Basic Problem Solving on Recursion

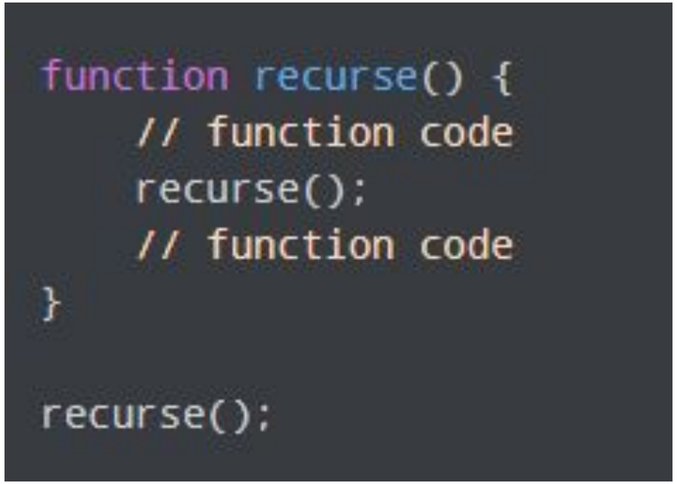
Relevel
by Unacademy

List of Concepts Involved

- Basic understanding of recursion
- Fibonacci sequence using recursion
- Print the nodes of a Binary Tree from left to right
- Check Palindrome in a String using recursion
- Sum triangle from array
- Subsequences Recursively
- Generate Parentheses Recursively
- Lexicographical printing Recursively
- Count maze path
- Permutations Recursively

Basic Understanding of Recursion

- Recursion is a process of calling itself. A function that calls itself is known as a recursive function.
- The syntax for a recursive function is as show below:
- Here, the function recurse() is a recursive function. It is calling itself inside the function

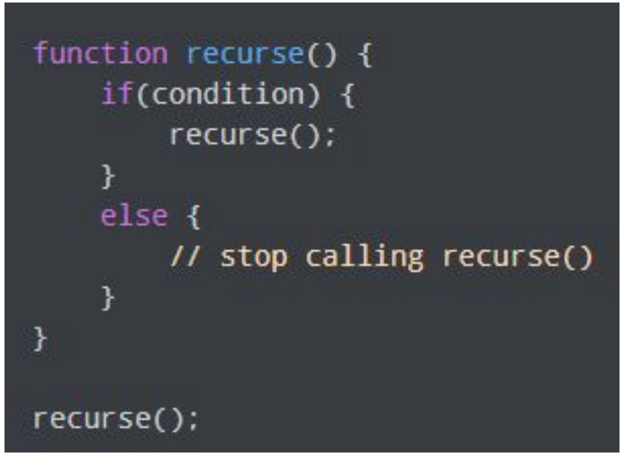


```
function recurse() {
    // function code
    recurse();
    // function code
}

recurse();
```

How Recursive function works:

- A recursive function must have a condition to stop calling itself. Otherwise, the function is called indefinitely.
- Once the condition is met, the function stops calling itself. This is called a base condition.
- To prevent infinite recursion, you can use conditional statements like if...else, while (or similar approach) where one branch makes the recursive call, and the other doesn't.
- So, generally is should look like this.



```
function recurse() {
    if(condition) {
        recurse();
    }
    else {
        // stop calling recurse()
    }
}

recurse();
```

Fibonacci Sequence using Recursion

Problem Statement:

Write a Fibonacci sequence of nth term using recursion.

Example 1:

Input: 7

Output: 0 1 1 2 3 5 8

Example 2:

Input: 10

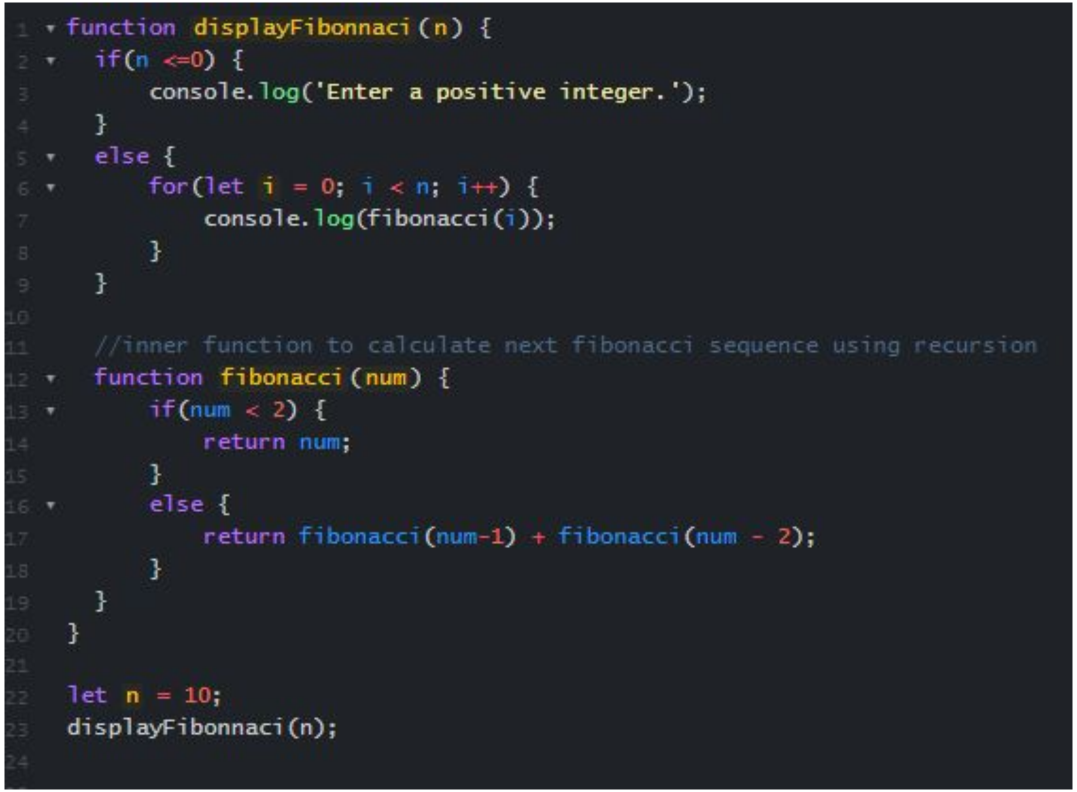
Output: 0 1 1 2 3 5 8 13 21 34

What is a Fibonacci sequence?

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

The Fibonacci sequence is the integer sequence where the first two terms are 0 and 1. After that, the next term is defined as the sum of the previous two terms. Hence, the nth term is the sum of (n-1)th term and (n-2)th term.

Code: https://jsfiddle.net/Lkn143g6/



```
function displayFibonnaci(n) {
  if(n <=0) {
      console.log('Enter a positive integer.');
  }
  else {
      for(let i = 0; i < n; i++) {
          console.log(fibonacci(i));
      }
  }

  //inner function to calculate next fibonacci sequence using recursion
  function fibonacci(num) {
      if(num < 2) {
          return num;
      }
      else {
          return fibonacci(num-1) + fibonacci(num - 2);
      }
  }
}

let n = 10;
displayFibonnaci(n);
```

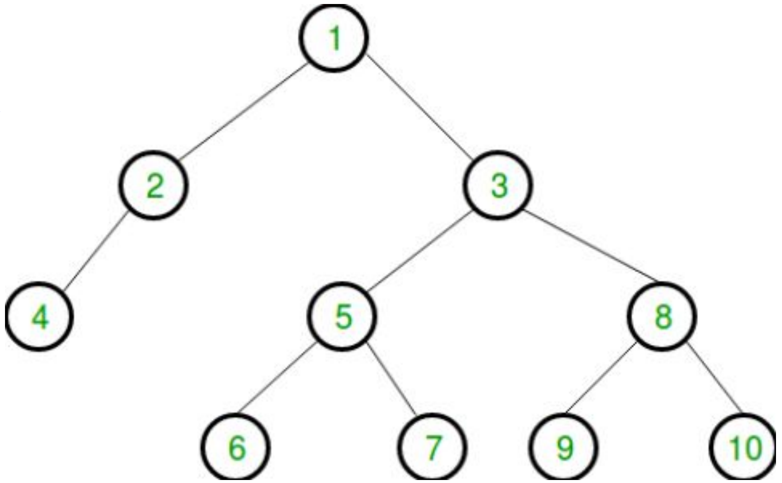
Print the nodes of a Binary Tree from left to right

Problem Statement:

Given a binary tree, write a program to print all the nodes of the given binary tree from left to right. That is, the nodes should be printed in the order they appear from left to right in the given tree.

For the shown binary tree, the output will be:

4 6 7 9 10



Understanding the problem:

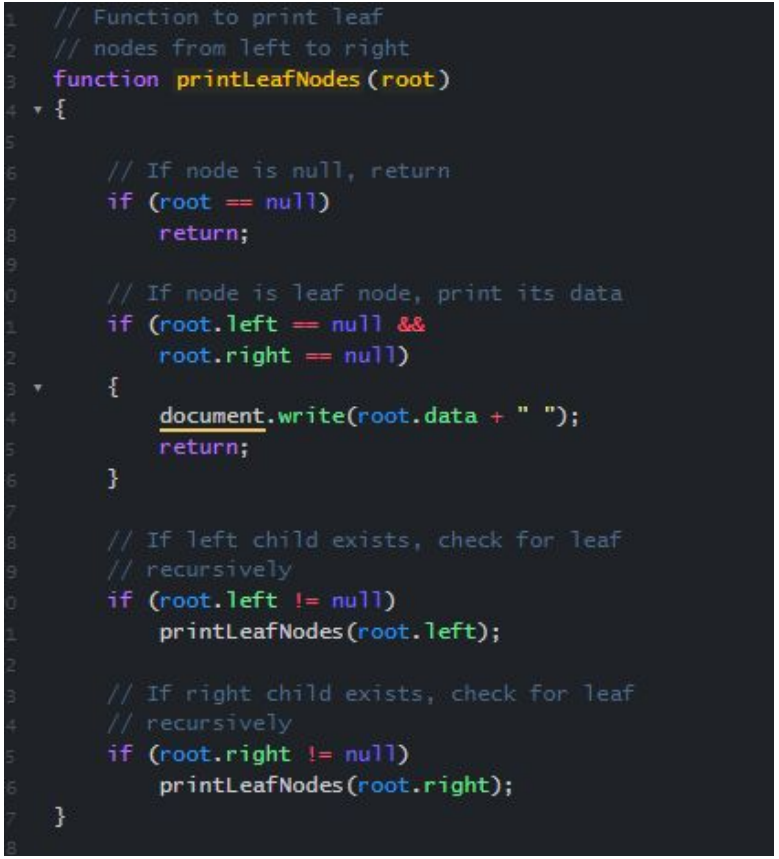
The idea behind this is very simple to understand once you have an idea about a tree.

In short, a binary tree has only 1 root node which can have 0, 1 or max 2 children and same goes for the child element as well. We will already be given with a tree structure (check the code link) all we need is to add the logic to traverse them in the required manner. Below is a step-by-step algorithm to do this:

1. Check if the given node is null. If null, then return from the function.
2. Check if it is a leaf node. If the node is a leaf node, then print its data.
3. If in the above step, the node is not a leaf node then check if the left and right children of node exist. If yes then call the function for left and right child of the node recursively.

Time Complexity: O(n) , where n is the number of nodes or leaves in the given binary tree.

Code: https://jsfiddle.net/yfv21sr9/



```
// Function to print leaf
// nodes from left to right
function printLeafNodes(root)
{

    // If node is null, return
    if (root == null)
        return;

    // If node is leaf node, print its data
    if (root.left == null &&
        root.right == null)
    {
        document.write(root.data + " ");
        return;
    }

    // If left child exists, check for leaf
    // recursively
    if (root.left != null)
        printLeafNodes(root.left);

    // If right child exists, check for leaf
    // recursively
    if (root.right != null)
        printLeafNodes(root.right);
}
```

Check Palindrome in a String using recursion

Problem Statement

Given a string, write a recursive function that checks if the given string is a palindrome or not a palindrome.

Example 1:

Input : malayalam

Output : Yes

Reverse of malayalam is also malayalam.

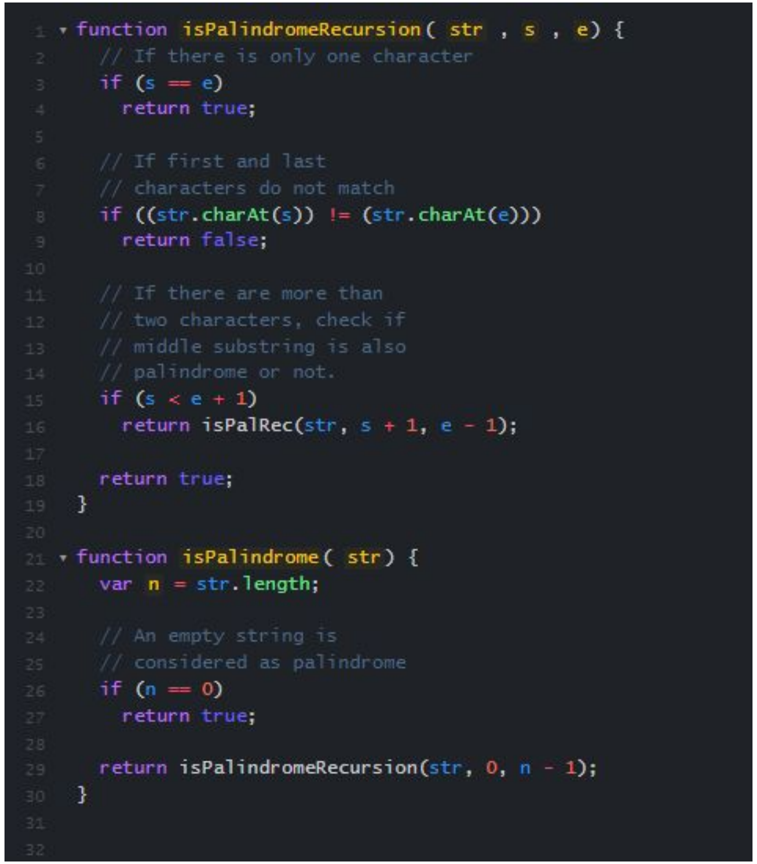
Example 2:

Input : max

Output : No

Reverse of max is not max.

Code: https://jsfiddle.net/cmfbdkh7/



```
function isPalindromeRecursion( str , s , e) {
  // If there is only one character
  if (s == e)
    return true;

  // If first and last
  // characters do not match
  if ((str.charAt(s)) != (str.charAt(e)))
    return false;

  // If there are more than
  // two characters, check if
  // middle substring is also
  // palindrome or not.
  if (s < e + 1)
    return isPalRec(str, s + 1, e - 1);

  return true;
}

function isPalindrome( str) {
  var n = str.length;

  // An empty string is
  // considered as palindrome
  if (n == 0)
    return true;

  return isPalindromeRecursion(str, 0, n - 1);
}
```

Sum triangle from array

Problem Statement

Given an array of integers, print a sum triangle from it such that the first level has all array elements. From then, at each level nu
less than the previous level and elements at the level is be the Sum of consecutive two elements in the previous level.

Example:

Input: A = {1, 2, 3, 4, 5}

Output: [48]

[20, 28]

[8, 12, 16]

[3, 5, 7, 9]

[1, 2, 3, 4, 5]

Explanation:

Here,

[48]

[20, 28] --> (20 + 28 = 48)

[8, 12, 16] --> (8 + 12 = 20, 12 + 16 = 28)

[3, 5, 7, 9] --> (3 + 5 = 8, 5 + 7 = 12, 7 + 9 = 16)

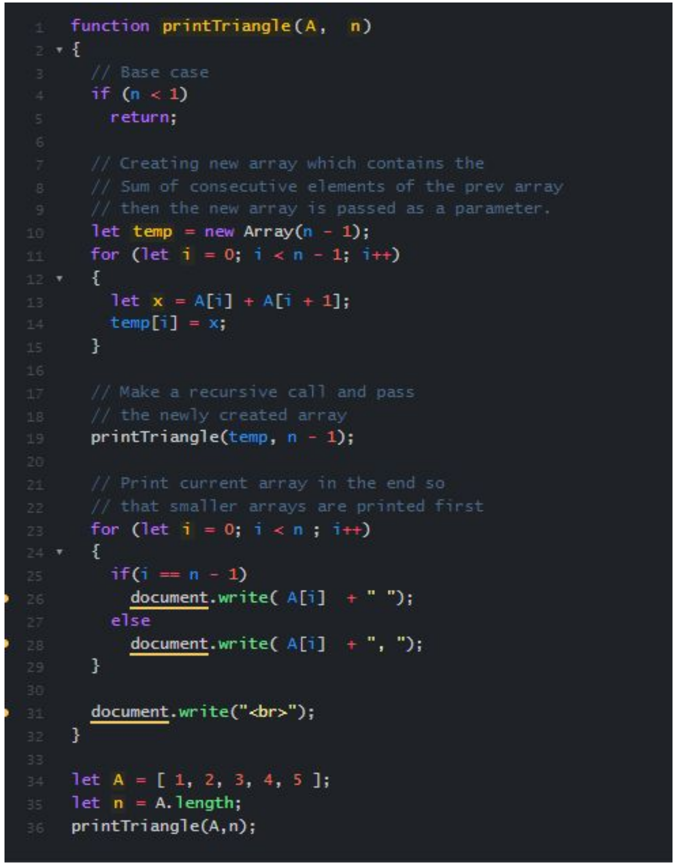
[1, 2, 3, 4, 5] --> (1 + 2 = 3, 2 + 3 = 5, 3 + 4 = 7, 4 + 5 = 9)

Understanding the algorithm

Let us first understand the algorithm to write the code:

1. At each iteration create a new array which contains the Sum of consecutive elements in the array passes as parameter.
2. Make a recursive call and pass the newly created array in the previous step.
3. While back tracking, print the array (to get smaller array printed first).

Code: https://jsfiddle.net/gfwmxh2p/



```
function printTriangle(A,  n)
{
  // Base case
  if (n < 1)
    return;

  // Creating new array which contains the
  // Sum of consecutive elements of the prev array
  // then the new array is passed as a parameter.
  let temp = new Array(n - 1);
  for (let i = 0; i < n - 1; i++)
  {
    let x = A[i] + A[i + 1];
    temp[i] = x;
  }

  // Make a recursive call and pass
  // the newly created array
  printTriangle(temp, n - 1);

  // Print current array in the end so
  // that smaller arrays are printed first
  for (let i = 0; i < n ; i++)
  {
    if(i == n - 1)
      document.write( A[i]  + " ");
    else
      document.write( A[i]  + ", ");
  }

  document.write("<br>");
}

let A = [ 1, 2, 3, 4, 5 ];
let n = A.length;
printTriangle(A,n);
```

Subsequences Recursively

Problem Statement:

Let’s take an input array, the array is an integer array

int arr[ ] = {1,2,3,4,5,6,7,8,9,10}; (sorted array)

SubSequence: May not be contiguous but maintain the relative order.

Elements in the subsequence appear in the same order as in the original array; the only difference from the sub array is that they may not be contiguous.

For example:

Consider original array as {1,2,3,4,5,6,7,8,9,10},

- {4,5,6,7} **is a subarray and a subsequence** of the original array
- {4,6,7} is **not the subarray** but **is a subsequence** of the original array (because 4,6,7 they
- appear in the same order as they appear in the original array)

- {1,9,10} **is not the subarray** but **a subsequence** of the original array because 1, 9, 10 their relative order is the same in the original array
- {5,4} **is not the subarray nor a subsequence** because the relative order of 5 and 4 is different in the original array
- {1,4,7,3} **is not the subarray nor a subsequence** because the order is maintained only up till 1,4,7 only.

Note: *All subarrays are subsequences but all subsequences are not subarray.*

For an array of ‘n’ elements,

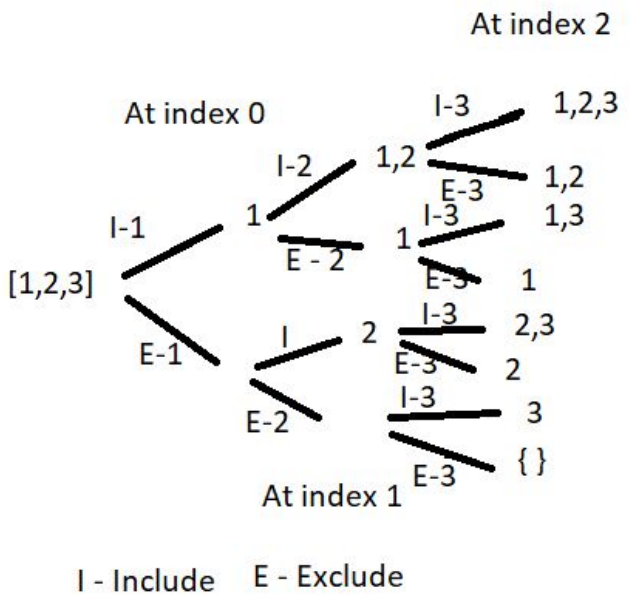
Total number of subsequences: 2^n (Including empty array)

Non-empty subsequence 2^n - 1

If we consider [1,3]: 1 is coming before 3 in a subsequence also 1 will come before 3

A subsequence is not like a permutation where we are arranging the elements. It is more like we may either add the element or not add an element.

And this is where we'll draw the logic from



First, we should start with the first element of the array; when we are at the first element of the array, the current index is 0. So, when the current index is 0, we have two choices either to include the current index or to exclude the current index (element at the current index)

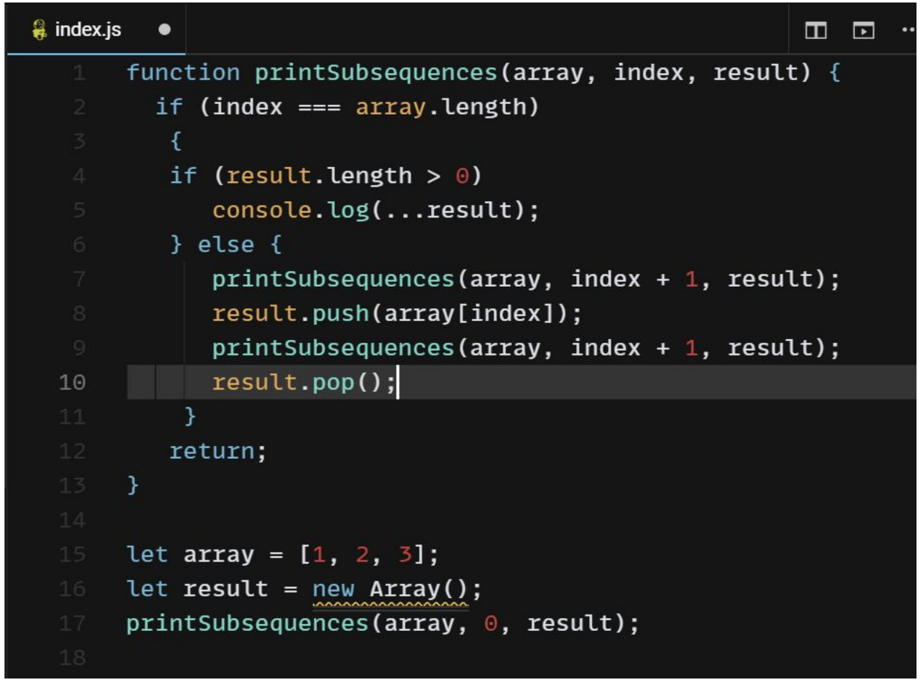
Then move to the next index, and apply this for every element in the array till we reach the last index.

The recursion tree for the array [1,2,3] is below

Now, print the subsequence once the last index is reached.

Output: [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3]

Implementation of the above approach


index.js

```
function printSubsequences(array, index, result) {
  if (index === array.length)
   {
   if (result.length > 0)
      console.log(...result);
   } else {
      printSubsequences(array, index + 1, result);
      result.push(array[index]);
      printSubsequences(array, index + 1, result);
      result.pop();
    }
   return;
}

let array = [1, 2, 3];
let result = new Array();
printSubsequences(array, 0, result);
```

Code link: https://codesandbox.io/s/floral-monad-bt7tk?file=/src/index.js

Generate Parentheses Recursively

Problem statement

You are given a ‘n’ pair of parentheses, write a function to generate all combinations of well-formed parentheses

If you are given n = 3, this means we need to generate well formed valid parentheses using 3 open brackets 3 close brackets

Then you have to return the combinations of all from parentheses, Below we have total 5 combinations of parentheses that we can generate when you have n = 3

```
[ “((()))”,
  “(()())”,
  “(())()”,
  “()(())”,
  “()()()” ]
```

- If you have n = 2, then we have to return the combinations of all parentheses

  [“(())”,

  “()()”]

- For n = 1,

  [“()”]

- For n=3,

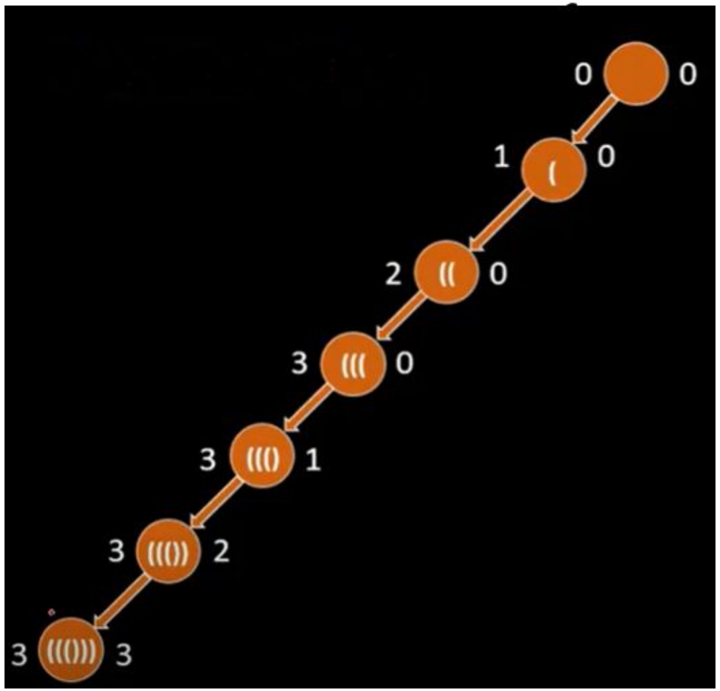
First we have ‘0’ opening bracket and ‘0’ closing bracket here we can make only once choice for all formed valid parentheses that we can only insert here 1 opening bracket **‘(‘** and there we can make two choice, first choice would be we can insert another opening bracket or we can insert here a closing bracket,

So, here then insert another opening bracket **‘((‘**, then here we have two opening bracket and 0 closing bracket. So, here we can again meet two choice we can insert here opening bracket or we can insert here a closing bracket, so let’s add here another opening bracket **‘(((‘**

Here we can’t add anymore opening bracket, here we can make only one choice that we can insert only closing bracket **‘((()’** here we left with two closing brackets

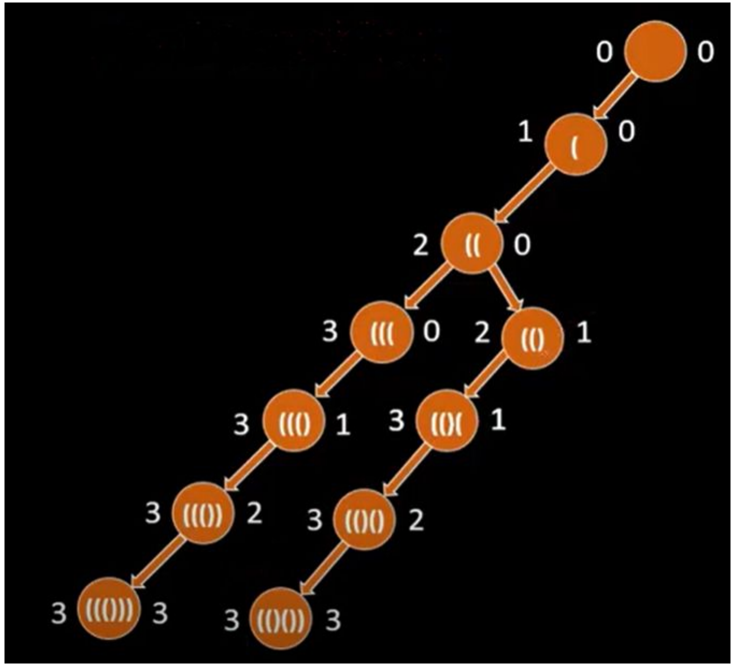
So, now let’s add another one ‘**((())**’ and another one **‘((()))’**

Below is the picturization:



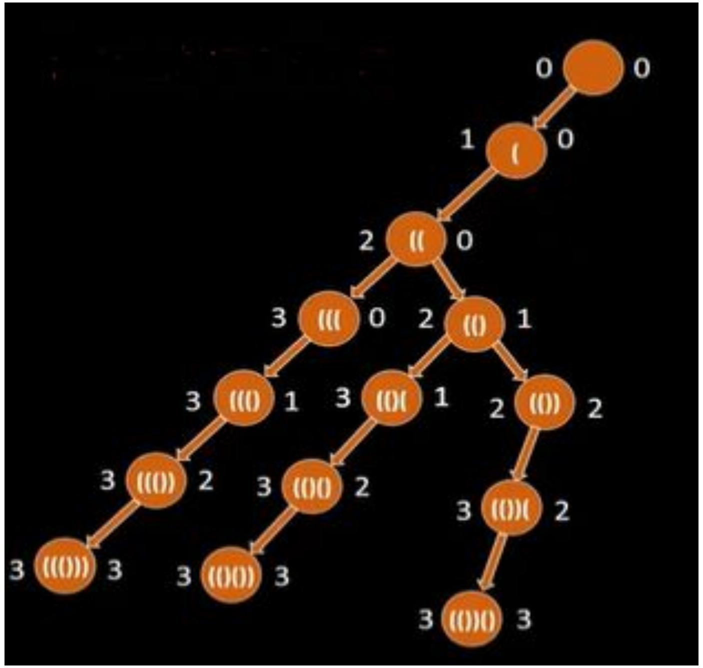
In this step **'(('**, we have another choice we can insert a closing bracket **'(()'**, now here we can insert opening bracket or closing bracket let's add here one opening bracket **'(()('**, (so now here we are done with the opening bracket) we can't add any more opening brackets, so we have only one choice that we can add closing bracket, so add one closing bracket **'(()()'**, and add another last closing bracket **'(()())'**

Check the below image for better understanding

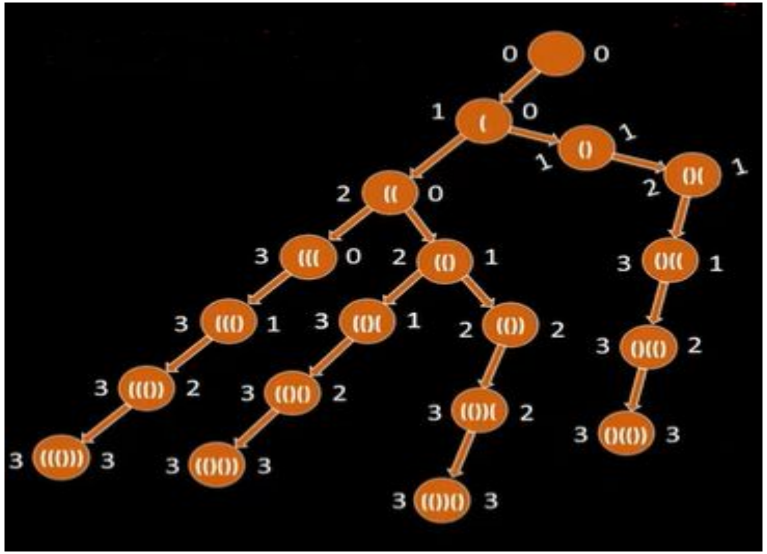


Now at this step **'(()'**, we had another choice we can add here closing brackets, let's add closing brackets **'(())'**, So here we had only one choice we can add opening bracket **'(())('**, here at this stage we left with only one closing bracket let's add this **'(())()'**,

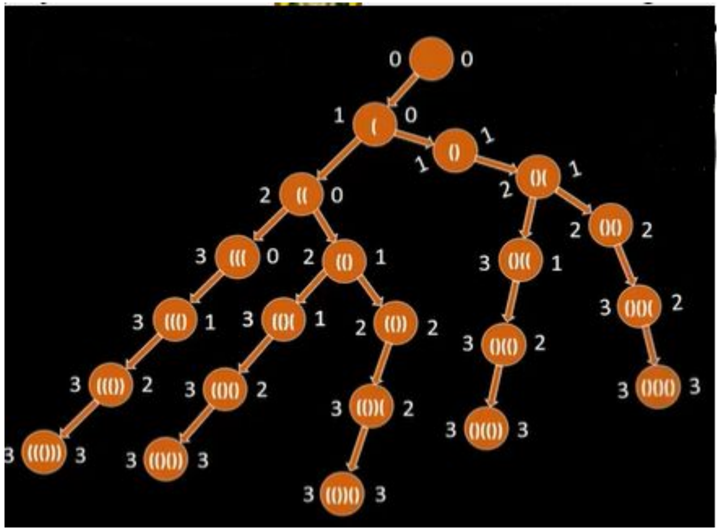
Check the below image for better understanding



Now, let's go back to this step **'('**, here we can add closing bracket **'()'**, and here we have only one choice we can only add one opening bracket **'()('**, now here we have two choices opening bracket and closing bracket so first let's add opening bracket **'()(('**, and here we have one choice we can only add closing bracket, **'()(()'**, and we left with one last choice we can add one closing bracket **'()(())'**,

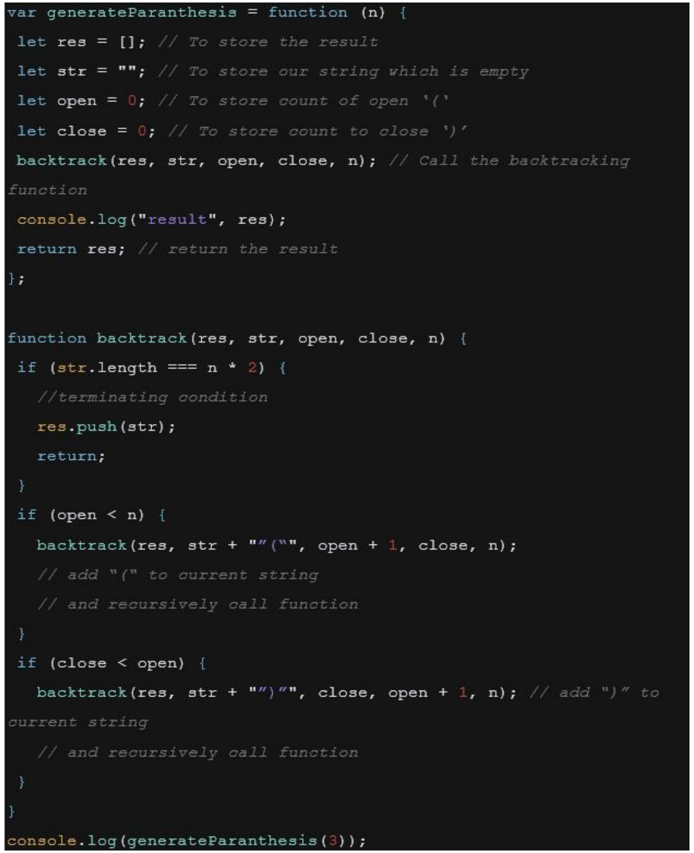


Now go back to this step **‘()(‘**, we had another choice we can add closing bracket **‘()()’**, here we have only one choice we can add only opening bracket let’s add this, **‘()()(‘**, and then add the closing bracket **‘()()()’**,



The recursive function call :

Code link:
https://codesandbox.io/s/dreamy-archimedes-u3g6s?file=/src/index.js:798-834



```
var generateParanthesis = function (n) {
 let res = []; // To store the result
 let str = ""; // To store our string which is empty
 let open = 0; // To store count of open '('
 let close = 0; // To store count to close ')'
 backtrack(res, str, open, close, n); // Call the backtracking
function
 console.log("result", res);
 return res; // return the result
};

function backtrack(res, str, open, close, n) {
 if (str.length === n * 2) {
   //terminating condition
   res.push(str);
   return;
 }
 if (open < n) {
   backtrack(res, str + ""("", open + 1, close, n);
   // add "(" to current string
   // and recursively call function
 }
 if (close < open) {
   backtrack(res, str + "")"", close, open + 1, n); // add ")" to
current string
   // and recursively call function
 }
}
console.log(generateParanthesis(3));
```

Lexicographical Printing Recursively

Problem Statement:

Given a number N, the task is to print all the numbers from 1 to N in lexicographical order.

Lexicographical order means that we are ordering by the digits with the leftmost digit giving highest priority

Input N = 14

Output: 1 10 11 12 13 14 2 3 4 5 6 7 8 9

Input: N = 19

Output: 1 10 11 12 13 14 15 16 17 18 19 2 3 4 5 6 7 8 9

Input N= 15

Output: 1 10 11 12 13 14 15 2 3 4 5 6 7 8 9

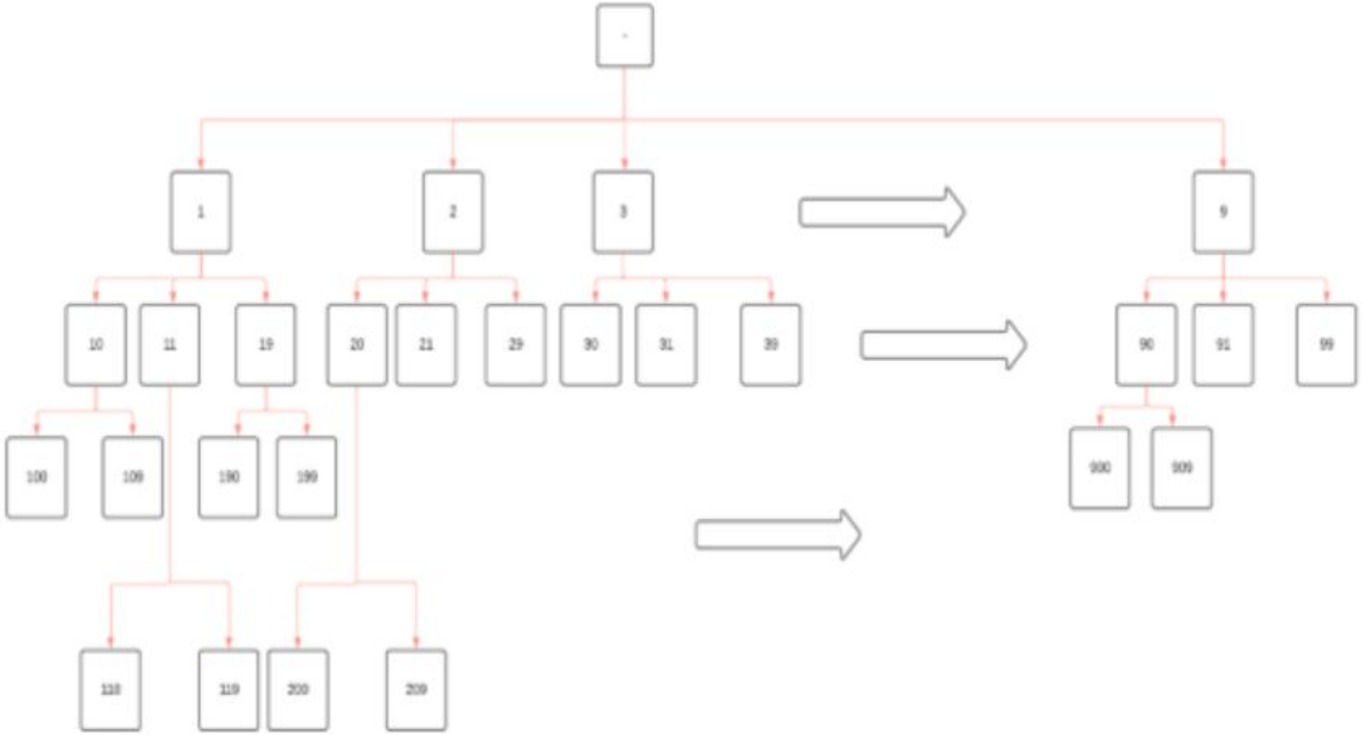
When there is no character then, 1st the family of 1

(1, 10, 100, 1000, 10000, ... 11, 110, 1100, ..., 12, ..., 19, 190,1900, ...,1999...)

will be printed less than N,

Next the family of 2(2, 20, 200, 2000, ..., 21, 210, 2100, ..., 22, ..., 29, 290, 2900, ..., 2999) will be printed less than N and so on till 9.

Below is the recursion diagram

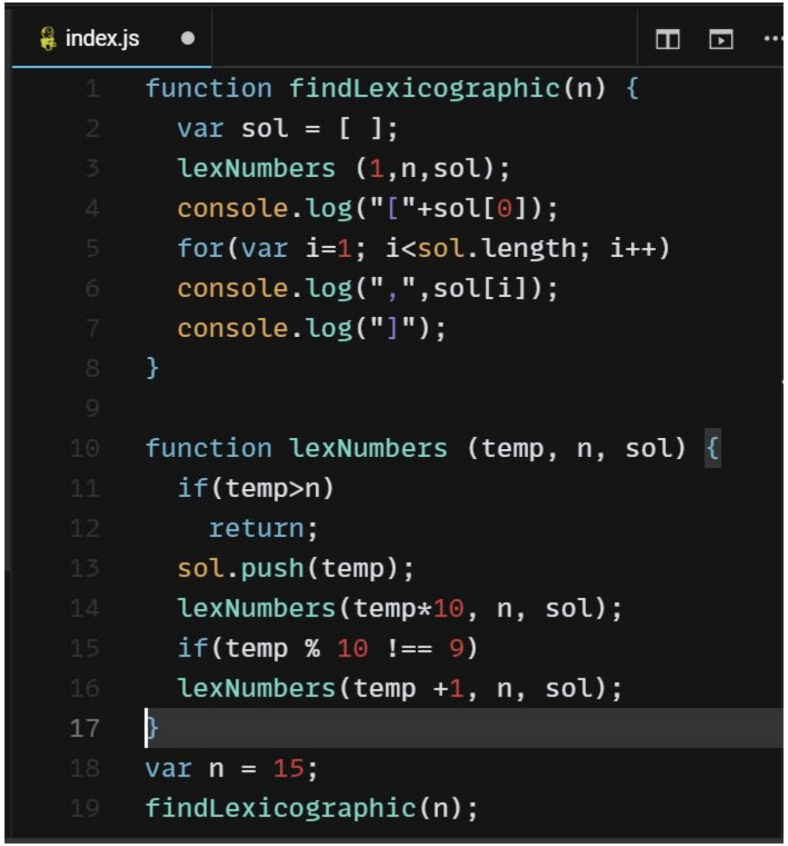


Here to generate all the families of the current number we will call the recursion function for all the numbers from 1 to 9 and then we check the condition, if the number is more than N, we return else we print it and call the recursion function again

Example:
Input N= 13
Output: 1 10 11 12 13 2 3 4 5 6 7 8 9

Here **1 10 11 12 13 14 15** these are the one digits, first 1 then 10, 11 (the second digit in 11 is 1 which is greater than second digit i.e., 0 in 10) and then 12, 13 and after 13 it stops because the max is 13 which is given (N) and then 2, 3, 4, 5, 6, 7, 8, 9 that's **lexicographical order**

JavaScript program for the above approach



index.js

```
function findLexicographic(n) {
  var sol = [ ];
  lexNumbers (1,n,sol);
  console.log("["+sol[0]);
  for(var i=1; i<sol.length; i++)
  console.log(",",sol[i]);
  console.log("]");
}

function lexNumbers (temp, n, sol) {
  if(temp>n)
    return;
  sol.push(temp);
  lexNumbers(temp*10, n, sol);
  if(temp % 10 !== 9)
  lexNumbers(temp +1, n, sol);
}
var n = 15;
findLexicographic(n);
```

Output:

```
[1,10,11,12,13,14,15,2,3,4,5,6,7,8,9]
```

Code link: https://jsfiddle.net/fsw80bcn/

Count maze problem

Problem Statement:

The problem is to count all the possible paths on an m x n grid from top left (grid[0][0]) to bottom right (grid[m-1][n-1])

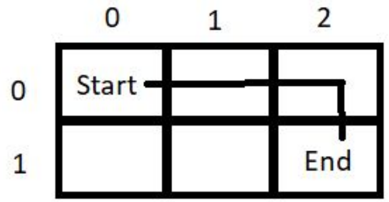
Having constraints that from each cell you can either move only to right or down

Input: m = 2, n = 3

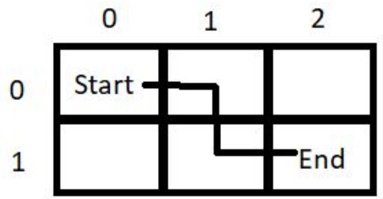
Output: 3

Here in this case, we have three possibilities,
Keeping the constraint in mind:
Moves allowed are only right and down

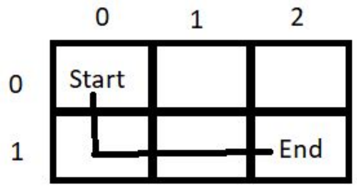
Step 1: Right -> Right -> Down
From (0,0) step into (0,1) then (0,2) and then (1,2



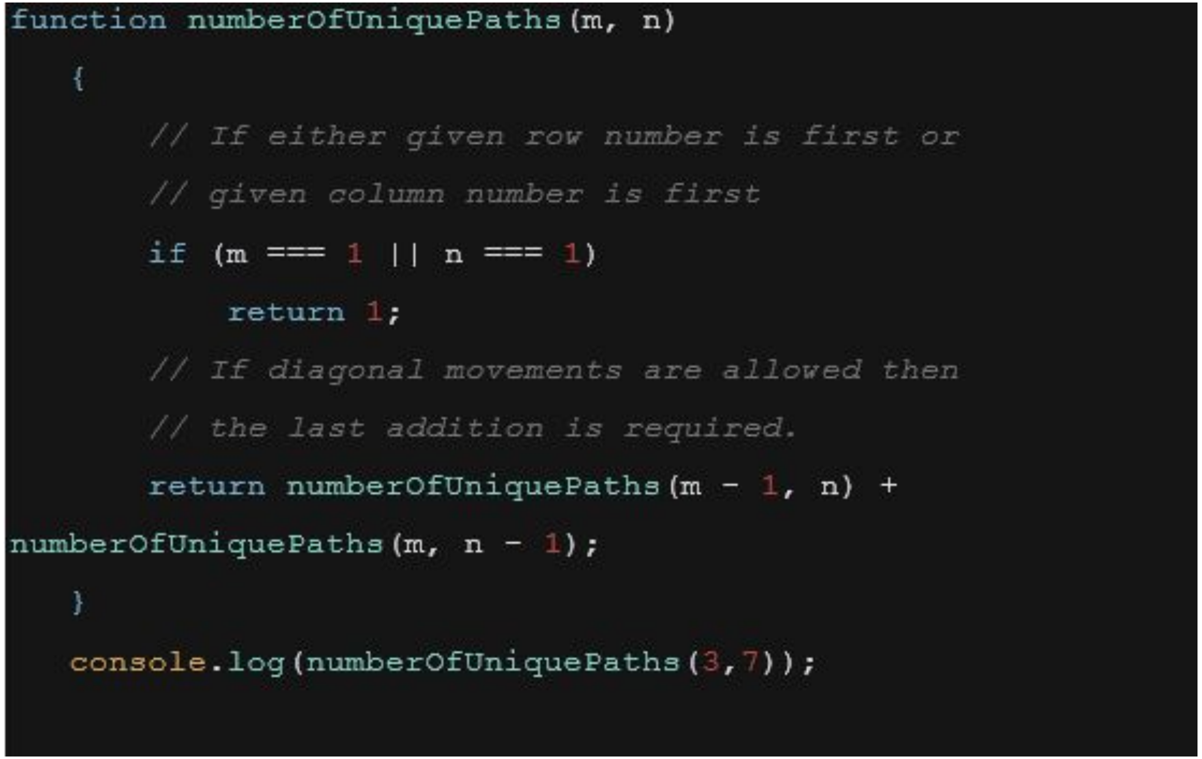
Step 2: Right -> Down -> Right
From (0,0) step into (0,1) then (1,1) then (1,2)



Step 3: Down -> Right -> Right
From (0,00 step into (1,0) then (1,1) then (1,2)



Code link: https://jsfiddle.net/4y28svhe/



```
function numberOfUniquePaths(m, n)
   {
       // If either given row number is first or
       // given column number is first
       if (m === 1 || n === 1)
            return 1;
       // If diagonal movements are allowed then
       // the last addition is required.
       return numberOfUniquePaths(m - 1, n) +
numberOfUniquePaths(m, n - 1);
   }
   console.log(numberOfUniquePaths(3,7));
```

Permutations Recursively

Here we will be discussing about String permutation problem i.e., Calculating the possible permutations of a string.

- Problem statement:
- Generate all permutations of a String
- Input: ABC
- Output: ABC, ACB, BAC, CAB, BCA, CBA

Understanding the concept:

Permutations of a string means generating all possible words that could be formed using the characters in the provided string

For example:

If the string is AB, possible words would be AB and BA

If the string is ABC, possible combinations would be ABC, ACB, BAC, CAB, BCA, CBA

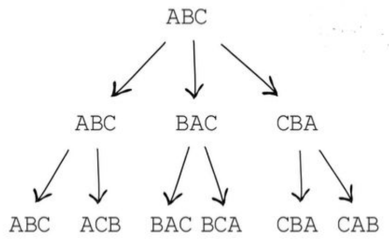
For the above string ABC, we want to generate all the possible variations where the character A is at the beginning of the word, then all the variations where A is at the middle of the word and when A is the last character in the word

The algorithm to achieve that

- Start from character A, and swap the character with A itself this means changing nothing in the string ABC
- Then swap the character A with B we get BAC
- Then swap A with C we get CBA
- Here we got all the possibilities of swapping with the other characters,

Here we will start from the second character from these [ABC, BAC, CBA] i.e., [B,A,B]
In ABC, (second character ‘B’)
Again, we will swap the character B with itself so swapping B with itself we get ABC,
Then B with C we get ACB
In BAC, (second character ‘A’)
Again, we will swap the character A with itself so swapping A with itself we get BAC,
Then A with C we get BCA
In CBA, (second character ‘B’)
Again, we will swap the character B with itself so swapping B with itself we get CBA,
Then B with A we get CAB

Now the third character, since it is the last character in the string this means that we already reached our solution, our solution is basically the last level
ABC, ACB, BAC, CAB, BCA, CBA



Here for string ABC, to generate all possible combinations required 3! Ways which is the numbers of the characters or items.
For **n** distinct items, the number of permutations would be n!

When we have duplicate items:

For example: consider ‘ABA’ here A is repeated 2 times, here we need to divide 3! by 2!

For a string of length n we have duplicate items, if one item is repeated m times, we need to

divide n! by m!

Input: ABA

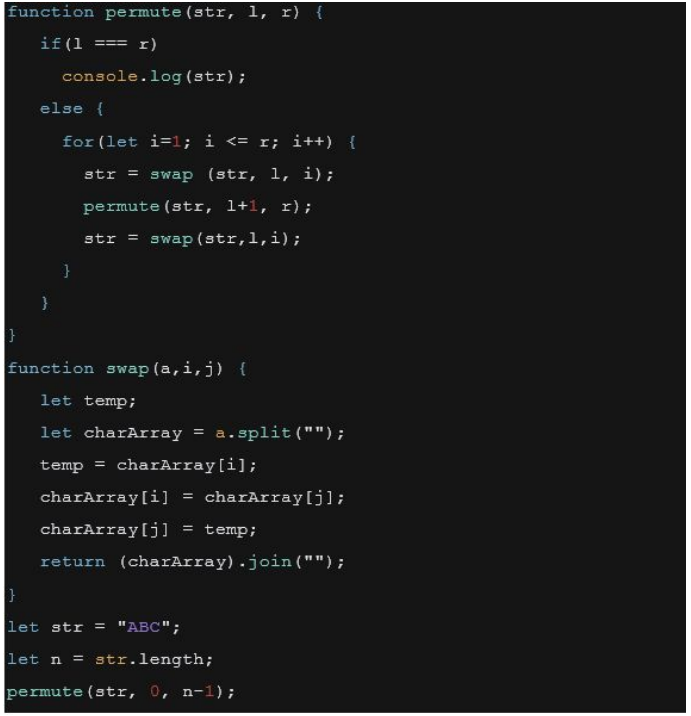
Output: ABA, AAB, BAA, BAA, ABA, AAB

Expected Output: ABA, BAA, AAB

In ‘ABA’, A is occurring twice so in permutation AAB if we replace A at 0th index and A at 1st index won’t produce different result so here AA can be arranged in 2! ways

3! / 2! = 3.

JavaScript program to print all permutations of a given string



```javascript
function permute(str, l, r) {
   if(l === r)
     console.log(str);
   else {
     for(let i=1; i <= r; i++) {
       str = swap (str, l, i);
       permute(str, l+1, r);
       str = swap(str,l,i);
     }
   }
}
function swap(a,i,j) {
   let temp;
   let charArray = a.split("");
   temp = charArray[i];
   charArray[i] = charArray[j];
   charArray[j] = temp;
   return (charArray).join("");
}
let str = "ABC";
let n = str.length;
permute(str, 0, n-1);
```

Output:

```
ABC
ACB
BAC
BCA
CBA
CAB
```

Code link: https://jsfiddle.net/1c39jews/

Practice H/W:

1. Write a JavaScript program to compute the exponent of a number. Note : The exponent of a number says how many times the base number is used as a factor.

   $8^2 = 8 \times 8 = 64$. Here 8 is the base and 2 is the exponent.

2. Write a JavaScript program for binary search.

   Sample array: [0,1,2,3,4,5,6]

   console.log(l.br_search(5)) will return '5'

Multiple Choice Questions

1. Choose the correct answer

a) Recursion is always better than iteration

b) Recursion uses more memory compared to iteration

c) Recursion uses less memory compared to iteration

d) Iterative function is always better and simpler to write than recursion

2. Which of the following statements is correct?

a) Iteration requires more system memory than recursion

b) Recursion requires more system memory than iteration

c) Both need equal amount of memory

d) Cannot be compared

3. In the absence of an exit condition in a recursive function, the following error is given

a) Compilation error

b) Run time error

c) Logical error

d) none

4. Choose the correct option

```
function something (int number) {
    if (number <= 0)
        return 1;
    else
        return number * something(number-1);
}
something (4);
```

a) 12

b) 24

c) 1

d) 0

5. Which of the following cannot be solved using recursion?

a) Tower of Hanoi

b) Fibonacci series

c) Tree Traversal

d) Problems without base case

Upcoming Class Teasers

Advanced Problem Solving on Recursion

- Rat In a Maze
- N Queen
- Sudoku Solver
- Knights tour
- Squareful array

THANK YOU