```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include "odd_even_sort.h"

const int RMAX = 100;



//由于vs中无法使用库中的random和srandom函数，故将库中原函数的代码写过来
#define ULONG_MAX  ((unsigned long)(~0L)) /* 0xFFFFFFFF*/
#define LONG_MAX   ((long)(ULONG_MAX >> 1))/* 0x7FFFFFFF*/
#define NULL (void *) 0
static long int RandomTable[32] = { 3,
      0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
      0xde3b81e0, 0xdf0a6fb5, 0xf103bc02, 0x48f340fb,
      0x7449e56b, 0xbeb1dbb0, 0xab5c5918, 0x946554fd,
      0x8c2e680f, 0xeb3d799f, 0xb11ee0b7, 0x2d436b86,
      0xda672e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,
      0xd7158fd6, 0x6fa6f051, 0x616e6b96, 0xac94efdc,
      0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,
      0xf5ad9d0e, 0x8999220b, 0x27fb47b9

};
static long int* fptr = &RandomTable[4];
static long int* rptr = &RandomTable[1];
static long int* state = &RandomTable[1];
static long int* end_ptr = &RandomTable[sizeof(RandomTable) /
sizeof(RandomTable[0])];
long int random()
{
    long int i;
    *fptr += *rptr;
    i = (*fptr >> 1) & LONG_MAX;
    fptr++;
    if (fptr >= end_ptr)
    {
        fptr = state;
        rptr++;
    }
```

```
    else
    {
        rptr++;
        if (rptr >= end_ptr)
            rptr = state;
    }
    return i;
}

void srandom(unsigned int seed)
{
    state[0] = seed;
    register long int i;
    for (i = 1; i < 31; ++i)
        state[i] = (1103515145 * state[i - 1]) + 1234;
    fptr = &state[3];
    rptr = &state[0];
    for (i = 0; i < 10 * 31; ++i)
        random();
}


int main(int argc, char* argv[]) {
    int my_rank, p,global_n,local_n;//local_n:各个进程中数组的大小
    char g_i;
    int* local_A;
    MPI_Comm comm;
    MPI_Init(&argc, &argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &my_rank);
    //处理终端输入的参数
    Get_args(argc, argv, &global_n, &local_n, &g_i, my_rank, p,
comm);
    local_A = (int*)malloc(local_n * sizeof(int));
    if (g_i == 'g') {
        //生成随机数
        random_list(local_A, local_n, my_rank);
        Print_Unsorted_Vector(local_A, local_n, my_rank, p, comm);
    }
    else {
        //用户从终端输入一串数字
```

```
        Get_Input(local_A, local_n, my_rank, p, comm);
    }

    Sort(local_A, local_n, my_rank, p, comm);

    Print_Sorted_Vector(local_A, local_n, my_rank, p, comm);

    free(local_A);

    MPI_Finalize();

    return 0;
}


//生成随机数，每一个进程生成local_n个随机数，存储到数组local_n中
void random_list(int local_A[], int local_n, int my_rank) {
    int i;
    srandom(my_rank + 1);
    for (i = 0; i < local_n; i++)
        local_A[i] = random() % RMAX;

}


//从终端读入global_n个整型数字
void Get_Input(int local_A[], int local_n, int my_rank, int p,
    MPI_Comm comm) {
    int i;
    int* temp=new int(1);
    if (my_rank == 0) {
        temp = (int*)malloc(p * local_n * sizeof(int));
        for (i = 0; i < p * local_n; i++)
            scanf("%d", &temp[i]);
    }
    //散射，读取向量并分发，分发到各个进程
    MPI_Scatter(temp, local_n, MPI_INT, local_A, local_n,
MPI_INT,0, comm);

    if (my_rank == 0)
        //释放内存
        free(temp);
```

```
}

//比较两个数大小的函数,用于快速排序的函数指针
int cmp(const void* a_p, const void* b_p) {
    int a = *((int*)a_p);
    int b = *((int*)b_p);
    if (a < b)
        return -1;
    else if (a == b)
        return 0;
    else
        return 1;
}

//排序函数，对local list进行排序，使用奇偶排序对global list进行排序
void Sort(int local_A[], int local_n, int my_rank,int p, MPI_Comm
comm) {
    int phase;
    int* temp1, * temp2;
    int even_partner;
    int odd_partner;
    temp1 = (int*)malloc(local_n * sizeof(int));
    temp2 = (int*)malloc(local_n * sizeof(int));
    //获取某个阶段，某个进程的通信伙伴
    if (my_rank % 2 != 0) {
        //奇通信阶段，奇数为通信双方的较小进程
        even_partner = my_rank - 1;
        odd_partner = my_rank + 1;
        if (odd_partner == p)
            odd_partner = MPI_PROC_NULL;
    }
    else {
        //偶通信阶段，偶数为通信双方的较小进程
        even_partner = my_rank + 1;
        if (even_partner == p)
            even_partner = MPI_PROC_NULL;
        odd_partner = my_rank - 1;
    }

    //用内置的qsort函数对local list进行快速排序
    qsort(local_A, local_n, sizeof(int), cmp);
    //定理:如果p个进程运行奇偶排序算法，那么p个阶段后，输入列表就有序
```

```
    for (phase = 0; phase < p; phase++)
        Odd_even_iteration(local_A, temp1, temp2, local_n,
phase,even_partner, odd_partner, my_rank, p, comm);
    //释放内存
    free(temp1);
    free(temp2);
}

//一次奇偶交换排序的迭代函数
void Odd_even_iteration(int local_A[], int temp1[], int temp2[],int
local_n, int phase, int even_partner, int odd_partner,int my_rank,
int p, MPI_Comm comm) {
    MPI_Status status;
    if (phase % 2 == 0) {
        //even phase
        if (even_partner >= 0) {
            //为了保证MPI程序的安全性，采用MPI自己提供的调度通信的办法
MPI_Sendrecv
            //与对方进程交换数据
            MPI_Sendrecv(local_A, local_n, MPI_INT, even_partner,
0,temp1, local_n, MPI_INT, even_partner, 0, comm,&status);
            if (my_rank % 2 != 0)
                Merge_high(local_A, temp1, temp2, local_n);
            else
                Merge_low(local_A, temp1, temp2, local_n);
        }
    }
    else {
        //odd phase
        if (odd_partner >= 0) {
            MPI_Sendrecv(local_A, local_n, MPI_INT, odd_partner,
0,temp1, local_n, MPI_INT, odd_partner, 0, comm, &status);
            if (my_rank % 2 != 0)
                Merge_low(local_A, temp1, temp2, local_n);
            else
                Merge_high(local_A, temp1, temp2, local_n);
        }
    }
}



//合并两个进程的数据，并取较小的一半数据
```

```
void Merge_low(int  my_keys[],int recv_keys[],int temp_keys[],int
local_n) {
    int m_i, r_i, t_i;
    m_i = r_i = t_i = 0;
    while (t_i < local_n) {
        if (my_keys[m_i] <= recv_keys[r_i]) {
            temp_keys[t_i++] = my_keys[m_i++];
        }
        else {
            temp_keys[t_i++] = recv_keys[r_i++];
        }
    }

    memcpy(my_keys, temp_keys, local_n * sizeof(int));
}

//合并两个进程的数据，并取较大的一半数据
void Merge_high(int local_A[], int temp1[], int temp2[],int
local_n) {
    int a_i, b_i, c_i;
    a_i = local_n - 1;
    b_i = local_n - 1;
    c_i = local_n - 1;
    while (c_i >= 0) {
        if (local_A[a_i] >= temp1[b_i]) {
            temp2[c_i--] = local_A[a_i--];
        }
        else {
            temp2[c_i--] = temp1[b_i--];
        }
    }

    memcpy(local_A, temp2, local_n * sizeof(int));
}


//打印分发给每个进程的数据，即local_A，排序以前的
void Print_list(int local_A[], int local_n, int rank) {
    int i;
    printf("process %d's data:  ", rank);
    for (i = 0; i < local_n; i++)
        printf("%d ", local_A[i]);
```

```
    printf("\n");

}

//打印所有进程的数据，排序以前的
void Print_Unsorted_Vector(int local_A[], int local_n, int my_rank,
int p, MPI_Comm comm) {
    int* A;
    int q;
    MPI_Status status;
    if (my_rank == 0) {
        A = (int*)malloc(local_n * sizeof(int));
        printf("\n排序前的数组元组:\n");
        Print_list(local_A, local_n, my_rank);
        for (q = 1; q < p; q++) {
            MPI_Recv(A, local_n, MPI_INT, q, 0, comm, &status);
            Print_list(A, local_n, q);
        }
        free(A);
    }
    else {
        MPI_Send(local_A, local_n, MPI_INT, 0, 0, comm);
    }
}



//打印排序后的数组元素
void Print_Sorted_Vector(int local_A[], int local_n, int my_rank,
int p, MPI_Comm comm) {
    //定义和初始化
    int* temp = new int(1);
    int i, n;

    if (my_rank == 0) {
        n = p * local_n;
        temp = (int*)malloc(n * sizeof(int));
        MPI_Gather(local_A, local_n, MPI_INT, temp, local_n,
MPI_INT, 0, comm);

 printf("\n**********************************************************
**************\n");
```

```
        printf("排序后的数组元素:\n");
        for (i = 0; i < n; i++)
            printf("%d ", temp[i]);
        free(temp);
        printf("\n");

    }
    else {
        MPI_Gather(local_A, local_n, MPI_INT, temp, local_n,
MPI_INT, 0, comm);
    }

}

//程序用法，需要注意的是，n要被p整除，即global_n个数需被进程平分，分别求和
void Usage(char* program) {
    fprintf(stderr, "请按以下格式输入:\nmpiexec -n <进程个数> <可执行文件
名称> <g|i> <数组元素个数>\n\n请注意，数组元素个数能够整除进程个数以便分发
\n\n", program);
    fflush(stderr);
}


//处理命令行参数
void Get_args(int argc, char* argv[], int* global_n_p, int*
local_n_p,char* gi_p, int my_rank, int p, MPI_Comm comm) {

    if (my_rank == 0) {
        if (argc != 3) {
            Usage(argv[0]);
            *global_n_p = -1;  //退出
        }
        else {
            *gi_p = argv[1][0];
            if (*gi_p != 'g' && *gi_p != 'i') {
                Usage(argv[0]);
                *global_n_p = -1; //退出
            }
            else {
                *global_n_p = atoi(argv[2]);
                if (*global_n_p % p != 0) {
                    Usage(argv[0]);
```

```
                    *global_n_p = -1;
                }
            }
        }
    }

    MPI_Bcast(gi_p, 1, MPI_CHAR, 0, comm);
    MPI_Bcast(global_n_p, 1, MPI_INT, 0, comm);

    if (*global_n_p <= 0) {
        MPI_Finalize();
        exit(-1);
    }
    //平均分给各个进程
    *local_n_p = *global_n_p / p;
}
```

odd_even_sort.h

```
#pragma once
void Usage(char* program);
void Print_list(int local_A[], int local_n, int rank);
void Merge_low(int local_A[], int temp1[], int temp2[],int
local_n);
void Merge_high(int local_A[], int temp1[], int temp2[],int
local_n);
void random_list(int local_A[], int local_n, int my_rank);
int  Compare(const void* a_p, const void* b_p);
void Get_args(int argc, char* argv[], int* global_n_p, int*
local_n_p,char* gi_p, int my_rank, int p, MPI_Comm comm);
void Sort(int local_A[], int local_n, int my_rank,int p, MPI_Comm
comm);
void Odd_even_iteration(int local_A[], int temp_B[], int
temp_C[],int local_n, int phase, int even_partner, int odd_partner,
int my_rank, int p, MPI_Comm comm);
void Print_Unsorted_Vector(int local_A[], int local_n,int my_rank,
int p, MPI_Comm comm);
void Print_Sorted_Vector(int local_A[], int local_n, int
my_rank,int p, MPI_Comm comm);
void Get_Input(int local_A[], int local_n, int my_rank, int p,
MPI_Comm comm);
```