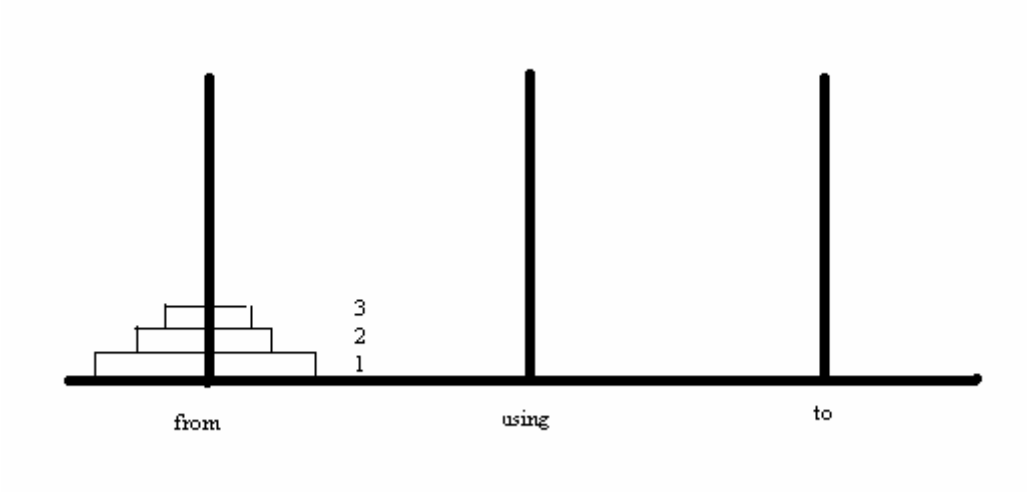
TOWER OF HANOI

Problem :



There are 3 pegs ‘from’ , ‘using ‘ and ‘to’. Some disks of different sizes are given which can slide onto any peg . Initially all of those are in ‘from’ peg in order of size with largest disk at the bottom and smallest disk at the top. We have to move all the disks from ‘from’ peg to ‘to’ peg . At the end ,’to’ peg will have disks in the same order of size . there are some rules :
1)only one disk can be moved from one peg to another peg at a time.
2) A disk can be placed only on top of a larger one .
3) A disk can be moved from top only.

The function is:

```
void Towerof Hanoi(int n, String  from, String using, String to)
{
        if(n>0)
        {
        Towerof Hanoi(n-1,   from,  to,  using);
        System.out.println(“Move disk ”+n+” from ”+from+” to ”+to);
        Towerof Hanoi(n-1,  using, from, to);
        }

}
```

So first recursive call moves n-1 disks from ‘from’ to ‘using’ using ‘to’. So after that call n-1 disks are in ‘using ‘ peg in order of size and the ‘from’ peg contains the nth disk i.e, the largest one. So, now move that disk from ‘from’ peg to ‘to’ peg. Then again by the 2^{nd} recursive call move n-1 disk from ‘using’ peg to ‘to’ peg using ‘from’ peg. So , after all these, ‘to’ peg contains all disks in order of size .

Time Complexity :

Let the time required for n disks is T(n) .
There are 2 recursive call for n-1 disks and one constant time operation to move a disk from 'from' peg to 'to' peg . Let it be k_1.
Therefore,
$T(n) = 2\ T(n-1) + k_1$

$T(0) = k_2$, a constant.
$T(1) = 2\ k_2 + k_1$
$T(2) = 4\ k_2 + 2k_1 + k_1$
$T(2) = 8\ k_2 + 4k_1 + 2k_1 + k_1$

Coefficient of $k_1 = 2^n$
Coefficient of $k_2 = 2^n - 1$
Time complexity is $O(2^n)$ or $O(a^n)$ where a is a constant greater than 1.
So it has exponential time complexity. For single increase in problem size the time required is double the previous one. This is computationally very expensive. Most of the recursive programs takes exponential time that is why it is very hard to write them iteratively .

Space Complexity:

Space for parameter for each call is independent of n i.e., constant. Let it be k .
When we do the 2^{nd} recursive call 1^{st} recursive call is over . So, we can reuse the space of 1^{st} call for 2^{nd} call . Hence ,

$T(n) = T(n-1) + k$
$T(0) = k$

$T(1) = 2k$
$T(2) = 3k$
$T(3) = 4k$
So the space complexity is O(n).

Here time complexity is exponential but space complexity is linear . Often there is a trade off between time and space complexity .

PERMUTATION GENERATION

Problem :

We are given an array of n distinct integers . Generate all possible (n!) of the numbers in that array .

Complexity of iterative program will be much higher than the recursive one .
Suppose we have a recursive program which generates all permutation of n numbers and after that it returns the original array .

1	**2**	3	4	5
2	1	**3**	4	5
3	1	2	**4**	5
4	1	2	3	**5**
5	1	2	3	4

Generate recursively all the permutation of n-1 numbers excluding first one . Then swap the 1st with 2nd . In the example the numbers which are swapped are in bold and the portion of the array for which recursive call is made is underlined . At the end the array obtained is not original one.But the requirement of the program was to return the original array. To get the original array give a left shift .

5 1 2 3 4 => 1 2 3 4 5

Why the original array is to be returned ?

Suppose the original array is not returned at the end . Then ,

1	2	3	**4**	**5**
1	2	**3**	**5**	4
1	2	5	**3**	**4**
1	2	**5**	4	**3**
1	2	**3**	4	**5**

We end up again in generating the same permutation . So, we have to return the original array at the end .

The function looks like :

```
void Permute(int a[] , int st )
{
}
```

a[] is the array of integers given and st is the start index .