3. **处理 ±1 RMQ**：
    1. 将 Euler 序列分块，用 ST 表处理跨块查询；
    2. 预处理块内所有可能的差分状态（因相邻深度差为 ±1），实现块内 O (1) 查询。

核心知识点

1. **笛卡尔树**：一种二叉树，满足父节点值 > 子节点值，左 / 右子树对应原序列的左 / 右子区间，可将区间最值查询转化为 LCA 问题。
2. **Euler 序列与 LCA**：DFS 遍历中，节点首次出现和离开子节点时的记录构成 Euler 序列，两节点的 LCA 是它们在 Euler 序列中出现位置之间深度最小的节点。
3. **±1 RMQ**：相邻元素深度差为 ±1 的特殊 RMQ 问题，可通过分块 + 预处理差分状态实现 O (1) 查询。
4. **ST 表**：基于倍增的区间最值预处理结构，支持 O (1) 查询，预处理时间 O (n log n)。

(1)①处应填（ ）

A. p->son[0] = S[top--]　　B. p->son[1] = S[top--]

C. S[top--]->son[0] = p　　D. S[top--]->son[1] = p

1) ①处应填（A）

分析：笛卡尔树构建使用单调栈维护递减序列（父节点值 > 子节点值）。当栈顶元素值 < 当前节点 p 的值时，栈顶元素需弹出并作为 p 的左儿子（因栈顶元素在原序列中位于 p 左侧，且值较小）。故①处为 `p->son[0] = S[top--]`，对应选项 A。

(2)②处应填（ ）

A. p->son[0] = S[top]　　B. p->son[1] = S[top]

C. S[top]->son[0] = p　　D. S[top]->son[1] = p

(2) ②处应填（D）

分析：弹出栈顶后，若栈非空，栈顶元素值 > 当前节点 p 的值（否则会被弹出），因此栈顶元素是 p 的父节点。由于 p 在原序列中位于栈顶元素右侧，故栈顶元素的右儿子应为 p。故②处为 `S[top]->son[1] = p`，对应选项 D。

(3)③处应填（ ）

A. x->dep < y->dep　　B. x < y

C. x->dep > y->dep　　D. x->val < y->val

(3) ③处应填（A）

分析：`min` 函数用于在 Euler 序列中找两节点的 LCA，而 LCA 是区间内深度最小的节点（深度越小，越靠近根）。因此需比较两节点深度，返回深度较小的节点。故③处为 `x->dep < y->dep`，对应选项 A。

(4)④处应填（ ）

A. A[i * b + j - 1] == A[i * b + j]->son[0]

B. A[i * b + j]->val < A[i * b + j - 1]->val

C. A[i * b + j] == A[i * b + j - 1]->son[1]

D. A[i * b + j]->dep < A[i * b + j - 1]->dep

(4) ④处应填（D）

分析：`Dif[i]`记录块内相邻元素的深度差分（±1 RMQ 特性）。若当前元素深度 `<` 前一元素深度（差分 `-1`），则用 bit 标记该状态。故④处条件为 `A[i*b +j]->dep < A[i*b +j-1]->dep`，对应选项 D。

(5)⑤处应填（ ）
A. v += (S >> i & 1) ? -1 : 1
B. v += (S >> i & 1) ? 1 : -1
C. v += (S >> (i - 1) & 1) ? 1 : -1
D. v += (S >> (i - 1) & 1) ? -1 : 1

(5) ⑤处应填（D）

分析：S 是块内差分状态的 bitmask（b-1 位），第 i-1 位表示第 i 个相邻对的差分：1 表示差分 -1，0 表示差分 +1。因此 v（深度偏移）需根据 S 的第 i-1 位更新：1 则 -1，0 则 +1。故⑤处为 `v += (S >> (i-1) & 1) ? -1 : 1`，对应选项 D。

(6)⑥处应填（ ）
A. (Dif[p] >> (r - p * b)) & ((1 << (r - l)) - 1)
B. Dif[p]
C. (Dif[p] >> (l - p * b)) & ((1 << (r - l)) - 1)
D. (Dif[p] >> ((p + 1) * b - r)) & ((1 << (r - l + 1)) - 1)

(6) ⑥处应填（C）

分析：块内查询需提取`[l, r]`区间的差分状态 S。l 在块 p 中的偏移为 `l - p*b`，区间长度为 `r - l`，故需从 `Dif[p]`中右移 `l - p*b` 位，再取 `r - l` 位。故⑥处为`(Dif[p] >> (l - p*b)) & ((1 << (r - l)) - 1)`，对应选项 C。

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

const int MAXN = 100000, MAXT = MAXN << 1;   // MAXT 为 Euler 序列最大长度（2*MAXN）
const int MAXL = 18, MAXB = 9, MAXC = MAXT / MAXB;   // MAXL 为 ST 表层级，MAXB 为块大小，MAXC 为块数量

struct node {
    int val;   // 节点值（用于笛卡尔树的最值判断）
    int dep, dfn, end;   // dep：深度；dfn：Euler 序列中首次出现位置；end：Euler 序列中最后出现位置
    node *son[2];   // son[0]左儿子，son[1]右儿子
} T[MAXN];

int n, t, b, c, Log2[MAXC + 1];   // t：Euler 序列长度；b：块大小；c：块数量；Log2：预处理对数
int Pos[(1 << (MAXB - 1)) + 5], Dif[MAXC + 1];   // Pos：块内差分状态的最值位置；Dif：块内差分信息（bitmask）
node *root, *A[MAXT], *Min[MAXL][MAXC];   // A：Euler 序列；Min：大块 ST 表（存储块内最值）

// 建立笛卡尔树（父节点值 > 子节点值，左/右子树对应原序列左/右区间）
void build() {
    static node *S[MAXN + 1];   // 单调栈，维护当前未确定父节点的节点
    int top = 0;   // 栈顶指针
    for (int i = 0; i < n; i++) {
```

```
        node *p = &T[i];   // 当前节点
        // 弹出栈顶所有值小于当前节点的元素（这些元素将作为当前节点的左子树）
        while (top && S[top]->val < p->val)
            p->son[0] = S[top--];   // ①：弹出的元素作为当前节点的左儿子
        // 若栈非空，栈顶元素值 > 当前节点，作为当前节点的父节点，当前节点为其右儿子
        if (top)
            S[top]->son[1] = p;   // ②：栈顶元素的右儿子设为当前节点
        S[++top] = p;   // 当前节点入栈
    }
    root = S[1];   // 栈底元素为根节点（整个序列的最大值）
}

// 构建 Euler 序列（DFS 遍历，进入节点记录一次，离开子节点返回时再记录一次）
void DFS(node *p) {
    A[p->dfn = t++] = p;   // 首次访问，记录位置 dfn
    for (int i = 0; i < 2; i++)   // 遍历左右儿子
        if (p->son[i]) {
            p->son[i]->dep = p->dep + 1;   // 子节点深度 = 父节点深度 + 1
            DFS(p->son[i]);   // 递归访问子节点
            A[t++] = p;   // 离开子节点返回时，再次记录当前节点
        }
    p->end = t - 1;   // 记录最后出现位置
}

// 比较两个节点的深度，返回深度较小的节点（用于找 LCA，Euler 序列中 LCA 是区间内深度最小的节点）
node *min(node *x, node *y) {
    return x->dep < y->dep ? x : y;   // ③：深度小的节点为更优（LCA）
}

// 初始化大块 ST 表（处理跨块的 RMQ）
void ST_init() {
    b = (int)(ceil(log2(t) / 2));   // 块大小 b = ?log2(t)/2?
    c = t / b;   // 块数量
    Log2[1] = 0;   // 预处理 log2（用于 ST 表查询）
    for (int i = 2; i <= c; i++)
        Log2[i] = Log2[i >> 1] + 1;
    // 初始化 ST 表第 0 层（每个块的最值）
    for (int i = 0; i < c; i++) {
        Min[0][i] = A[i * b];   // 块内第一个元素
        for (int j = 1; j < b; j++)
            Min[0][i] = min(Min[0][i], A[i * b + j]);   // 比较块内所有元素，取最值
    }
    // 向上构建 ST 表（倍增）
    for (int i = 1, l = 2; l <= c; i++, l <<= 1)
        for (int j = 0; j + l <= c; j++)
            Min[i][j] = min(Min[i - 1][j], Min[i - 1][j + (l >> 1)]);   // 合并两个子块的最值
}

// 预处理块内信息（±1 RMQ 的差分状态）
void small_init() {
    // 计算每个块的差分信息 Dif[i]（bitmask）
```

```
    for (int i = 0; i <= c; i++)
        for (int j = 1; j < b && i * b + j < t; j++)
            // ④：若当前元素深度 < 前一个元素深度（差分-1），设置对应 bit
            if (A[i * b + j]->dep < A[i * b + j - 1]->dep)
                Dif[i] |= 1 << (j - 1);   // 用 bit j-1 标记差分-1
    // 预处理所有可能的差分状态 S（共 2^(b-1)种）的最值位置
    for (int S = 0; S < (1 << (b - 1)); S++) {
        int mx = 0, v = 0;   // v：当前深度相对于块起始的偏移；mx：最小偏移（对应深度最小位置）
        for (int i = 1; i < b; i++) {
            // ⑤：根据 S 的 bit (i-1)更新偏移 v（1→-1，0→+1）
            v += (S >> (i - 1) & 1) ? -1 : 1;
            if (v < mx) {   // 记录最小偏移的位置（深度最小）
                mx = v;
                Pos[S] = i;
            }
        }
    }
}

// 大块 ST 表查询（[l, r]块区间的最值）
node *ST_query(int l, int r) {
    int g = Log2[r - l + 1];   // 取 log2 长度，确定 ST 表层级
    return min(Min[g][l], Min[g][r - (1 << g) + 1]);   // 合并两个子区间的最值
}

// 块内查询（[l, r]在同一个块内）
node *small_query(int l, int r) {
    int p = l / b;   // 块编号
    // ⑥：提取块 p 内[l, r]区间的差分状态 S（从 l 的块内偏移开始，长度为 r-l）
    int S = (Dif[p] >> (l - p * b)) & ((1 << (r - l)) - 1);
    return A[l + Pos[S]];   // 根据预处理的 Pos[S]获取最值位置
}

// 完整查询：处理任意区间[l, r]的 RMQ（转化为 Euler 序列上的 LCA 查询）
node *query(int l, int r) {
    if (l > r)
        return query(r, l);   // 确保 l <= r
    int pl = l / b, pr = r / b;   // 左右端点所在块
    if (pl == pr) {   // 同一块内，用块内查询
        return small_query(l, r);
    } else {   // 跨块：合并左块剩余部分、中间整块、右块起始部分的最值
        node *s = min(small_query(l, pl * b + b - 1), small_query(pr * b, r));
        if (pl + 1 <= pr - 1)   // 若有中间整块，用 ST 表查询
            s = min(s, ST_query(pl + 1, pr - 1));
        return s;
    }
}

int main() {
    int m;
    cin >> n >> m;
    for (int i = 0; i < n; i++)
        cin >> T[i].val;   // 输入序列值
```

```
    build();   // 建立笛卡尔树
    root->dep = 0;   // 根节点深度为 0（补充初始化）
    DFS(root);   // 生成 Euler 序列
    ST_init();   // 初始化大块 ST 表
    small_init();   // 初始化块内信息
    while (m--) {   // 处理 m 次查询
        int l, r;
        cin >> l >> r;   // 原序列的区间[l, r]（0-based）
        // 转化为 Euler 序列上的[T[l].dfn, T[r].dfn]，查询其 LCA（即区间最值）
        cout << query(T[l].dfn, T[r].dfn)->val << endl;
    }
    return 0;
}
```