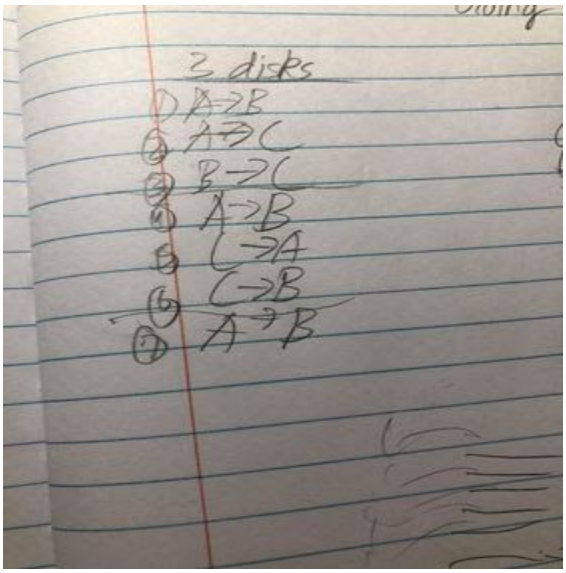
CSE13s Fall 2020
Assignment 3: The Tower Of Brahma

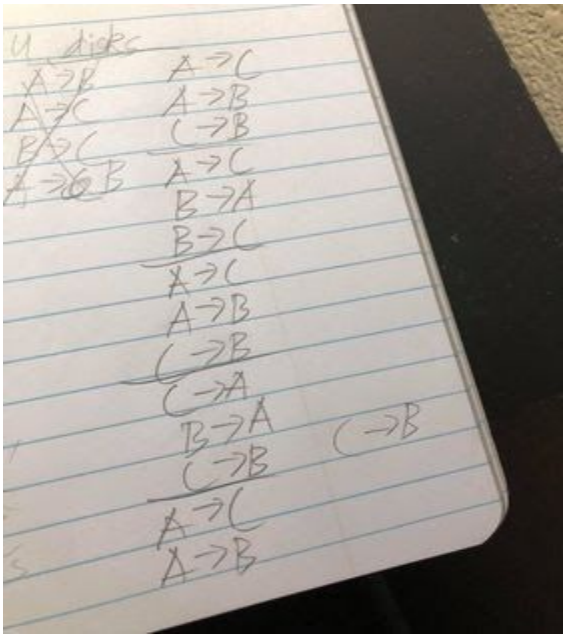
This lab is largely based on solving the puzzle of tower of hanoi, but instead of trying to get the disks from peg A to peg C, you try to get them to peg B. Try playing tower of hanoi to get a better idea. Link: https://www.mathsisfun.com/games/towerofhanoi.html

Top Level Design for tower.c:

- Int main():
    - The main() function deals with parsing command line arguments using getopt(), prints headers, and call either the recursive or the iterative functions to solve the puzzle. User types "-n" followed by a number to set a number of disks in the game, '-r' to call recursive function on it, and 's' to call an iterative function on it. Using getopt() we parse the command line arguments using while loop and switch statements. I use function atoi() to convert the given value to int. We a couple of boolean variables to denote whether recursive function or stack function have been called. Once all the arguments have been parsed, we check the boolean variables and print the headers then call the functions.
- Void stack(int n)
    - This function takes in the number of disks it has to solve the puzzle for and it solves it using stacks and loops. It creates 3 stacks (A,B,C) and in stack A pushes items in a desending order so the last item is technically the first and smallest disk. Then there is a pattern to tower of hanoi that I noticed and used to implement this solution.
    - I built a list in which I denote which pegs were used in each move. Like in the image:
    -



- I noticed that dealings between pegs for 3 moves were the same. Referencing to the image above, I noticed first move dealt with A-B, then A-C, then B-C and then the pattern repeated all over again. I checked this with 5 disks and 7 disks as well and found the same pattern(work far too messy to include an image here). BUT I noticed that pattern for even disks was a bit different
- (Reference to the image below) Even disks also had a pattern for every three moves but their pattern was different. For 4 disks, the pattern went like A-C, A-B,C-B and then repeat all over again.



- Because of this different pattern, I added a condition that checks if disks are odd or even.
- But in both cases, I loop until the top of stack B is smaller than the number of disks. I have a counter variable, “moves”, that keeps tracks of how many moves I have made and I use it to check if its the first, second, or third, move in the pattern of the 3 moves chunk by taking its modulus of 3. (REFERENCE TO PSEUDO-CODE FOR CLARITY)
- Inside the the modulus condition, I check the top disk on both pegs of the pair and whichever one is smaller moves to other one UNLESS one of them is empty then it moves to the empty one. (REFERENCE TO PSEUDO-CODE FOR CLARITY)
- After that it’s simple push and pop and printing.
- **INSPIRATION TO SEE THE PATTERN CAME FROM THIS VIDEO**: https://www.youtube.com/watch?v=ZWNK34T0YKM&ab_channel=PooyaTaheri

- Void toh(int n, char from, char using, char to):
  - This function is short and concise. This function took more thinking than actually writing. It deals with solving the puzzle using the recursive method.
  - **INSPIRATION TO SEE RECURSIVE PATTERN CAME FROM THIS VIDEO**:https://www.youtube.com/watch?v=q6RicK1FCUs&t=503s&ab_channel=AbdulBari

- The video, and piazza in general, made it clear that the recursive pattern was to move (n-1) disks to aux peg first then move it to the destination peg.
- After seeing this pattern, it was really just a matter of writing the code in literal terms. I set my base case as n == 1 to move the first disk to destination. The recursive algorithm just calls the function with one less disk and switch the aux and destination peg with each other. Then I print and recall the function to move from the aux to destination, this time using the original source peg.
- The function does not deal with stacks, it only prints.

Top Level Design for stack.c:

Note: This entire c file is largely inspired first by Eugene Cho's lab section and professor's lecture slides

- Stack * stack_create(**int** capacity, **char** name):
  - Largely inspired by professor's and Eugene's slides. Using the malloc function, space for data type Stack is allocated in memory and stored in *s. We may use this space as we like.
  - I use calloc function to allocate the 'capacity' size of memory to store int data types for items. The 'name' and 'capacity' are what's given in the parameters. And top which is really just a pointer that points to the top disk is set to 0.
- Void stack_delete(Stack *s):
  - Meant to delete stack given in parameter. Uses free() to release the allocated of iteams first then, sets them to NULL, and then frees the stack.
- Void stack_push(Stack*s, int item):
  - Checks first if there is any room by seeing it adding one more to top will push it to the max capacity. If that's the case then allocate more space using realloc(), used for resizing memory, and allocate space for two more items.
  - The items at top is now the item given in parameter and top is incremented.
- **bool** stack_empty(Stack *s):
  - Returns if the top value is 0 since that means no disk is there.
- **int** stack_pop(Stack *s):
  - First checks if the stack is empty. If it is, then it returns -1 else top is decremented and the item at top is returned. Top always points to the next empty slot.
- **int** stack_peek (Stack *s)
  - Returns -1 if stack is empty else returns the item value at top-1.

Pseudo-code

#include necessary libraries
#define OPTIONS "srn:"
main(){
Bool recursive_call, stack_call = false;
Int disk = 5;

```
while(c = getopt(argc, argv, OPTIONS) != -1) //Parsing through the command line till no
character is return, which means returning -1
switch(c){
Case ‘n’:
set x to optarge; //using atoi to convert
}
Case ‘r’:
Set recursive call boolean to true
Case ‘s’:
Set stack call boolean to true

If stack_call or recursive_call are true:
 Print headers;
 Call them();
}

Void solve_stack(int n){
Moves = 0 //Use to count moves taken
Create three Stacks A,B,C
Push all disks in stack A
If n is odd{
        while(all disks are not on peg B){
                If this is third move{
                        Move disk to whichever one is empty or
                        move the smaller disk between A-B if both have disks;
                If this is fourth move{
                        Move disk to whichever one is empty or
                        move the smaller disk between A-C if both have disks;
                If this is fifth move{
                        Move disk to whichever one is empty or
                        move the smaller disk between B-C if both have disks;
}
}
}
}
}else{//Meaning # of disks is odd
        while(all disks are not on peg B){
                If this is third move{
                        Move disk to whichever one is empty or
                        move the smaller disk between A-C if both have disks;
                If this is fourth move{
                        Move disk to whichever one is empty or
                        move the smaller disk between A-B if both have disks;
                If this is fifth move{
```

```
                    Move disk to whichever one is empty or
                    move the smaller disk between C-B if both have disks;
}
}
}
}

}
Delete all the stacks;
}

Void toh(int n, from, using, to){
If disk is 1{
Move it to ‘to’ from ‘from’; //printing only
}else{
toh(n-1, from, to, using) // Move it to auxiliary
Print that you moved from ‘from’ to ‘to’
toh(n-1, using, from, to)//Move from ‘using’ to ‘to’
}
}
```