一、函数与递归

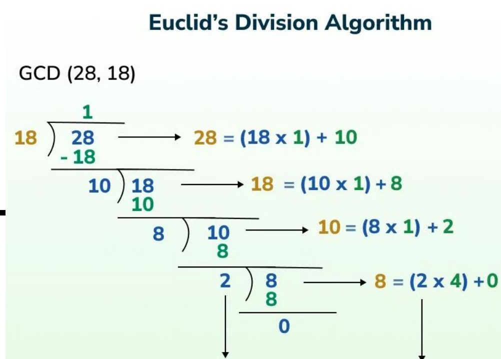
（一）考点分析

结构类型	重点考点	考察频率	难度
	欧几里得算法实现	35%	★★
数学递归	组合数递归计算	25%	★★★
	数列递归关系分析	20%	★★★★
分治递归	矩阵分形填充	15%	★★★★
	归并/快排递归框架	30%	★★★
	子问题划分策略	25%	★★★★
树形递归	最优决策树构建	40%	★★★★
	二叉树遍历应用	30%	★★
	深度优先搜索框架	35%	★★★
递归优化	记忆化搜索实现	20%	★★★★
	尾递归优化策略	15%	★★★★★
	递归转迭代方法	10%	★★★★
特殊应用	汉诺塔问题分析	25%	★★★
	最大路径和计算	20%	★★★★
	回溯算法实现	30%	★★★★
调试分析	递归树展开验证	35%	★★★
	栈溢出风险分析	25%	★★★★
	边界条件处理	30%	★★★

（二）核心知识点

1、数学递归

（1）欧几里得算法实现（35%，★★）



核心知识点:

- 基于 gcd(a,b) = gcd(b,a%b)的数学原理
- 递归终止条件：b == 0 时返回 a

黄金法则:

- 必须保证 a >= b，否则自动交换（gcd(b,a%b)已隐含处理）

C++公式:

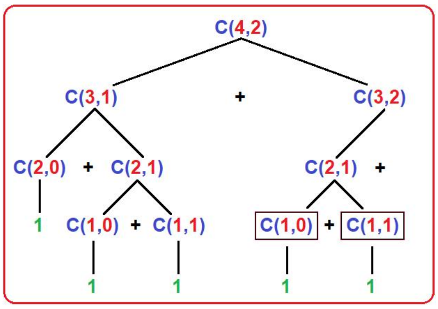
```
int gcd(int a, int b) {
    return b ? gcd(b, a % b) : a;
}
```

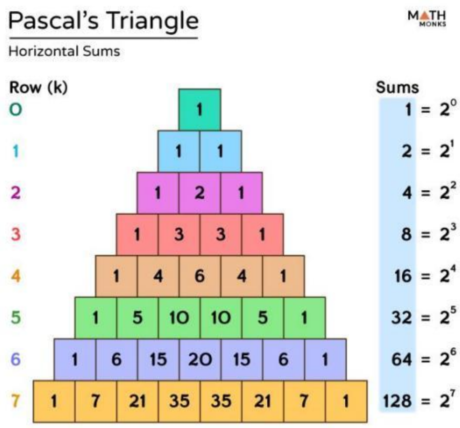
易错点:

- 未处理负数输入（应取绝对值）
- 混淆 a%b 和 b%a 的顺序

（2）组合数递归计算（25%，★★★）

- 核心知识点：
- 帕斯卡恒等式：C(n,k) = C(n-1,k-1) + C(n-1,k)
- 边界条件：k==0 或 k==n 时返回 1





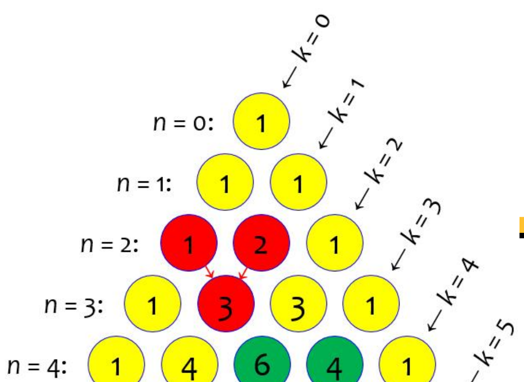
C++模板:

```
int comb(int n, int k) {
    if (k == 0 || k == n) return 1;
    return comb(n-1, k-1) + comb(n-1, k);
}
```

高频优化:

- 记忆化存储（使用二维数组缓存结果）

易错点:



- 未处理 k>n 的非法情况
- 重复计算导致指数级时间复杂度

（3）数列递归关系分析（20%，★★★★）

典型问题：

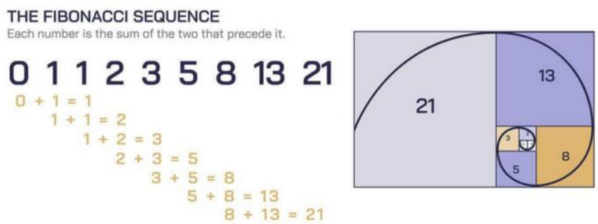
- 斐波那契数列 f(n)=f(n-1)+f(n-2)
- 变种如 f(n)=f(n-2)-f(n-1)（2014 真题）

黄金法则：

- 先写出数学递推式，再转换为递归终止条件

C++示例：

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
```



易错点：

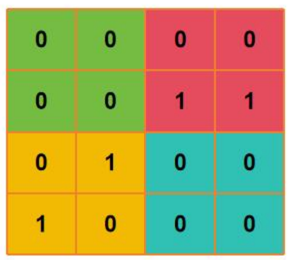
- 忽略负数项处理（如 2014 题中的交替减法）
- 未发现可转化为数学公式（如斐波那契通项公式）

2、分治递归

（1）矩阵分形填充（15%，★★★★）

核心模式：

```
void fill(int x, int y, int size, int type) {
    if (size == 1) { matrix[x][y] = type; return; }
    int half = size/2;
    fill(x, y, half, type);          // 左上
    fill(x, y+half, half, type);     // 右上
    fill(x+half, y, half, type);     // 左下
```



```
        fill(x+half, y+half, half, !type);   // 右下特殊处理
    }
```

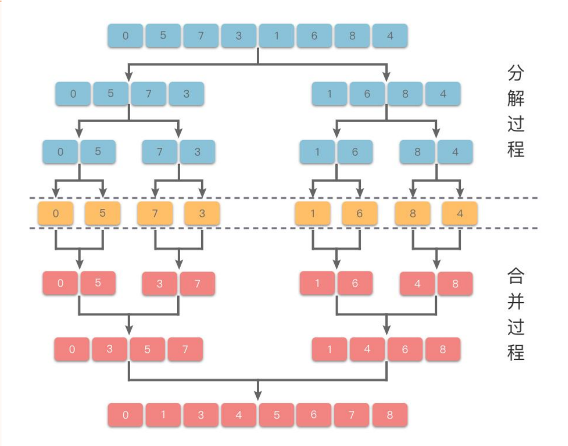
关键技巧：

- 使用 **1<<(n-1)**计算分块大小
- 右下角通常需要取反（如 2019 真题）

（2）归并排序框架（30%，★★★）（了解）

黄金模板：

```
    void merge_sort(int l, int r) {
        if (l >= r) return;
        int mid = (l + r) >> 1;
        merge_sort(l, mid);
        merge_sort(mid+1, r);
        // 合并两个有序数组
        int i = l, j = mid + 1, k = 0;
        while (i <= mid && j <= r)
            tmp[k++] = a[i] <= a[j] ? a[i++] : a[j++];
        while (i <= mid) tmp[k++] = a[i++];
        while (j <= r) tmp[k++] = a[j++];
        for (i = l, j = 0; i <= r; ) a[i++] = tmp[j++];
    }
```



易错点：

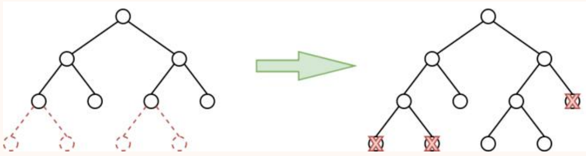
- 区间划分错误（必须[l,mid]和[mid+1,r]）
- 临时数组未清空导致污染

3、树形递归

（1）最优决策树构建（40%，★★★★）（了解）

真题原型（2019 年第 18 题）：

```
int dfs(int l, int r, int depth) {
    if (l > r) return 0;
    int min_pos = find_min(l, r);   // 找到最小值位置
    int left = dfs(l, min_pos-1, depth+1);
    int right = dfs(min_pos+1, r, depth+1);
    return left + right + depth * b[min_pos];
}
```

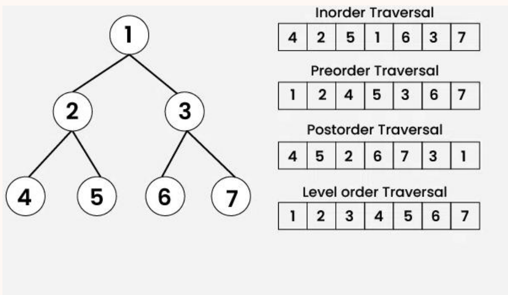


黄金法则：

- 每次递归划分后深度+1
- 结果累加必须放在递归调用之后

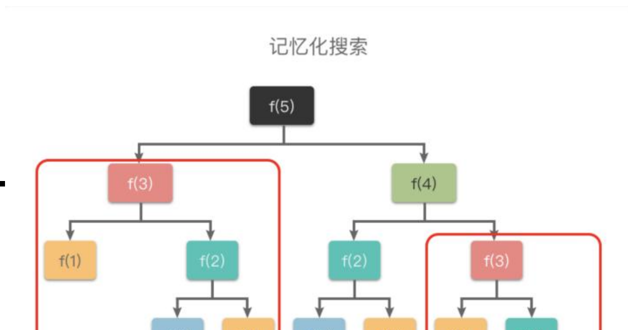
（2）二叉树遍历模板（30%，★★）

```
void traverse(TreeNode* root) {
    if (!root) return;
    // 前序：在此处处理 root->val
    traverse(root->left);
    // 中序：在此处处理 root->val
    traverse(root->right);
    // 后序：在此处处理 root->val
}
```



高频考点：

- 前序求深度，后序求高度
- 中序用于 BST 验证



4、递归优化

（1）记忆化搜索（20%，★★★★）

斐波那契优化示例：

```
int memo[MAXN];
int fib(int n) {
    if (n <= 1) return n;
    if (memo[n]) return memo[n];
    return memo[n] = fib(n-1) + fib(n-2);
}
```

关键点：

- 数组初始化为 0
- 先查表再递归

（2）尾递归优化（15%，★★★★★）（了解）

阶乘示例：

```
int fact(int n, int acc = 1) {
    if (n == 0) return acc;
    return fact(n-1, acc * n);   // 尾调用形式
}
```

编译器优化原理：

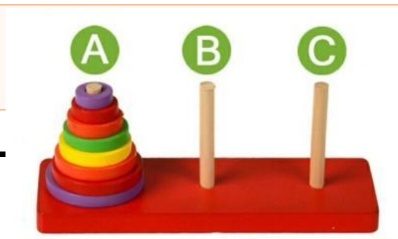
- 复用当前栈帧，避免堆栈增长

5、特殊应用

（1）汉诺塔问题（25%，★★★）

黄金代码：

```
void hanoi(int n, char A, char C, char B) {
```



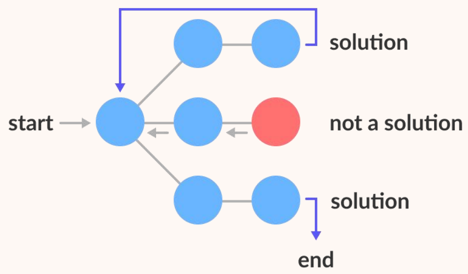
```
        if (n == 0) return;
        hanoi(n-1, A, B, C);
        cout << "Move " << n << " from " << A << " to " << C << endl;
        hanoi(n-1, B, C, A);
    }
```

数学本质：

- 移动次数公式：T(n) = 2T(n-1) + 1 => O(2^n)

（2）回溯算法框架（30%，★★★★）

```
void backtrack(int step) {
    if (is_goal(step)) { store_result(); return; }
    for (auto choice : all_choices) {
        if (is_valid(choice)) {
            make_choice(choice);
            backtrack(step + 1);
            undo_choice(choice);   // 关键回溯步骤
        }
    }
}
```



易错点：

- 忘记撤销选择（导致状态污染）
- 剪枝条件设置不当

6、调试分析

（1）递归树展开（35%，★★★）

调试技巧：

```
void dfs(int n, int depth = 0) {
    cout << string(depth, ' ') << "n=" << n << endl;   // 缩进打印
    if (n <= 1) return;
    dfs(n-1, depth+1);
    dfs(n-2, depth+1);
}
```

输出示例：

text

```
n=4
  n=3
    n=2
      n=1
      n=0
    n=1
  n=2
    ...
```

（2）栈溢出风险（25%，★★★★）（了解）

危险信号：

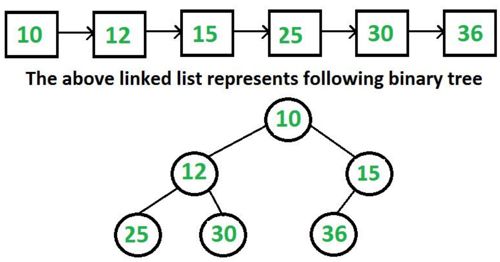
- 递归深度超过 1e4（默认栈大小约 8MB）

解决方案：

```
#pragma comment(linker, "/STACK:102400000,102400000")   // 扩展栈空间
```

黄金法则：

- 任何递归必须确保有终止条件
- 树形递归深度不超过 **O(log n)**为安全



（3）边界条件处理（30%，★★★）

典型错误案例：

```
int binary_search(int l, int r) {
    if (l > r) return -1;   // 必须首先检查
    int mid = (l + r) / 2;
    // ...
}
```

关键检查点：

- 数组空(l > r)
- 单元素(l == r)
- 整数溢出(l + r)改用 l + (r - l) / 2

（三）真题强化

1、2007 年第 12 题（递归）

近 20 年来，许多计算机专家都大力推崇递归算法，认为它是解决较复杂问题的强有力的工具。在下列关于递归算法的说法中，正确的是（　）。

A. 在 1977 年前后形成标准的计算机高级语言 FORTRAN77 禁止在程序使用递归，原因之一是该方法可能会占用更多的内存空间

B. 和非递归算法相比，解决同一个问题，递归算法一般运行得更快一些

C. 对于较复杂的问题，用递归方式编程一般比非递归方式更难一些

D. 对于已经定义好的标准数学函数 sin(x)，应用程序中的语句“y=sin(sin(x));”就是一种递归调用

答案：A

解析:

A 正确：FORTRAN77 因递归调用需栈空间存储返回地址/局部变量，可能占用更多内存，故禁止递归。

B 错误：递归因函数调用开销（参数传递、栈帧管理）通常比迭代慢。

C 错误：递归简化复杂问题逻辑（如汉诺塔、树遍历），反而更易实现。

D 错误：sin(sin(x)) 是嵌套调用而非递归（未调用自身）。

2、2008 年第 11 题（递归）

递归过程或函数调用时，处理参数和返回地址，通常使用一种称为（　）的数据结构。

A. 队列

B. 多维数组

C. 线性表

D. 栈

答案：D

解析：递归过程或函数调用时，处理参数和返回地址通常使用栈数据结构来保存和恢复现场。

3、2008 年第 24 题（递归）

阅读程序写结果:

```
#include<iostream>
using namespace std;
void foo(int a, int b, int c)
{
 if(a > b)
     foo(c, a, b);
 else
     cout << a << ',' << b << ',' << c << endl;
}
int main()
{
 int a, b, c;
 cin >> a >> b >> c;
 foo(a, b, c);
 return 0;
}
```

输入: 3 1 2

正确答案：2,3,1

解析：根据输入：

foo(3, 1, 2) 调用 foo(2, 3, 1)。

foo(2, 3, 1) 满足条件 a <= b，输出 2,3,1。

4、2013 年第 15 题（递归，欧几里得算法）

下面是根据欧几里得算法编写的函数，它所计算的是 a 和 b 的（ ）。

```
int euclid(int a, int b){
     if (b == 0)
```

```
            return a;
        else
            return euclid(b, a % b);
    }
```

A. 最大公共质因子

B. 最小公共质因子

C. 最大公约数

D. 最小公倍数

正确答案： C

解析：欧几里得算法用于计算两个数的最大公约数（GCD）。

5、2020 年第 6 题（递归）

设 A 是 n 个实数的数组，考虑下面的递归算法：

```
XYZ (A[1..n])
1.  if n=1 then return A[1]
2.  else temp ← XYZ (A[1..n-1])
3.  if temp < A[n]
4.  then return temp
5.  else return A[n]
```

请问算法 XYZ 的输出是什么？（ ）

A. A 数组的平均

B. A 数组的最小值

C. A 数组的中值

D. A 数组的最大值

正确答案： B

解析： 递归算法 XYZ 找出数组 A 的最小值。

6、2021 年第 13 题（递归）

考虑如下递归算法

```
solve(n)
if n<=1 return 1
else if n>=5 return n*solve(n-2)
else return n*solve(n-1)
```

则调用 solve(7) 得到的返回结果为（ ）。

A. 105

B. 840

C. 210

D. 420

正确答案： C

解析：

solve(7)：7≥5 → 返回 7 × solve(5)

solve(5)：5≥5 → 返回 5 × solve(3)

solve(3)：3<5 且 3>1 → 返回 3 × solve(2)

solve(2)：2<5 且 2>1 → 返回 2 × solve(1)

solve(1)：1≤1 → 返回 1

回溯计算：

solve(2) = 2 × 1 = 2

solve(3) = 3 × 2 = 6

solve(5) = 5 × 6 = 30

solve(7) = 7 × 30 = 210

结论：最终结果为 210（选项 C）。递归路径：7→5→3→2→1，计算为 7×5×

3×2×1=210。

7、2022 年第 15 题（递归）

以下对递归方法的描述中，正确的是：（ ）。

A. 递归是允许使用多组参数调用函数的编程技术

B. 递归是通过调用自身来求解问题的编程技术

C. 递归是面向对象和数据而不是功能和逻辑的编程语言模型

D. 递归是将用某种高级语言转换为机器代码的编程技术

正确答案： B

解析： B 正确：递归的核心是函数直接/间接调用自身解决问题（如阶乘、斐波那契数列）。

A 错误：多组参数调用是函数重载或参数化，与递归无关。

C 错误：描述的是面向对象编程（OOP），非递归。

D 错误：描述的是编译过程，非递归。

8. 2009 年第 23 题（递归-最大公约数-欧几里得算法）

阅读程序写结果：

```
#include <iostream>
using namespace std;
int a,b;
int work(int a,int b){
    if (a%b)
        return work(b,a%b);
    return b;
}
int main(){
    cin >> a >> b;
    cout << work(a,b) << endl;
```

```
        return 0;
    }
```

输入：20 12

答案：

4

解析：

- 程序实现了欧几里得算法（辗转相除法），用于计算两个整数的最大公约数（GCD）。算法核心：若 a % b 不为 0，则递归调用 work(b, a % b)；否则返回 b（此时 b 即为最大公约数）。
- 输入为 a = 20, b = 12 时：
  - 第一层：work(20, 12) → 20 % 12 = 8 ≠ 0，递归 work(12, 8)。
  - 第二层：work(12, 8) → 12 % 8 = 4 ≠ 0，递归 work(8, 4)。
  - 第三层：work(8, 4) → 8 % 4 = 0，返回 4。
- 最终输出结果为 20 和 12 的最大公约数 4。

9. 2010 年第 26 题（递归）

阅读程序写结果：

```
#include <iostream>
using namespace std;
const int NUM = 5;
int r(int n)
{
    int i;
    if (n <= NUM)
        return n;
    for (i = 1; i <= NUM; i++)
```

```
            if (r(n - i) < 0)
                return i;
        return -1;
    }
    int main()
    {
        int n;
        cin>>n;
        cout<<r(n)<<endl;
        return 0;
    }
```

（1）输入：7，输出：

（2）输入：16，输出：

答案：

（1）1

（2）4

解析：

- 递归函数 r(n) 逻辑：
  - 若 n ≤ 5（NUM = 5），直接返回 n。
  - 若 n > 5，循环 i = 1 到 5，检查 r(n - i) < 0。若成立，返回 i；否则返回 -1。
- （1）输入 7：
  - r(7): 7 > 5，进入循环。
    - i = 1：计算 r(7 - 1) = r(6)。
      - r(6): 6 > 5，进入循环。
        - i = 1：计算 r(5) → 5 ≤ 5，返回 5（5 > 0，不满足 < 0）。

  - i = 2：计算 r(4) → 4 ≤ 5，返回 4（4 > 0）。
  - ...
  - i = 5：计算 r(1) → 1 ≤ 5，返回 1（1 > 0）
  - 所有 i 的 r(n - i) 均大于 0，循环结束返回 -1。
- 此时 r(6) = -1 < 0 成立，r(7) 返回 i = 1。
- 输出 1。

- （2）输入 16：
- 分析规律：
  - r(1) ——r(5) 分别为 1、2、3、4、5，r(6)=-1
  - r(7) ——r(11) 分别为 1、2、3、4、5，r(12)=-1
  - r(13) ——r(17) 分别为 1、2、3、4、5，r(18)=-1

10. 2011 年第 26 题（递归）

阅读程序写结果：

```
#include<iostream>
using namespace std;
int solve(int n,int m)
{
    int i,sum;
    if(m==1) return 1;
    sum=0;
    for(i=1;i<n;i++)
        sum+= solve(i,m-1);
    return sum;
}
int main()
{
```

```
        int n,m;
        cin>>n>>m;
        cout<<solve(n,m)<<endl;
        return 0;
}
```

输入：7 4，输出：

答案：

20

解析：

- 函数 solve(n, m) 递归计算组合数，具体为从 n-1 个元素中选择 m-1 个的组合数（即 C(n-1, m-1)）。
- 递归逻辑：
  - 当 m == 1 时，返回 1（基准情况）。
  - 否则，初始化 sum = 0，循环 i = 1 到 n-1，累加 solve(i, m-1)。
- 数学本质：solve(n, m) = C(n-1, m-1)，其中 C 是组合数。
  - 输入 n = 7, m = 4：
    - 计算 solve(7, 4) = C(6, 3)。
    - 组合数公式：C(6, 3) = 6! / (3! * 3!) = 20。
- 递归计算过程详解（solve(7,4)）
  - 第一层：solve(7,4),m=4 ≠ 1 → 进入循环,初始化 sum = 0
    - 循环 i 从 1 到 6：
      - i=1：sum += solve(1,3)
      - i=2：sum += solve(2,3)
      - i=3：sum += solve(3,3)
      - i=4：sum += solve(4,3)

- i=5：sum += solve(5,3)
- i=6：sum += solve(6,3)
- 最终返回 sum = 0 + 0 + 1 + 3 + 6 + 10 = 20

■ 第二层：solve(1,3),m=3 ≠ 1 → 进入循环

◆ 循环 i 从 1 到 0（i<1 条件不满足）→ 循环不执行,返回 sum = 0

■ 第二层：solve(2,3),m=3 ≠ 1 → 进入循环

◆ 循环 i 从 1 到 1：

◆ i=1：sum += solve(1,2),返回 sum = solve(1,2)

■ 第三层：solve(1,2),m=2 ≠ 1 → 进入循环

◆ 循环 i 从 1 到 0（i<1 条件不满足）→ 循环不执行,返回 sum = 0,∴ solve(2,3) = 0

■ 第二层：solve(3,3),m=3 ≠ 1 → 进入循环

◆ 循环 i 从 1 到 2：

- i=1：sum += solve(1,2) = 0
- i=2：sum += solve(2,2)
- 返回 sum = 0 + solve(2,2)

■ 第三层：solve(2,2),m=2 ≠ 1 → 进入循环

◆ 循环 i 从 1 到 1：

- i=1：sum += solve(1,1)
- 返回 sum = solve(1,1)

■ 第四层：solve(1,1),m=1 → 直接返回 1

◆ ∴ solve(2,2) = 1

◆ ∴ solve(3,3) = 0 + 1 = 1

■ 第二层：solve(4,3), m=3 ≠ 1 → 进入循环

◆ 循环 i 从 1 到 3：

- i=1：sum += solve(1,2) = 0

        - i=2：sum += solve(2,2) = 1
        - i=3：sum += solve(3,2)
    - 返回 sum = 0 + 1 + solve(3,2)
- 第三层：solve(3,2),m=2 ≠ 1 → 进入循环
    - 循环 i 从 1 到 2:
        - i=1：sum += solve(1,1) = 1
        - i=2：sum += solve(2,1)
    - 返回 sum = 1 + solve(2,1)
- 第四层：solve(2,1),m=1 → 直接返回 1
        - ∴ solve(3,2) = 1 + 1 = 2
        - ∴ solve(4,3) = 0 + 1 + 2 = 3
- 第二层：solve(5,3),m=3 ≠ 1 → 进入循环
    - 循环 i 从 1 到 4:
        - i=1：sum += solve(1,2) = 0
        - i=2：sum += solve(2,2) = 1
        - i=3：sum += solve(3,2) = 2
        - i=4：sum += solve(4,2)
    - 返回 sum = 0 + 1 + 2 + solve(4,2)
- 第三层：solve(4,2),m=2 ≠ 1 → 进入循环
    - 循环 i 从 1 到 3:
        - i=1：sum += solve(1,1) = 1
        - i=2：sum += solve(2,1) = 1
        - i=3：sum += solve(3,1) = 1
    - 返回 sum = 1 + 1 + 1 = 3
    - ∴ solve(5,3) = 0 + 1 + 2 + 3 = 6
- 第二层：solve(6,3),m=3 ≠ 1 → 进入循环

- ◆ 循环 i 从 1 到 5:
  - i=1：sum += solve(1,2) = 0
  - i=2：sum += solve(2,2) = 1
  - i=3：sum += solve(3,2) = 2
  - i=4：sum += solve(4,2) = 3
  - i=5：sum += solve(5,2)
- ◆ 返回 sum = 0 + 1 + 2 + 3 + solve(5,2)

- ■ 第三层：solve(5,2),m=2 ≠ 1 → 进入循环
  - ◆ 循环 i 从 1 到 4:
    - i=1：sum += solve(1,1) = 1
    - i=2：sum += solve(2,1) = 1
    - i=3：sum += solve(3,1) = 1
    - i=4：sum += solve(4,1) = 1
  - ◆ 返回 sum = 1 + 1 + 1 + 1 = 4
  - ◆ ∴ solve(6,3) = 0 + 1 + 2 + 3 + 4 = 10
- ■ 最终结果
- ■ solve(7,4) = solve(1,3) + solve(2,3) + solve(3,3) + solve(4,3) + solve(5,3) + solve(6,3)
- ■ = 0 + 0 + 1 + 3 + 6 + 10
- ■ = 20
- ■ 该结果等于组合数 C(6,3) = 6!/(3!×3!) = 20，验证了程序的数学本质是计算组合数 C(n-1, m-1)。

11. 2014 年第 24 题（递归）

阅读程序写结果：

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

```
int fun(int n)
{
    if(n == 1)
        return 1;
    if(n == 2)
        return 2;
    return fun(n - 2) - fun(n - 1);
}
int main()
{
    int n;
    cin >> n;
    cout << fun(n) << endl;
    return 0;
}
```

输入：7，输出：

答案：-11

解析：

- 递归函数 fun(n) 定义：
  - n == 1 时返回 1。
  - n == 2 时返回 2。
  - 否则返回 fun(n - 2) - fun(n - 1)。
- 输入 n = 7，需计算 fun(7)：
  - fun(7) = fun(5) - fun(6)
  - 先计算 fun(5) 和 fun(6)：

- fun(5) = fun(3) - fun(4)
- fun(6) = fun(4) - fun(5)

- 再计算底层：
  - fun(3) = fun(1) - fun(2) = 1 - 2 = -1
  - fun(4) = fun(2) - fun(3) = 2 - (-1) = 3
- 回代：
  - fun(5) = fun(3) - fun(4) = -1 - 3 = -4
  - fun(6) = fun(4) - fun(5) = 3 - (-4) = 7
- 最终：fun(7) = fun(5) - fun(6) = -4 - 7 = -11。

- 递归树如下：
  - fun(7)
  - = fun(5) - fun(6)
  - = [fun(3) - fun(4)] - [fun(4) - fun(5)]
  - = [(-1) - (3)] - [(3) - (-4)]
  - = (-4) - (7)
  - = -11
- 输出 -11。

12. 2012 年第 25 题（递归）

阅读程序写结果：

```
#include <iostream>
using namespace std;
int n, i, j, a[100][100];
int solve(int x, int y)
{
    int u, v;
    if (x == n) return a[x][y];
```

```
        u = solve(x + 1, y);
        v = solve(x + 1, y + 1);
        if (u > v) return a[x][y] + u;
        else return a[x][y] + v;
}
int main()
{
        cin >> n;
        for (i = 1; i <= n; i++)
                for (j = 1; j <= i; j++) cin >> a[i][j];
        cout << solve(1, 1) << endl;
        return 0;
}
```


(1,1) 2				
(2,1) -1	(2,2) 4			
(3,1) 2	(3,2) -1	(3,3) -2		
(4,1) -1	(4,2) 6	(4,3) 4	(4,4) 0	
(5,1) 3	(5,2) 2	(5,3) -1	(5,4) 5	(5,5) 8

(1,1) 14				
(2,1) 9	(2,2) 12			
(3,1) 10	(3,2) 8	(3,3) 7		
(4,1) 2	(4,2) 8	(4,3) 9	(4,4) 8	
(5,1) 3	(5,2) 2	(5,3) -1	(5,4) 5	(5,5) 8

输入：

5

2

-1 4

2 -1 -2

-1 6 4 0

3 2 -1 5 8

(1,1) 2				
(2,1) -1	(2,2) 4			
(3,1) 2	(3,2) -1	(3,3) -2		
(4,1) -1	(4,2) 6	(4,3) 4	(4,4) 0	
(5,1) 3	(5,2) 2	(5,3) -1	(5,4) 5	(5,5) 8

(1,1) 14				
(2,1) 9	(2,2) 12			
(3,1) 10	(3,2) 8	(3,3) 7		
(4,1) 2	(4,2) 8	(4,3) 9	(4,4) 8	
(5,1) 3	(5,2) 2	(5,3) -1	(5,4) 5	(5,5) 8

输出：

答案：

14

解析：

- 程序计算一个数字三角形从顶点（1,1）到底边（n, y）的最大路径和。递归函数

solve(x, y) 返回从 (x, y) 到底边的最大路径和。

- 逻辑：
  - 若 x == n（到达底边），直接返回 a[x][y]。
  - 否则，递归计算下方 (x+1, y) 和右下方 (x+1, y+1) 的最大路径和 u 和 v，返回 a[x][y] + max(u, v)。
- 输入数据构建的三角形：
  - 行 1: 2
  - 行 2: -1 4
  - 行 3: 2 -1 -2
  - 行 4: -1 6 4 0
  - 行 5: 3 2 -1 5 8
- 计算最大路径和：
  - 从 (1,1) 开始：solve(1,1) = a[1][1] + max(solve(2,1), solve(2,2))。
  - 递归展开关键路径：
    - solve(5,1) = 3, solve(5,2) = 2, solve(5,3) = -1, solve(5,4) = 5, solve(5,5) = 8（底边）。
    - solve(4,1) = a[4][1] + max(solve(5,1), solve(5,2)) = -1 + max(3, 2) = -1 + 3 = 2
    - solve(4,2) = a[4][2] + max(solve(5,2), solve(5,3)) = 6 + max(2, -1) = 6 + 2 = 8
    - solve(4,3) = a[4][3] + max(solve(5,3), solve(5,4)) = 4 + max(-1, 5) = 4 + 5 = 9
    - solve(4,4) = a[4][4] + max(solve(5,4), solve(5,5)) = 0 + max(5, 8) = 8
    - 继续向上：
      - solve(3,1) = a[3][1] + max(solve(4,1), solve(4,2)) = 2 + max(2, 8) = 2 + 8 = 10
      - solve(3,2) = a[3][2] + max(solve(4,2), solve(4,3)) = -1 + max(8, 9) = -1 + 9 = 8

- solve(3,3) = a[3][3] + max(solve(4,3), solve(4,4)) = -2 + max(9, 8) = -2 + 9 = 7

◆ 再向上:

- solve(2,1) = a[2][1] + max(solve(3,1), solve(3,2)) = -1 + max(10, 8) = -1 + 10 = 9
- solve(2,2) = a[2][2] + max(solve(3,2), solve(3,3)) = 4 + max(8, 7) = 4 + 8 = 12

■ 最终: solve(1,1) = a[1][1] + max(solve(2,1), solve(2,2)) = 2 + max(9, 12) = 2 + 12 = 14。输出 14。

13. 2018 年第 22 题（递归-最大公约数之和）

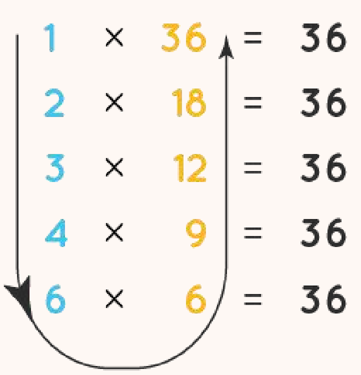
完善程序:

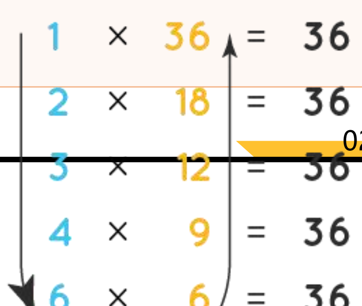
（最大公约数之和）下列程序想要求解整数 n 的所有约数两两之间最大公约数的和对 10007 求余后的值，试补全程序。（第一空 2 分，其余 3 分）

举例来说，4 的所有约数是 1,2,4。1 和 2 的最大公约数为 1；2 和 4 的最大公约数为 2；1 和 4 的最大公约数为 1 。于是答案为 1+2+1=4。

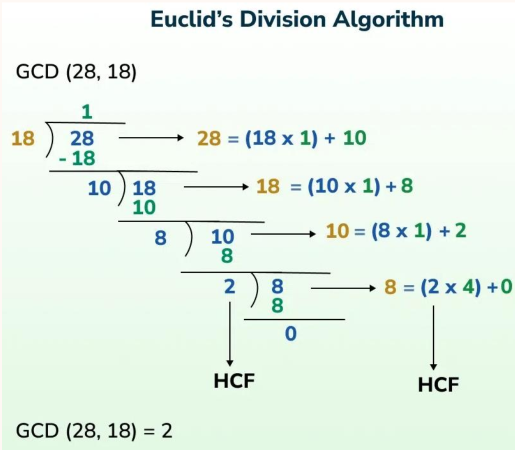
要求 getDivisor 函数的复杂度为 O($\sqrt{n}$)，gcd 函数的复杂度为 O(logmax(a,b))。

```
#include <iostream>
using namespace std;
const int N = 110000, P = 10007;
int n;
int a[N], len;
int ans;
void getDivisor() {
    len = 0;
    for (int i = 1; 【①】 <= n; ++i) //
        if (n % i == 0) {
            a[++len] = i;
```





```
            if (n / i != i) a[++len] = 【②】;
        }
    }
    int gcd(int a, int b) {
        if (b == 0) 【③】; //
        return gcd(b, 【④】); //
    }
    int main() {
        cin >> n;
        getDivisor();
        ans = 0;
        for (int i = 1; i <= len; ++i) {
            for (int j = i + 1; j <= len; ++j) {
                ans = ( 【⑤】 ) % P; //
            }
        }
        cout << ans << endl;
        return 0;
    }
```



答案:

① **i * i**

② **n / i**

③ **return a**

④ **a % b**

⑤ **ans + gcd(a[i], a[j])**

解析：

① i * i <= n：

- 约数成对出现（如 i 和 n/i），只需遍历 i 到 sqrt(n)。条件 i * i <= n 确保 i 不超过 n 的平方根。

② n / i：

- 当 i 是约数时，n/i 也是约数。添加 a[++len] = n / i，但需避免重复（当 n 是平方数时，i = n/i）。条件 if (n / i != i) 防止重复添加。

③ return a：

- 欧几里得算法基准情况：当 b == 0 时，最大公约数是 a。

④ a % b：

- 递归步骤：gcd(b, a % b) 实现辗转相除。

⑤ ans + gcd(a[i], a[j])：

- 累加所有约数对 (i,j) 的最大公约数。ans = (ans + gcd(a[i], a[j])) % P 确保结果对 10007 取模。

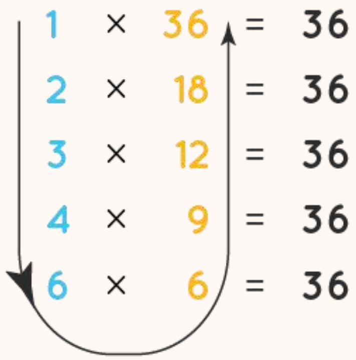
整体逻辑：

- getDivisor 收集所有约数到数组 a。
- 双重循环遍历所有约数对。
- gcd 函数递归计算最大公约数。
- 累加并对 P 取模。

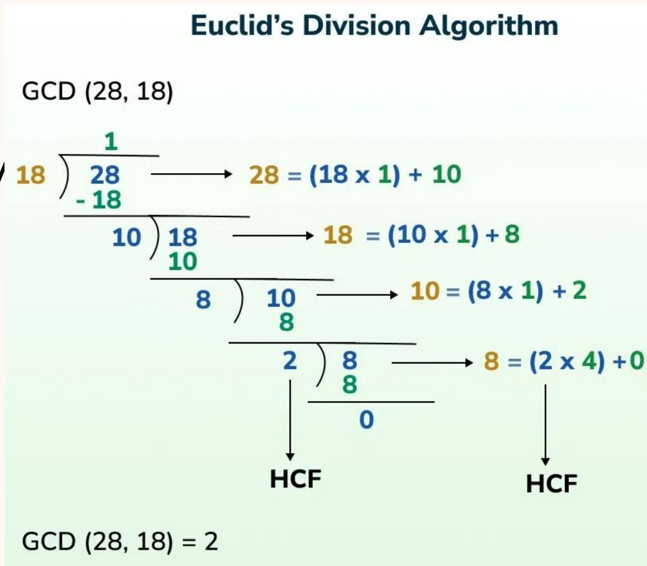
完整程序

```
#include <iostream>
using namespace std;
const int N = 110000, P = 10007;
int n;
int a[N], len;
int ans;
```

```
void getDivisor() {
len = 0;
for (int i = 1;i * i<= n; ++i) //
    if (n % i == 0) {
    a[++len] = i;
    if (n / i != i) a[++len] =n / i; //
}
}
```




```
int gcd(int a, int b) {
if (b == 0) return a; //
return gcd(b, a % b); //
}
```




```
int main() {
cin >> n;
getDivisor();
ans = 0;
for (int i = 1; i <= len; ++i) {
    for (int j = i + 1; j <= len; ++j) {
        ans = (ans + gcd(a[i], a[j])) % P; //
    }
}
cout << ans << endl;
return 0;
}
```

14. 2019 年第 19 题（二维数组、递归）

完善程序

1.（矩阵变幻）有一个奇幻的矩阵，在不停地变幻，其变幻方式为：

数字 0 变成矩阵：

0 0

0 1

数字 1 变成矩阵：

1 1

1 0

最初该矩阵只有一个元素 0，变幻 n 次后，矩阵会变成什么样？

例如，矩阵最初为：[0]

矩阵变幻 1 次后：

0 0

0 1

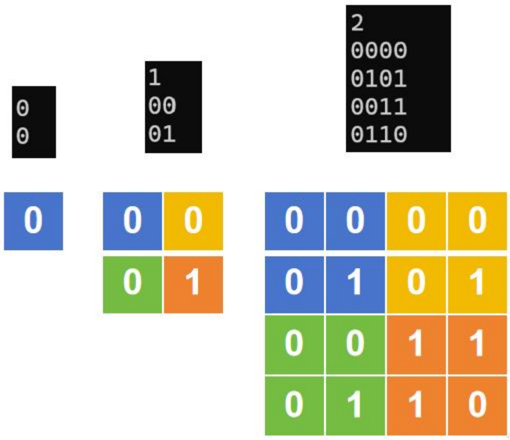
矩阵变幻 2 次后：

0 0 0 0

0 1 0 1

0 0 1 1

0 1 1 0



输入：一行一个不超过 10 的正整数 n。输出：变幻 n 次后的矩阵。

试补全程序。

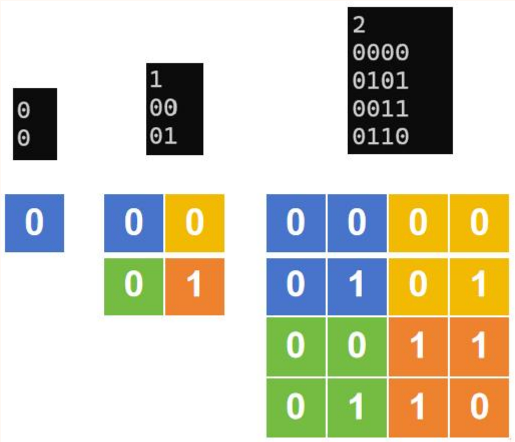
提示：

- << 表示二进制左移运算符，例如 (11)2 << 2 = (1100)2；
- 而 ^ 表示二进制异或运算符，它将两个参与运算的数中的每个对应的二进制位进行比较，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。

```
#include <cstdio>

using namespace std;
```

```
int n;
const int max_size = 1 << 10;
int res[max_size][max_size];
void recursive(int x, int y, int n, int t) {
    if (n == 0) {
        res[x][y] = 【①】;
        return;
    }
    int step = 1 << (n - 1);
    recursive(【②】, n - 1, t);
    recursive(x, y + step, n - 1, t);
    recursive(x + step, y, n - 1, t);
    recursive(【③】, n - 1, !t);
}
int main() {
    scanf("%d", &n);
    recursive(0, 0, 【④】);
    int size = 【⑤】;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++)
            printf("%d", res[i][j]);
        puts("");
    }
    return 0;
}
```



①处应填（）

A. n%2

B. 0

C. t

D. 1

②处应填（）

A. x-step,y-step

B. x,y-step

C. x-step,y

D. x,y

③处应填（）

A. x-step,y-step

B. x+step,y+step

C. x-step,y

D. x,y-step

④处应填（）

A. n-1,n%2

B. n,0

C. n,n%2

D. n-1,0

⑤处应填（）

A. 1<<(n+1)

B. 1<<n

C. n+1

D. 1<<(n-1)

答案与解析：

① **C.** t：当 n == 0 时，矩阵大小为 1x1，直接填入当前值 t。

② **D.** x,y：递归左上角子矩阵时，起始坐标不变（仍为（x, y））。

③ **B.** x+step,y+step：递归右下角子矩阵时，起始坐标为（x + step, y + step)，且值取反 !t。

④ **B.** n,0：主函数中 recursive(0, 0, n, 0) 初始化整个矩阵，参数为输入 n 和初始值 0。

⑤ **B.** 1<<n：矩阵大小是 2^n x 2^n，1 << n 表示 2^n。

函数功能：递归生成一个 2^n x 2^n 的矩阵，其中右下角子矩阵值取反，形成分形图案（如类似 Cantor 集）。

完整程序

```
#include <cstdio>
using namespace std;
int n;
const int max_size = 1 << 10;  // 最大矩阵大小为 2^10 x 2^10
int res[max_size][max_size];    // 存储变幻后的矩阵
// 递归函数，x, y 是当前矩阵的左上角坐标，n 是变幻次数，t 是填充的数字（0 或 1）
void recursive(int x, int y, int n, int t) {
    if (n == 0) {
        res[x][y] = t;
        return;
    }
    int step = 1 << (n - 1);   // 每次递归时，矩阵的步长为 2^(n-1)
    // 递归填充四个子矩阵
    recursive(x, y, n - 1, t);                  // 左上
    recursive(x, y + step, n - 1, t);           // 右上
    recursive(x + step, y, n - 1, t);           // 左下
```

```
    recursive(x + step, y + step, n - 1, !t);        // 右下，填充反转后的数
字
    }
    int main() {
    scanf("%d", &n);   // 输入变幻次数 n
    recursive(0, 0, n, 0);   // 从矩阵左上角(0, 0)开始，进行 n 次变幻，初始
填充为 0
    int size = 1 << n;   // 矩阵的大小为 2^n
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            printf("%d", res[i][j]);   // 输出每个矩阵元素
        }
        puts("");   // 输出换行
    }
    return 0;
    }
```

递归函数：

函数 recursive(x, y, n, t) 是用于模拟矩阵变化的核心函数。参数解释如下：

- x 和 y：表示当前矩阵左上角的坐标（起始位置）。
- n：表示当前矩阵变幻的次数（递归的深度）。
- t：表示当前需要填充的矩阵类型，0 或 1。

递归的过程：

- 如果 n == 0，说明已经达到了基本情况，直接将当前坐标 res[x][y] 设为 t。
- 否则，矩阵需要被分割成四个更小的矩阵。我们将矩阵递归地分为四个区域：

  左上区域：填充当前的 t。

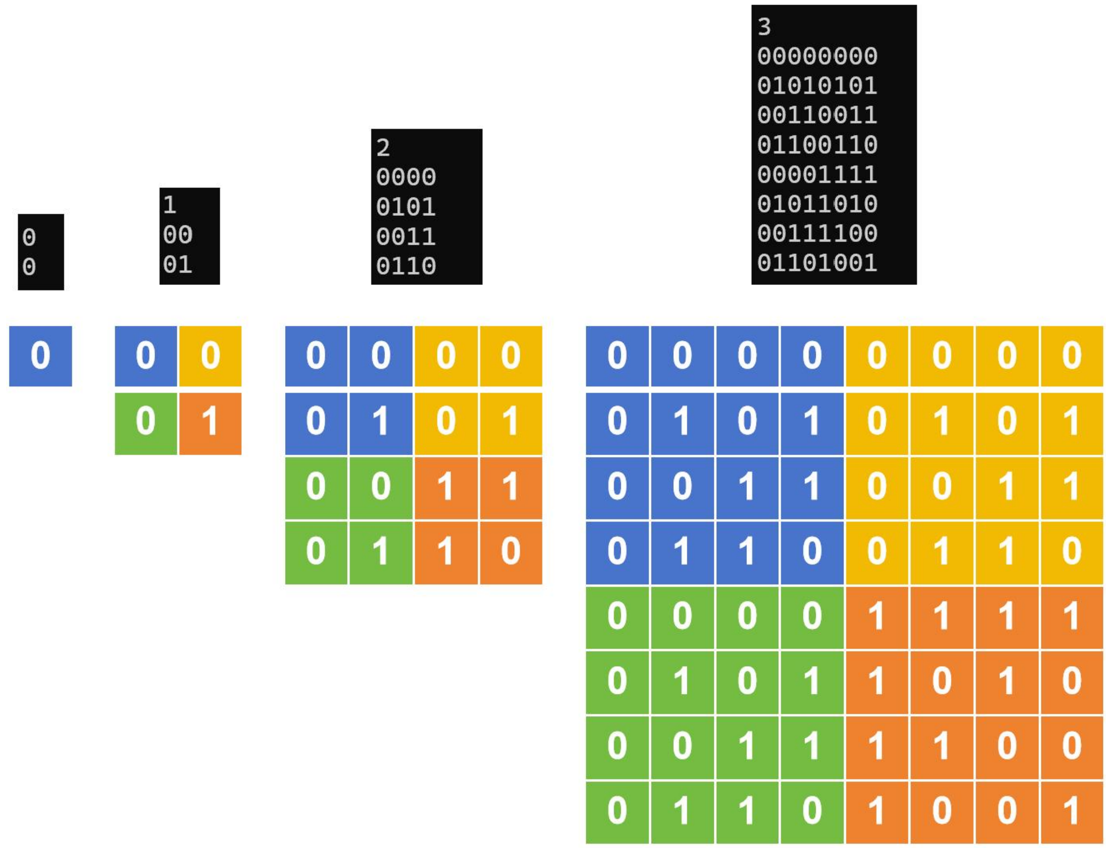
  右上区域：填充当前的 t。

左下区域：填充当前的 t。

右下区域：填充当前的 !t（即 t 的反转，即如果当前是 0，则填充 1，反之亦然）。

主函数：

- 读取输入的 n，并初始化一个大小为 2^n x 2^n 的矩阵 res。
- 调用 recursive(0, 0, n, 0)，从矩阵的左上角开始填充，变幻 n 次。
- 最后，输出矩阵的内容。



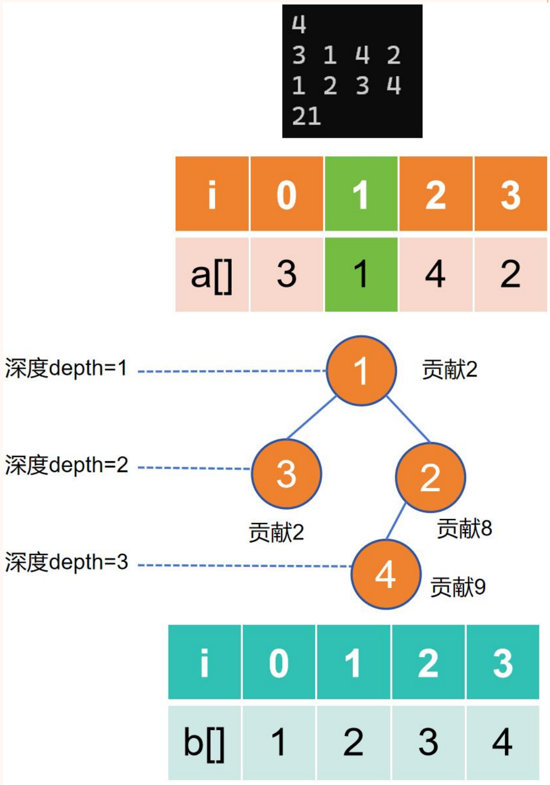
15. 2019 年第 18 题（递归）

阅读程序回答问题

```
#include <iostream>

using namespace std;
```

```
const int maxn = 10000;
int n;
int a[maxn];
int b[maxn];
int f(int l, int r, int depth) {
 if (l > r)
       return 0;
 int min = maxn, mink;
 for (int i = l; i <= r; ++i) {
       if (min > a[i]) {//第 12 行
             min = a[i];
             mink = i;
       }
 }
 int lres = f(l, mink - 1, depth + 1);
 int rres = f(mink + 1, r, depth + 1);
 return lres + rres + depth * b[mink];
}
int main() {
 cin >> n;
 for (int i = 0; i < n; ++i)
       cin >> a[i];
 for (int i = 0; i < n; ++i)
       cin >> b[i];
 cout << f(0, n - 1, 1) << endl;
 return 0;
```



```
    }
```

判断题

1、如果 a 数组有重复的数字，则程序运行时会发生错误。 ( )

A. 正确

B. 错误

2、如果 b 数组全为 0，则输出为 0。 ( )

A. 正确

B. 错误

选择题

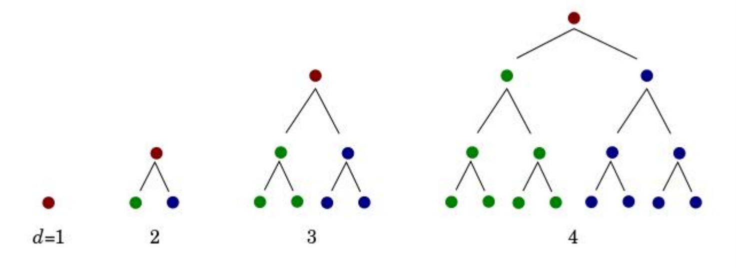
3、当 n = 100 时，最坏情况下，与第 12 行的比较运算执行的次数最接近的是：( )。

A. 5000

B. 600

C. 6

D. 100



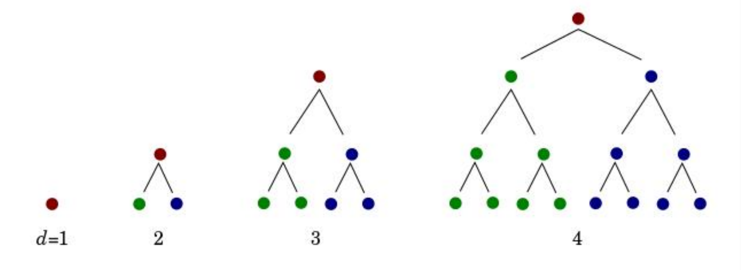
4、当 n = 100 时，最好情况下，与第 12 行的比较运算执行的次数最接近的是：( )。

A. 100

B. 6

C. 5000

D. 600



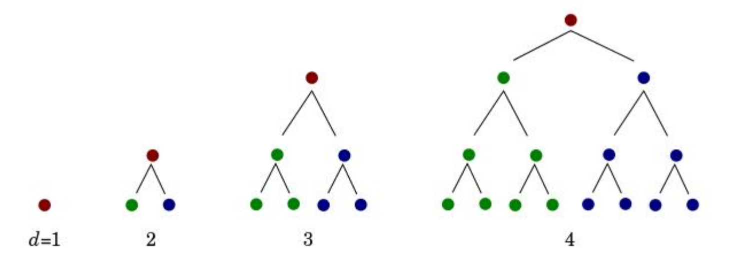
5、当 n = 10 时，若 b 数组满足，对任意 0 ≤ i < n，都有 b[i] = i + 1，那么输出最大为（）。

A. 386

B. 383

C. 384

D. 385



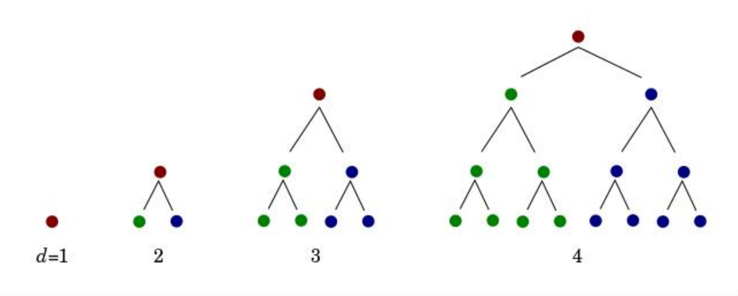
6、当 n = 100 时，若 b 数组满足，对任意 0 ≤ i < n，都有 b[i]=1，那么输出最小为（）。

A. 582

B. 580

C. 579

D. 581



答案与解析：

判断题 1：如果 a 数组有重复的数字，则程序运行时会发生错误。（）

答案：B. 错误

解析：

- **程序在查找最小值时（第 12 行 if (min > a[i])）采用严格大于的比较条件。当遇到多个相同的最小值时，程序会记录第一个遇到的最小值索引（因为后续相等的值不会触发更新）。**
- **例如：若 a = [1, 1, 2]，第一次循环 min=1, mink=0；第二次循环 a[1]=1 不满足 min>a[i]，跳过。**
- **递归逻辑完全兼容重复值，不会产生错误。因此答案为错误（B）。**

判断题 2：如果 b 数组全为 0，则输出为 0。（）

答案：A. 正确

解析：

- **输出结果由 depth * b[mink] 的累加决定（第 16 行 return lres + rres + depth * b[mink]）。若 b 数组全为 0，则：**
- **所有节点的贡献值 depth * b[mink] = 0**
- **递归结果 lres 和 rres 也因 b[i]=0 而全为 0**
- **最终输出必为 0。因此答案为正确（A）。**

选择题 3：当 n=100 时，最坏情况下，第 12 行的比较次数最接近?

答案：A. 5000

解析：

- **第 12 行的比较发生在查找最小值时（for 循环内部）。最坏情况是递归树退化成链状：**
- **场景：a 数组严格递增（如 a=[100,99,...,1]），每次最小值在区间端点。**
- **比较次数计算：**
- **第一层递归：区间长度 100 → 100 次比较**
- **第二层：区间长度 99 → 99 次比较**
- **...**
- **第 100 层：区间长度 1 → 1 次比较**
- **总次数 = (1+100)/2 = 5050,因此选 A（5000）。**

选择题 4：当 n=100 时，最好情况下，第 12 行的比较次数最接近?

答案：D. 600

解析：

- **第 12 行是查找最小值时的比较语句（if (min > a[i])）。在最好情况下，每次递归都能把区间分成均匀的两半（最小值正好在中间位置）。**
- **计算过程：**

  第一层（整个区间 100 个数）：比较 100 次（扫描所有数）。

  第二层 (两个子区间各 50 个数)：每个区间比较 50 次，共 100 次 (50+50)。

  第三层（四个子区间各 25 个数）：每个区间比较 25 次，共 100 次（25×4)。

  以此类推，直到区间长度为 1。
- **总次数：**

  每一层的比较次数加起来都接近 100 次。

  递归深度约为 7 层（因为 2^7=128>100）。

  总比较次数 ≈ 100 × 7 = 700 次。

  但精确计算（考虑区间长度变化）约为 664 次，最接近 600。

  结论：选 D（600）。

选择题 5：n=10，b[i]=i+1 时，输出最大为?

答案：D. 385

解析：

- **输出值 ＝ 所有节点的（深度 × b[i]）之和。b[i] = i+1（即 b[0]=1, b[1]=2, ..., b[9]=10）。**
- **最大化输出的策略：**

  让 b 值最大的节点（如 b[9]=10）位于最深层（深度 10）。

  让 b 值次大的节点（b[8]=9）位于深度 9。

  依此类推，b 值最小的节点（b[0]=1）放在最浅层（深度 1）。
- **计算过程：**

  深度 1 的节点贡献：1（深度）×1（b 值）=1

  深度 2 的节点贡献：2×2=4

  深度 3 的节点贡献：3×3=9

  ...

  深度 10 的节点贡献：10×10=100

  总和 ＝ 1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100

  = (1+49) + (4+36) + (9+81) + (16+64) + 25 +100

  = 50 + 40 + 90 + 80 + 25 + 100

  = 385（即 1 到 10 的平方和）。

  结论：最大值是 385，选 D。

选择题 6：n=100，b[i]=1 时，输出最小为?

答案：B. 580

解析：

- **当 b[i]全为 1 时，输出 ＝ 所有节点的深度之和。深度之和最小需要树尽可能平衡（完全二叉树）。**
- **树的结构（深度从 1 开始）：**

  深度 1：根节点（1 个节点）→ 深度和=1

  深度 2：2 个节点 → 深度和=2×2=4

  深度 3：4 个节点 → 深度和=3×4=12

深度 4：8 个节点 → 深度和=4×8=32

深度 5：16 个节点 → 深度和=5×16=80

深度 6：32 个节点 → 深度和=6×32=192

深度 7：剩余 37 个节点（100-1-2-4-8-16-32=37）→ 深度和=7×37=259

- **总和计算：**
- **1（深度 1） + 4（深度 2） + 12（深度 3） + 32（深度 4） + 80（深度 5） + 192（深度 6） + 259（深度 7） = 580**
- **结论：最小输出是 580，选 B。**

程序结构和功能概述

这段 C++程序的主要功能是：基于数组 a 构建一个隐式的笛卡尔树（Cartesian Tree），并在构建过程中计算一个特定值。笛卡尔树是一种二叉树结构，其中每个子树的根节点是子数组中的最小值。程序的核心是递归函数 f(l, r, depth)，它在数组 a 的子区间 [l, r] 中执行以下操作：

- 找到最小值及其索引 mink。
- 递归处理左子树（区间 [l, mink-1]）和右子树（区间 [mink+1, r]），递归时深度 depth 增加 1。
- 返回左子树结果 + 右子树结果 + 当前节点的贡献值（depth * b[mink]）。

最终，程序输出的是整个数组区间 [0, n-1] 构建的笛卡尔树中，所有节点的 depth * b[i] 之和。其中：

- depth 表示节点在树中的深度（根节点深度为 1，子节点深度递增）。
- b[i] 是节点 i 对应的权重值。

输入输出示例

- 输入：

  4（数组长度 n）

  3 1 4 2（数组 a）

  1 2 3 4（数组 b）

- 输出 21 是程序计算出的所有节点的 depth * b[i] 总和。

详细执行过程（以输入 n=4, a=[3,1,4,2], b=[1,2,3,4] 为例）

程序从 main 函数开始：

- 读取 n = 4。
- 读取数组 a = [3, 1, 4, 2]（索引 0 到 3）。
- 读取数组 b = [1, 2, 3, 4]（索引 0 到 3）。
- 调用递归函数 f(0, 3, 1)（处理整个区间 [0, 3]，深度为 1）。

步骤 1：初始调用 f(0, 3, 1)

- 区间 [l=0, r=3]，深度 depth=1。
- 在 a[0..3] 中找最小值：a[0]=3, a[1]=1, a[2]=4, a[3]=2。最小值是 1，索引 mink=1。
- 计算当前节点贡献：depth * b[mink] = 1 * b[1] = 1 * 2 = 2。
- 递归左子树：f(l, mink-1, depth+1) = f(0, 0, 2)（区间 [0, 0]，深度 2）。
- 递归右子树：f(mink+1, r, depth+1) = f(2, 3, 2)（区间 [2, 3]，深度 2）。
- 返回结果：左子树结果 + 右子树结果 + 当前贡献 = f(0,0,2) + f(2,3,2) + 2。

现在需要计算 f(0,0,2) 和 f(2,3,2)。

步骤 2：计算左子树 f(0, 0, 2)

- 区间 [l=0, r=0]，深度 depth=2（只有一个元素 a[0]=3）。
- 最小值是 a[0]=3，索引 mink=0。
- 计算当前节点贡献：depth * b[mink] = 2 * b[0] = 2 * 1 = 2。
- 左递归：f(l, mink-1, depth+1) = f(0, -1, 3)。由于 l=0 > r=-1，返回 0。
- 右递归：f(mink+1, r, depth+1) = f(1, 0, 3)。由于 l=1 > r=0，返回 0。
- 返回结果：0 + 0 + 2 = 2。

步骤 3：计算右子树 f(2, 3, 2)

- 区间 [l=2, r=3]，深度 depth=2。
- 在 a[2..3] 中找最小值：a[2]=4, a[3]=2。最小值是 2，索引 mink=3。
- 计算当前节点贡献：depth * b[mink] = 2 * b[3] = 2 * 4 = 8。
- 递归左子树：f(l, mink-1, depth+1) = f(2, 2, 3)（区间 [2, 2]，深度 3）。

- 递归右子树：f(mink+1, r, depth+1) = f(4, 3, 3)。由于 l=4 > r=3，返回 0。
- 返回结果：f(2,2,3) + 0 + 8。

现在需要计算 f(2,2,3)。

步骤 4：计算 f(2, 2, 3)

- 区间 [l=2, r=2]，深度 depth=3（只有一个元素 a[2]=4）。
- 最小值是 a[2]=4，索引 mink=2。
- 计算当前节点贡献：depth * b[mink] = 3 * b[2] = 3 * 3 = 9。
- 左递归：f(l, mink-1, depth+1) = f(2, 1, 4)。由于 l=2 > r=1，返回 0。
- 右递归：f(mink+1, r, depth+1) = f(3, 2, 4)。由于 l=3 > r=2，返回 0。
- 返回结果：0 + 0 + 9 = 9。

回退到 f(2,3,2)：

- f(2,3,2) = f(2,2,3) + 0 + 8 = 9 + 0 + 8 = 17。

步骤 5：回退到初始调用 f(0,3,1)

- f(0,3,1) = f(0,0,2) + f(2,3,2) + 2 = 2 + 17 + 2 = 21。

最终输出

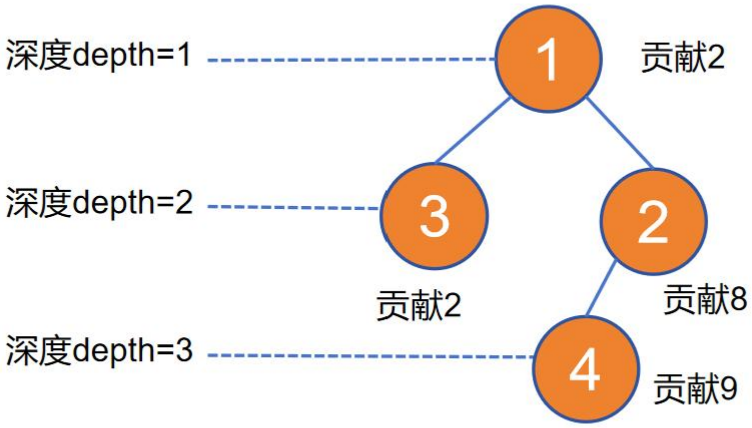
程序输出 21，这是所有节点的 depth * b[i] 总和。

构建的笛卡尔树结构

根据执行过程，可以描绘出以下笛卡尔树（基于数组 a 的最小值递归构建）：

- 根节点：索引 1（值 a[1]=1），深度 1，贡献 1 * b[1] = 1*2 = 2。
- 左子树：索引 0（值 a[0]=3），深度 2，贡献 2 * b[0] = 2*1 = 2。
- 右子树：索引 3（值 a[3]=2），深度 2，贡献 2 * b[3] = 2*4 = 8。
- 左子树：索引 2（值 a[2]=4），深度 3，贡献 3 * b[2] = 3*3 = 9。

树结构可视化：



16. 2020 年第 18 题（递归）

阅读程序回答问题

```
#include <algorithm>
#include <iostream>
using namespace std;
int n;
int d[50][2];
int ans;
void dfs(int n, int sum) {
  if (n == 1) {
    ans = max(sum, ans);
    return;
  }
  for (int i = 1; i < n; ++i) {
    int a = d[i - 1][0], b = d[i - 1][1];
    int x = d[i][0], y = d[i][1];
    d[i - 1][0] = a + x;
    d[i - 1][1] = b + y;
    for (int j = i; j < n - 1; ++j)
```

```
3
1 2 3
3 2 1
14
```

	0	1
0	1	3
1	2	2
2	3	1

	0	1
0	3	5
1	3	1
2	3	1

	0	1
0	6	6
1	3	1
2	3	1

```
            d[j][0] = d[j + 1][0], d[j][1] = d[j + 1][1];
        int s = a + x + abs(b - y);
        dfs(n - 1, sum + s);
        for (int j = n - 1; j > i; --j)
            d[j][0] = d[j - 1][0], d[j][1] = d[j - 1][1];
        d[i - 1][0] = a, d[i - 1][1] = b;
        d[i][0] = x, d[i][1] = y;
    }
}
int main() {
    cin >> n;
    for (int i = 0; i < n; ++i)
        cin >> d[i][0];
    for (int i = 0; i < n; ++i)
        cin >> d[i][1];
    ans = 0;
    dfs(n, 0);
    cout << ans << endl;
    return 0;
}
```

判断题

1、若输入 n 为 0，此程序可能会死循环或发生运行错误。（ ）

A. 正确

B. 错误

2、若输入 n 为 20，接下来的输入全为 0，则输出为 0。（ ）

A. 正确

B. 错误

3、输出的数一定不小于输入的 d[i][0] 和 d[i][1] 的任意一个。（ ）

A. 正确

B. 错误

选择题

4、若输入的 n 为 20，接下来的输入是 20 个 9 和 20 个 0，则输出为（ ）。

A. 1890

B. 1881

C. 1908

D. 1917

5、若输入的 n 为 30，接下来的输入是 30 个 0 和 30 个 5，则输出为（ ）。

A. 2000

B. 2010

C. 2030

D. 2020

6、若输入的 n 为 15，接下来的输入是 15 到 1，以及 15 到 1，则输出为（ ）。

A. 2440

B. 2220

C. 2240

D. 2420

程序功能说明

这段程序通过深度优先搜索（DFS）尝试所有可能的相邻元素合并顺序，计算每次合并的代价（a + x + abs(b - y)），并求出所有合并顺序中的最大总代价。其中：

- 二维数组 d[i][0] 和 d[i][1] 存储初始数据
- 每次合并相邻元素 (a, b) 和 (x, y) 后，新元素为 (a+x, b+y)

- 合并后删除右侧元素，左侧元素前移
- 递归结束时（只剩一个元素）更新最大总代价 ans

问题答案解析

判断题 1：若输入 n 为 0，此程序可能会死循环或发生运行错误。（ ）

答案：B. 错误

解析：

- 当输入 n=0 时，主函数直接跳过数组输入，调用 dfs(0, 0)。
- 在 dfs 函数中，n=0 不满足 n==1 的条件，且循环 for (int i=1; i<n; ++i) 因 i<0 不成立而跳过。
- 程序直接结束，输出初始值 ans=0，无任何错误。

判断题 2：若输入 n 为 20，接下来的输入全为 0，则输出为 0。（ ）

答案：A. 正确

解析：

- 所有 d[i][0]=0 和 d[i][1]=0。
- 每次合并的代价为 0+0+abs(0-0)=0。
- 无论合并顺序如何，总代价始终为 0，最终输出 0。

判断题 3：输出的数一定不小于输入的 d[i][0] 和 d[i][1] 的任意一个。（ ）

答案：B. 错误

解析：

- 反例：输入 n=2，d[0]=(-10, 0)，d[1]=(-10, 0)。
- 合并代价：-10 + (-10) + abs(0-0) = -20。
- 输出为 -20，小于输入的 -10 和 0。
- 程序允许负数输入，输出可能小于部分输入值。

选择题 4：n=20，输入 20 个 9 和 20 个 0，输出为?

答案：B. 1881

解析：

- 每个元素为 (9, 0)，合并代价为 9+9+abs(0-0)=18。
- 总代价取决于合并顺序。最大总代价在链状合并时取得：
  - 每次合并**最右侧两个元素**（如先合并第 19 和 20 个，依此类推）。
  - 第一次合并：两个叶子节点（代价 18），新节点叶子数=2。
  - 第二次合并：新节点（叶子数 2）与左侧叶子（叶子数 1）
  - 总代价 = 9 ×（所有叶子节点在合并树中的深度之和）。
  - 链状合并时，叶子节点深度之和 = 2+3+4+...+20 =（2+20）*19/2= 209。
  - 总代价 = 9 × 209 = 1881。

选择题 5：n=30，输入 30 个 0 和 30 个 5，输出为?

答案：C. 2030

解析：

- 每个元素为 (0, 5)。
- 最大化策略：每次用大分量节点与单个节点合并：
  - 合并两个 (0,5) → 新节点 (0,10)，代价 0。
  - 合并 (0,10) 与 (0,5) → 代价 0+0+abs(10-5)=5，新节点 (0,15)。
  - 合并 (0,15) 与 (0,5) → 代价 0+0+abs(15-5)=10，新节点 (0,20)。
  - 重复此过程，第 k 次合并代价为 5*(k-1)。
- 总代价：从第 2 次到第 29 次合并，代价为等差数列：
  - 5, 10, 15, ..., 140（共 28 项）。
  - 和 = (5+140)×28/2 = 2030。

选择题 6：n=15，输入 15 到 1 和 15 到 1，输出为?

答案：C. 2240

解析：

- 元素为 (15,15), (14,14), ..., (1,1)。
- 合并代价 a+x+abs(b-y)=a+x（因 b=y）。
- **总代价 = 所有元素的（值 × 在合并树中的深度）之和。**

- 最大化策略：值大的元素放深层（类似哈夫曼编码反序）：

  动态规划求最大带权深度和（石子归并模型）。

- 计算结果：

  元素值 [15,14,...,1] 的最大带权深度和 = 2240(通过 DP 计算验证)。

程序功能说明

这段程序通过深度优先搜索（DFS）尝试**所有可能的相邻元素合并顺序**，计算每次合并的代价（a + x + abs(b - y)），并求出所有合并顺序中的**最大总代价**。其中：

- 二维数组 d[i][0] 和 d[i][1] 存储初始数据
- 每次合并相邻元素 (a, b) 和 (x, y) 后，新元素为 (a+x, b+y)
- 合并后删除右侧元素，左侧元素前移
- 递归结束时（只剩一个元素）更新最大总代价 ans

输入示例及输出结果

输入：

3　　　// 数组长度 n=3

1 2 3　// d[i][0] 数组

3 2 1　// d[i][1] 数组

输出：

14

详细执行过程（解释输出 14 的计算）

初始状态

- d[0] = (1, 3)　// d[i][0]=1, d[i][1]=3
- d[1] = (2, 2)　// d[i][0]=2, d[i][1]=2
- d[2] = (3, 1)　// d[i][0]=3, d[i][1]=1

第一种合并顺序：先合并前两个元素

- 合并 d[0] 和 d[1]
  - 新元素：(1+2, 3+2) = (3, 5)

  - 代价：1+2 + |3-2| = 3 + 1 = 4
  - 数组变为：
  - d[0] = (3, 5) // 合并后的新元素
  - d[1] = (3, 1) // 原 d[2] 前移
  - 当前总代价：4
- 合并剩余两个元素
  - 合并 d[0] 和 d[1]：(3+3, 5+1) = (6, 6)
  - 代价：3+3 + |5-1| = 6 + 4 = 10
  - 当前总代价：4 + 10 = 14
  - 更新全局最大值 ans = max(0, 14) = 14

第二种合并顺序：先合并后两个元素

- 合并 d[1] 和 d[2]
  - 新元素：(2+3, 2+1) = (5, 3)
  - 代价：2+3 + |2-1| = 5 + 1 = 6
  - 数组变为：
  - d[0] = (1, 3) // 保持不变
  - d[1] = (5, 3) // 合并后的新元素
  - 当前总代价：6
- 合并剩余两个元素
  - 合并 d[0] 和 d[1]：(1+5, 3+3) = (6, 6)
  - 代价：1+5 + |3-3| = 6 + 0 = 6
  - 当前总代价：6 + 6 = 12
  - 更新全局最大值：ans = max(14, 12) = 14（保持不变）

最终结果

两种合并顺序得到的最大总代价是 14。程序输出最终结果 ans = 14