PARALLEL PROGRAMMING LAB
TASK ONE

ΑΝΤΙΒΑΧΗΣ ΑΝΤΩΝΙΟΣ

To compile and run...

This project makes use of a makefile. Simply run `make`, in the project root, to compile and link the source code into a binary named `arccheck`.

After that, run it with

```
mpirun -n N <--use-hwthread-cpus> arccheck
```

Where N is the desired number of processors. If it is desired, we can also use the flag in the brackets <>, to specify a higher core count, by utilizing the cpu threads.

To remove build/binary files, run make clean.

Attention

1. This is a Linux build, and will not work on Windows operating systems.
2. When running the program, and entering invalid input, like characters, they will be ignored. However the terminal will not be cleared - this is due to a bug I could not resolve in good time. It does not affect program functionality, however, and input works correctly.

Further information

This application utilizes the openmpi implementation of the MPI standard.

TASK 1 - Input Reading

To read input from stdin, the following function is used

```
void read_int(int* target){
    int a;
    char buf[1024]; // use 1KiB just to be sure
    int success; // flag for successful conversion

    int success = 0
    do
    {
        if (!fgets(buf, 1024, stdin))
        {
            // reading input failed:
            MPI_Abort(MPI_COMM_WORLD, 1);
        }

        // have some input, convert it to integer:
        char *endptr;

        errno = 0; // reset error number
        a = strtol(buf, &endptr, 10);
        ...
    }while(!success)
```

We will use about 1KiB of memory on the stack, and assign to it input from stdin. If for any reason this returns NULL, the program aborts with MPI_abort(...), and all processes are terminated.

Otherwise, our input is converted to a base 10 int. After that, and this is ommited from this document for the sake of space, we will check for overflow, input beyond our assigned stack space, or any other false input. Please see the source code for details.

Menu

This function is then called through the menu() function, in process 0. See lines 18-39 in `main.c`

The only way to terminate the program is by inputing 2, or aborting with control-C

TASK 2 - Sort check

```
for (int i = proc_id - 1; i < n - 1; i += (proc_c - 1)) {
    if (arr[i] > arr[i+1]) {
        sorted = i+1;
        break;
    }
}
```

Because we only want processes 1 - N to do the actual logic, while 0 reads input from the user, we subtract from proc_id, to correctly index from 0. The limit is up to n - 1 elements, so that the if statement does not index outside the array.

Finally, by adding to the index with processes available to us for indexing, (proc_c - 1), we ensure that each process receives enough 'pairs' of numbers, so as to evenly distribute across any number of processes.

Inevitably however, if more processes than numbers exist, some will terminate without doing anything.

If sorting breaks at any point, the process will break out of the loop, and send a non-zero int to process rank 0 (index where sorting breaks), to signify a fail.

TASK 3 - Reporting

If process 0 receives any return greater than 0 from the other processes, than that will signify a break in the sorting, and the index at which that happens. So if that happens at indexes 0 and 1, the return will be 1.

```
for (int i = 1; i < proc_c; i++){
    MPI_Recv(&res, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

    if (res > 0) { returns[count] = res; count++; }
}
```

Because we dont know which process will terminate first, and there might be more than one point of failure, we should store the results in an array, and pick the smallest result

Visuals.

In order to more clearly communicate results with the user, the following function was implemented.

```
void display_arr(int* arr, int count, int term_index, int
num_or_comma){
    putchar('[');
    for (int i = 0; i < count; i++) {
        if (i == count-1) {
            printf("%d",arr[i]);
            break;
        }
        printf("%d,",arr[i]);
    }
    putchar(']');

    if (term_index != -1) {
        putchar('\n');
        if (num_or_comma == 0) putchar(' ');

        //calculate required white space
        int sum = 0;
        for (int i = 0; i < term_index; i++){
            sum += int_len(arr[i])+1;
        }

        for (int i = 0; i < sum; i++) {
            printf(" ");
        }
        putchar('^');
    }
    putchar('\n');
}
```

This function takes a pointer to an array, it's length and two more parameters, term_index and num_or_comma.

The first specifies where the index character (^) should be, with a negative input as a disable.

The second, specifies wether to point between numbers, or on numbers.

Expected output

```
// Clear terminal
Choose action:
1) Check list
2) Abort

// Clear terminal
Give size of list : 5

// Clear terminal
Numbers left: 5
[0,0,0,0,0]
 ^
4

// Clear terminal
Numbers left: 4
[4,0,0,0,0]
   ^
3

// Clear terminal
Numbers left: 3
[4,3,0,0,0]
     ^
2

// Clear terminal
Numbers left: 2
[4,3,2,1,0]
       ^
1

// Clear terminal
Numbers left: 1
[4,3,2,1,0]
         ^
0

// Clear terminal
```

```
[4,3,2,1,0]
  ^
Ascending order breaks on these indexes: 0-1
Press enter to continue...
```

This will loop indefinately, until the user chooses to abort.