# 题目

给出基数为 -2 的两个数 arr1 和 arr2，返回两数相加的结果。

数字以 数组形式 给出：数组由若干 0 和 1 组成，按最高有效位到最低有效位的顺序排列。例如，arr = [1,1,0,1] 表示数字 (-2)^3 + (-2)^2 + (-2)^0 = -3。数组形式 中的数字 arr 也同样不含前导零：即 arr == [0] 或 arr[0] == 1。

返回相同表示形式的 arr1 和 arr2 相加的结果。两数的表示形式为：不含前导零、由若干 0 和 1 组成的数组。

示例 1：

输入：arr1 = [1,1,1,1,1], arr2 = [1,0,1]

输出：[1,0,0,0,0]

解释：arr1 表示 11，arr2 表示 5，输出表示 16 。

示例 2：

输入：arr1 = [0], arr2 = [0]

输出：[0]

示例 3：

输入：arr1 = [0], arr2 = [1]

输出：[1]

提示：

1 <= arr1.length, arr2.length <= 1000

arr1[i] 和 arr2[i] 都是 0 或 1

arr1 和 arr2 都没有前导0

# 分析

要解决基数为-2的两个数相加问题，核心是理解-2进制的加法规则——与十进制类似需处理“进位”，但因基数为负数，进位规则需特殊调整，同时要避免前导零（除非结果为0）。

解题思路

1、核心背景：-2 进制的特点

- -2 进制的每一位权重为 (-2)^k（k 从0开始，对应最低有效位到最高有效位）。

- 加法需处理“当前位和”与“进位”，但因基数为负，进位规则与十进制不同：

设当前位和为 sum = a + b + carry（a、b为两数当前位，carry 为上一位进位），需通过调整使当前位结果为0 或 1（-2 进制仅含 0/1），并计算新的进位。

2、-2 进制加法规则（关键）

对当前位和 sum，分情况处理：

| sum | 当前位结果 | 新进位 | 逻辑说明 |

|------|------------|--------|----------|

| 0 | 0 | 0 | sum 直接满足 0/1，无进位 |

| 1 | 1 | 0 | sum 直接满足 0/1，无进位 |

| 2 | 0 | -1 | 2 = (-2)\*(-1) + 0 → 当前位为0，进位-1 |

| 3 | 1 | -1 | 3 = (-2)\*(-1) + 1 → 当前位为1，进位-1 |

| -1 | 1 | 1 | -1 = (-2)\*1 + 1 → 当前位为1，进位1 |

3、算法步骤

- 逆序遍历：从两数的最低有效位（数组末尾）开始相加（因数组按“最高有效位→最低有效位”存储）。

- 处理进位：按上述规则计算当前位结果和新进位，直到两数遍历完毕且进位为0。

- 去前导零：结果数组按“最低有效位→最高有效位”存储，逆序后需移除前导零（若结果非0）。

代码实现（含详细注释）：

class Solution {

public:

vector<int> addNegabinary(vector<int>& arr1, vector<int>& arr2) {

vector<int> result; // 存储结果（先按“最低有效位→最高有效位”顺序）

int carry = 0; // 进位（初始为0）

int i = arr1.size() - 1; // arr1的指针（从末尾，即最低有效位开始）

int j = arr2.size() - 1; // arr2的指针（从末尾开始）

// 遍历两数所有位，且处理完剩余进位

while (i >= 0 || j >= 0 || carry != 0) {

// 取当前位的值（若指针已越界，视为0）

int a = (i >= 0) ? arr1[i] : 0;

int b = (j >= 0) ? arr2[j] : 0;

// 计算当前位的总和（当前位两数之和 + 上一位进位）

int sum = a + b + carry;

// 按-2进制规则计算当前位结果和新进位

if (sum >= 2) {

// sum=2 → 结果0，进位-1；sum=3 → 结果1，进位-1

result.push\_back(sum % 2);

carry = -1;

} else if (sum == -1) {

// sum=-1 → 结果1，进位1（因 -1 = (-2)\*1 + 1）

result.push\_back(1);

carry = 1;

} else {

// sum=0或1 → 结果直接为sum，无进位

result.push\_back(sum);

carry = 0;

}

// 指针向前移动（处理更高一位，即数组左侧）

i--;

j--;

}

// 步骤1：逆序结果数组，转为“最高有效位→最低有效位”顺序

reverse(result.begin(), result.end());

// 步骤2：移除前导零（仅保留一个0，若结果全为0）

// 找到第一个非零元素的位置

auto first\_non\_zero = result.begin();

while (first\_non\_zero != result.end() && \*first\_non\_zero == 0) {

first\_non\_zero++;

}

// 若全为0，返回[0]；否则移除前导零

if (first\_non\_zero == result.end()) {

return {0};

} else {

result.erase(result.begin(), first\_non\_zero);

}

return result;

}

};

代码解释

1、逆序遍历与进位处理：

数组按“最高有效位→最低有效位”存储，因此从末尾（i = arr1.size()-1、j = arr2.size()-1）开始遍历，对应 -2 进制的最低有效位。每次计算 sum = a + b + carry，按规则更新当前位结果和进位，确保当前位仅为 0 或 1。

2、结果调整：

- 逆序：结果数组初始按“最低有效位→最高有效位”存储，逆序后转为题目要求的“最高有效位→最低有效位”。

- 去前导零：遍历找到第一个非零元素，移除其左侧的所有零（若全为零，返回 [0]，避免空数组或全零数组）。

复杂度分析

- 时间复杂度：O(max(len1, len2))，其中 len1、len2 分别为 arr1、arr2 的长度。遍历过程最多处理 max(len1, len2) + 1 位（含最终进位），逆序和去前导零均为线性时间。

- 空间复杂度：O(max(len1, len2))，结果数组的长度最多为 max(len1, len2) + 1（含进位扩展的一位）。

该方法严格遵循 -2 进制的加法规则，高效处理进位和结果格式，确保输出符合题目要求（无空数组、无多余前导零）。