```
/* ==================================================================
 *
 * CS 566 - Assignment 03
 * Camillo Lugaresi, Cosmin Stroe
 *
 * Common operations, including serial matrix multiplication algorithms
 * and their helper functions (naive and Strassen), matrix allocation functions,
 * matrix reordering functions, and various other helper functions.
 *
 * ================================================================== */

#include "common.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void alloc_matrix(struct matrix *m, int n)
{
        m->n = n;
        m->data = malloc(sizeof(*m->data) * m->n * m->n);
}

void alloc_matrix2(struct matrix2 *m, int n)
{
        m->n = m->stride = n;
        m->data = malloc(sizeof(*m->data) * m->n * m->n);
}

void make_matrix2(struct matrix *m, struct matrix2 *m2)
{
        m2->n = m2->stride = m->n;
        m2->data = m->data;
}

void make_submatrix(struct matrix2 *m, struct matrix2 *sub, int r, int c, int blksz)
{
        sub->stride = m->stride;
        sub->data = m->data + m->stride*r + c;
        sub->n = blksz;
}

/* C = C + A*B */
void naive_matrix_mult_add(struct matrix *C, struct matrix *A, struct matrix *B)
{
        int i,j,k;
        int n = C->n;
        for (i = 0; i < n; i++) {
                for (j = 0; j < n; j++) {
                        int c = 0;
                        for (k = 0; k < n; k++) c += CELL(A,i,k)*CELL(B,k,j);
                        CELL(C,i,j) += c;
                }
        }
}

/* C = A*B */
void naive_matrix2_mult(struct matrix2 *C, struct matrix2 *A, struct matrix2 *B)
{
        int i,j,k;
        int n = C->n;
        for (i = 0; i < n; i++) {
                for (j = 0; j < n; j++) {
                        int c = 0;
                        for (k = 0; k < n; k++) c += CELL2(A,i,k)*CELL2(B,k,j);
                        CELL2(C,i,j) = c;
                }
```

```c
	}
}

/* C = A+B */
void matrix2_sum(struct matrix2 *C, struct matrix2 *A, struct matrix2 *B)
{
	int i,j;
	int n = A->n;
	for (i = 0; i < n; i++) {
		for (j = 0; j < n; j++) {
			CELL2(C,i,j) = CELL2(A,i,j) + CELL2(B,i,j);
		}
	}
}

/* C = A-B */
void matrix2_diff(struct matrix2 *C, struct matrix2 *A, struct matrix2 *B)
{
	int i,j;
	int n = A->n;
	for (i = 0; i < n; i++) {
		for (j = 0; j < n; j++) {
			CELL2(C,i,j) = CELL2(A,i,j) - CELL2(B,i,j);
		}
	}
}

void strassen_split(struct matrix2 *A, struct matrix2 As[2][2])
{
	int blksz = A->n/2;
	make_submatrix(A, &As[0][0], 0, 0, blksz);
	make_submatrix(A, &As[0][1], 0, blksz, blksz);
	make_submatrix(A, &As[1][0], blksz, 0, blksz);
	make_submatrix(A, &As[1][1], blksz, blksz, blksz);
}

/* C = A*B */
void strassen_matrix2_mult(struct matrix2 *C, struct matrix2 *A, struct matrix2 *B)
{
	/* if we can't divide it in 4 parts, use naive algorithm */
	if (C->n % 2) {
		naive_matrix2_mult(C, A, B);
		return;
	}

	struct matrix2 As[2][2];
	strassen_split(A, As);
	struct matrix2 Bs[2][2];
	strassen_split(B, Bs);
	struct matrix2 Cs[2][2];
	strassen_split(C, Cs);

	int blksz = A->n/2;
	int i;
	struct matrix2 M[7];
	for (i = 0; i < 7; i++) alloc_matrix2(&M[i], blksz);

	struct matrix2 tmp1, tmp2;
	alloc_matrix2(&tmp1, blksz);
	alloc_matrix2(&tmp2, blksz);

	// M1 = (A11 + A22)*(B11 + B22)
	matrix2_sum(&tmp1, &As[0][0], &As[1][1]);
	matrix2_sum(&tmp2, &Bs[0][0], &Bs[1][1]);
	strassen_matrix2_mult(&M[0], &tmp1, &tmp2);

	// M2 = (A21 + A22)*B11
```

```
        matrix2_sum(&tmp1, &As[1][0], &As[1][1]);
        strassen_matrix2_mult(&M[1], &tmp1, &Bs[0][0]);

        // M3 = A11*(B12-B22)
        matrix2_diff(&tmp1, &Bs[0][1], &Bs[1][1]);
        strassen_matrix2_mult(&M[2], &As[0][0], &tmp1);

        // M4 = A22*(B21-B11)
        matrix2_diff(&tmp1, &Bs[1][0], &Bs[0][0]);
        strassen_matrix2_mult(&M[3], &As[1][1], &tmp1);

        // M5 = (A11+A12)*B22
        matrix2_sum(&tmp1, &As[0][0], &As[0][1]);
        strassen_matrix2_mult(&M[4], &tmp1, &Bs[1][1]);

        // M6 = (A21-A11)*(B11+B12)
        matrix2_diff(&tmp1, &As[1][0], &As[0][0]);
        matrix2_sum(&tmp2, &Bs[0][0], &Bs[0][1]);
        strassen_matrix2_mult(&M[5], &tmp1, &tmp2);

        // M7 = (A12-A22)*(B21+B22)
        matrix2_diff(&tmp1, &As[0][1], &As[1][1]);
        matrix2_sum(&tmp2, &Bs[1][0], &Bs[1][1]);
        strassen_matrix2_mult(&M[6], &tmp1, &tmp2);

        // C11 = M1 + M4 - M5 + M7
        matrix2_sum(&tmp1, &M[0], &M[3]);
        matrix2_diff(&tmp2, &tmp1, &M[4]);
        matrix2_sum(&Cs[0][0], &tmp2, &M[6]);

        // C12 = M3 + M5
        matrix2_sum(&Cs[0][1], &M[2], &M[4]);

        // C21 = M2 + M4
        matrix2_sum(&Cs[1][0], &M[1], &M[3]);

        // C22 = M1 - M2 + M3 + M6
        matrix2_diff(&tmp1, &M[0], &M[1]);
        matrix2_sum(&tmp2, &tmp1, &M[2]);
        matrix2_sum(&Cs[1][1], &tmp2, &M[5]);

        for (i = 0; i < 7; i++) free(M[i].data);
        free(tmp1.data);
        free(tmp2.data);
}

/* C = A*B */
void strassen_matrix_mult(struct matrix *C, struct matrix *A, struct matrix *B)
{
        struct matrix2 A2,B2,C2;
        make_matrix2(A, &A2);
        make_matrix2(B, &B2);
        make_matrix2(C, &C2);

        struct matrix2 tmp;
        alloc_matrix2(&tmp, A->n);
        strassen_matrix2_mult(&tmp,&A2,&B2);
        matrix2_sum(&C2, &C2, &tmp);
        free(tmp.data);
}

/* C = A*B */
void strassen_matrix_mult_add(struct matrix *C, struct matrix *A, struct matrix *B)
{
        struct matrix2 A2,B2,C2;
        make_matrix2(A, &A2);
        make_matrix2(B, &B2);
        make_matrix2(C, &C2);
        strassen_matrix2_mult(&C2,&A2,&B2);
}

int parse_args(int argc, char *argv[], int *n, int *k, struct input_params *m_in)
{
        int ch;
        m_in->print = 0;
        m_in->lu2d = 0;
        m_in->strassen = 0;
        while ((ch = getopt(argc, argv, "p2s")) != -1) {
                switch (ch) {
                case 'p':
                        m_in->print = 1;
                        break;
                case '2':
                        m_in->lu2d = 1;
                        break;
                case 's':
                        m_in->strassen = 1;
                        break;
                }
        }
        argc -= optind;
        argv += optind;

        if (argc < 2) {
                fprintf(stderr, "ERROR: must specify matrix size and power");
                return 1;
        }
        *n = atoi(argv[0]);
        *k = atoi(argv[1]);

        if (argc == 4) {
                m_in->mode = 1;
                m_in->u.prob[0] = atof(argv[2]);
                m_in->u.prob[1] = atof(argv[3]);
        } else if (argc == 6) {
                m_in->mode = 2;
                m_in->u.pattern[0] = atoi(argv[2]);
                m_in->u.pattern[1] = atoi(argv[4]);
                m_in->u.pattern[2] = atoi(argv[5]);
                m_in->u.pattern[3] = atoi(argv[6]);
        } else {
                fprintf(stderr, "ERROR: Invalid number of arguments (%d).\n", argc);
                return 1;
        }

        return 0;
}


/*
  Input method 1: Probabilities of -1 and +1.
        Arguments: n p1 p2

  Input method 2: Four numbers that get repeated with a shift.
        Arguments: n x1 x2 x3 x4

*/
void fill_matrix(struct matrix *m, struct input_params *m_in)
{
        int count, i;

        count = m->n*m->n;

        // Input method 1
        if (m_in->mode == 1) {
                double pm = m_in->u.prob[0];
                double pp = m_in->u.prob[1];

                for (i = 0; i < count; i++) {
                        double x = random()/(double)RAND_MAX;
                        if (x < pm) m->data[i] = -1;
                        else if (x > 1.0 - pp) m->data[i] = 1;
                        else m->data[i] = 0;
                }
        } else {
                int shift = 0;
                int r, c;

                for (r = 0; i < m->n; r++) {
                        shift += (r+1);
                        for (c = 0; c < m->n; c++) {
                                CELL(m,r,c) = m_in->u.pattern[(shift+c) % 4];
                        }
                }
        }
}

void copy_block(int blksz, struct matrix *sm, int sr, int sc,
                                                    struct matrix *dm, int dr, int dc)
{
        int i;

        int *src = sm->data + sm->n * sr + sc;
        int *dst = dm->data + dm->n * dr + dc;

        for (i = 0; i < blksz; i++) {
                memcpy(dst, src, sizeof(*dm->data)*blksz);
                src += sm->n;
                dst += dm->n;
        }
}

/* rearrange data into blocks for scatter */
void matrix_to_blocks(struct matrix *m, int *blockdata, int blksz, int column_first)
{
        struct matrix tmp;
        int i, j, s;

        tmp.data = blockdata;
        tmp.n = blksz;
        s = m->n / blksz;
        if (column_first) {
                for (j = 0; j < s; j++) {
                        for (i = 0; i < s; i++) {
                                copy_block(blksz, m, i * blksz, j * blksz, &tmp, 0, 0);
                                tmp.data += tmp.n*tmp.n;
                        }
                }
        } else {
                for (i = 0; i < s; i++) {
                        for (j = 0; j < s; j++) {
                                copy_block(blksz, m, i * blksz, j * blksz, &tmp, 0, 0);
                                tmp.data += tmp.n*tmp.n;
                        }
                }
        }
}

void blocks_to_matrix(struct matrix *m, int *blockdata, int blksz, int column_first)
{
        struct matrix tmp;
        int i, j, s;

        tmp.data = blockdata;
        tmp.n = blksz;
        s = m->n / blksz;
        if (column_first) {
                for (j = 0; j < s; j++) {
                        for (i = 0; i < s; i++) {
                                copy_block(blksz, &tmp, 0, 0, m, i * blksz, j * blksz);
                                tmp.data += tmp.n*tmp.n;
                        }
                }
        } else {
                for (i = 0; i < s; i++) {
                        for (j = 0; j < s; j++) {
                                copy_block(blksz, &tmp, 0, 0, m, i * blksz, j * blksz);
                                tmp.data += tmp.n*tmp.n;
                        }
                }
        }
}

/* rearrange data into blocks for scatter */
void matrix_to_rowblocks_cyclic(struct matrix *m, int *blockdata, int blksz)
{
        int i, j, s;
        int *dst = blockdata;

        s = m->n / blksz;
        for (i = 0; i < blksz; i++) {
                int *src = m->data + m->n*i;
                for (j = 0; j < s; j++) {
                        memcpy(dst, src, sizeof(*dst)*m->n);
                        dst += m->n;
                        src += m->n*blksz;
                }
        }
}

void print_matrix(struct matrix *m)
{
        int i;
        int count = m->n*m->n;
        int *p = m->data;

        for (i = 1; i <= count; i++) {
                printf("% d", *p);
                if (i % m->n == 0) printf("\n");
                else printf(" ");
                p++;
        }
}

/*
   Integer computation of the log base 2 of n.
*/
int intlog2( int n ) {

        int power = 0;
        int value = 1;

        while( (value << 1) <= n ) {
                power++;
                value <<= 1;
        };

        return power;
```

```
}

/*
    Integer power of 2.
*/
int intpow2( int power ) {

        int pwr = 0;
        int value = 1;

        while( (pwr + 1) <= power ) {
                power++;
                value <<= 1;
        };

        return value;
}

int count_swaps(int *reorder_all, int n)
{
        /* to count the number of swaps performed, follow the cycles in the perm vector */
        int count = 0, i;
        for (i = 0; i < n; i++) {
                if (reorder_all[i] == -1) continue;
                int p = i, q;
                do {
                        q = p;
                        p = reorder_all[p];
                        reorder_all[q] = -1;
                        count++;
                } while (p != i);
                count--;
        }
        return count;
}
```