Permutations (medium)

We'll cover the following

Problem Statement

Given a set of distinct numbers, find all of its permutations.

Permutation is defined as the re-arranging of the elements of the set. For example, {1, 2, 3} has the following six permutations:

1. {1, 2, 3}
2. {1, 3, 2}
3. {2, 1, 3}
4. {2, 3, 1}
5. {3, 1, 2}
6. {3, 2, 1}

If a set has 'n' distinct elements it will have $n!$ permutations.

Example 1:

```
Input: [1,3,5]
Output: [1,3,5], [1,5,3], [3,1,5], [3,5,1], [5,1,3], [5,3,1]
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
const find_permutations = function(nums) {
  result = [];
  // TODO: Write your code here
  return result;
};


console.log(`Here are all the permutations: ${find_permutations([1, 3, 5])}`)

```

RUN SAVE RESET

Solution

This problem follows the Subsets pattern and we can follow a similar **Breadth First Search (BFS)** approach. However, unlike Subsets, every permutation must contain all the numbers.

Let's take the example-1 mentioned above to generate all the permutations. Following a BFS approach, we will consider one number at a time:

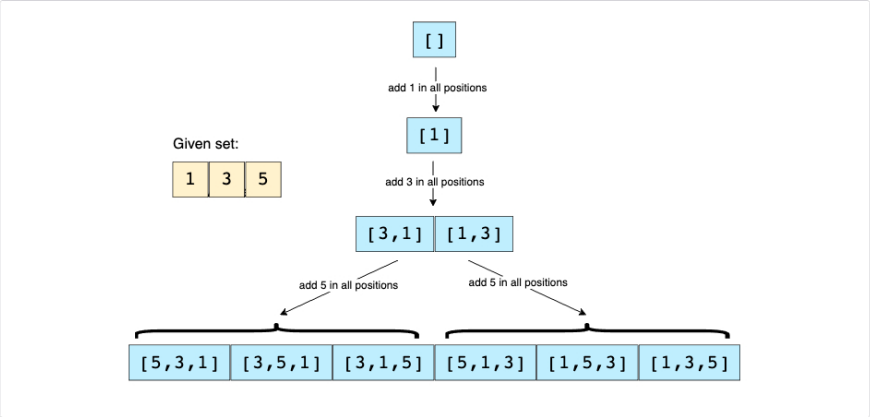
1. If the given set is empty then we have only an empty permutation set: []
2. Let's add the first element (1), the permutations will be: [1]
3. Let's add the second element (3), the permutations will be: [3,1], [1,3]
4. Let's add the third element (5), the permutations will be: [5,3,1], [3,5,1], [3,1,5], [5,1,3], [1,5,3], [1,3,5]

Let's analyze the permutations in the 3rd and 4th step. How can we generate permutations in the 4th step from the permutations of the 3rd step?

If we look closely, we will realize that when we add a new number (5), we take each permutation of the previous step and insert the new number in every position to generate the new permutations. For example, inserting '5' in different positions of [3,1] will give us the following permutations:

1. Inserting '5' before '3': [5,3,1]
2. Inserting '5' between '3' and '1': [3,5,1]
3. Inserting '5' after '1': [3,1,5]

Here is the visual representation of this algorithm:



Code

Here is what our algorithm will look like:

Java Python3 C++ JS

```
permutations.push([]);
for (let i = 0; i < nums.length; i++) {
  const currentNumber = nums[i];
  // we will take all existing permutations and add the current number to create new permutations
  const n = permutations.length;
  for (let p = 0; p < n; p++) {
    const oldPermutation = permutations.shift();
    // create a new permutation by adding the current number at every position
    for (let j = 0; j < oldPermutation.length + 1; j++) {
      const newPermutation = oldPermutation.slice(0); // clone the permutation
      newPermutation.splice(j, 0, currentNumber); // insert currentNumber at index 'j'
      if (newPermutation.length === numsLength) {
        result.push(newPermutation);
      } else {
        permutations.push(newPermutation);
      }
    }
  }
}

return result;
}

console.log('Here are all the permutations: find_permutations(nums: any): any[]
const result = find_permutations([1, 3, 5,6]);
result.forEach((permutation) => {
  console.log(permutation);
});
```

RUN SAVE * RESET

Close

Output 5.473s

```
Here are all the permutations:
[ 6, 5, 3, 1 ]
[ 5, 6, 3, 1 ]
[ 5, 3, 6, 1 ]
[ 5, 3, 1, 6 ]
[ 6, 3, 5, 1 ]
[ 3, 6, 5, 1 ]
[ 3, 5, 6, 1 ]
[ 3, 5, 1, 6 ]
```

Time complexity

We know that there are a total of $N!$ permutations of a set with 'N' numbers. In the algorithm above, we are iterating through all of these permutations with the help of the two 'for' loops. In each iteration, we go through all the current permutations to insert a new number in them on line 17 (line 23 for C++ solution). To insert a number into a permutation of size 'N' will take $O(N)$, which makes the overall time complexity of our algorithm $O(N * N!)$.

Space complexity

All the additional space used by our algorithm is for the result list and the queue to store the intermediate permutations. If you see closely, at any time, we don't have more than $N!$ permutations between the result list and the queue. Therefore the overall space complexity to store $N!$ permutations each containing N elements will be $O(N * N!)$.

Recursive Solution

Here is the recursive algorithm following a similar approach:

Mark Course as Completed

Java | Python3 | C++ | JS

```
function generate_permutations(nums) {
  const result = [];
  generate_permutations_recursive(nums, 0, [], result);
  return result;
}


function generate_permutations_recursive(nums, index, currentPermutation, result) {
  if (index === nums.length) {
    result.push(currentPermutation);
  } else {
    // create a new permutation by adding the current number at every position
    for (let i = 0; i < currentPermutation.length + 1; i++) {
      newPermutation = currentPermutation.slice(0); // clone the permutation
      newPermutation.splice(i, 0, nums[index]); // insert nums[index] at index 'i'
      generate_permutations_recursive(nums, index + 1, newPermutation, result);
    }
  }
}

console.log('Here are all the permutations:');
const result = generate_permutations([1, 3, 5]);
result.forEach((permutation) => {
  console.log(permutation);
});
```

RUN SAVE RESET

Close

Output 3.585s

```
Here are all the permutations:
[ 5, 3, 1 ]
[ 3, 5, 1 ]
[ 3, 1, 5 ]
[ 5, 1, 3 ]
[ 1, 5, 3 ]
[ 1, 3, 5 ]
```


MARK AS COMPLETED