7.1.6. Using the procedure `write(x)` for `x` $= 0\ldots 9$, write a recursive procedure that prints the decimal representation of a positive integer n.

Solution. The recursive solution allows us to produce digits from right to left but print them from left to right:

```
procedure print (n:integer); {n>0}
begin
| if n<10 then begin
| | write (n);
| end else begin
| | print (n div 10);
| | write (n mod 10);
| end;
end;
```

□

7.1.7. The "Towers of Hanoi" puzzle consists of three vertical sticks and N rings of different sizes. The rings are put on one of the sticks in such a way that larger rings are beneath smaller ones. We are to move this tower onto another stick one ring at a time. While moving the rings from one stick to another, we are not permitted to put a larger ring onto a smaller one. Write a program that shows the list of movements required to solve the problem.

Solution. The following recursive procedure moves `i` upper rings from the `m`-th stick to the `n`-th stick (we assume that the remaining rings on all sticks are larger and remain untouched); `m` and `n` are different numbers among $\{1, 2, 3\}$:

```
procedure move(i,m,n: integer);
  var s: integer;
begin
  if i = 1 then begin
    writeln ('move ', m, '->', n);
  end else begin
    s:=6-m-n; {s is the third stick; 1+2+3 = 6}
    move (i-1, m, s);
    writeln ('move ', m, '->', n);
    move (i-1, s, n);
  end;
end;
```

(The first recursive call moves a tower of `i-1` rings onto the third stick. After that the `i`-th ring becomes free and is moved to the remaining stick. The second recursive call moves the tower onto the `i`-th ring.) □

https://towardsdatascience.com/tower-of-hanoi-a-recursive-approach-12592d1a7b20
The problem involves three rods and a number of disks of different sizes that can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top. The objective of the puzzle is to move the entire stack to another rod, 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 on an empty rod.
3. No disk may be placed on top of a smaller disk.

Here's a recursive Java program for solving the Tower of Hanoi problem:
java

```java
public class TowerOfHanoi {
    public static void towerOfHanoi(int n, char sourceRod, char targetRod, char
auxiliaryRod) {
        if (n == 1) {
            System.out.println("Move disk 1 from " + sourceRod + " to " + targetRod);
            return;
        }

        towerOfHanoi(n - 1, sourceRod, auxiliaryRod, targetRod);
        System.out.println("Move disk " + n + " from " + sourceRod + " to " +
targetRod);
        towerOfHanoi(n - 1, auxiliaryRod, targetRod, sourceRod);
    }
    public static void main(String[] args) {
        int numberOfDisks = 3;
        towerOfHanoi(numberOfDisks, 'A', 'C', 'B');
    }
}
```

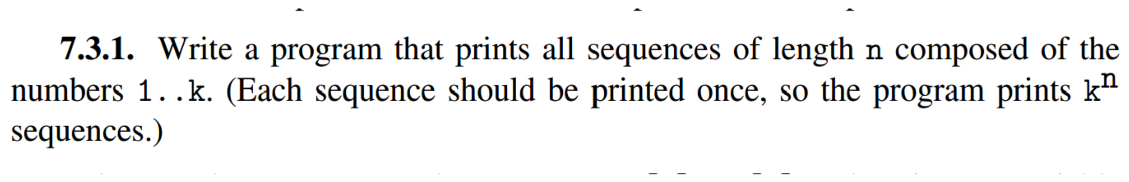
In this Java program:

- The `towerOfHanoi` method is a recursive function that takes the number of disks `n`, the source rod (`sourceRod`), the target rod (`targetRod`), and the auxiliary rod (`auxiliaryRod`) as parameters.

- The base case (`n == 1`) handles the situation when there is only one disk. In this case, it directly moves the disk from the source rod to the target rod.

- For cases where there are more than one disk, the program follows the recursive steps:
  1. Move `n-1` disks from the source rod to the auxiliary rod.
  2. Move the remaining disk from the source rod to the target rod.
  3. Move the `n-1` disks from the auxiliary rod to the target rod.

- In the `main` method, we initialize the number of disks to 3 and call the `towerOfHanoi` method to solve the Tower of Hanoi problem, moving disks from rod 'A' to rod 'C' with the help of rod 'B'.



7.3.1. Write a program that prints all sequences of length n composed of the numbers 1..k. (Each sequence should be printed once, so the program prints k^n sequences.)

Recursive:

```
procedure generate_sequences(sequence, n, k):
    if length(sequence) == n:
        print(sequence)
        return

    for i from 1 to k:
        append i to sequence
        generate_sequences(sequence, n, k)
        remove the last element from sequence

procedure print_all_sequences(n, k):
    sequence = empty list
    generate_sequences(sequence, n, k)

# Example usage:
print_all_sequences(3, 2)
```

Time Complexity Analysis:

Best Case:

In the best case, where n is 0, and k is any positive integer, the algorithm immediately prints an empty sequence, and the time complexity is constant.

Best Case Time Complexity:

O(1)

Average Case:

For the average case, the algorithm explores all possible sequences of length n composed of numbers 1 to k using a recursive approach. It makes a recursive call for each possible choice at each position in the sequence. The number of recursive calls grows exponentially with n, and for each recursive call, there is a loop that iterates k times.

The number of recursive calls can be represented as O(k^n)   in the average case. Therefore, the average time complexity is exponential.

Average Case Time Complexity: O(k^n)

Worst Case:

The worst case occurs when the algorithm needs to generate and print all possible sequences of length n composed of numbers from 1 to k. In the worst case, the algorithm explores all possible combinations, and the number of recursive calls remains O(k^n)
Worst Case Time Complexity: O(k^n)

Space Complexity Analysis:

The space complexity of the algorithm is determined by the maximum depth of the recursion, which is the length of the sequence n. In each recursive call, a constant amount of space is used for the sequence.

Space Complexity:O(n)

In summary, the algorithm has an exponential time complexity in the average and worst cases, making it inefficient for large values of n or k. The space complexity is linear with respect to the length of the sequence.

4.3-2

The solution to the recurrence $T(n) = 4T(n/2) + n$ turns out to be $T(n) = \Theta(n^2)$. Show that a substitution proof with the assumption $T(n) \le cn^2$ fails. Then show how to subtract a lower-order term to make a substitution proof work.

Let us assume $T(n) \le cn^2$ for all $n \ge n_0$, where c and n_0 are positive constants.

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + n \\ &\le 4c\left(\frac{n}{2}\right)^2 + n \\ &= cn^2 + n \end{aligned}$$

With this we cannot prove our assumption in it's exact form.

Now, let us assume $T(n) \le cn^2 - bn$ for all $n \ge n_0$, where b, c, and n_0 are positive constants.

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + n \\ &\le 4\left(c\left(\frac{n}{2}\right)^2 - \frac{bn}{2}\right) + n \\ &= 4\left(c\frac{n^2}{4} - \frac{bn}{2}\right) + n \\ &= cn^2 - 2bn + n \\ &= cn^2 - bn - (b-1)n \\ &\le cn^2 - bn \end{aligned}$$

The last step holds as long as $(b-1)n$ is positive.

> Solve the recurrence $T(n) = 3T(\sqrt{n}) + \log n$. by making a change of variables. Your solution should be asymptotically tight. Do not worry about whether values are integral.

Note: I don't why suddenly log is used in this problem statement, whereas till now everywhere lg was used. I'm assuming in this context, log still means $\log_2 = \lg$.

Let us assume $n = 2^m$, $m = \lg n$.

Hence, the recurrence takes the form

$T(2^m) = 3T(2^{m/2}) + m.$

Now, assuming $S(m) = T(2^m)$, the recurrence takes the form

$S(m) = 3S(m/2) + m.$

Let's assume, $S(m) \le cm^{\lg 3}$, for all $m \ge m_0$, where c and m_0 are some positive constants.

$$\begin{aligned} S(m) &= 3S(m/2) + m \\ &\le 3c(m/2)^{\lg 3} + m \\ &= 3c\frac{m^{\lg 3}}{2^{\lg 3}} + m \\ &= cm^{\lg 3} + m \end{aligned}$$

With this we cannot prove our assumption in it's exact form.

So, let us modify our assumption by subtracting a lower-order term.

Let's assume $S(m) \le cm^{\lg 3} - bm$.

$$\begin{aligned} S(m) &= 3S(m/2) + m \\ &\le 3(c(m/2)^{\lg 3} - b(m/2)) + m \\ &= 3c\frac{m^{\lg 3}}{2^{\lg 3}} - 3bm/2 + m \\ &= cm^{\lg 3} - bm - (b/2 - 1)m \\ &\le cm^{\lg 3} - bm \end{aligned}$$

The last step holds as long as $(b/2 - 1)m$ is positive.

If we set $m_0 = 1$, we need $b \ge 2$.

Changing our variable back to n, we have:

$$T(n) = O(m^{\lg 3}) = O((\lg n)^{\lg 3}) = O(\lg^{\lg 3} n)$$

> Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 3T(\lfloor n/2 \rfloor) + n$. Use the substitution method to verify your answer.

RECURSION TREE

Ignoring the floor function, the recursion takes the form:

$$T(n) = 3T(n/2) + n$$

Rate of increase in number of subproblems in each recursion = 3

Rate of decrease in subproblem size = 2

Hence at depth $i = 0, 1, 2, \ldots, \lg n$ of the tree, there are 3^i nodes, each of cost $n/2^i$.

Hence, total cost of the tree is:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\lg n} 3^i \frac{n}{2^i} \\
&= n \cdot \sum_{i=0}^{\lg n} \left(\frac{3}{2}\right)^i \\
&= n \cdot \frac{(3/2)^{\lg n+1} - 1}{(3/2) - 1} \\
&= 2n \cdot ((3/2) \cdot (3/2)^{\lg n} - 1) \\
&= 3n \cdot (3/2)^{\lg n} - 2n \\
&= 3n \cdot \frac{3^{\lg n}}{2^{\lg n}} - 2n \\
&= 3n \cdot \frac{3^{\lg n}}{n} - 2n \\
&= 3 \cdot 3^{\lg n} - 2n \\
&= 3 \cdot n^{\lg 3} - 2n \\
&= O(n^{\lg 3})
\end{aligned}$$

The last step comes from the fact that $n^{\lg 3} \approx n^{1.58} > n$.

VERIFICATION USING SUBSTITUTION

If you notice carefully, the recurrence has almost exact same form (ignoring floor) as the modified recurrence in the previous exercise after changing variables.

Another

https://atekihcan.github.io/CLRS/04/E04.04-05/

https://atekihcan.github.io/CLRS/04/E04.05-01/
https://atekihcan.github.io/CLRS/04/E04.05-02/

Problems:

https://github.com/bollwarm/DataStructuresAlgorithms

Replace every array element with the product of every other element without using a division operator

Given an integer array, replace each element with the product of every other element without using the division operator.

For example,

```
Input:  { 1, 2, 3, 4, 5 }
Output: { 120, 60, 40, 30, 24 }


Input:  { 5, 3, 4, 2, 6, 8 }
Output: { 1152, 1920, 1440, 2880, 960, 720 }
```

```
#include <stdio.h>
 int findProduct(int arr[], int n, int left, int i)
{
    // base case: no elements left on the right
    if (i == n) {
        return 1;
    }
    // take backup of the current element
    int curr = arr[i];
    // calculate the product of the right subarray
    int right = findProduct(arr, n, left * arr[i], i + 1);
    // replace the current element with the product of the left and right
subarray
    arr[i] = left * right;
    // return product of right the subarray, including the current element
    return curr * right;
}
```

```c
int main(void)
{
    int arr[] = { 5, 3, 4, 2, 6, 8 };
    int n = sizeof(arr) / sizeof(arr[0]);

    findProduct(arr, n, 1, 0);

    // print the modified array
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```