作业 5 数组与广义表解答

题 1 m 维数组表示多项式的输出

思路：多项式按维度拉平成一维数组 `coeffs`，每个维度的上界存于 `shape[i]`。通过递归枚举指数向量 $(e_1,\ldots,e_m)$，按字典序逆序访问每个组合；遇到非零系数时输出对应项。

```c
#include <stdio.h>

typedef struct {
    int dim;
    const int *shape;
    const int *coeffs;
} PolyGrid;

static int index_of(const PolyGrid *poly, const int expo[]) {
    int idx = 0;
    int stride = 1;
    for (int i = poly->dim - 1; i >= 0; --i) {
        idx += expo[i] * stride;
        stride *= (poly->shape[i] + 1);
    }
    return idx;
}

static void print_term(int coeff, const int expo[], int dim, int
first_term) {
    if (!first_term) {
        putchar(coeff >= 0 ? '+' : '-');
    } else if (coeff < 0) {
        putchar('-');
    }
    int abs_c = coeff >= 0 ? coeff : -coeff;
    int use_var = 0;
    if (abs_c != 1) {
        printf("%d", abs_c);
    }
    for (int i = 0; i < dim; ++i) {
        if (expo[i] == 0) continue;
        use_var = 1;
        printf("x%d", i + 1);
        if (expo[i] != 1) printf("^%d", expo[i]);
    }
    if (!use_var && abs_c == 1) printf("1");
}

static void walk_poly(const PolyGrid *poly, int level,
                      int expo[], int *count) {
    if (level == poly->dim) {
        int coeff = poly->coeffs[index_of(poly, expo)];
```

```
        if (coeff != 0) {
            print_term(coeff, expo, poly->dim, *count == 0);
            (*count)++;
        }
        return;
    }
    for (int e = poly->shape[level]; e >= 0; --e) {
        expo[level] = e;
        walk_poly(poly, level + 1, expo, count);
    }
}

void print_polynomial(const PolyGrid *poly) {
    int expo[16] = {0};   // 假定维度不超过 16
    int count = 0;
    walk_poly(poly, 0, expo, &count);
    if (count == 0) printf("0");
    putchar('\n');
}
```

时间复杂度：需要访问所有指数排列，次数为 $\prod_{i=1}^{m}(shape[i]+1)$，故时间复杂度为 $O!\left(\prod(shape[i]+1)\right)$，额外空间为常数级数组。

题 2 三元组顺序表稀疏矩阵加法（原地）

思路：三元组序列采用行优先顺序。使用双指针 `i`、`j` 扫描矩阵 A、B，`k` 表示写回 A 的位置；比较 `(row, col)`，较小者直接写入，位置相同则相加并依据结果是否为零决定是否保留。

```
typedef struct {
    int row;
    int col;
    int val;
} Triple;

void add_sparse_inplace(Triple A[], int *lenA, const Triple B[], int lenB)
{
    int i = 0, j = 0, k = 0;
    while (i < *lenA && j < lenB) {
        int ar = A[i].row, ac = A[i].col;
        int br = B[j].row, bc = B[j].col;
        if (ar < br || (ar == br && ac < bc)) {
            if (k != i) A[k] = A[i];
            ++i; ++k;
        } else if (ar > br || (ar == br && ac > bc)) {
            A[k++] = B[j++];
        } else {
            int sum = A[i].val + B[j].val;
            ++i; ++j;
            if (sum != 0) {
                A[k].row = ar;
                A[k].col = ac;
```

```
                A[k].val = sum;
                ++k;
            }
        }
    }
    while (i < *lenA) {
        if (k != i) A[k] = A[i];
        ++i; ++k;
    }
    while (j < lenB) {
        A[k++] = B[j++];
    }
    *lenA = k;
}
```

时间复杂度：每个非零三元组至多处理一次，若 A、B 的非零元数分别为 m、n，则时间复杂度为 $O(m+n)$，额外空间为常数。

题 3 V + B 结构的稀疏矩阵加法

思路：矩阵的布尔数组 bitmap 记录非零位置，vals 中保存对应值。按行优先顺序遍历 bitmap，同步维护 pa、pb、pc 三个指针，分别指向 A、B 和结果的非零列表。

```
typedef struct {
    int rows;
    int cols;
    const int *bitmap;
    const int *vals;
    int nnz;
} MatrixVB;

typedef struct {
    int *bitmap;
    int *vals;
    int nnz;
} MatrixVBResult;

MatrixVBResult add_matrix_vb(const MatrixVB *A, const MatrixVB *B) {
    MatrixVBResult C = {0};
    int total = A->rows * A->cols;
    C.bitmap = (int *)calloc(total, sizeof(int));
    C.vals = (int *)malloc((A->nnz + B->nnz) * sizeof(int));
    int pa = 0, pb = 0, pc = 0;
    for (int idx = 0; idx < total; ++idx) {
        int flagA = A->bitmap[idx];
        int flagB = B->bitmap[idx];
        if (flagA && flagB) {
            int sum = A->vals[pa++] + B->vals[pb++];
            if (sum != 0) {
                C.bitmap[idx] = 1;
                C.vals[pc++] = sum;
```

```
            }
        } else if (flagA) {
            C.bitmap[idx] = 1;
            C.vals[pc++] = A->vals[pa++];
        } else if (flagB) {
            C.bitmap[idx] = 1;
            C.vals[pc++] = B->vals[pb++];
        }
    }
    C.nnz = pc;
    return C;
}
```

时间复杂度：必须遍历整张布尔矩阵，代价为 $O(rows \times cols)$；额外空间与非零元总数成正比。

题 4 广义表表示的多元多项式加法

思路：节点记录指数 `exp` 与类型标记 `tag`。链表按指数升序排列，递归比较当前节点指数；若指数相同则递归合并子表（`tag=1`），空子表表示该项抵消并应删除。

```
typedef struct Term Term;

struct Term {
    int exp;
    int tag;        // 0: 系数, 1: 子表
    union {
        double coef;
        Term *sub;
    } data;
    Term *next;
};

static Term *dup_list(const Term *src) {
    if (!src) return NULL;
    Term *node = (Term *)malloc(sizeof(Term));
    *node = *src;
    if (src->tag == 1) {
        node->data.sub = dup_list(src->data.sub);
    }
    node->next = dup_list(src->next);
    return node;
}

Term *add_poly_list(const Term *A, const Term *B) {
    if (!A) return dup_list(B);
    if (!B) return dup_list(A);
    if (A->exp < B->exp) {
        Term *node = dup_list(A);
        node->next = add_poly_list(A->next, B);
        return node;
    }
```

```
    if (A->exp > B->exp) {
        Term *node = dup_list(B);
        node->next = add_poly_list(A, B->next);
        return node;
    }
    Term *node = (Term *)malloc(sizeof(Term));
    node->exp = A->exp;
    node->next = NULL;
    if (A->tag == 0 && B->tag == 0) {
        double sum = A->data.coef + B->data.coef;
        if (sum == 0.0) {
            free(node);
            return add_poly_list(A->next, B->next);
        }
        node->tag = 0;
        node->data.coef = sum;
    } else {
        node->tag = 1;
        node->data.sub = add_poly_list(A->data.sub, B->data.sub);
        if (!node->data.sub) {
            free(node);
            return add_poly_list(A->next, B->next);
        }
    }
    node->next = add_poly_list(A->next, B->next);
    return node;
}
```

时间复杂度：设两多项式的非零项总数为 `t`，每项在递归中最多访问一次，因此时间复杂度为 $O(t)$；递归深度不超过变量层数。

题 5 判别两个广义表是否相等

思路：沿用题 4 的节点结构。递归比较对应节点的指数与类型；若节点为子表则继续递归比较子表，最终通过 `next` 指针同步遍历两链表。

```
int equal_poly_list(const Term *A, const Term *B) {
    if (!A && !B) return 1;
    if (!A || !B) return 0;
    if (A->exp != B->exp || A->tag != B->tag) return 0;
    if (A->tag == 0) {
        if (A->data.coef != B->data.coef) return 0;
    } else {
        if (!equal_poly_list(A->data.sub, B->data.sub)) return 0;
    }
    return equal_poly_list(A->next, B->next);
}
```

时间复杂度：若两个广义表完全相同，则需要遍历全部节点，设节点总数为 `k`，时间复杂度为 $O(k)`，递归深度取决于广义表最大层数。