ECE26400 Programming Assignment #3

This assignment is related to PE05 and PE06. We continue to build on what you did in PE05 and PE06. In particular, you will now have to generate a tiling solution when you are provided a valid floor length and row-column coordinates of the 4-tile.

In addition to those outlined in PE05 and PE06, the main learning goal is:

1. How to generate a tiling solution using a divide-and-conquer approach.

1 Getting started

You should unzip on eceprog.ecn.purdue.edu the zip file `pa03_files.zip` using the following command:

```
unzip pa03_files.zip
```

The zip file `pa03_files.zip` contains a folder, named `PA03`, and one file within the folder:

1. `answer03.h`: This is a "header" file and it declares the functions you will be writing for this assignment.

In this assignment, you have to create a file called `answer03.c` to define all functions declared in `answer03.h` and a file called `pa03.c` for the `main` function.

To get started, read this document in its entirety. We assume that you are familiar with the tiling problem that we want you to solve (see the description for PE05). We also assume that you are familiar with the functions that you have written in `answer06.c` for PE06.

2 Functions you have to define

You are going to reuse all the functions that you have written in `answer06.c` for PE06. Please refer to the PE06 description for the details of those functions. There is only one new function that you have to write in `answer03.c`.

2.1 Functions to be written in `answer03.c`

We will continue to use the `tiling_solution` structure in this assignment.

```
typedef struct _tiling_solution {
   int floor_length;      // length of square floor, power of two >= 1
   int row, column;       // valid row and column coordinates
   char **floor_content;  // address of 2-dimensional array for floor content
} tiling_solution;
```

This is how we will use the structure in this assignment. We will declare a variable of type `tiling_solution` in the `main` function. The `main` function will process the arguments passed to the executable to fill in valid fields `floor_length`, `row`, and `column` in the variable of type `tiling_solution`. The address of this variable will be passed to the three of the following five functions that you have to define in `answer03.c`. (The prototypes of these five functions are in `answer03.h`.)

```
char **allocate_2d_array(int floor_length);

void free_2d_array(char **array, int floor_length);

int determine_tiling_solution_category(char *filename, tiling_solution *ts);

int save_tiling_solution(char *filename, tiling_solution *ts);

void generate_tiling_solution(tiling_solution *ts);
```

The first four functions should have been completed in PE06. So, you only have to work on the `generate_tiling_solution` function. Given the floor length and the row-column coordinates of the 4-tile in `*ts`, the function should allocate space for a 2-dimensional array, generate the tiling solution, and store the tiling solution in the 2-dimensional array. We expect the 2-dimensional array to store only combinations of `'0'`, `'1'`, `'2'`, `'3'`, and `'4'`. The highest-level address of the 2-dimensional array should be saved in the `floor_content` field of `*ts`.

If you do not have a way of generating a tiling solution, it is recommended that you assign `NULL` to the `floor_content` field of `*ts` to avoid memory issues when we run your executable(s). The caller function will use the `floor_content` field to detect the presence of a tiling solution.

Please see Section 4 on how you may write this function.

2.2 `main` function in `pa03.c`

The purpose of the executable `pe06` is fairly straightforward: It takes in five arguments in the following order that correspond to respectively:

1. mode of executable
2. length of square floor
3. row coordinate of the 4-tile
4. column coordinate of the 4-tile
5. name of the (input or output) file

There are two modes of operation: `"-v"` and `"-g"`. The `"-v"` mode stands for validation of a tiling solution and the `"-g"` mode stands for the generation of a tiling solution. You will rely on the functions written in PE06 for the `"-v"` mode. The `"-g"` mode relies on the generation function you have to write in this assignment.

Look at PE03, PE04, and PA02 on how you can convert the floor length, and row and column coordinates from the relevant `argv[i]`.

If these converted numbers are valid, the `main` function treats the next argument as an input file if it is in the `"-v"` mode. If the mode is `"-g"`, it treats the next argument as an output file.

For the `"-v"` mode, the `main` function determines the category of the input file. If the `main` function reaches here, the function should complete the tasks outlined in the next paragraph and return `EXIT_SUCCESS` at the end. Otherwise, the function should return `EXIT_FAILURE` immediately.

To determine the category of the input file, the `main` function should declare a variable `ts`, for example, of type `tiling_solution`, and initialize the fields `floor_length`, `row`, and `column` properly. The name of the input file and the address of `ts` should be used to determine the category of the input file. The category of the input file should be printed to `stdout` with the format `"%d\n"`. Be aware that for a valid solution, the `ts.floor_content` contains a valid 2-dimensional array. You should free the memory before you return from the `main` function.

For the `"-g"` mode, the `main` function should generate a tiling solution and save the tiling solution to the output file. The `main` function should return `EXIT_SUCCESS` only if the solution can be generated and be saved to the output file. If the solution cannot be generated (because of invalid floor length, row-column coordinates, or lack of memory) or the tiling solution cannot be written to the output file, the `main` function should return `EXIT_FAILURE`. You should also free the memory before you return from the `main` function. In the `"-g"` mode, nothing should be printed to the `stdout`.

When you complete this assignment, this executable allows you to generate a solution and to evaluate the generated solution. In other words, you can test your solution (program).

2.3 Printing, helper functions and macros

All debugging, error, or log statements in `answer03.c` and `pa03.c` should be printed to `stderr`.

You may define your own helper functions in `pa03.c` and `answer03.c`. All these functions should be declared and defined as static functions. In other words, the scope of these functions are only within the files that they are found. Declaring and defining these functions as static eliminate the conflicts in function names.

You should not use macros that start with `T_` (Upper case T and underscore) in their names. We will use such macros in the evaluation of your submissions. If you use such macros, we may not be able to evaluate your submission properly.

3 Compiling your program

We use the same flags introduced in PE01 to compile the program for debugging purpose.

```
gcc -std=c99 -Wall -Wshadow -Wvla -Werror -g -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pa03.c answer03.c -o pa03
```

When we evaluate your program, we also use the optimization flag `-O3` (uppercase letter `O` and number three) instead of the `-g` flag as follows:

```
gcc -std=c99 -Wall -Wshadow -Wvla -Werror -O3 -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pa03.c answer03.c -o pa03
```

You are recommended to copy the `Makefile` from PE01 and modify it appropriately for PA03.

4 Writing, running and testing your program

In addition to the test cases that we have provided in PE05 and PE06, you can generate more test cases from your generation function.

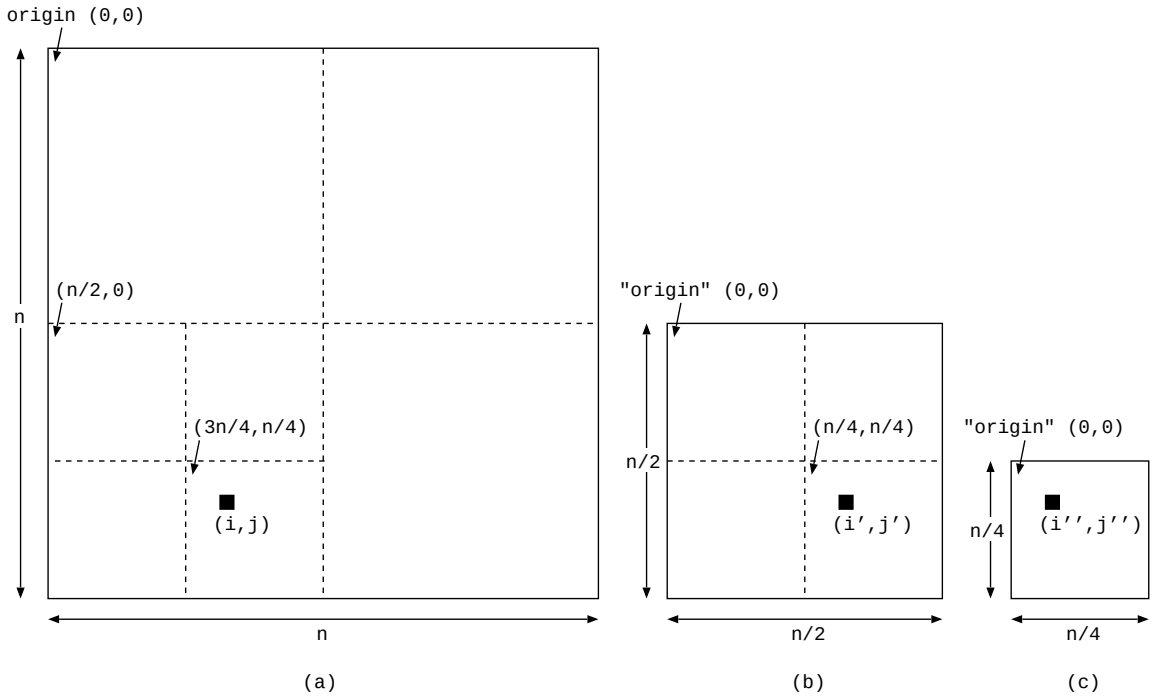
You should also run `./pa03` with appropriate arguments under `valgrind`.

Please see PE01 description about `valgrind`.

Now, we will elaborate on a strategy that you may use to develop your generation function. It is best to record your solution as you go. Therefore, the presence of a 2-dimensional array of the correct size would be useful throughout the entire process. In PE05, we mentioned the division of a $n \times n$ problem into four $n/2 \times n/2$ problems and use the solutions of the four $n/2 \times n/2$ problems as the solution of the $n \times n$ problem.

For ease of programming and testing, we instead suggest that you take the approach of dividing a $n \times n$ problem into four $n/2 \times n/2$ problem, and "solve" a particular $n/2 \times n/2$ problem. After you have written and tested this version of the function, you proceed to generalize the function to handle all four $n/2 \times n/2$ problems.

The following figure shows a $n \times n$ problem with a 4-tile depicted by the black square. As shown in (a), there are a few parameters that are useful in specifying the problem: The origin of the $n \times n$ problem has row-column coordinates of $(0,0)$, the length of the floor is n, and the row-column coordinates of the 4-tile is (i, j). Be aware that we are using row-column coordinates here, not the usual (x,y) coordinates defined using a horizontal x-axis and a vertical y-axis.



Now, if we divide the $n \times n$ floor into four $n/2 \times n/2$ floors, and we want to solve the tiling problem of the particular $n/2 \times n/2$ floor that contains the 4-tile, how can we specify the problem in (b)? If we were to assume that this $n/2 \times n/2$ floor is the only floor that exists, we can again use the parameters of "origin" $(0,0)$, floor length $n/2$, and coordinates of the 4-tile (i', j') to specify the problem.

How do you derive (i', j') from (i, j) from the original problem? For this particular example,

$$i' = i - n/2,$$

and

$$j' = j - 0.$$

The row-column offsets of $(n/2,0)$ allow us to obtain the row-column coordinates of any location in this $n/2 \times n/2$ problem from the coordinates in the original problem.

In fact, you will realize that the offsets of $(n/2,0)$ are actually the row-column coordinates of the top-left location of the $n/2 \times n/2$ floor under consideration in the original floor. In (a), we also show the top-left row-column coordinates of the $n/2 \times n/2$ floor that contains the 4-tile, as well as the top-left row-column coordinates of the $n/4 \times n/4$ floor that contains the 4-tile. The coordinates are $(n/2,0)$ and $(3n/4,n/4)$, respectively.

The coordinates of $(3n/4,n/4)$ in the original $n \times n$ floor becomes $(3n/4-n/2,n/4-0) = (n/4,n/4)$ in the $n/2 \times n/2$ floor that contains the 4-tile.

These should also inform you that given the row-column coordinates of a tile in the smaller floor, you could find out the row-column coordinates of a tile in the larger original floor.

Now, if we consider the $n/4 \times n/4$ floor that contains the 4-tile, the offsets from the original floor are $(3n/4,n/4)$, which are the coordinates of the top-left corner of the the $n/4 \times n/4$ floor under consideration in the original floor.

Therefore, the key to mapping the coordinates of a tile in a smaller floor to the coordinates of the same tile in the original floor, and vice versa, is to figure out the coordinates of top-left corner of the smaller floor in the original floor.

It is most likely that you have to write helper functions for the generation function. You will have to decide what parameters should be passed to a helper function. Go through the figure given and determine what parameters should be passed to a function and how you should compute those parameters to be passed to a function.

If you do this part correctly, you should be able to reach a 1×1 floor that contains the 4-tile, and the offsets of this 1×1 floor are the row-column coordinates of the 4-tile in the original floor.

4.1 Divide-and-conquer

With this, you are ready to solve the $n \times n$ tiling problem: Given a $n \times$n floor with $n = 2^k$, where $k \geq 0$, and a particular location of the floor is occupied by a 1×1 tile, cover the rest of the floor with L-shape tiles with 3-unit squares in different orientations without overlapping tiles.

The recommended approach is to transform the $n \times n$ floor with a location occupied by a 1×1 tile into four smaller $n/2 \times n/2$ tiling problems, use the same function to solve each of the four $n/2 \times n/2$ tiling problems independently, and use these individual solutions together to form the overall solution of the original problem.

In the bigger $n \times n$ tiling problem, one location is occupied with a unit-square tile. If you want to use the same function to solve a smaller $n/2 \times n/2$ tiling problem, each smaller $n/2 \times n/2$ floor must also have a location occupied with a unit-square tile.

From the preceding description, we know that one such smaller $n/2 \times n/2$ tiling problem is naturally defined: a $n/2 \times n/2$ floor with a location filled with a unit-square 4-tile.

For the remaining three smaller $n/2 \times n/2$ tiling problems, we have to fill in each $n/2 \times n/2$ floor, a location with a unit-square tile. However, we are allowed to use only one 4-tile in the bigger $n \times n$ tiling problem. What should you use to create three smaller $n/2 \times n/2$ tiling problems?

5 Submission

You must submit a zip file called `PA03.zip`, which contains two files:

1. `answer03.c`
2. `pa03.c`

Assuming that you are in the folder that contains `answer03.c` and `pa03.c`, use the following command to zip your files:

```
zip PA03.zip answer03.c pa03.c
```

Make sure that you name the zip file as `PA03.zip`. Moreover, the zip file should not contain any folders. Submit `PA03.zip` through Brightspace. You may submit as many versions as you like. Brightspace will keep only the most recent submission, and we will grade only the final submission.

6 Grading

All debugging messages should be printed to `stderr`. In the `"-v"` mode, we expect only a single `int` to be printed to `stdout`. In the `"-g"` mode, we expect nothing to be printed to `stdout`.

It is important that if the instructor has a working version of `pa03.c`, it should be compilable with your `answer03.c` to produce an executable. For evaluation purpose, we will use different combinations of your submitted `.c` files and our `.c` files to generate different executables. If a particular combination does not allow an executable to be generated, you do not get any credit for the function(s) that the executable is supposed to evaluate. We use both `-g` and `-O3` flags (separately) to compile your submission. Your submission is evaluated only when compilation using each of the flags is successful.

It is important to note that the `.c` files from the instructor do not assume the presence of global variables that are declared by the students. Of course, the `.c` files from the instructor may use global variables that are in C libraries, such as `errno`, `stdout`, `stderr`, and so on. If your submission contains global variables that are declared by you, it is unlikely that your executable will work correctly.

Be aware that we set a time-limit for each test case based on the size of the test case. If your program does not complete its execution before the time limit for a test case, it is deemed to have failed the test case. We will not announce the time-limits that we will use. They will be very reasonable.

The `allocate_2d_array` function and the `free_2d_array` function together account for 10%. The `determine_tiling_solution_category` function accounts for 20%. The `save_tiling_solution` function accounts for 10%. The `generate_tiling_solution` function accounts for 50%. The `main` function accounts for 10%.

The occurrence of any memory issues (memory errors or memory leaks flagged in a `valgrind` report) will result in 50-point penalty.

7 A few points to remember

All debugging messages should be printed to `stderr`. Other than the category of the input file, you should not be printing anything else to `stdout`. If the output of your program is not as expected, you get 0 for that test case.

You can declare and define additional static functions that you have to use in `pa03.c` and `answer03.c`.

You should not use macros that start with `T_`.

Grading of programming exercises and assignments is performed on machines with similar setup as eceprog.ecn.purdue.edu. You should perform testing of your work on eceprog.ecn.purdue.edu before submission. Correct output on your computer does not translate into correct output on eceprog.ecn.purdue.edu.