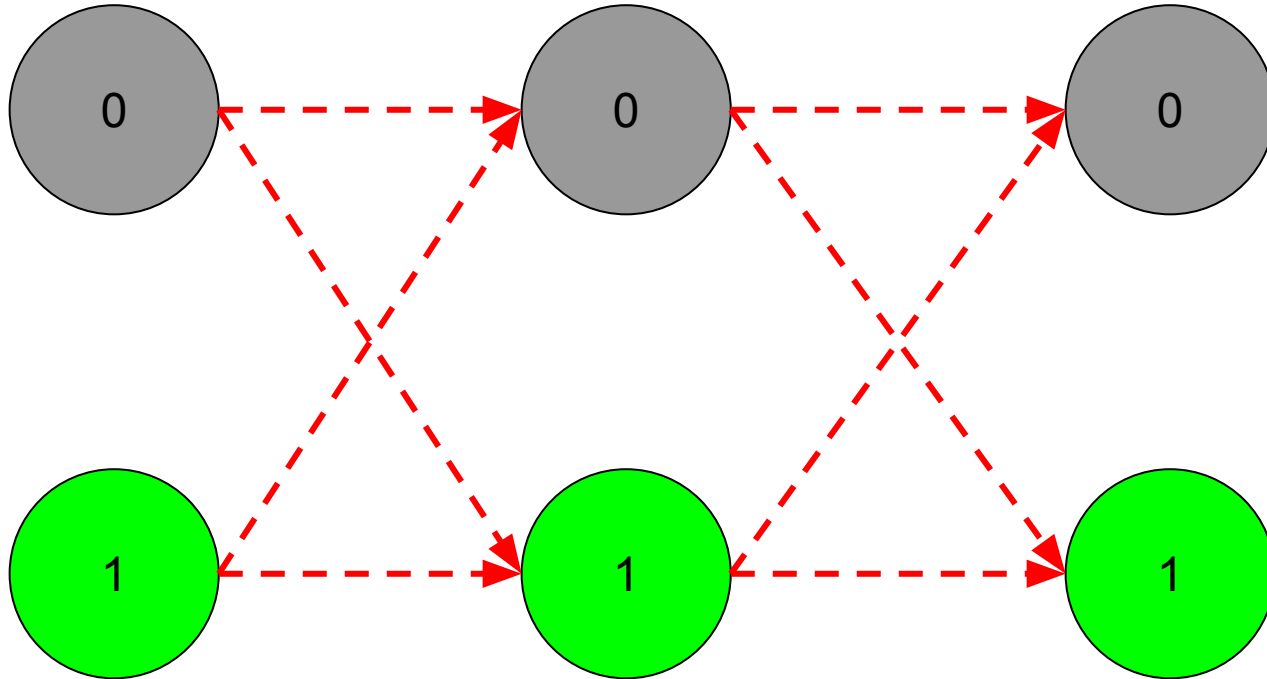


Solutions for the permutation problem

Binary numbers



Why 3 bits has 2^3 possibilities?

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

000

001

...

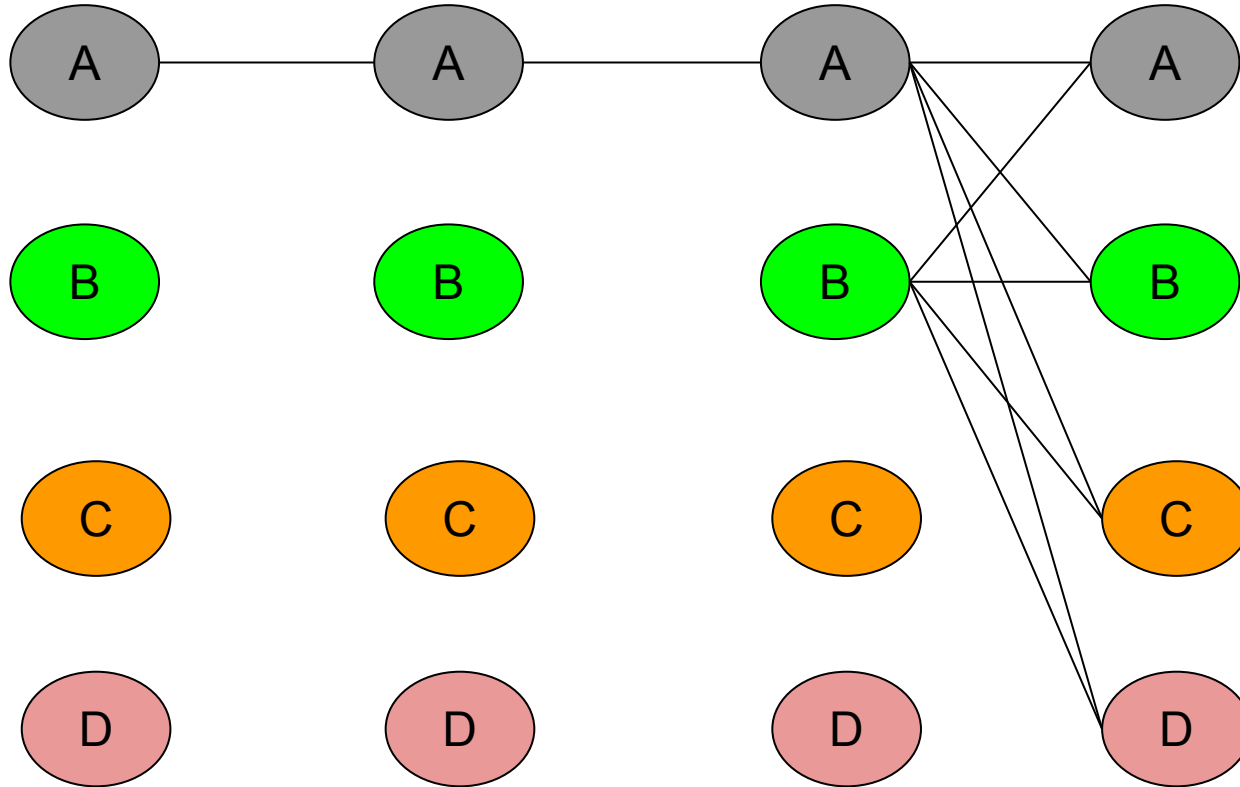
111

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”

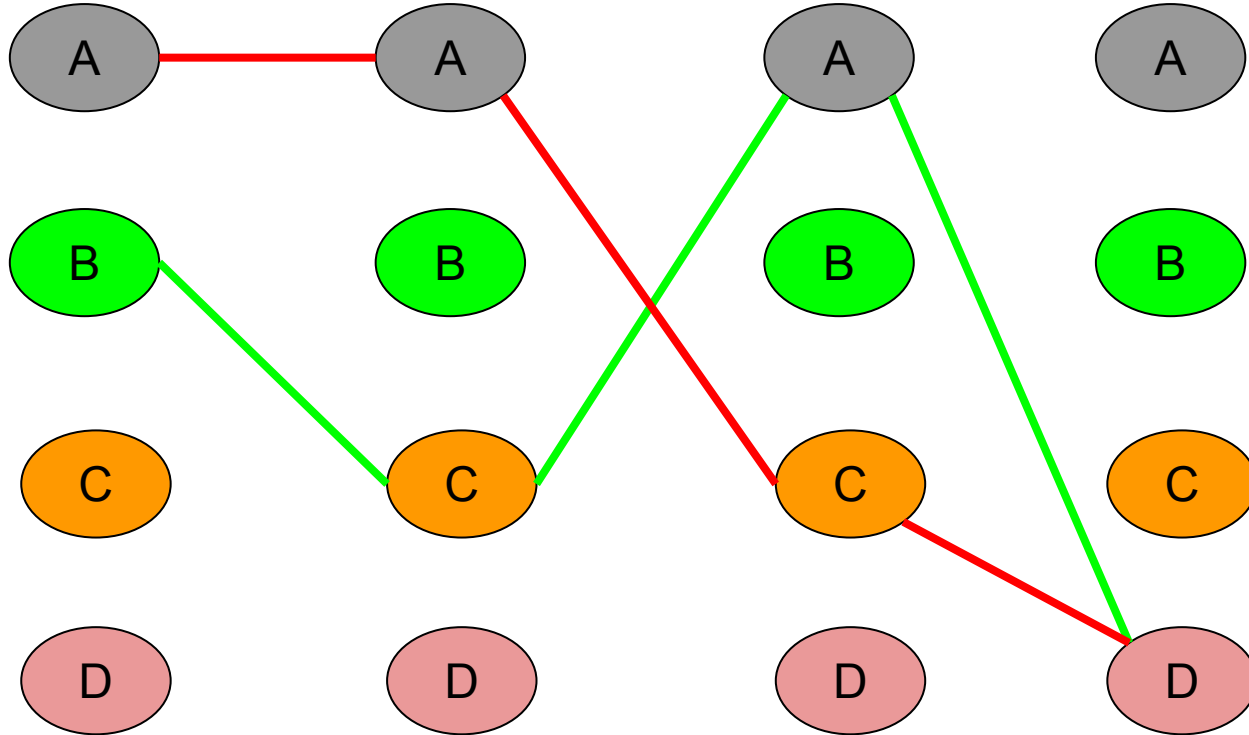


The plot is incomplete.

Yes.

We have 4^4 cases

Permutation of(ABCD) = 4!



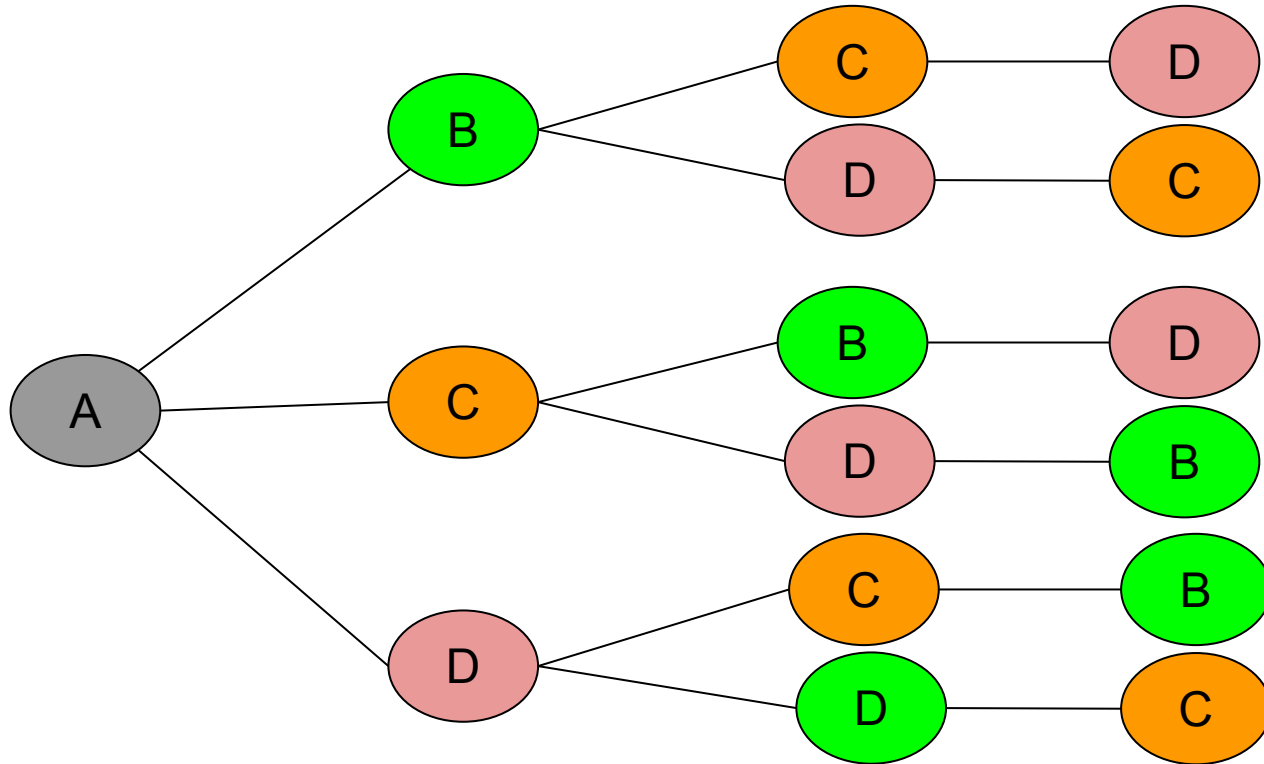
Permutation is based on this base 4 system.

However, repeating is not allowed.

So AACD is invalid

BCAD is good

Why $4! = 4 \times 3 \times 2 \times 1$




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

Second step we have 3 choices

...

First step pick up A is shown left

Solution of a similar problem of HW1 permutation

 LeetCode

[Explore](#) [Problems](#) [Mock](#) [Contest](#) [Articles](#) [Discuss](#) [Store](#)

[Description](#) [Solution](#) [Submissions](#) [Discuss \(999+\)](#)

46. Permutations

Medium [3045](#) [93](#) [Add to List](#) [Share](#)

Given a collection of **distinct** integers, return all possible permutations.

Example:

Input: [1,2,3]

Output:

```
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

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 ($O(n^n)$)


```

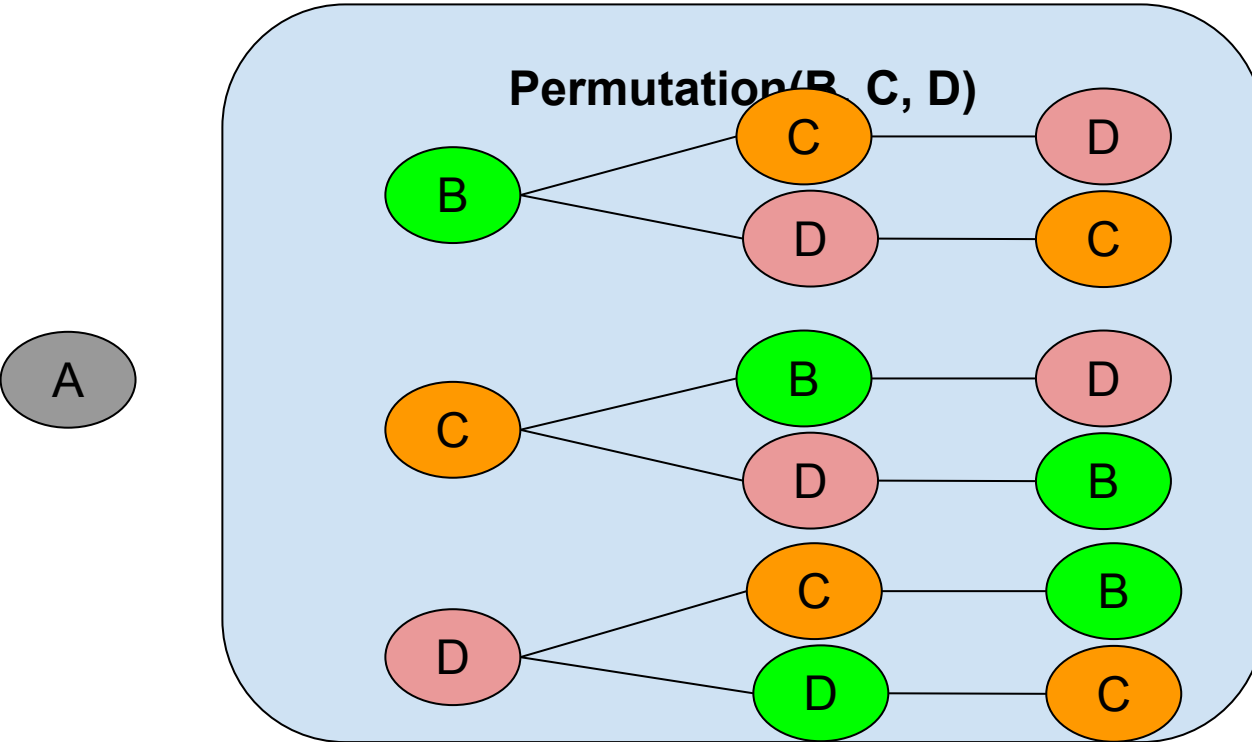
10 ▾ class Solution {
11     public:
12 ▾         vector<vector<int>> permute(vector<int>& nums) {
13             // productions may have repeatings. It is supposed to have n^n elements.
14             vector<vector<int>> productions={{}};
15             vector<vector<int>> permutations;
16             // iteratively generate productions.
17 ▾             for (int i=0;i<nums.size();++i){
18                 vector<vector<int>> new_productions;
19 ▾                 for(auto &num:nums){
20 ▾                     for(auto& prod:productions){
21                         vector<int> new_prod = prod;
22                         new_prod.push_back(num);
23                         new_productions.push_back(new_prod);
24                     }
25                 }
26                 productions = new_productions;
27             }
28             // generate the final results by removing results which has repeating.
29 ▾             for(auto&prod:productions){
30                 // compare the size of set s to nums.size(). If s.size()==nums.size(), there is no repeating.
31                 unordered_set<int> s(prod.begin(), prod.end());
32 ▾                 if (s.size()==nums.size()){
33                     vector<int> permu(prod.begin(), prod.end());
34                     permutations.push_back(permu);
35                 }
36             }
37             return permutations;
38         }
39     };

```

Python code for reference

```
1  """
2  jimmy shen
3  02/20/2020
4  A code to implement the naive  $O(n^n)$  permutation algorithm.
5  runtime 72 ms
6  time complexity  $O(n^n)$ 
7  space complexity  $O(n)$ 
8  """
9
10 class Solution:
11     def permute(self, nums: List[int]) -> List[List[int]]:
12         return [perm for perm in itertools.product(*[nums]*len(nums)) if len(perm)==len(set(perm))]
```

Recursion



Permutation(A,B,C,D)

=

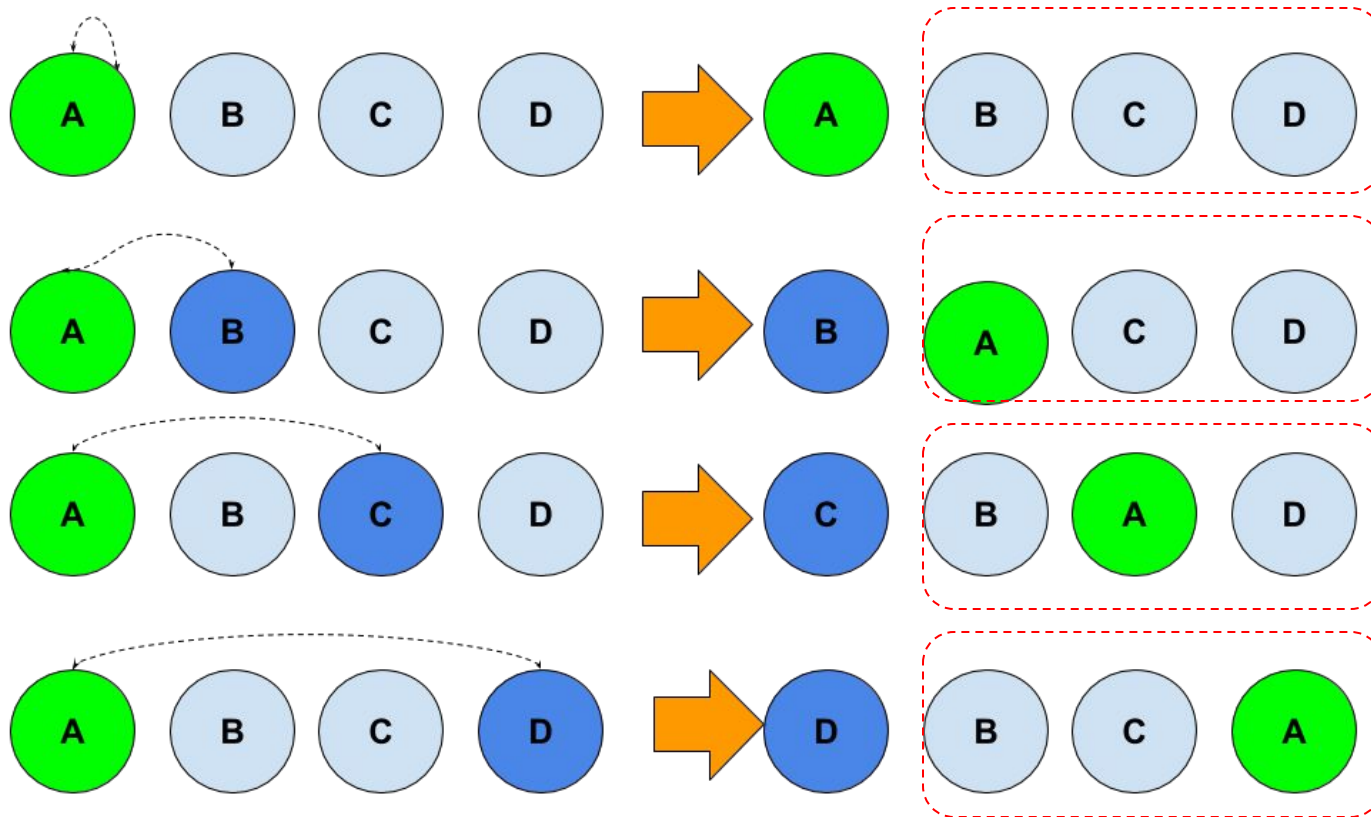
A with
Permutation(B,C,D)

+B with
Permutation(A,C,D)

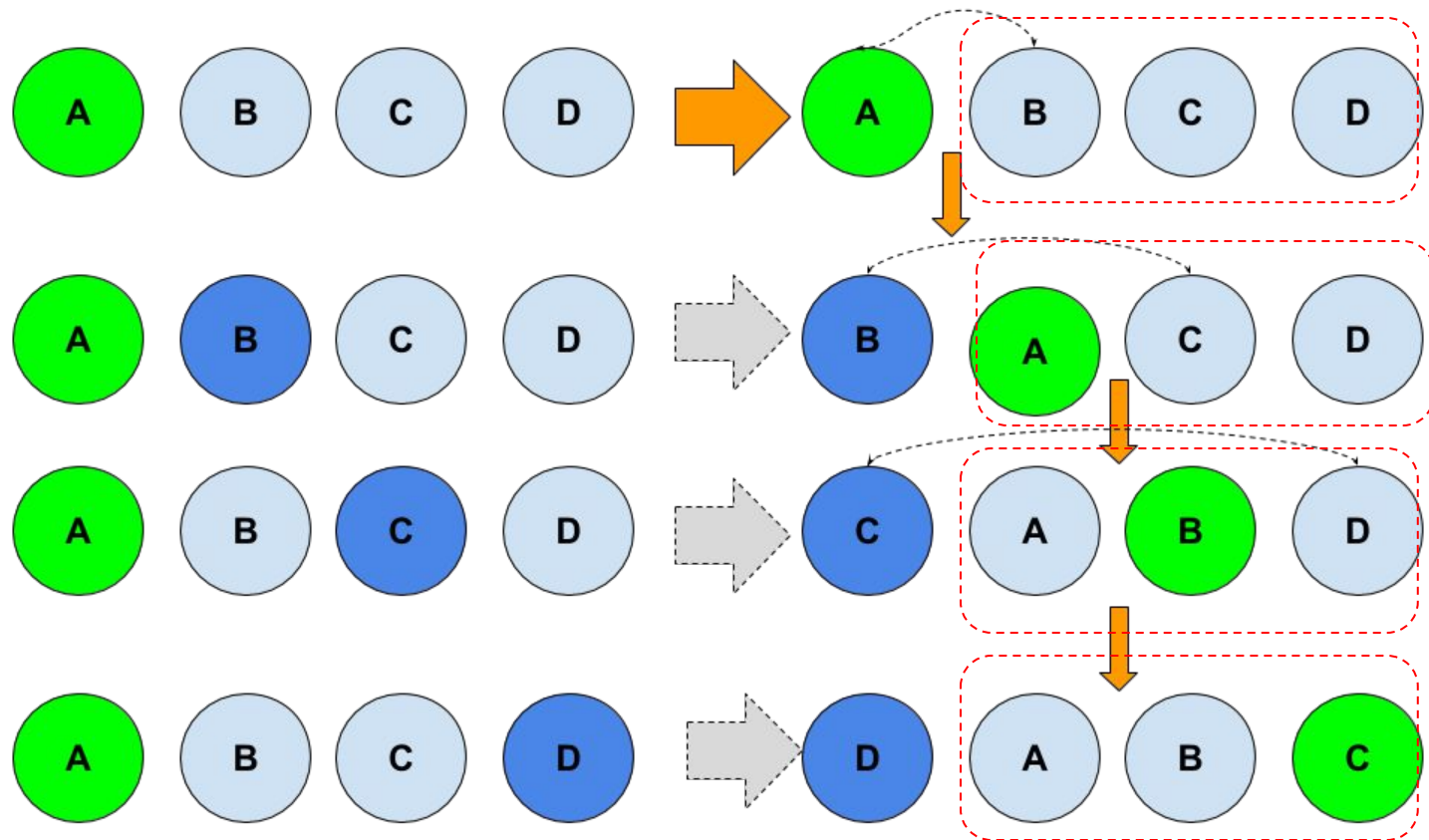
+C with
Permutation(A,B,D)

+D with
Permutation(A,B,C)

Naive swap to get a subproblem



Using a smarter way to keep the original order



```

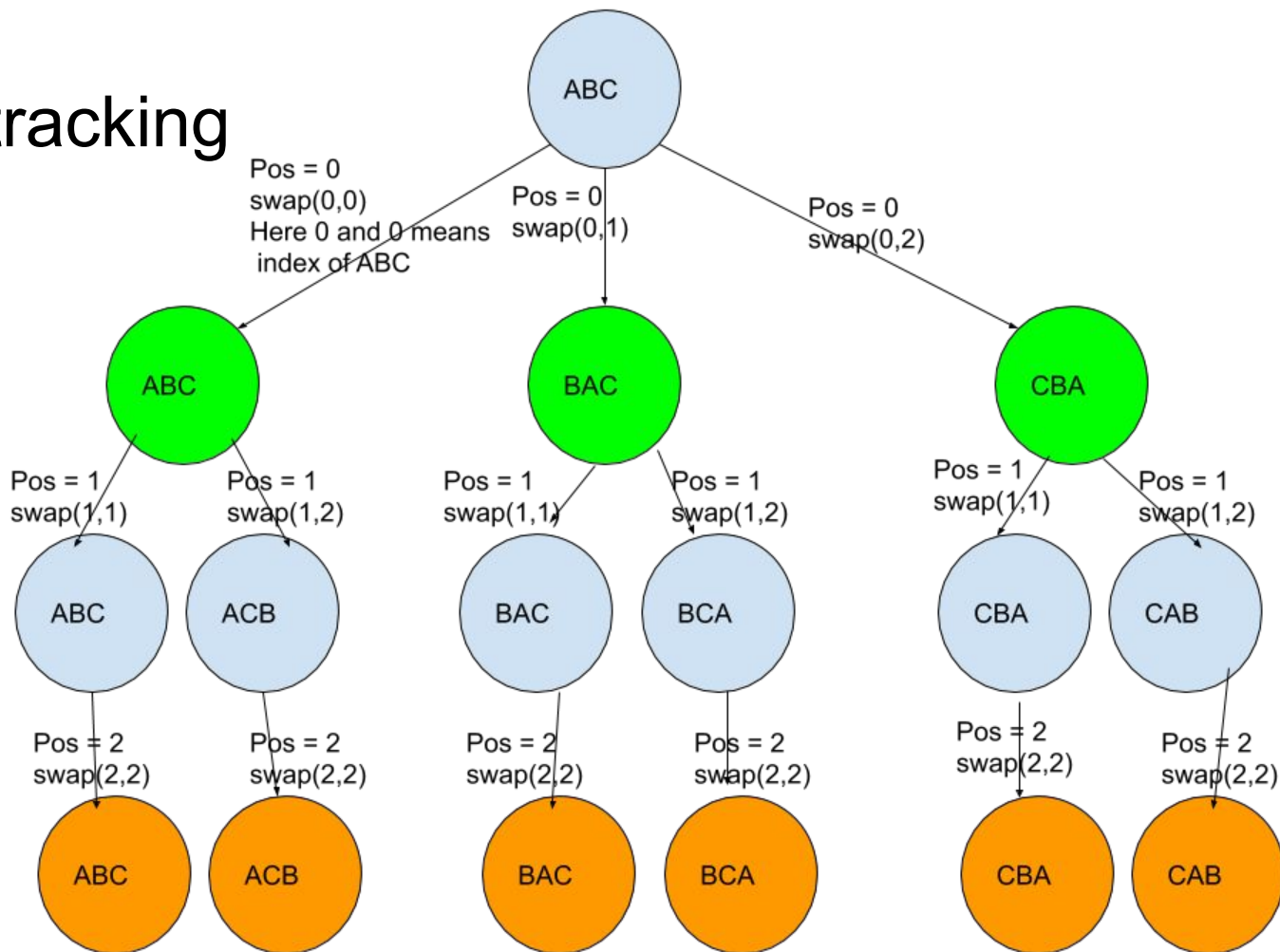
1  /*
2  jimmy shen
3  02/20/2020
4  A code to implement the naive  $O(n^n)$  permutation algorithm.
5  runtime 12 ms
6  time complexity  $O(n!)$ 
7  space complexity  $O(n)$  as we have temp
8  */
9
10 class Solution {
11 public:
12     vector<vector<int>> permute(vector<int>& nums) {
13         vector<vector<int>> res;
14         recursive_permute(nums, res, 0);
15         return res;
16     }
17     void recursive_permute(vector<int>&nums, vector<vector<int>> &res, int pos){
18         // if we reach the size of nums, we are done.
19         vector<int> temp = nums;
20         if(pos == temp.size()-1){
21             res.push_back(temp);
22             return;
23         }
24         else{
25             for(int i=pos; i<temp.size(); i++){
26                 swap(temp[pos], temp[i]);
27                 recursive_permute(temp, res, pos+1);
28             }
29         }
30     }
31 };
32

```

Python

```
1  """
2  jimmy shen
3  02/20/2020
4  A code to implement the recursion algorithm.
5  runtime 40 ms
6  time complexity O(n!)
7  """
8
9  class Solution:
10     def permute(self, nums: List[int]) -> List[List[int]]:
11         if len(nums) <= 1: return [nums]
12         return [[n]+p
13                 for i, n in enumerate(nums)
14                 for p in self.permute(nums[:i]+nums[i+1:])] ]
15
```

Backtracking




```

1  ▾  /*
2     jimmy shen
3     02/20/2020
4     A code to implement the naive  $O(n^n)$  permutation algorithm.
5     runtime 12 ms
6     time complexity  $O(n!)$ 
7     space complexity  $O(1)$ 
8     */
9
10 ▾  class Solution {
11     public:
12 ▾      vector<vector<int>> permute(vector<int>& nums) {
13          vector<vector<int>> res;
14          dfs(nums, res, 0);
15          return res;
16      }
17 ▾      void dfs(vector<int>&nums, vector<vector<int>> &res, int pos){
18          // if we reach the size of nums, we are done.
19 ▾          if(pos >= nums.size()){
20              res.push_back(nums);
21              return;
22          }
23 ▾          else{
24 ▾              for(int i=pos; i<nums.size(); i++){
25                  swap(nums[pos], nums[i]);
26                  dfs(nums, res, pos+1);
27                  //recover the nums to do backtracking
28                  swap(nums[pos], nums[i]);
29              }
30          }
31      }
32  };

```

Python

```
1  """
2  jimmy shen
3  02/20/2020
4  A code to implement the recursive with backtracking algorithm.
5  runtime 40 ms
6  time complexity O(n!)
7  """
8
9  class Solution:
10     def permute(self, nums: List[int]) -> List[List[int]]:
11         res = []
12         def dfs(pos):
13             if pos==len(nums)-1:
14                 # using deep copy here to harvest the result
15                 res.append(nums[:])
16             for i in range(pos, len(nums)):
17                 # swap
18                 nums[pos], nums[i] = nums[i], nums[pos]
19                 dfs(pos+1)
20                 nums[pos], nums[i] = nums[i], nums[pos]
21         dfs(0)
22         return res
```

Thanks