Completed search - Backtracking

Group 7

Trần Xuân Minh
Nguyễn Quốc Trường

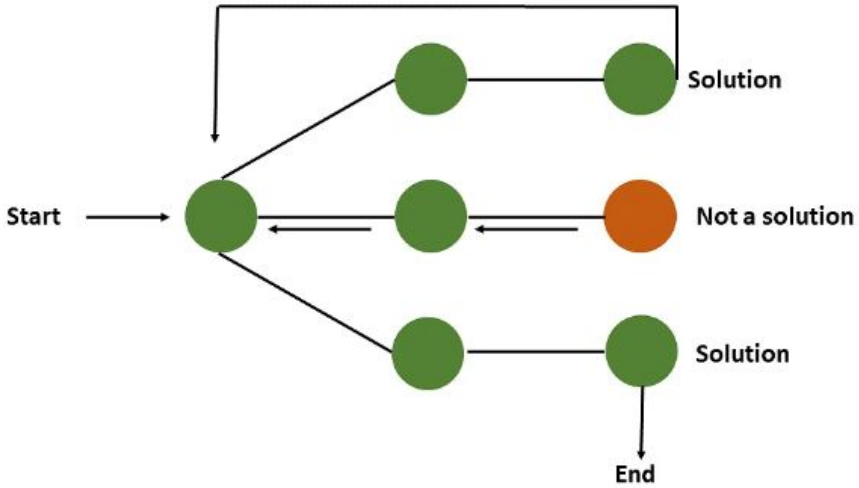
I. Introduction

a. What is Backtracking Algorithm?

It finds a solution by building a solution step by step, increasing levels over time, using recursive calling. A search tree known as the state-space tree is used to find these solutions. Each branch in a state-space tree represents a variable, and each level represents a solution.

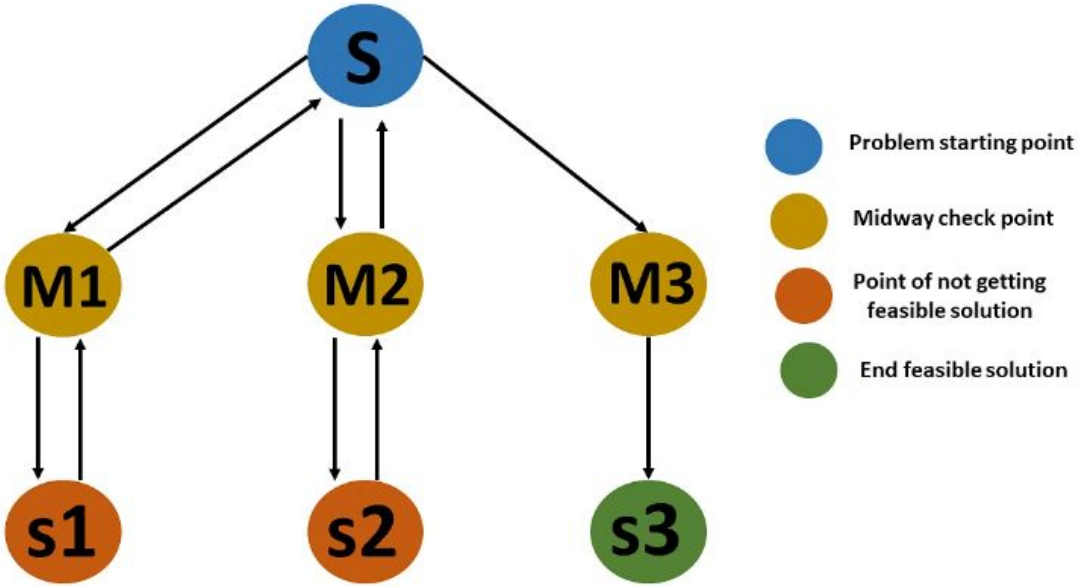
A backtracking algorithm uses the depth-first search method. When the algorithm begins to explore the solutions, the abounding function is applied so that the algorithm can determine whether the proposed solution satisfies the constraints. If it does, it will keep looking. If it does not, the branch is removed, and the algorithm returns to the previous level.

A space state tree is a tree that represents all of the possible states of the problem, from the root as an initial state to the leaf as a terminal state.



b. How does a Backtracking Algorithm work?

In any backtracking algorithm, the algorithm seeks a path to a feasible solution that includes some intermediate checkpoints. If the checkpoints do not lead to a viable solution, the problem can return to the checkpoints and take another path to find a solution. Consider the following scenario:


S
M1
M2
M3
s1
s2
s3
Problem starting point
Midway check point
Point of not getting feasible solution
End feasible solution

When you look at this example, you can see that we go through all possible combinations until you find a viable solution. As a result, you refer to backtracking as a brute-force algorithmic technique.

A "space state tree" is the above tree representation of a problem. It represents all possible states of a given problem (solution or non-solution).

The final algorithm is as follows:

- Step 1: Return success if the current point is a viable solution.
- Step 2: Otherwise, if all paths have been exhausted (i.e., the current point is an endpoint), return failure because there is no feasible solution.
- Step 3: If the current point is not an endpoint, backtrack and explore other points, then repeat the preceding steps.

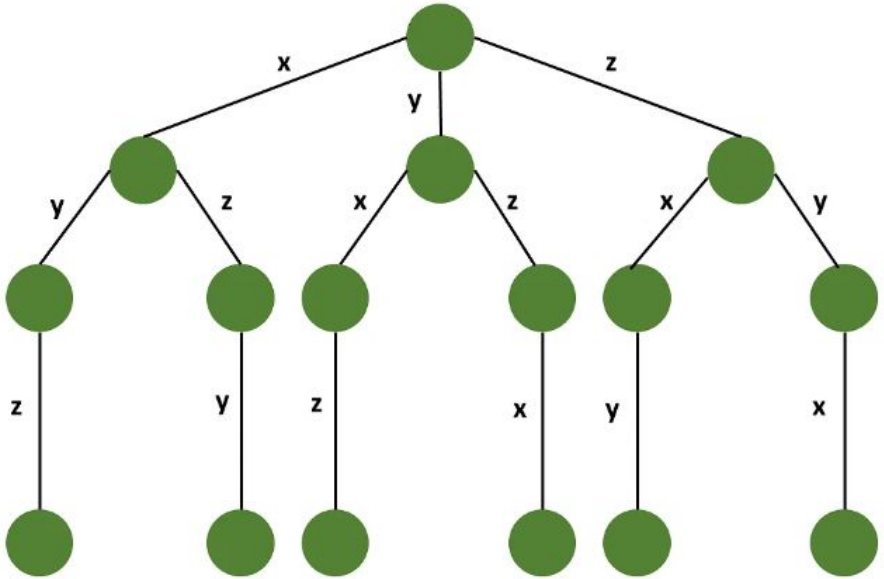
b. Pseudocode for Backtracking algorithm.

```
1  Backtracking(k) {
2    for([each option i (in set D)]) {
3      if ([i is acceptable]) {
4        [select i for X[k]];
5        if ([success]) {
6          [output the result];
7        } else {
8          Backtracking(k+1);
9          [unselect i for X[k]];
10       }
11     }
12   }
13 }
```

C. Example

You need to arrange the three letters x, y, and z so that z cannot be next to x.

According to the backtracking, you will first construct a state-space tree. Look for all possible solutions and compare them to the given constraint.



The following are possible solutions to the problems: (x,y,z), (x,z,y), (y,x,z), (y,z,x), (z,x,y) (z,y,x).

Nonetheless, valid solutions to this problem are those that satisfy the constraint that keeps only (x,y,z) and (z,y,x) in the final solution set.

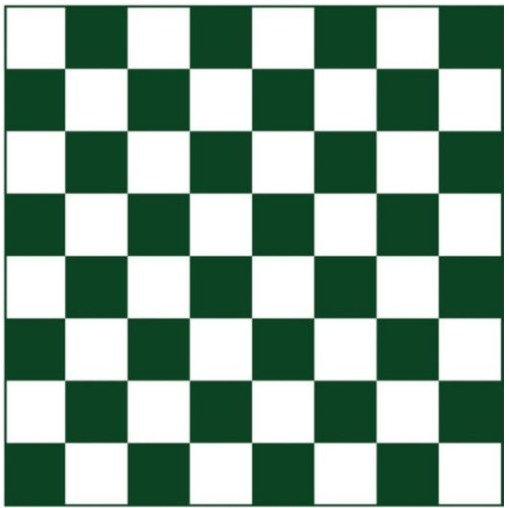
II. Applications

1. n-Queens Problem
2. Hamiltonian Circuit Problem
3. Subset-Sum Problem

1. n-Queens Problem

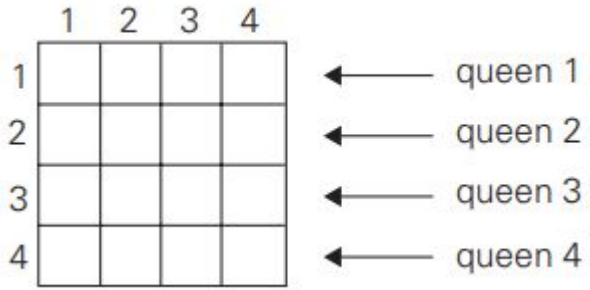
a. Problems

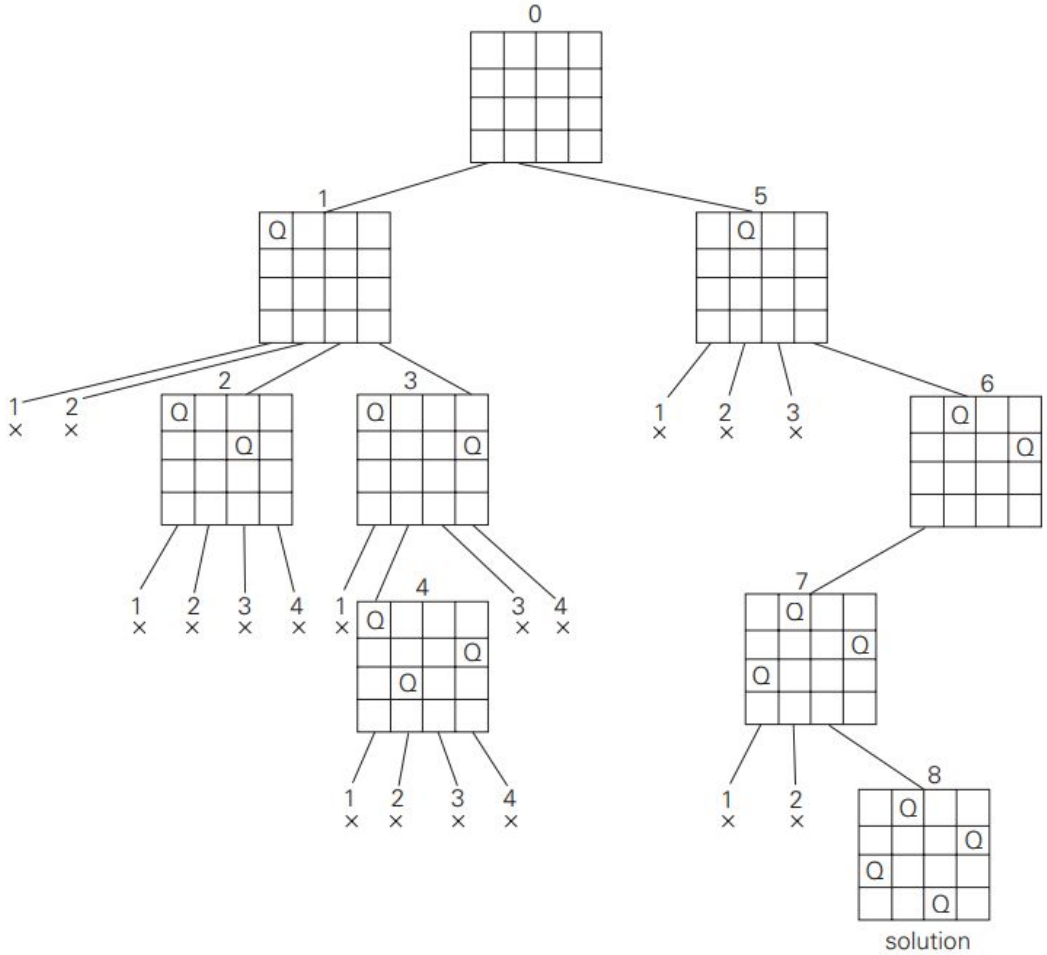
The n-queens problem is to place n queens on an n × n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.



b. Example

For n = 1, the problem has a trivial solution, and it is easy to see that there is no solution for n = 2 and n = 3. So let us consider the four-queens problem and solve it by the backtracking technique. Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board




0
1
5
2
3
6
4
7
8
solution

State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

To prevent the queens from attacking each other, we need to arrange n queens on n rows of the chessboard. We use an array Q[1, 2, ..., n], where Q[i]=j if the queen on the i row is placed on column j.

	0	1	2	3	
0		Q			Q[0] = 1
1				Q	Q[1] = 3
2	Q				Q[2] = 0
3			Q		Q[3] = 2

c. Pseudo-code

```
RecursiveQueen(Q[1, 2, ..., n], r):
  if(r = n + 1)
    print Q
  else
    for j ← 1 to n
      legal ← True
      for i ← 1 to r − 1                ≪ check conflict ≫
        if (Q[i] = j) or (Q[i] = j + r − i) or (Q[i] = j − r + i)
          legal ← False
      if legal = True
        Q[r] ← j
        RecursiveQueen(Q[1, 2, ..., n], r + 1)
```

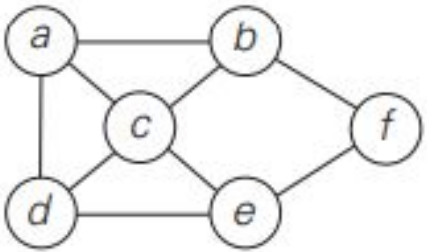
Time Complexity: O(N!)

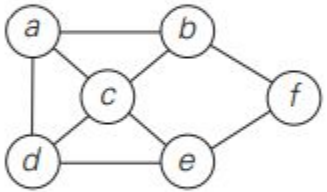
Auxiliary Space: O(N)

2. Hamiltonian Circuit Problem

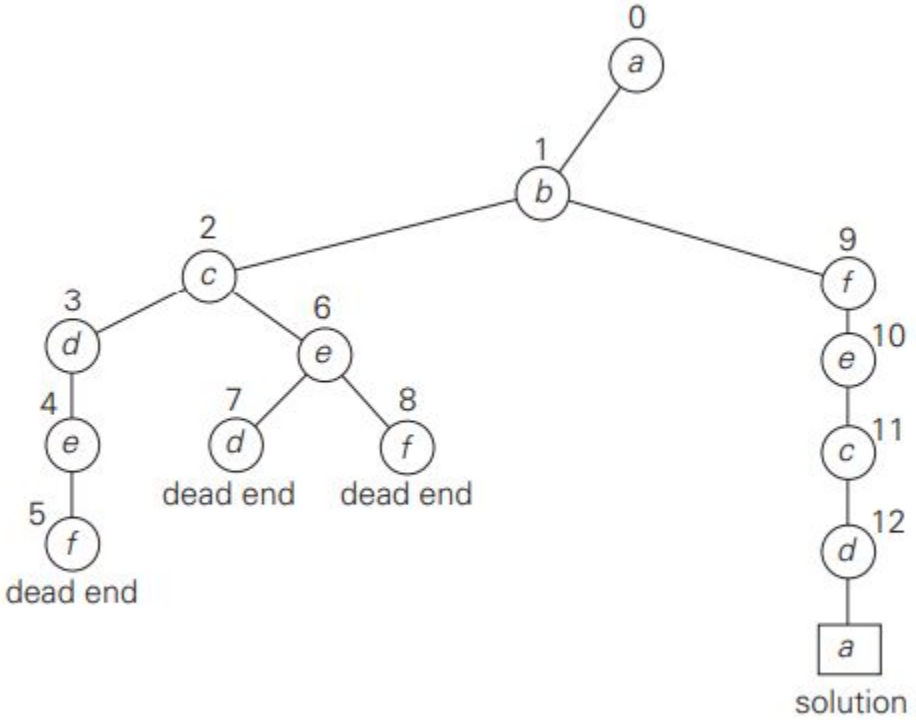
a. Problems

As our next example, let us consider the problem of finding a Hamiltonian circuit in the graph.




a
b
c
d
e
f

(a)


0
a
1
b
2
c
9
f
3
d
6
e
10
e
4
e
7
d
8
f
11
c
5
f
dead end
dead end
dead end
12
d
a
solution

(b)

(a) Graph. (b) State-space tree for finding a Hamiltonian circuit. The numbers above the nodes of the tree indicate the order in which the nodes are generated.

b. Pseudo-code

isValid(v, k)

Input – Vertex v and position k.

Output – Checks whether placing v in the position k is valid or not.

```
Begin
   if there is no edge between node(k-1) to v, then
      return false
   if v is already taken, then
      return false
   return true; //otherwise it is valid
End
```

cycleFound(node k)

Input − node of the graph.

Output − True when there is a Hamiltonian Cycle, otherwise false.

```
Begin
   if all nodes are included, then
      if there is an edge between nodes k and 0, then
         return true
      else
         return false;

   for all vertex v except starting point, do
      if isValid(v, k), then //when v is a valid edge
         add v into the path
         if cycleFound(k+1) is true, then
            return true
         otherwise remove v from the path
   done
   return false
End
```

Time Complexity : O(N!), where N is number of vertices.

Auxiliary Space : O(1), since no extra space used.

3. Subset-Sum Problem

a. Problems

Find a subset of a given set A = {a1,...,an} of n positive integers whose sum is equal to a given positive integer d.

b. Pseudo-code

Input: a set A with n elements and a value x

Output: true if there exists a subset of A with sum x, false otherwise.

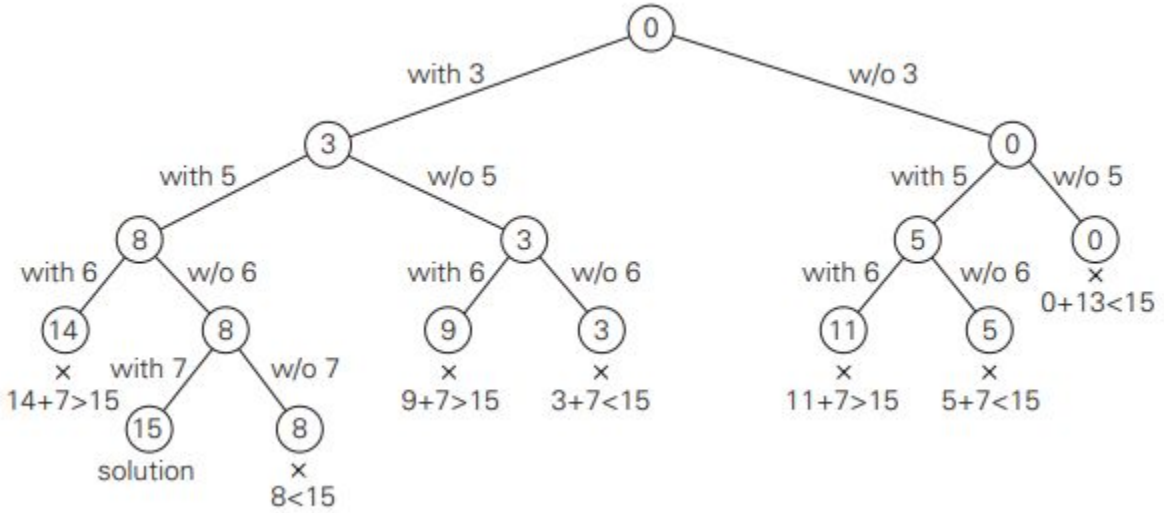
```
function subset_sum(A, n, x):
        if x == 0:
                return true
        if n == 0 and x != 0:
                return false
        if A[n-1] > x:
                return subset_sum(A, n-1, x)
        return subset_sum(A, n-1, x) or subset_sum(A, n-1, x-A[n-1])
```

Time Complexity: O(2^(n/2))

Space Complexity: O(n)

c. Example

Example 1: A = {3, 5, 6, 7} and d = 15


0
with 3
w/o 3
3
0
with 5
w/o 5
with 5
w/o 5
8
3
5
0
with 6
w/o 6
with 6
w/o 6
with 6
w/o 6
0+13<15
14
8
9
3
11
5
14+7>15
with 7
w/o 7
9+7>15
3+7<15
11+7>15
5+7<15
15
8
solution
8<15

If s is equal to d, we have a solution to the problem
If s is not equal to d, we can terminate the node as nonpromising if either of the following two inequalities holds:

$$s + a_{i+1} > d \quad \text{(the sum } s \text{ is too large)},$$

$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum } s \text{ is too small)}.$$

Example 2: A = {1, 3, 5, 6, 7, 9} and d = 17

What is the answer?

III. General Remarks

- Backtracking algorithm is a recursive search method in which we explore all possible states and eliminate those that do not lead to a solution.
- A common approach in Backtracking algorithms is to search for values for variables arranged in a specific order such that the chosen values satisfy pre-defined constraints.

- In a Backtracking algorithm, the output can be thought of as an n-tuple (x1, x2,...,xn) in which each value xi is an element of a finite linearly ordered set Si.
- These sets can be referred to as S1, S2,..., Sn, respectively, for each element of the n-tuple.
- The elements in each set can be arranged in a specific order, such as in ascending or descending order.

- The Backtracking algorithm can be applied to solve various problems, such as searching for all subsets of a set with a given sum or finding all paths in a graph from a starting vertex to an ending vertex.
- In each case, the Backtracking algorithm can be applied to search for values for variables arranged in a specific order such that the chosen values satisfy pre-defined constraints.
- The final result of the algorithm will be a set of values chosen for the variables that satisfy the pre-defined constraints.

To start a backtracking algorithm, the following pseudocode can be called for i = 0 ; X[1..0] represents the empty tuple.

ALGORITHM *Backtrack*$(X[1..i])$

```
//Gives a template of a generic backtracking algorithm
//Input: X[1..i] specifies first i promising components of a solution
//Output: All the tuples representing the problem's solutions
if X[1..i] is a solution write X[1..i]
else      //see Problem 9 in this section's exercises
    for each element x ∈ S_{i+1} consistent with X[1..i] and the constraints do
        X[i + 1] ← x
        Backtrack(X[1..i + 1])
```

IV. Pros and Cons

What are the pros and cons of the backtracking algorithm?

1. Pros:

- Backtracking can be applied to solve a wide range of difficult combinatorial problems, for which no efficient algorithms for finding exact solutions possibly exist.
- It is a systematic and complete search method that can guarantee finding all possible solutions.
- Backtracking can be a useful tool for generating all possible solutions, which can be valuable in its own right.
- It can be enhanced by evaluating the quality of partially constructed solutions, making it suitable for optimization problems.

2. Cons

- Backtracking can be very slow, especially for large instances of a problem. In the worst case, it may have to generate all possible candidates in an exponentially growing state space.
- The success of backtracking depends heavily on the ability to prune enough branches of its state-space tree before running out of time or memory.
- It may be difficult to estimate the size of the state-space tree of a backtracking algorithm, making it challenging to predict its efficiency.
- Backtracking may not be suitable for problems with many constraints or those with continuous variables.

Reduce the size of a state-space tree

- Exploit the symmetry often present in combinatorial problems
- Preassign values to one or more components of a solution

Estimate the size of a state-space tree

Knuth [Knu75] suggested generating a random path from the root to a leaf and using the information about the number of choices available during the path generation for estimating the size of the tree.

- Let c1 be the number of values of the first component x1 that are consistent with the problem's constraints
- Randomly select one of these values (with equal probability 1/c1) to move to one of the root's c1 children
- Repeating this operation for c2 possible values for x2 that are consistent with x1 and the other constraints, we move to one of the c2 children of that node
- Continue this process until a leaf is reached after randomly selecting values for x1, x2,...,xn
- By assuming that the nodes on level i have ci children on average, we estimate the number of nodes in the tree as $1 + c1 + c1c2 + \ ... \ + c1c2 \ ... \ cn$

V. Exercises

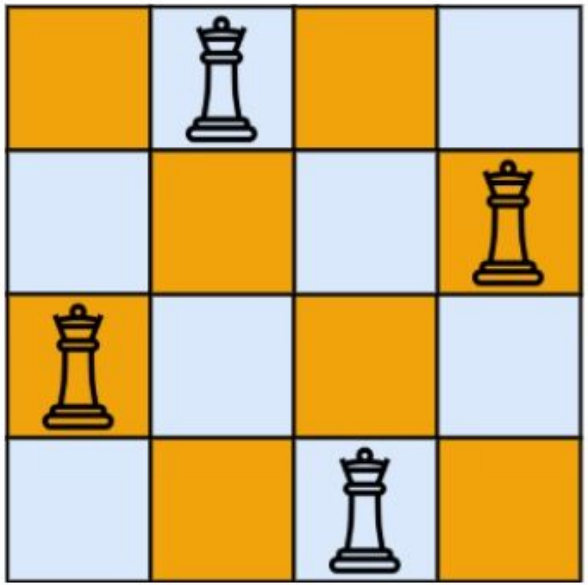
Exercise 1

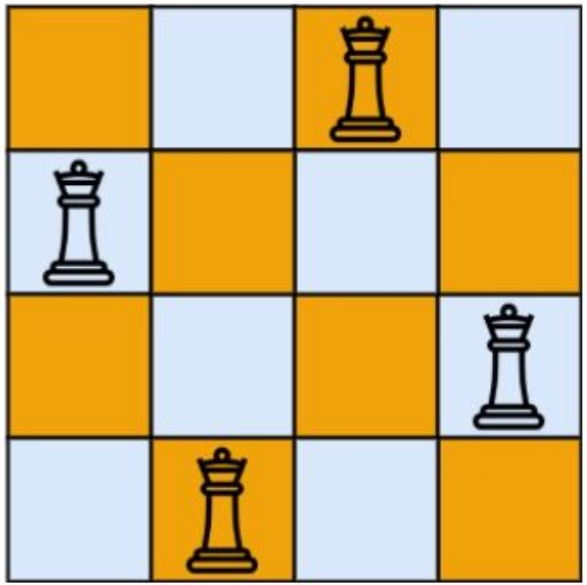
The **n-queens** puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.

Given an integer n, return *all distinct solutions to the **n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

Example 1:





```
Input: n = 4
Output: [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
Explanation: There exist two distinct solutions to the 4-queens puzzle as shown
above
```

Example 2:

```
Input: n = 1
Output: [["Q"]]
```

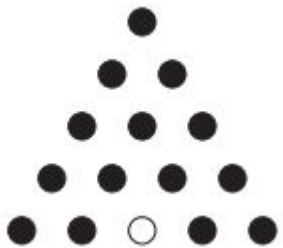
Constraints:

- `1 <= n <= 9`

Exercise 2

Puzzle pegs

This puzzle-like game is played on a board with 15 small holes arranged in an equilateral triangle. In an initial position, all but one of the holes are occupied by pegs, as in the example shown below. A legal move is a jump of a peg over its immediate neighbor into an empty square opposite; the jump removes the jumped-over neighbor from the board.



Design and implement a backtracking algorithm for solving the following versions of this puzzle.

a. Starting with a given location of the empty hole, find a shortest sequence of moves that eliminates 14 pegs with no limitations on the final position of the remaining peg.

b. Starting with a given location of the empty hole, find a shortest sequence of moves that eliminates 14 pegs with the remaining peg at the empty hole of the initial board.