DESIGN

PSEUDOCODE FOR TOWER.C:

Include statements:

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include "stack.h"

Function:-

calc_min() -> calculate minimum moves

recursive_exec() -> Main recursive exection

execute_iteration() -> called by the stack function where iteration takes place

stack_execute() -> Main stack function which calls execute_iteration as per cases provided

main() -> main function which has input and calls all other functions as per input provided

---

Pseudocode:

BRIEF - # The following function takes in the total number of disks i.e the capacity (n) and calculated the minimum number of moves. It does so by simply multiplying 2 to itself by the number of times of n given to obtain 2^n. Then that value is subtracted by 1 and returned.

```
calc_min ( total number of disks )        ## total_number_of_disks = n;
{
        final;                            ## variable
        ans = 1;                          ## varible ans = 1

        loop
        {

        ans = ans * 2;                    ## store 2^n in ans.

        }
        final = ans - 1;                  ## 2^n - 1
```

return final; ## give back thw result
}

BRIEF - ## This is where the recursion implementation and printing happens
We taken in the number of disks and the start,end and mediating peg.
Firstly we check if the number of disks provided to us isn't less than 0. If not then we start with the execution of recursion (more detail below) otherwise we return.

recursive_exec (total number of disks , a , b, c) {

if (total number of disks is less than 0):
{
return 0;
}

if (total number of disks is greater than or equal to 0)
{

call function again (total number of disks - 1 , a , c, b) by changing auxillary and destination pegs i.e store on C first.
print which disk moves from which peg to which peg;
call function again (total number of disks - 1 , c, b, a) by changing destination and auxillary pegs again i.e store on B first.
}

return 0;
}

BRIEF - ## This function is called from the main function
This function is where the actual stack iteration begins.
We take in the peg we want to remove a disk from and the peg we want to push that disk on. We also take the actual disks as input.
First we pop from the pegs and then go to the pegs with whatever condition passes (discussed in detail below) and do the actual push. Before pushing we actually check if the stack isn't full. That piece of code to check if a stack is full is in inside the stack.c's stack_push function itself.
After we succesfully push, we just print the latest push record and return.

execute_iteration(FROM, TO, source, destination)
{

small_disk = Pop from the 'FROM' stack and store in a varible

large_disk = Pop from the 'TO' stack and store in a varible

if (small_disk output is empty i.e source peg Is empty)
{

Check if FROM stack is not full;
Push disk popped from 'TO' onto 'FROM'
print which disk moves from which peg to which peg;

}

}

else if (large_disk output is empty i.e destination peg is empty)
{
Check if TO stack is not full;
Push disk popped from 'FROM' onto 'TO'
print which disk moves from which peg to which peg;
}
}

else if (small_disk output is greater than large_disk output i.e disk on source stack is greater than that of destination stack)
{
Check if FROMstack is not full;
Push onto FROM the larger disk;
Push onto FROM the smaller disk;
Print final push result
}
}

else (i.e large_disk output is greater than small_disk output i.e disk on destination stack is greater than that of source stack)
{
Check if TO stack is not full;
Push onto TO the larger disk;
Push onto TO the smaller disk;
Print final push result
}
}

}

BRIEF - ## This is where the actual main stack implementation begins.
Before going to 'execute_iteration', we first check the disks and other conditions here.
Firstly we custom make 3 stacks and fill the first stack with disks i.e we will peg A with disks depedning on user input of disks.
Then we check the number of disks and whether they're even or odd.
We follow the pattern that even disks will first go to the farthest peg i.e peg C first.
So for even disks, we begin with swapping the B and C peg and make C our temporary destiantion peg.
Initially We do not account for odd pegs and leave them as it is cause odd pegs simply go to destination pegs first.
We introduce a loop and bunch of repetitive conditions (given in detail below) and then call execute iteration accordingly.
Once the loop reaches the sentinel of the minimum_moves, the 3 pegs are deleted and the program returns.

stack_exec(total disks, minimum moves){

*struct Stack *peg_A, *peg_B, *peg_C;* *## Initialiaze 3 pegs i.e stacks*

peg_A = call stack_create to build a stack of size of the 'total disks' provided and name it A
peg_C = call stack_create to build a stack of size of the 'total disks' provided and name it C

peg_B = call stack_create to build a stack of size of the 'total disks' provided and name it B

begin = peg_A -> name # Store peg_A stacks name in begin variable
endpoint = peg_B -> name # Store peg_B stacks name in endpoint variable
middle = peg_C -> name; # Store peg_C stacks name in middle variable

if (total_disks provided are even in number)
{
Swap the endpoint and middle. i.e we want to start by storing in the farthest peg first i.e peg_C.
}

Loop{
Fill peg_A with the total number of disks assigned by user.
}

MAIN LOOP (I =1 ; if I is less than minimum moves ; increament i)
{
// As we haven't accounted for the odd numbers, we will find the remainder of each counter with 3.

if the remainder of I%3 is 1
{
Call execute_iteration with peg_A, peg_B, begin, endpoint;
}

else if the remainder of I%3 is 2{
Call execute_iteration with peg_A, peg_C, begin, middle;}

else the remainder of I%3 is 0{
execute_iteration with peg_C, peg_B, middle, endpoint;
}

After entire execution is complete delete all 3 stacks
delete(peg_A);
delete(peg_B);
delete(peg_C);

return 0;
}

BRIEF - ## This is the main function where program execution begins.
Initially we intiliaze default value, minimum_moves calculation variable and set flags for recursion and stacks to false
We then begin with a loop and get all the command-line/terminal arguments by given by the user.
In the loop, using switch we go to the options and set flags to true and get input disks if given. If not given, then the default value of 5 remains.
After switch, we introduce a bunch of if's and else if's to handle the flags and begin program execution accordingly. (Given in more detail below).
main(int argc, char **argv)
{

Initialize variables

total_disks = 5
minimum_moves
opt;
r_val = false
s_val = false

##################

LOOP (opt = Get command-line/ terminal arguments){

switch (opt)
{

case n:

total_disks = get the total disks to start with from the cmd.
break;

case s:
set s_val to true
break;

case r:
set r_val to true
break;

default:
if after -n no argument is provided, an error will be given.
}
}

if Both r and s values are true
{

Print stack header

minimum_moves = calculate minimum moves

Call stack_exec(total_disks,minimum_moves);

Print new line

print Number of moves

print new line

Print recursive header

Call recursive_exec(total_disks,'A','B','C');

Print new line
print number of moves

return 0;

}

else if (r_val == true and s_val == false)
{

minimum_moves = calculate minimum moves

Print recursive header

Call recursive_exec(total_disks,'A','B','C');

Print new line
Print number of moves

return 0;

}

else if (s_val == true and r_val == false)
{

Print stack header

minimum_moves = calculate minimum moves

Call stack_exec(total_disks,minimum_moves);

Print new line
print number of moves
return 0;
}

return 0;
}

PSEUDOCODE FOR STACK.C:

#include <stdio.h>
#include "stack.h"
#include <stdlib.h>
#include <stdbool.h>

Stack *stack_create (int capacity,char name)
{
if capacity is less than 1)
{

force set capacity to 1
}

struct Stack current_stack = (struct Stack*) malloc(sizeof(struct Stack));*

set current_stack -> name as name provided;
current_stack -> capacity as capacity provided;
current_stack -> top as -1;

set current_stack -> items to take dynamic input by multiplying it with the capacity provided.

return current_stack;

}

void stack_delete (struct Stack *current_stack)
{
use free() to empty current_stack
}

int stack_pop(struct Stack* current_stack)
{
if(current_stack is empty)
{
return -2;
}
else{
int val = get current disk on top of stack
decreament top counter
return val;
}
}

void stack_push(struct Stack *current_stack, int item)
{
if current stack is not full {
Increament top counter
push item on stack;
return;}

else if stack is already full {
return;
}
}

```
bool stack_empty(struct Stack* current_stack)
{
        if (current_stack -> top is full ){
        return true;}
        else If it isnt full {
        return false;}
}

int stack_peek(struct Stack *current_stack){

  if (current_stack if full ){
    return -3;
  }
  else {
  int peek_value = get the value of topmost disk on stack
        return peek_value;
}
}
```