Technische Universität München

# Informatics I for MSE

**MSE** IN8011
Prof. Bjoern Menze, Giles Tetteh
Philipp Kaeß

**WS 19/20**
Homework 6

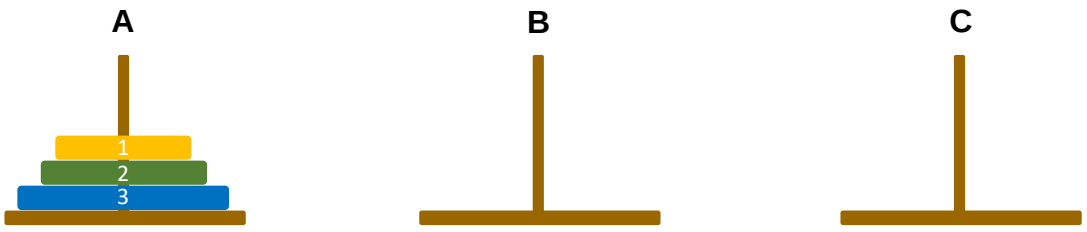
**Due Date:** 22.12.19, 23:59

**Release Date:** 13.12.19

## Homework 7: Towers of Hanoi

**Topics:** Recursion, Dynamic Memory Allocation, Pointers, Game Logic

**Introduction**
"Towers of Hanoi" is one of the most famous mathematical games of modern time. The game setup is very basic: It consists of three rods and $n$ disks of different diameter, which can slide onto any rod. Let us look at an example setup of the Towers of Hanoi for $n = 3$:




The objective of the game is to move all the disks from the initial position (all disks on rod A, in ascending order of diameter from top to bottom) to the rod on the very right, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack (or an empty rod).
3. No larger disk can be placed on top of a smaller disk.

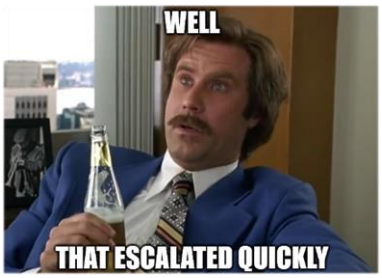
It turns out that this is possible of any positive number of disks $n$. Your task will be to create an algorithm to deliver the ideal solution path for any $n$. Therefore, we will take a closer look to possible approaches, assuming the disks are numbered from 1 to $n$:

**For $n = 1$** we get the most trivial case: We can simply put disk 1 from our source rod $A$ to the target rod $C$, and do not need our auxiliary rod $B$. (Path: $A \rightarrow C$)

**For $n = 2$** it is still simple: As we can't put disk 2 to the target rod immediately, we will first have to move disk 1 onto the auxiliary rod $B$. Then we can move disk 2 from $A$ to $C$ and finish by putting disk 1 from $B$ to $C$. (Path: $A \rightarrow B, A \rightarrow C, B \rightarrow C$)

**For $n = 3$**, things get a little more complex. We would like to move disk 3 to our target rod $C$ (as the largest disk should be the lowest disk on $C$), but we can't because disks 1 and 2 are still on top of 3. We will first have to move disk 1 and 2 to our auxiliary rod $B$, then put disk 3 from $A$ to $C$, and then once again move the disks 1 and 2 to the final target $C$. Luckily, we already know how to move two disks from the above $n = 2$ case. The only difference is, that our target rod for the first $n = 2$ operation is not $C$, but $B$, with $C$ being the auxiliary rod. And the second $n = 2$ operation is moving two disks from source rod $B$ to target rod $C$ by the help of rod $A$. (Path: $\underbrace{A \to C, A \to B, C \to B,}_{\text{Move 2 disks from A to B}}$ $\underbrace{A \to C,}_{\text{Move lowest disk from A to C}}$ $\underbrace{B \to A, B \to C, A \to C}_{\text{Move 2 disks from B to C}}$)

This algorithm can be extended for higher $n$s. In theory, we are able to compute the ideal path for every positive $n$ with that algorithm. In practice it turns out, that the number of moves rises exponentially. In fact, the minimum number of moves is $2^n - 1$. If you need one second per move, you will need seven seconds to solve for $n = 3$, about 17 minutes for $n = 10$, 12 days for $n = 20$ and 35 million years for $n = 50$.




When we compute the path for some $n$, we want to store this path in an array, to be able to print it or to compare it to the solution. But in fact, we don't know how large this array has to be, as it depends on the input $n$ and we can't declare an array with a variable as length (e.g. `char array[n]`). This is caused by the fact that static memory is allocated before the run of the main, and even though the value of n may be theoretically determined, the compiler does not know the value of $n$ yet. We will have to allocate the memory dynamically during runtime.

To save memory, we use the following convention: The rods are identified by a single `char` (just as `'A'`, `'B'` and `'C'` above) and a move is stored as two consecutive `char`s in the trace array. E.g. for $n = 3$, source rod = `'A'`, target rod = `'C'`, helper rod = `'B'` the trace array should be `"ACABCBACBABCAC"`. With that convention we will need a char array of size $2 \cdot (2^n - 1)$, but as we would like to NULL-terminate our array and have a little bit of security buffer, we will allocate an array of size $2 \cdot 2^n = 2^{n+1}$. This allocated array can then be filled with the necessary game moves.

**Specifications**

The function `char *allocate_trace(int disks)` with the input parameter

- `disks` , the number of disks for the upcoming game

should dynamically allocate a char array of size $2^{disks+1}$, **initialize all elements to 0** and return a pointer to the newly allocated array.

The function `void move(int disks, char source, char target, char helper, char *trace)` with the input parameters

- `disks` , the number of disks
- `source` , the name of the source rod as char
- `target` , the name of the target rod as char
- `helper` , the name of the auxiliary rod as char
- `trace` , the pointer to the allocated char array

should **recursively** simulate the ideal solution for the “Towers of Hanoi” and write the moves into the `trace` array according to the above convention with the help of the provided function `write()` (description see below).

You may always assume that your functions will get valid, true positive inputs.

**Tasks**

You are provided with the template file ***hanoi.c*** on Moodle. Complete the functions `allocate_trace()` and `move()`, so that the program works as specified above. Please note:

- In this template file we predefined the function `write(char *array, char source, char target)`. You can use this function to write a step into the `trace` array. Therefore, simply call the function with the pointer to the `trace` array, and the respective source and target rod names. The function will then automatically take care of the right index to store the two chars in the array. Simply ignore the return type and statement, they are only used for correction. You can treat this function as if it was a void function.

- You are only allowed to write code in the spots indicated in the functions. Do not change anything else. Also do not include or remove any libraries or headers.

- Do not print anything

- You are provided with additional files:
  - ***main_hanoi.c*** is the main file we provide you to test your functions. Do not change anything in this file. The main will first ask the user to input the number of disks, then call your function `allocate_trace()` to allocate the needed memory. The returned pointer will then be passed on to the function `move()`, together with the number of disks and the names `'A'`, `'C'` and `'B'` for source, target and helper rod. After the run of your functions, the array is printed in a nice format and the allocated memory is freed (**so you don’t need to free the memory**). Please check by yourself if the printed trace is the correct and ideal solution.
  - ***hanoi.h*** is the header to the template file ***hanoi.c***. It is included in ***hanoi.c*** as well as in ***main_hanoi.c***.

- To run your functions, compile the two C-Files together (`gcc -Wall -Werror main_hanoi.c hanoi.c`).

- When you are finished, put ***hanoi.c*** directly in a zip-File named “HW7_Mustermann_Max” (with your name of course) and upload this file to Moodle. Do not upload ***hanoi.h*** or ***main_hanoi.c***, and do not put any additional folders into the zip-File.

- Fun fact: For some versions of C it actually **is** possible to define arrays with variables as size. We strongly recommend not to use this tool at all, as it will only confuse you, and in this specific case will lead to accessing and writing on unreserved memory (as the array would be freed automatically directly after the end of `allocate_trace()`)

**Example Output**

After you compiled and executed the main together with your functions, your terminal could look like this:

```
Number of disks: 3
Testing allocate_trace...
Calculating trace array...
Calculated trace of your program:
A -> C (Move No. 1)
A -> B (Move No. 2)
C -> B (Move No. 3)
A -> C (Move No. 4)
B -> A (Move No. 5)
B -> C (Move No. 6)
A -> C (Move No. 7)

Please check by yourself whether these steps yield the shortest possible solution.
```

**Tool Box**

The explanations below **may** help you to solve this problem:

**RECURSION**

A function is called recursive iff it calls itself. Usually a recursive function consists of two parts, a recursive case and a base case. We will try to understand this at the example of the faculty function. We want to calculate the faculty of e.g. $5$. Well, we don't know what exactly the value is, but if we knew the faculty of $4$, then we could multiply it by $5$ and get the solution.

In general: The faculty of $n$ is $n$ times the faculty of $n-1$. It then is enough to know the value of the faculty of one case, the base case, which is in this case $n=1$. The function could then look like this:

```
int faculty(int n){
        if(n==1){return 1;} //base case
        else{return n*faculty(n-1);} //recursive case
}
```

**DYNAMIC MEMORY ALLOCATION**

To dynamically allocate memory, you may need one of the following functions:

- `malloc()`
- `calloc()`
- `realloc()`

To slowly introduce you to the sad but true reality that a large part of coding is actually looking for documentation and googling, we do not provide any information about these functions here, but encourage you to investigate by yourself.

**Threshold**

To pass this homework, your program has to fulfill the following conditions:

- Your file ***hanoi.c*** must be compilable and runnable together with the main file without any errors and with the flags -Wall -Werror enabled
  (`gcc -Wall -Werror -o test main_hanoi.c hanoi.c`)

- After the run of your functions, the ideal solution path has to be stored according to the above defined conventions in the correctly allocated and initialized trace array.

- Your functions will be tested on a total of 25 cases. (5 cases for correct allocation and initialization, and 20 cases for correct solution path (n from 1 to 20). To pass this assignment, you have to pass at least 80% (=20) of the test cases.

## Information on Plagiarism and Copyright

As already announced in the lecture and the tutorials, you are supposed to do this homework alone. This homework is an official part of the exam at the end of this semester, and therefore fraud or plagiarism will be pursued and punished.

You are encouraged to discuss **specific** problems with other students (like e.g. “I don’t know how to print a string, can you help me?”), but you are forbidden to share ideas on the approach/structure of your functions, and of course to show other students your solution. This will result in a similar control-flow and structure of your submissions, which we will be able to detect even if you change variable names or loops etc.

We will compare your submissions with a code-optimized plagiarism scanner and if we find plagiarized code, you will fail this submission. If you are found guilty of plagiarism twice, you will lose the homework bonus, and in severe cases we will take further steps.

**This homework is intellectual property of the TUM, namely the Chair of Image-Based Biomedical Computing. Publishing either this homework task itself or your solutions to it in any form (including on the MSE-Cloud) is strictly forbidden and can be pursued TUM-internally as well as legally.**