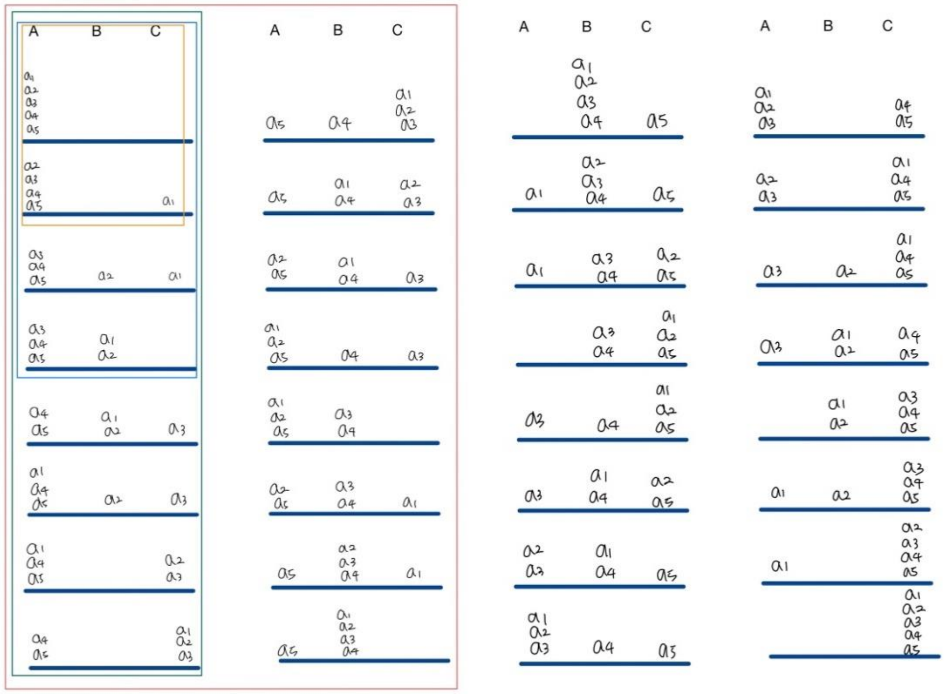
1.



□: move 4 discs from A to B

□: move 3 discs (a_1, a_2, a_3) from A to C

□: move 2 discs (a_1, a_2) from A to B

□: move 1 discs (a_1) from A to C

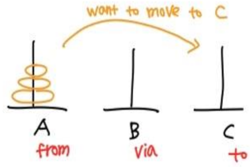
2.

```
tower_of_hanoi ( n, from, via, to) {

If  n is even
    swap ( via, to);

int total_num_of_moves = 2^n -1

for i=0 to total_num_of_moves {

  if ( i%3 ==1)
     legal movement of top disk between source pole and destination pole
  if ( i%3 ==2)
     legal movement of top disk between source pole and auxiliary pole
  if ( i%3 ==0)
     legal movement of top disk between auxillary pole and destination pole

  }

}
```

Time complexity : $\boxed{O(2^n)}$

∵ The tower of hanoi problem with 3 pole and n disks takes 2^n-1 moves to solve,
so, to enumerate the moves. we obviously can't do better than $t(n) = 2^n-1$
since enumerating k things is $O(k)$

<Technical report>



1. The reason for setting parameter

void tower_of_hanoi (int n, char from, char via, char to)

n → num of disks
from → source pole
via → auxilary pole
to → destination pole

n: used to get total num of moves $(2^n - 1)$

from, via, to: have to know move from where to where

A-B
B-C
C-A

3 possible interaction

2. Theoretical explanation

after a disk other than the smallest is moved, the next to be moved must be the smallest disk because it is the top disk resting on the spare pole and there are no other choices to move a disk.

A smple Algorithm using this principle is to alternate moves between the smallest and the not when moving the smallest piece, always move it to the next position in the same direction. (to the right if the number of disks is even, to the left if the number of disks is odd). If there is no tower position in the chosen direction, move the piece to the opposite end, but then continue to move in correct direction. For example, if you started three pieces, we would move the smallest piece to the opposite end, then continue moving in the left direction after that. When the turn is to move the non-smallest piece, there is only one *legal move. Doing this will complete the puzzle in the fewest moves.

- case of *legal movement

1. when one of the two poles is empty, we must move the disk from non empty to empty
2. when the top disk of one pole is smaller than the other, we have no choice but to move the smaller disk onto the larger disk

→ the definite implimentation of legal movement can be embodied using stack

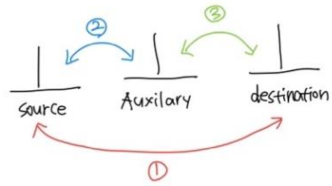
Considering those constraints after the first move, there is only one legal move at every subsequent turn

The sequence of these unique moves is an optimal solution to the problem equivalent to the problem in iterative manner.

3. performance analysis and result

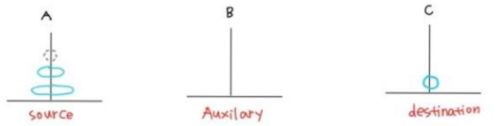
let's assume that we want move 3 disks from A to C

then. total_num_of_moves required $2^3-1 = 7$

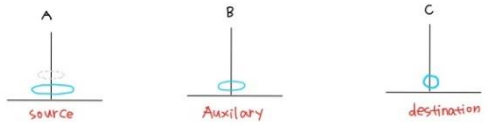


for i=1 to 7, taking enable movement between two poles

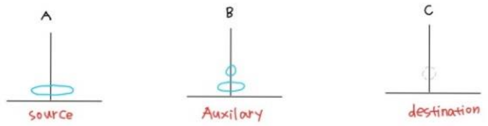
i=1 i%3 == 1 → legal movement between source and destination



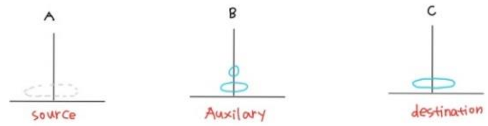
i=2 i%3 == 2 → legal movement between source and Auxilary



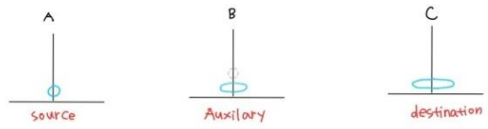
i=3 i%3 == 0 → legal movement between Auxilary and destination



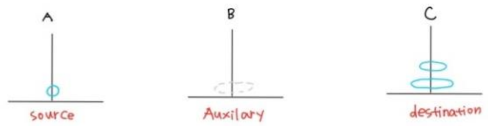
i=4 i%3 == 1 → legal movement between source and destination



i=5 i%3 == 2 → legal movement between source and Auxilary

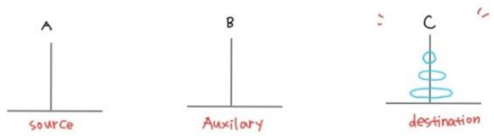


i=6 i%3 == 0 → legal movement between Auxilary and destination



. The result

i=7 i%3 == 1 → legal movement between source and destination



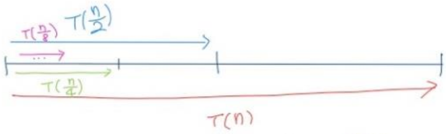
⇒ for 2^3-1 loop, we can move all disks from source to destination.

after every three moves, the destination pole is in state that it can accept a new disk.

so after every three moves, we can go with the other three moves

and so untill we exhaust all our moves.

1) $T(n) = T(\frac{n}{2}) + c$



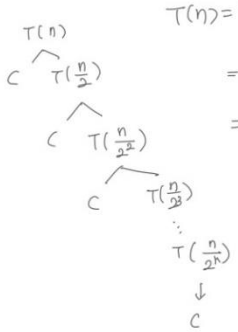
for loop, the size of problem become half.

let's assume that $T(1) = c$

$$T(n) = \begin{cases} c & n=1 \\ T(\frac{n}{2}) + c & n>1 \end{cases}$$

$n=1 \rightarrow$ the end of recursion

let k the number of total implementation



$$\left.\begin{aligned} T(n) &= T(\frac{n}{2}) + c \\ &= T(\frac{n}{2^2}) + c + c \\ &= T(\frac{n}{2^3}) + c + c + c \\ &\vdots \\ &= T(\frac{n}{2^k}) + k \cdot c \end{aligned}\right\} k \text{ times}$$

when $\frac{n}{2^k} = 1$ recursive call is ended and return c

so for $n > 2^k$ we have

$$T(n) = T(\frac{n}{2^k}) + k \cdot c$$

if n is a power of 2, say $n = 2^k$

Rewrite k in terms of n. $k = \lg n$

$$\begin{aligned} T(n) &= T(\frac{n}{2^{\lg n}}) + c\lg n \\ &= T(\frac{n}{n}) + c\lg n \\ &= T(1) + c\lg n \\ &= c + c\lg n \\ &= c(\lg n + 1) + \bigcirc \end{aligned}$$

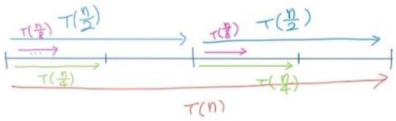
when n is not power of 2, ($2^{k-1} < n < 2^k$) some constants can exist here.

but constant has little effect on time complexity. so it can be ignored.

$\therefore$ we can say that

$$T(n) = \boxed{O(\log n)}$$

2) $T(n) = 2 \cdot T(\frac{n}{2}) + c$



for loop, the size of problem become half

but the number of problem become twice at the same time

$$T(n) \begin{cases} c & n=1 \\ 2T(\frac{n}{2}) + c & n>1 \end{cases}$$

let k the number of total implementation

$$\begin{aligned} T(n) &= 2 \cdot T(\frac{n}{2}) + c \\ &= 2 \cdot (2 \cdot T(\frac{n}{2^2}) + c) + c \\ &= 2 \cdot 2 \cdot (2T(\frac{n}{2^3}) + c) + c + 2c \\ &= 2 \cdot 2 \cdot 2 \cdot (2T(\frac{n}{2^4}) + c) + c + 2c + 2^2c \\ &\vdots \\ &= 2^k \cdot T(\frac{n}{2^k}) + \underline{c + 2c + 2^2c + 2^3c \cdots 2^{k-1}c} \\ &= 2^k \cdot T(\frac{n}{2^k}) + c(2^k - 1) \end{aligned}$$

$$\frac{a(r^n - 1)}{r-1} = \frac{c \cdot (2^k - 1)}{2-1} = c(2^k - 1)$$

so for $n > 2^k$ we have

$$T(n) = 2^k \, T(\frac{n}{2^k}) + (2^k - 1)c$$

if n is a power of 2, say $n = 2^k$

$k = \lg n$

$$\begin{aligned} T(n) &= 2^{\lg n} \cdot T(n/2^{\lg n}) + (2^{\lg n} - 1)c \\ &= n\,T(n/n) + (n-1)c \\ &= n\,T(1) + (n-1)c \\ &= nc + (n-1)c = (2n-1)c \end{aligned}$$

$$\therefore \boxed{O(n)}$$