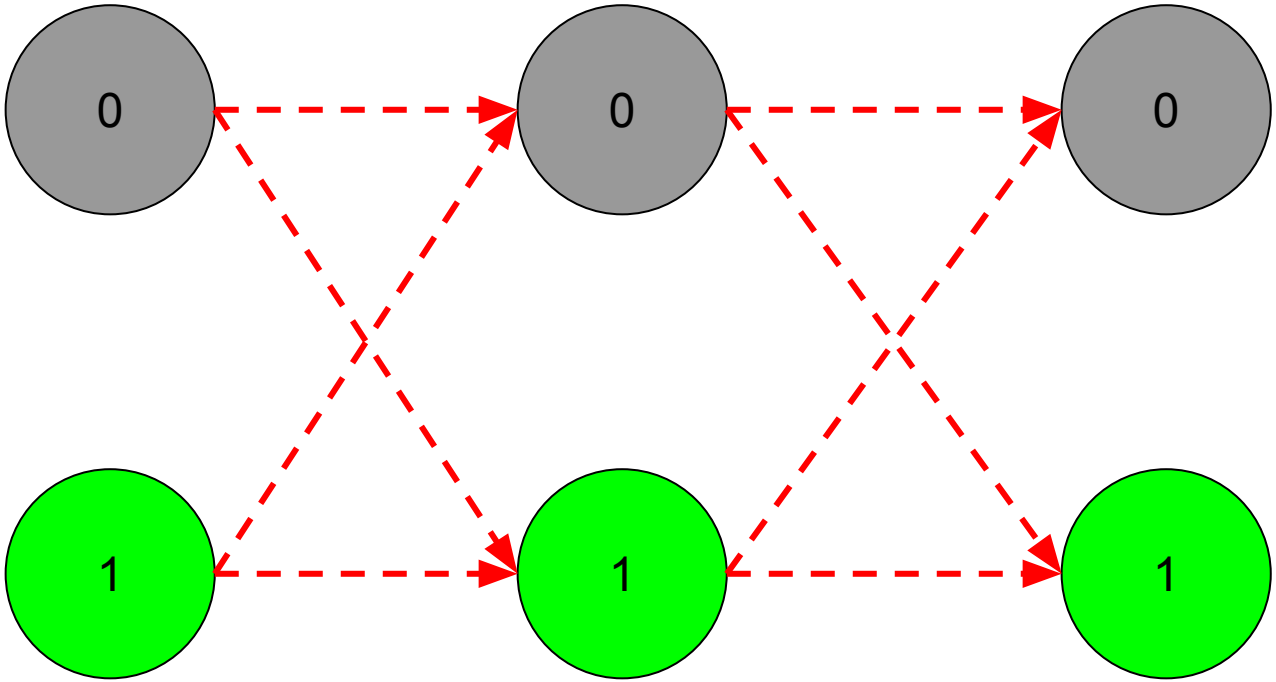
Permutation and backtracking

CSCI 335, 2021 SUMMER

Binary numbers



Why 3 bits has 2^3 possibilities?

Each bit has 2 options and 3 bit is 2^3

000

001

…

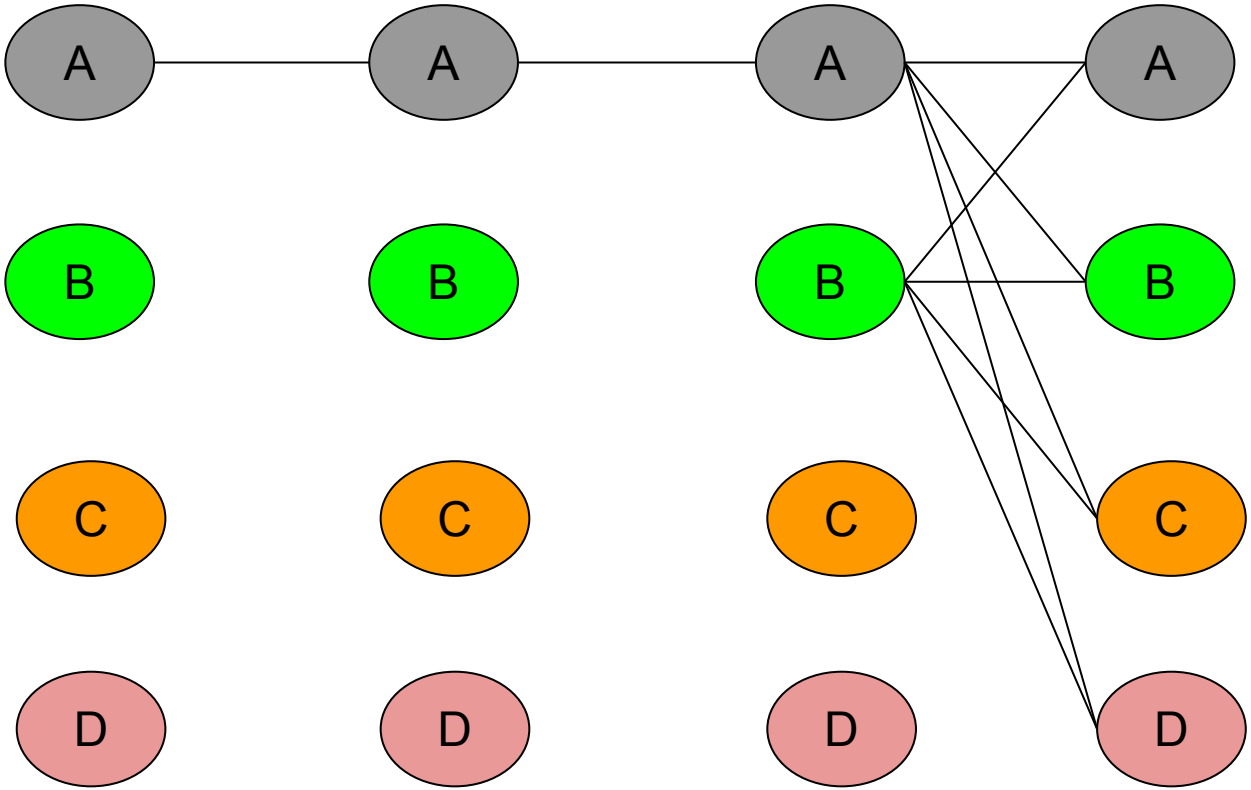
111

Base 4 system

What if we have a base 4 system and has 4 symbols?

Let’s use ABCD to build this base 4 system

Base 4 system with 4 “BITs”

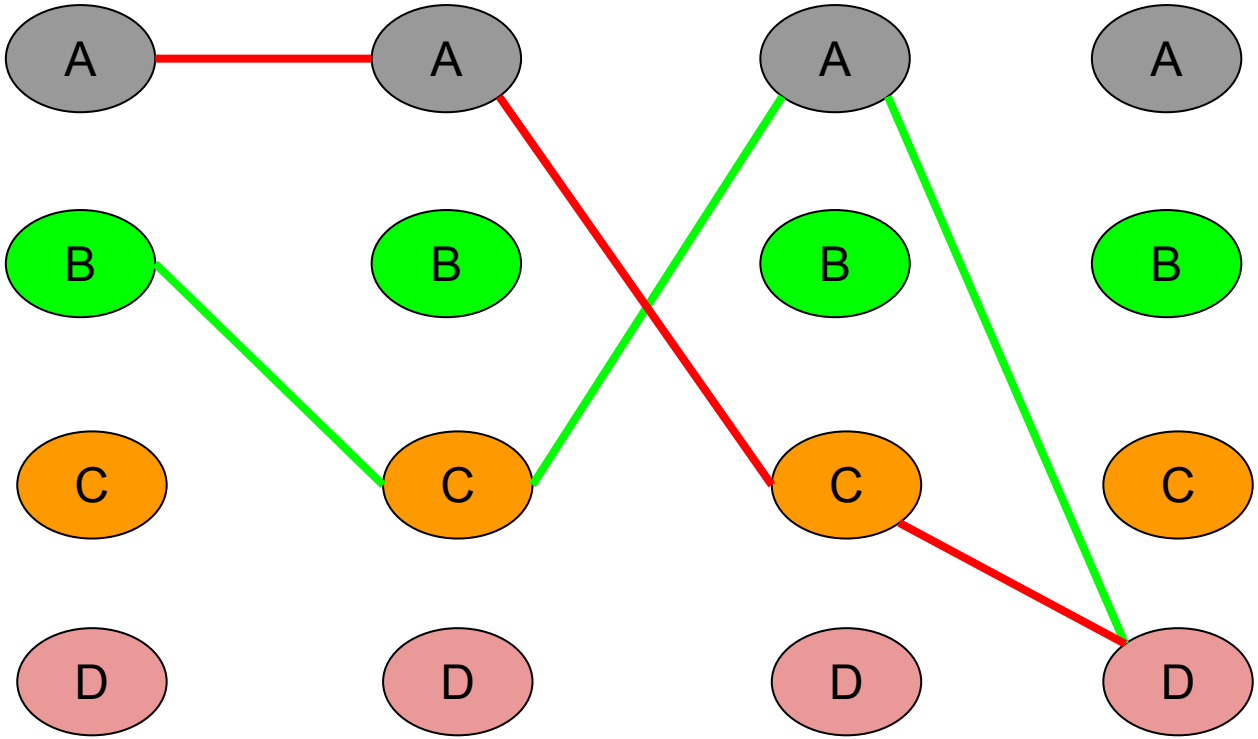


The plot is incomplete.

Yes.

We have 4^4 cases

Permutation of(ABCD) = 4!



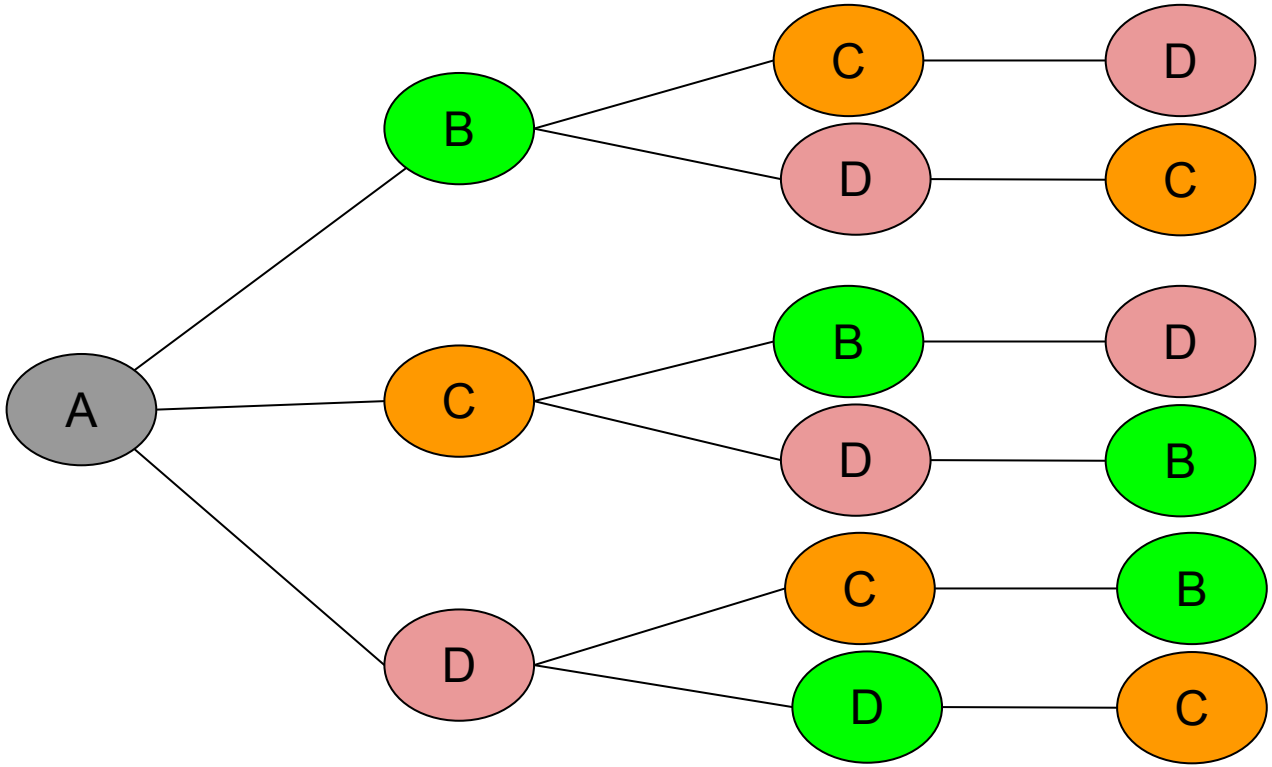
Permutation is based on this base 4 system.

However, repeating is not allowed.

So AACD is invalid

BCAD is good

Why 4! = 4*3*2*1



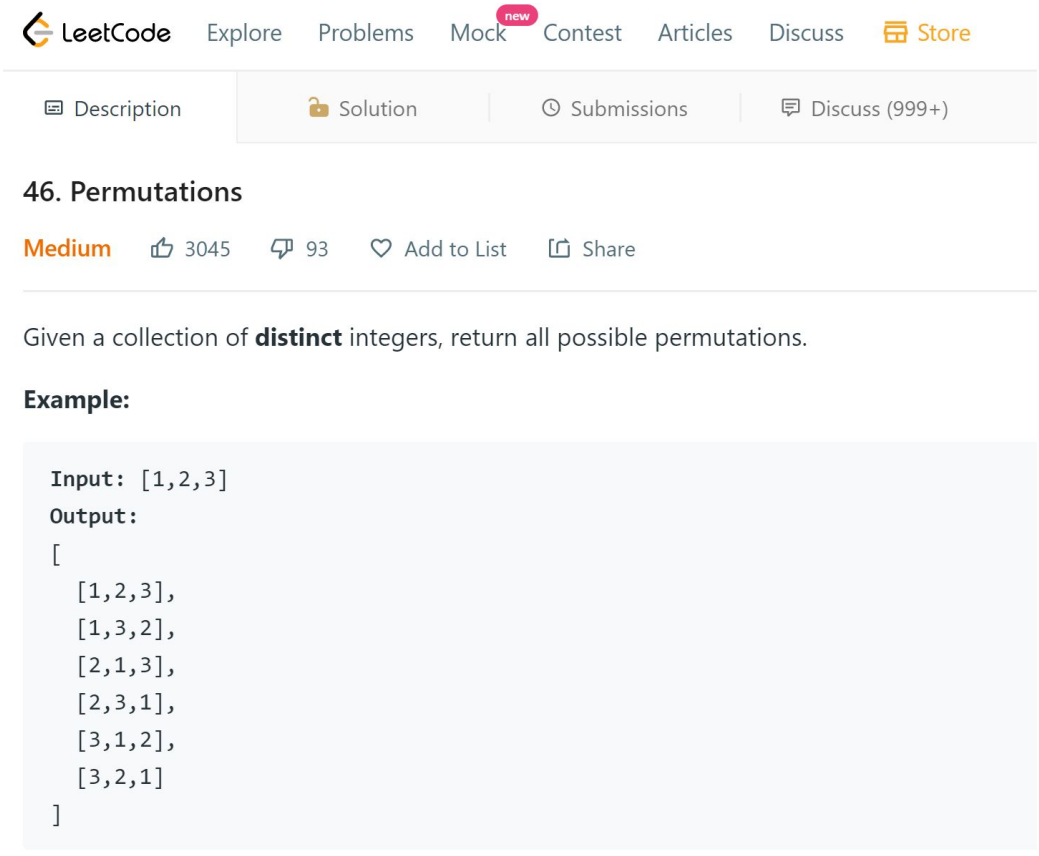
First step we have 4 choices: A, B,C or D

Second step we have 3 choices

…

Fist step pick up A is shown left

Solution of the permutation problem from LC



Solution 1: Naive enumerating (time complexity O(n^n))

Naive enumerate all possibilities:

- If no repeating, collect this result
- If it has repeating, do not collect this result.

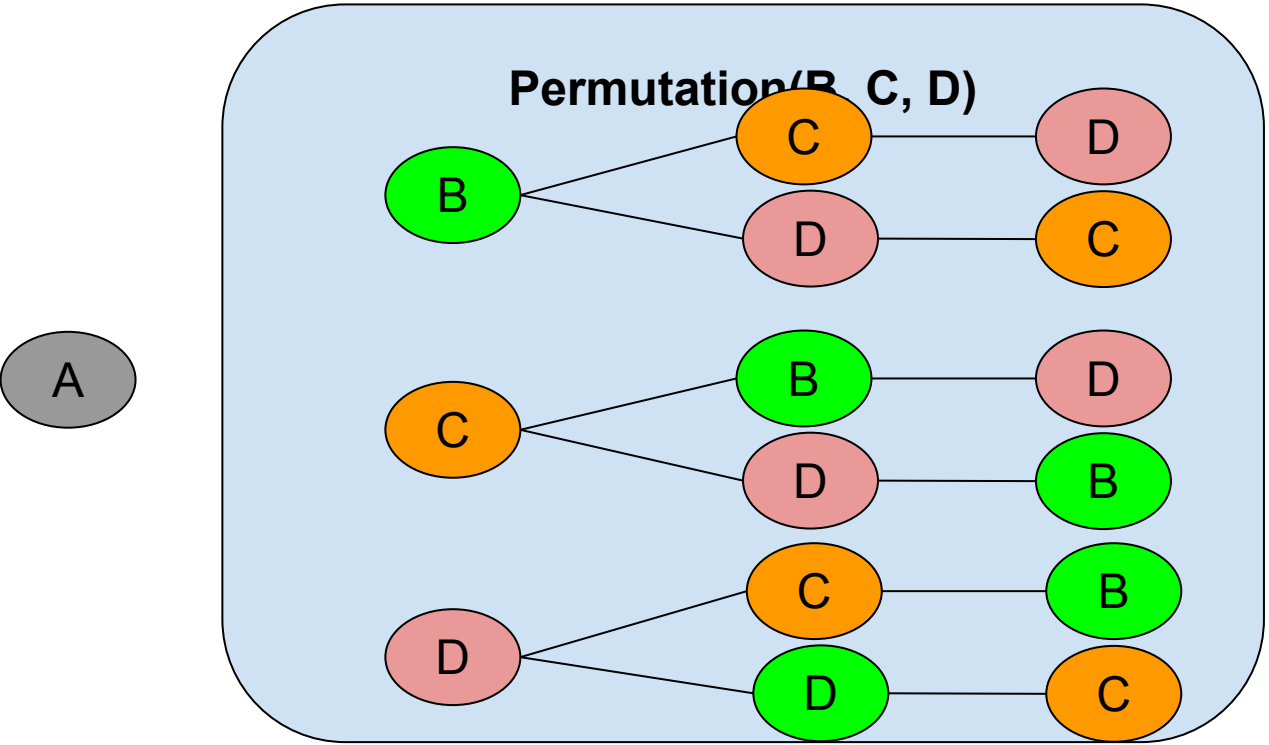
Time complexity (O(n^n))

```cpp
class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        // productions may have repeatings. It is supposed to have n^n elements.
        vector<vector<int>> productions={{}};
        vector<vector<int>> permutations;
        // iteratively generate productions.
        for (int i=0;i<nums.size();++i){
            vector<vector<int>> new_productions;
            for(auto &num:nums){
                 for(auto& prod:productions){
                    vector<int> new_prod = prod;
                    new_prod.push_back(num);
                    new_productions.push_back(new_prod);
                 }
            }
            productions = new_productions;
        }
        // generate the final results by removing results which has repeating.
       for(auto&prod:productions){
           // compare the size of set s to nums.size(). If s.size()==nums.size(), there is no repeating.
            unordered_set<int> s(prod.begin(), prod.end());
            if (s.size()==nums.size()){
                vector<int> permu(prod.begin(), prod.end());
                permutations.push_back(permu);
            }
       }
    return permutations;
}
};
```

Python code for reference

```
"""
jimmy shen
02/20/2020
A code to implement the naive o(n^n) permutation algorithm.
runtime 72 ms
time complexity O(n^n)
space complexity O(n)
"""

class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        return [perm for perm in itertools.product(*[nums]*len(nums)) if len(perm)==len(set(perm))]
```

Recursion



Permutation(A,B,C,D)
=

A with
Permutation(B,C,D)

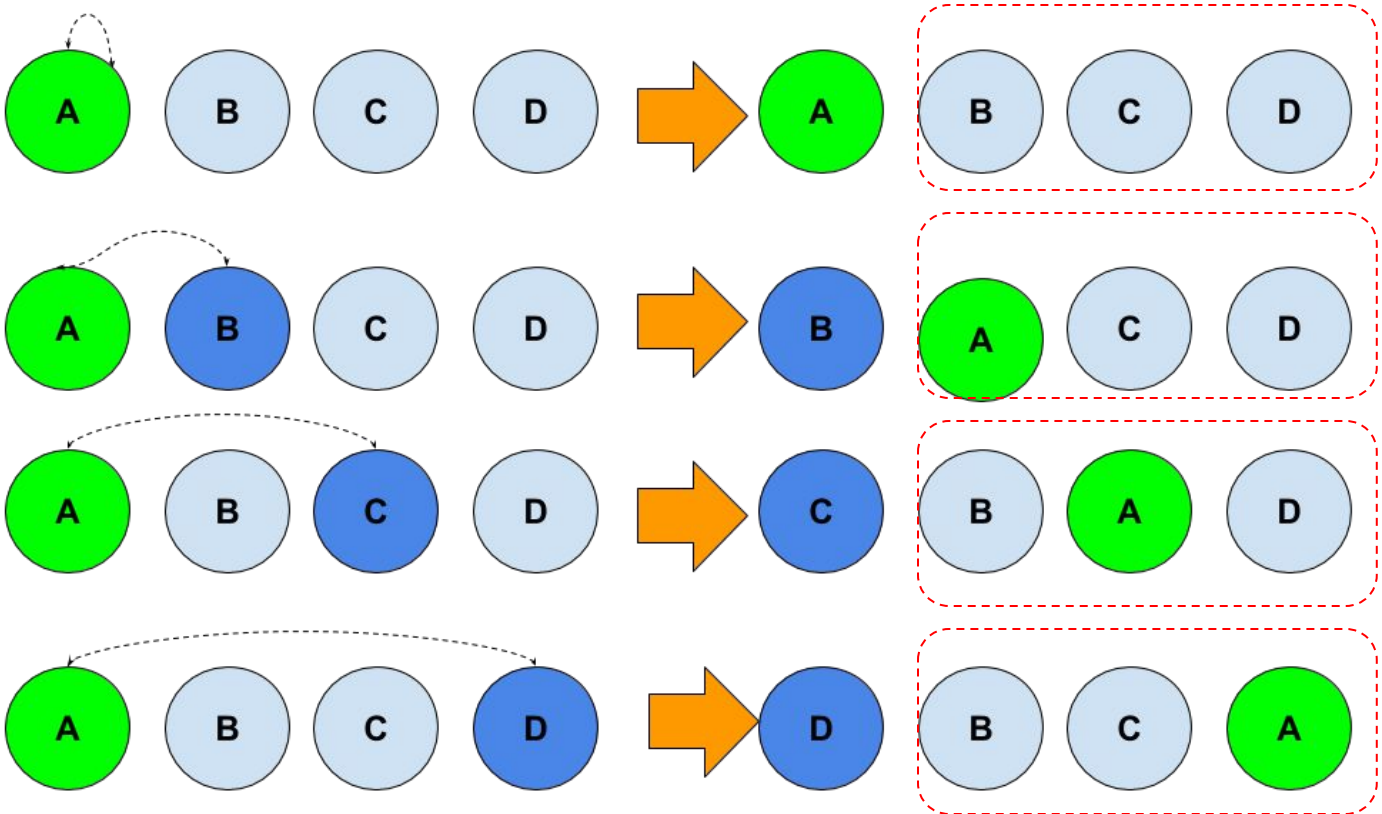
+B with
Permutation(A,C,D)

+C with

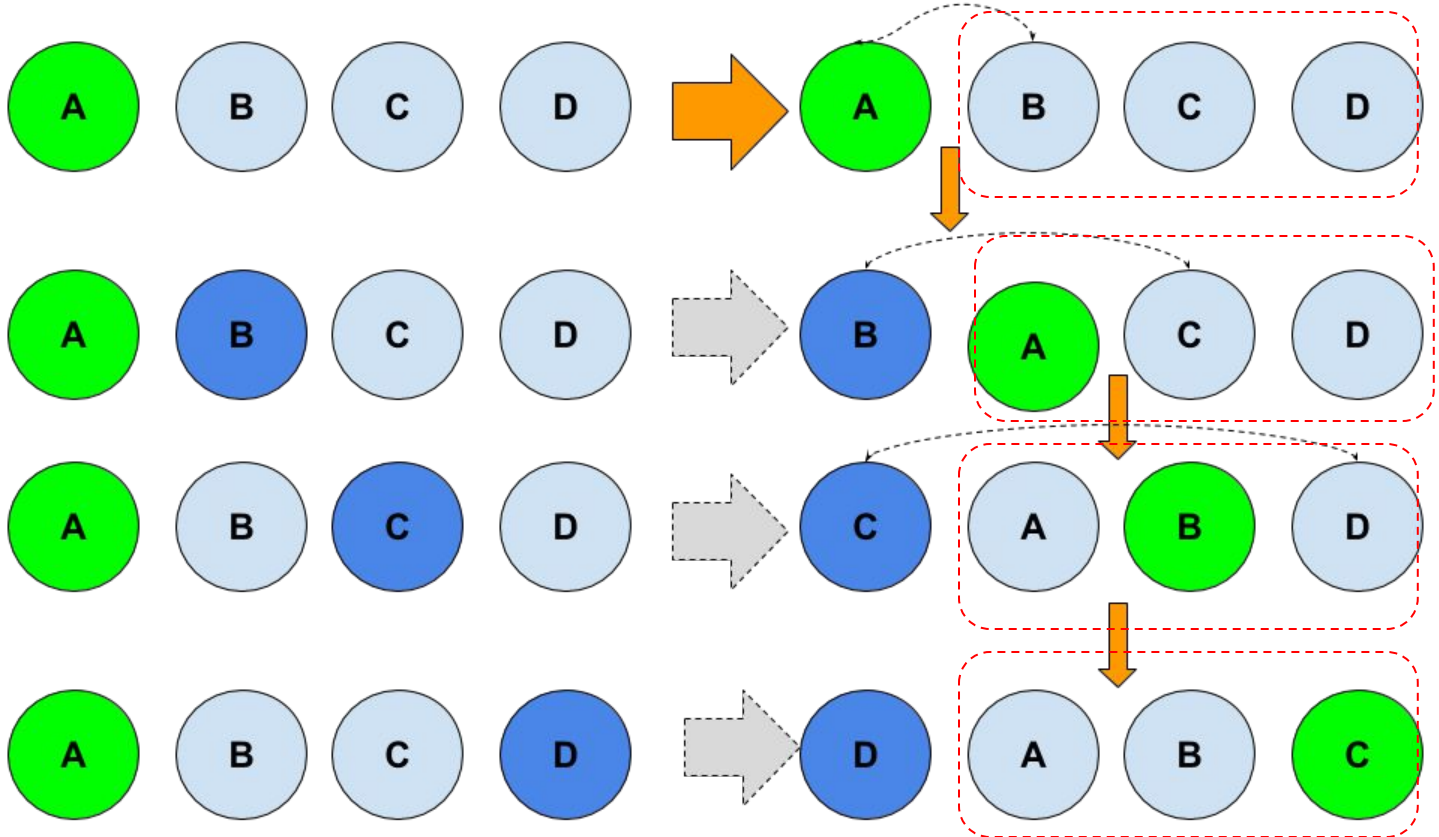
Permutation(A,B,D)

+D with
Permutation(A,B,C)

Naive swap to get a subproblem



Using a smarter way to keep the original order

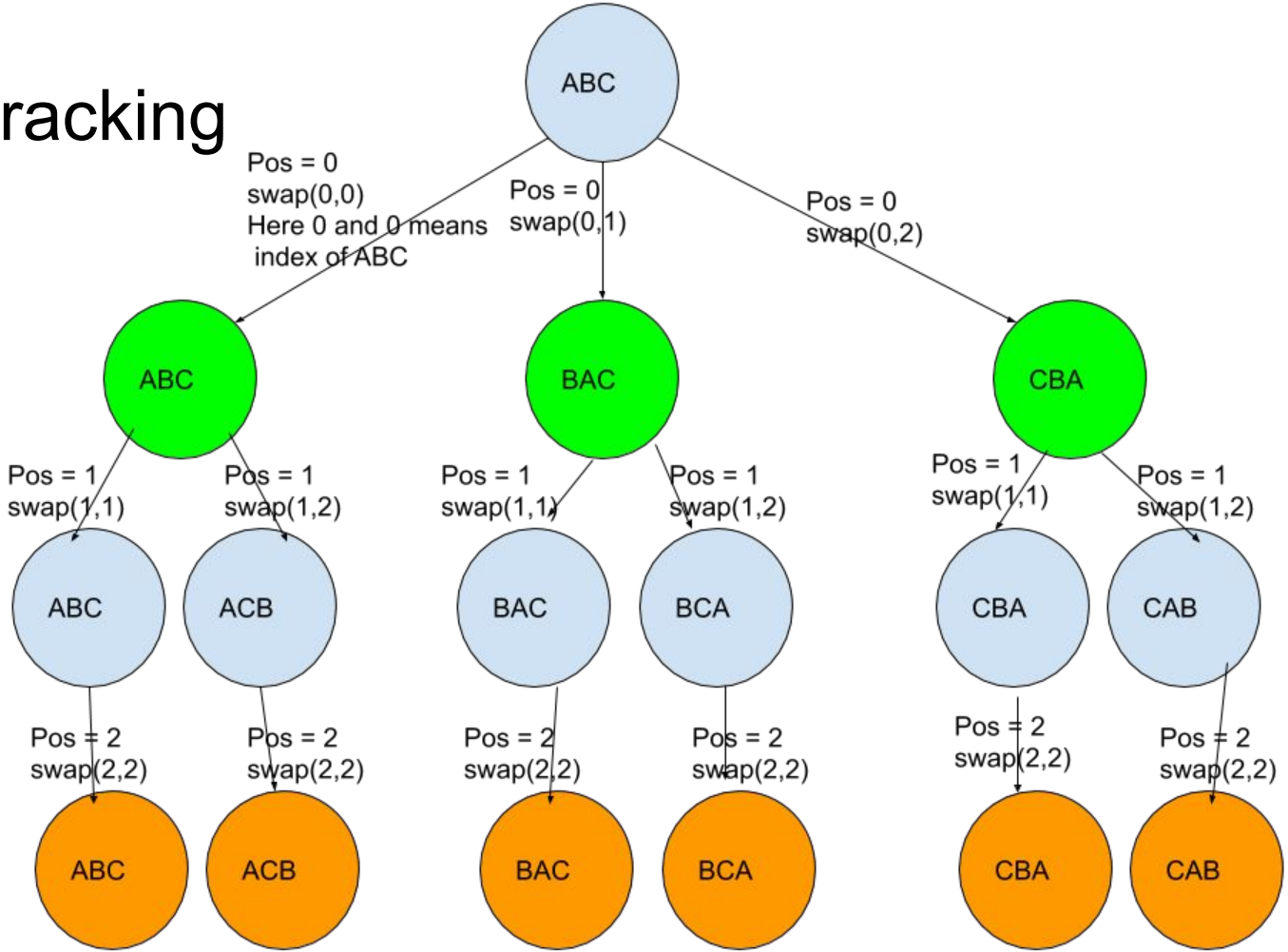


```cpp
/*
jimmy shen
02/20/2020
A code to implement the naive o(n^n) permutation algorithm.
runtime 12 ms
time complexity O(n!)
space complexity O(n) as we have temp
*/

class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>> res;
        recursive_permute(nums, res, 0);
        return res;
    }
    void recursive_permute(vector<int>&nums, vector<vector<int>> &res, int pos){
        // if we reach the size of nums, we are done.
        vector<int> temp = nums;
        if(pos == temp.size()-1){
            res.push_back(temp);
            return;
        }
        else{
            for(int i=pos; i<temp.size(); i++){
                swap(temp[pos], temp[i]);
                recursive_permute(temp, res, pos+1);
            }
        }
    }
};
```

Python

```python
"""
jimmy shen
02/20/2020
A code to implement the recursion algorithm.
runtime 40 ms
time complexity O(n!)
"""

class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        if len(nums) <=1:return [nums]
        return [[n]+p
                for i, n in enumerate(nums)
                for p in self.permute(nums[:i]+nums[i+1:]) ]
```

Backtracking



```cpp
/*
jimmy shen
02/20/2020
A code to implement the naive o(n^n) permutation algorithm.
runtime 12 ms
time complexity O(n!)
space complexity O(1)
*/

class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>> res;
        dfs(nums, res, 0);
        return res;
    }
    void dfs(vector<int>&nums, vector<vector<int>> &res, int pos){
        // if we reach the size of nums, we are done.
        if(pos >= nums.size()){
            res.push_back(nums);
            return;
        }
        else{
            for(int i=pos; i<nums.size(); i++){
                swap(nums[pos], nums[i]);
                dfs(nums, res, pos+1);
                //recover the nums to do backtracking
                swap(nums[pos], nums[i]);
            }
        }
    }
};
```

Python

```python
"""
jimmy shen
02/20/2020
A code to implement the recursive with backtracking algorithm.
runtime 40 ms
time complexity O(n!)
"""

class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        res = []
        def dfs(pos):
            if pos==len(nums)-1:
                # using deep copy here to harvest the result
                res.append(nums[:])
            for i in range(pos, len(nums)):
                # swap
                nums[pos], nums[i] = nums[i], nums[pos]
                dfs(pos+1)
                nums[pos], nums[i] = nums[i], nums[pos]
        dfs(0)
        return res
```

Try to solve this one by yourself (Attention, this problem is hard)

https://leetcode.com/problems/n-queens/

Thanks