Data Structure & Algorithms

Lec 15: Algorithms Design Techniques (Recursion and BackTracking)

Fall 2017 - University of Windsor
Dr. Sherif Saad

Outlines

Algorithm Design Techniques

Recursion and BackTracking

Applying Backtracking (N-Queens, Map Coloring, Knapsack, etc)

Algorithms Design techniques

An algorithm design technique is a general problem-solving paradigm for a set of problems that share similar characteristics.

There is an overlapping between different algorithm design techniques. (e.g., the same problem could be solved by more than one technique)

Typically, for a given problem one or more algorithms design techniques could solve this problem. However, usually, one technique will be superior to the others

Some problems will require applying one or more algorithm design technique.

Algorithms Design techniques

The most common algorithms design techniques are:

1. Brute Force
2. Divide and Conquer
3. Greedy Algorithms
4. Dynamic Programming
5. Transform and Conquer
6. Randomized Algorithms
7. Backtracking

Brute Force Algorithms

Brute force, straightforward solution for any problem that attempts all candidate solutions in a systematic approach, without excluding any candidate solution.

It is effortless and straightforward to implement, but it is not the most efficient approach and possibly not feasible when the problem size increase.

Some problems require a brute force algorithm to find the solution

All brute force algorithms for a given problem are sufficient but not necessary.

Brute Force Algorithms

Given an integer s, write an algorithm to find all the pairs (a, b) of integers in whose sum is equal to s.

s = 8 → {(2, 6), (1, 7), (4, 4), (5, 3), (8, 0)}

Divide and Conquer

A divide and conquer approach solve the problem by dividing the actual problem into several smaller subproblems, then independently solve each subproblem, finally combine the solutions of the subproblems to yield the solution of the original problem.

If we divide the problem into m subproblems and solve all the m subproblems than this a divide and conquer problem, but if we end up solving only (m-i) subproblems, then this is a decrease and conquer problem.

Examples: Binary Search is a decrease and conquer, while Mergesort is a divide and conquer.

Greedy Algorithms

A technique to find a solution for the problem through a sequence of steps and at each step, the algorithm makes the locally optimal choice at each stage with the hope of finding a global optimum.

The algorithm solves the problem one step at a time and in each step selects the option that returns the most obvious and immediate benefits.

Greedy algorithms are straightforward to design an implement but do not guarantee to find the optimal solution.

Greedy algorithms work the best when the nature problem at hand is greedy.

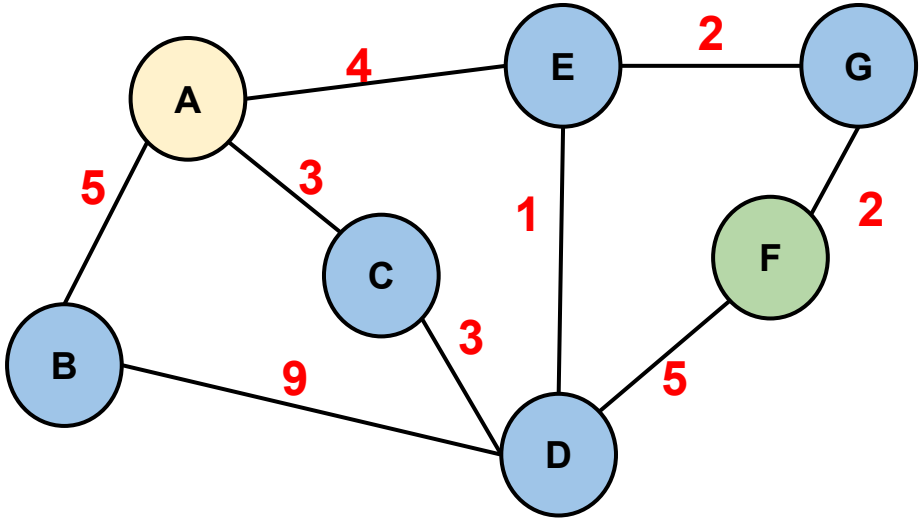
Greedy Algorithms

Using greedy algorithm to find the shortest path from A to F

A → C → D → E → G → F

At every node select the shortest distance to the next unvisited adjacent node.

How is that different from Dijkstra Algorithm?



Dynamic Programming

Dynamic programming solves a complex problem by breaking it down into a collection of simpler subproblems, and then solve each of those subproblems just once, and storing their solutions.

When the subproblems are identical or some of them repeatable, divide and conquer are inefficient, since it performs unnecessary computation.

Dynamic programming methods memorize the solution of solved subproblems and avoid resolving them.

In general, dynamic programming is either applied in top-down approach or bottom-up approach.

Transform and Conquer

The problem is transformed (represented) into a more manageable and easy to solve representation.

In general, the problem is transformed using one of the following methods

- Instance simplification: the instances of the problem can be transformed into a more accessible instance to solve.
- Representation change: the data structure can be transformed so that it is more efficient.
- Problem reduction: the problem can be transformed into a more straightforward problem to solve.

Randomized Algorithms

Randomized algorithms apply some degree of randomization as part of its logic to solve a given problem.

This technique usually helps solve complex problems where brute force technique could take infinite time to find the solution.

Randomized algorithms typically do not guarantee the optimal solution.

Examples: Quicksort, Genetic Algorithm

Backtracking

A backtracking approach is an effective method to solve constraint satisfaction problems.

Backtracking algorithms systematically examine all possible solutions and incrementally construct the solutions by combining different candidates, as soon as it discovers that the combination of a candidate cannot possibly result in a valid solution it backtracks by eliminating the last added candidate.

Backtrack algorithms represent the solution space as a search tree and traverse the tree using a depth-first traverse.

Recursive and Backtracking

When a function call itself , it is called recursion.

Recursion is useful in solving problem which can be broken into smaller problems of the same type.

Recursion is an algorithm implementation technique. In general, every recursion algorithm has an iterative version.

Writing an iterative algorithm for problems that are naturally recursive most of the time is difficult than writing a recursive algorithm.

Recursion

Any recursive algorithm must have a terminating condition, otherwise it will never terminate.

Every recursion function has a base case, the base case occurs when the recursion function can solve the problem without breaking it into smaller problems

Structure of a Recursion Function

```
1. if (test for the BaseCase):
2.     return some base case value
3. else (test for anoher BaseCase):
4.     return some base case value
5. else
6.     do some work  to break it into smaller problems then
7.     recursive call
```

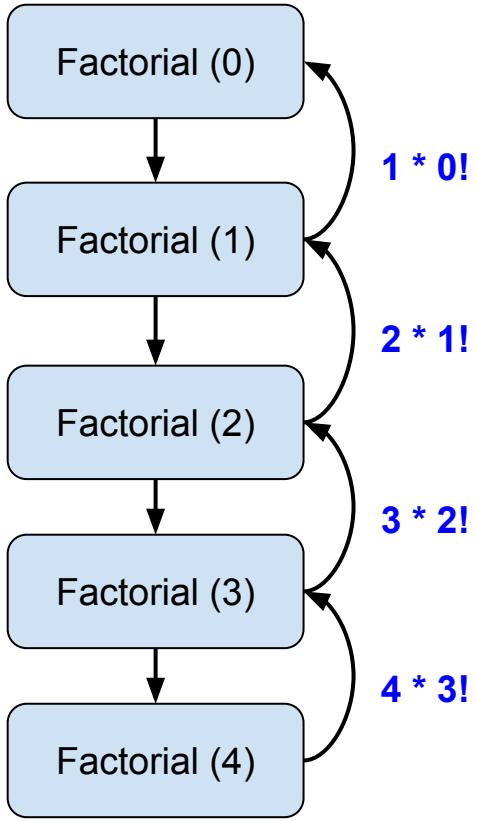
Structure of a Recursion Function

The base case of a factorial function is 1 when n = 0. In this case the function return 1 and no need to break the problem into smaller sub problems. Otherwise we break it into smaller problems where n! = n * (n-1)!

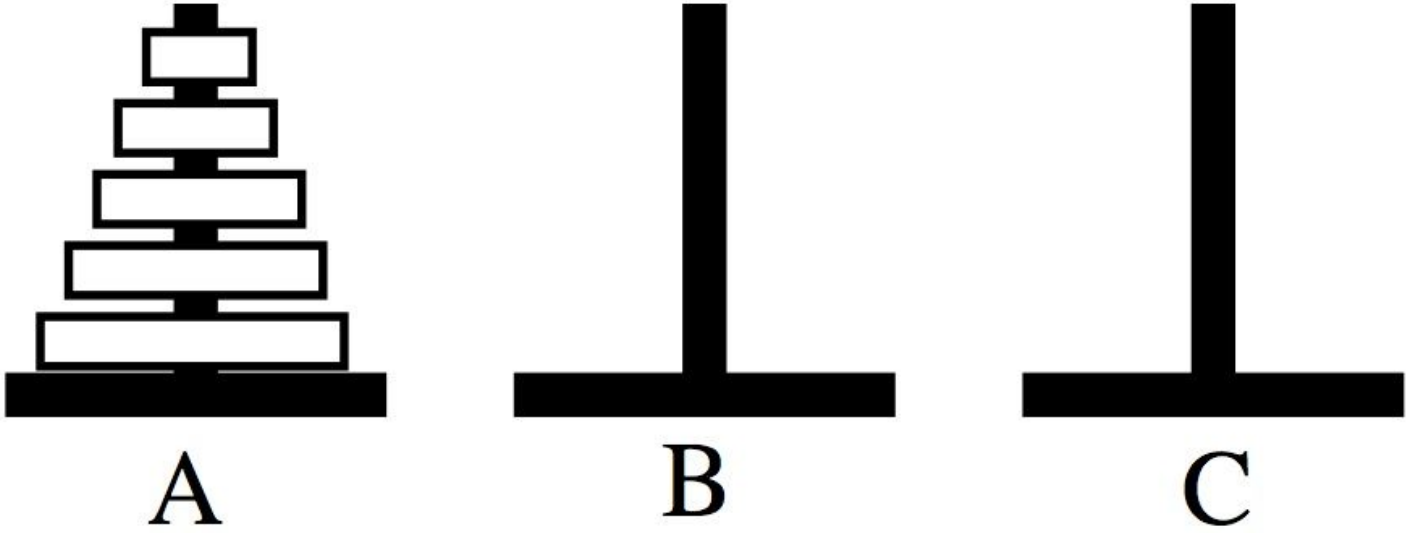
$$factorial = \begin{cases} n! = 1 & n = 0 \\ n! = n * (n-1)! & n > 0 \end{cases}$$

Structure of a Recursion Function

```
1. int Factorial(int n){
2.     if(n == 1 || n== 0){                    // base case
3.         return 1;
4.     }
5.     else{
6.         return n * Factorial(n-1);     // break into smaller problems
7.     }
8. }
```



Recursion Problems: Towers of Hanoi

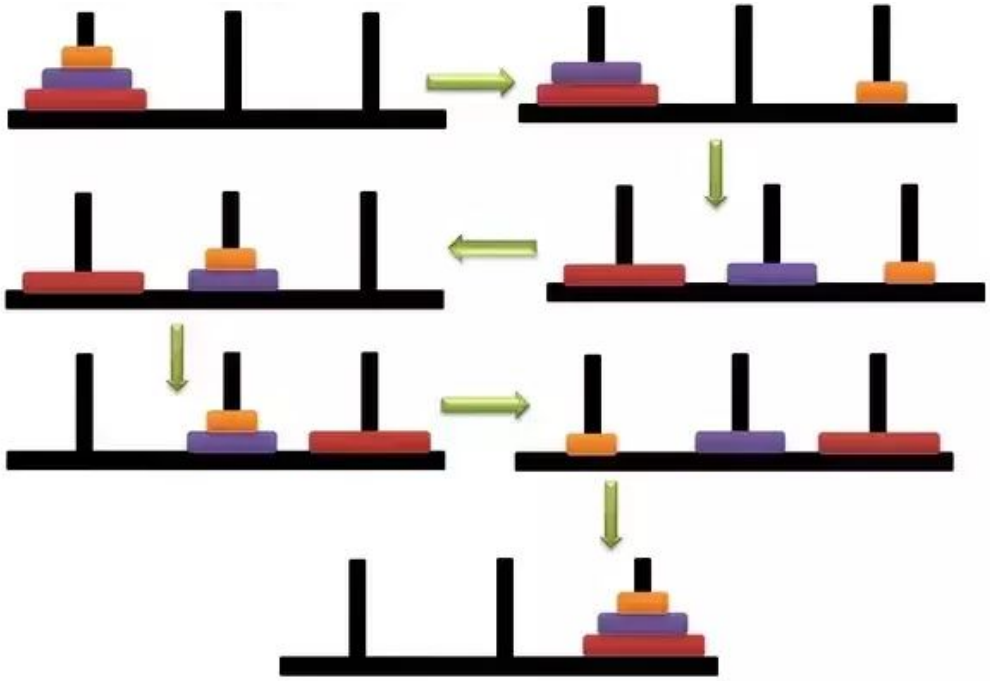


Recursion Problems: Towers of Hanoi

Tower of Hanoi is a mathematical puzzle. This puzzle consists of three towers (A, B, and C) in which disks can slide. There is some number of disks are placed on a tower in increasing order of their disk size from top to bottom. Now we have to transfer all the disk to another tower. Keeping in mind that.

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the towers and placing it on top of another tower.
3. No disk may be placed on top of a smaller disk.

Recursion Problems: Towers of Hanoi



Recursion Problems: Towers of Hanoi

```
1. algorithm TowerHanoi(n, s: source, d: destination, t: temporary)
2.    if n == 1:
3.        move n from s to d
4.    else:
5.        TowerHanoi(n-1, s, t, d)
6.        move N from s to d
7.        TowerHanoi(n-1, t, d, s)
8. end-algorithm
```

Recursion Problems: Towers of Hanoi

```
1. int TowerHanoi(int n, char a, char b, char c){
2.     if (n == 1){
3.         cout<< move disk from<< a << " to " << c << endl;
4.     }else{
5.         // move n-1 disk from a to b using c
6.         TowerHanoi(n-1, a, c, b);
7.         cout<< move disk from<< a << " to " << c << endl;
8.         // moving n-1 disk from  b to c using a
9.        TowerHanoi(n-1, b, a, c);
10.    }
11. }
```

Recursion Problems: Fibonacci Sequence

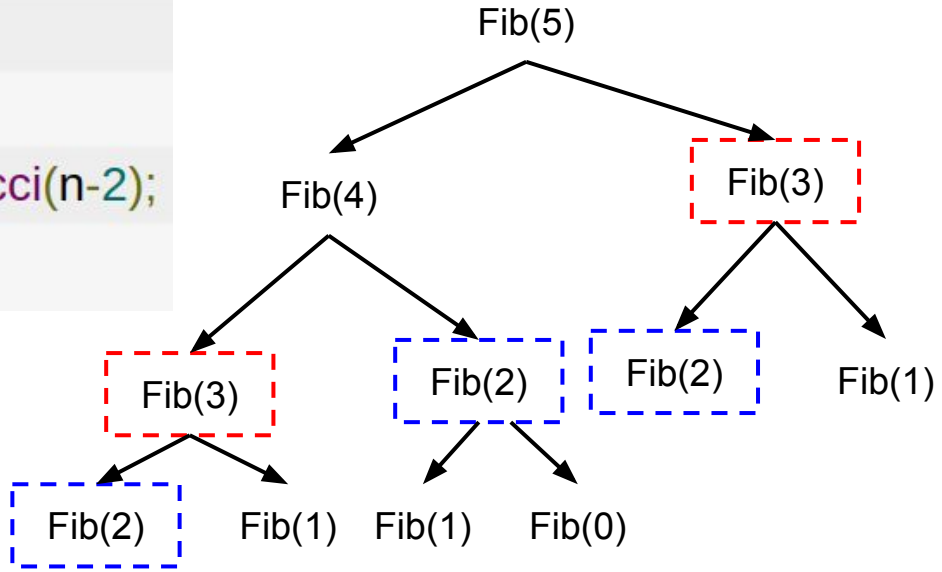
The Fibonacci sequence is a series where the next element is the sum of previous two elements. The first two elements of the Fibonacci sequence is 0 followed by 1.

Fibonacci Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21

Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

Recursion Problem: Fibonacci Sequence

```
1. int Fibonacci(int n) {
2.    if (n==1 || n== 0)
3.        return n
4.    else return Fibonacci(n-1) +Fibonacci(n-2);
5. }
```



Recursion vs Iteration

Recursion

- Terminate when reach the base case.
- Each recursive call require extra space in the "call stack" or "execution stack"
- We could result in stack overflow for large problem size.
- The solutions of some problems are easily expressed using recursion.

Iteration

- Terminate when the iterative condition becomes false
- Does not require extra space.
- The solutions of some problems are hard to implement using iterative approach.

BackTracking

Backtracking is an improvement of the brute force approach.

It is systematically search for a solution to a problem among all possible solutions.

Backtracking algorithms are mostly implemented using recursion.

For some problems we can represent the solution space (the set of all candidate solutions) as a tree. Rather than traversing the entire tree to examine all the solutions we use backtracking for pruning the tree and speed the brute force process.

Backtracking and Recursion

Backtracking is easily implemented with recursion because:

- The execution stack takes care of keeping track of the choices that got us to a given point.
- When a failure occurred, we can get to the previous choice (backtrack) by merely returning a failure code from the recursive call.

Backtracking algorithms are the most common technique to solve constraints satisfaction problems

Constraint Satisfaction Problems

What are Constraint Satisfaction Problem (CSPs)?

CPSs are mathematical problems where one must find states or objects that satisfy a number of constraints or conditions

Formally a CSP is defined by a triple (V, D, C):

- V is a set of variables
- D is a domain of values
- C is a set of constraints

A **constraint** is a combination of valid values for the variables.

Constraint Satisfaction Problems (cont)

A state of the problem is defined by an **assignment** of values to some or all of the variables.

A **solution** to a CSP is an assignment that satisfies all the constraints

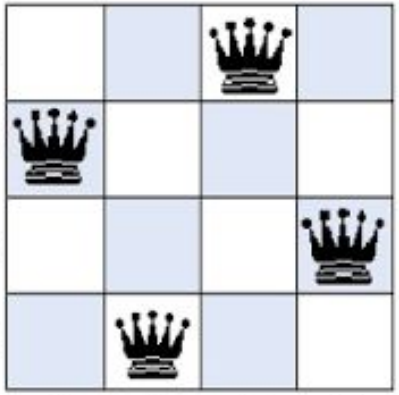
Examples:

- N-Queens ??
- 8-Puzzels ??
- Map Coloring ??
- Cryptarithmetic (**Verbal arithmetic**) ??

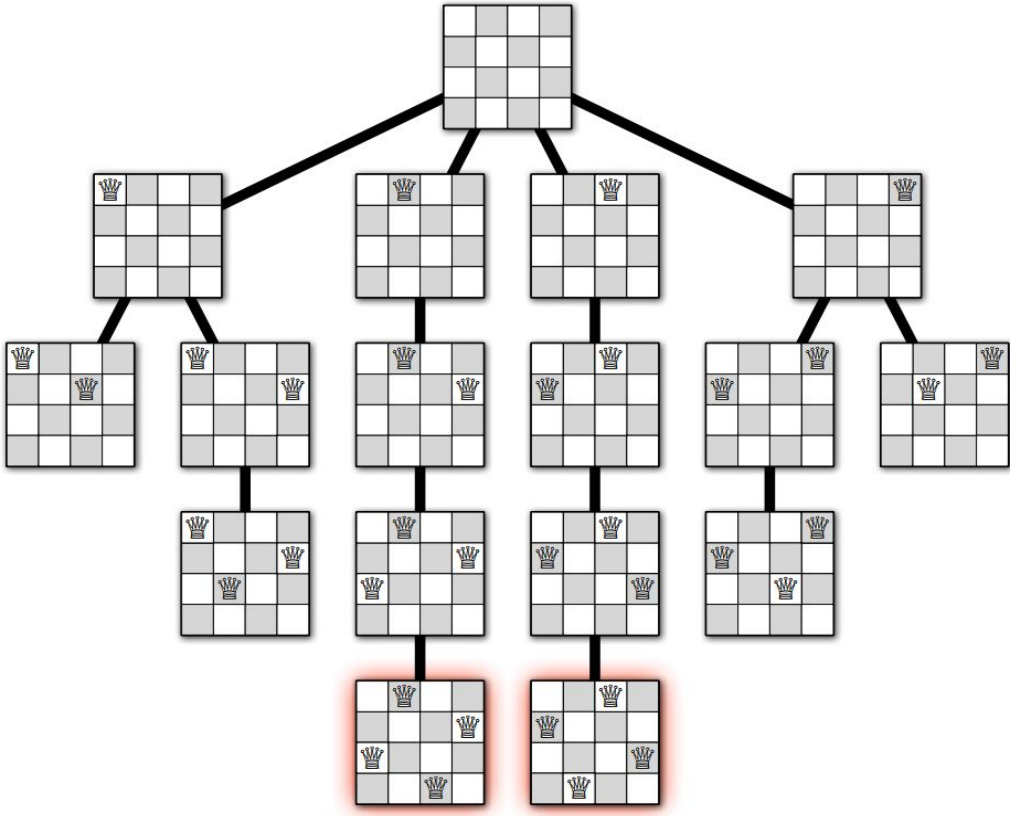
Backtracking Problems: N-Queens

N-Queens: *Given an* ***n x n*** *chessboard, arrange n queens so that none is attacking another.*

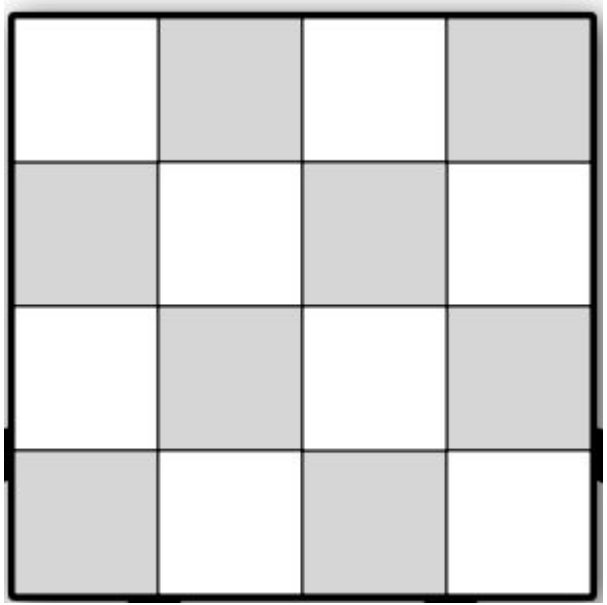
- *Variables:* Q_i *for each row i of the board*
- *Domain: {1, 2, 3, …, n} for position in row*
- *Constraints: ??*



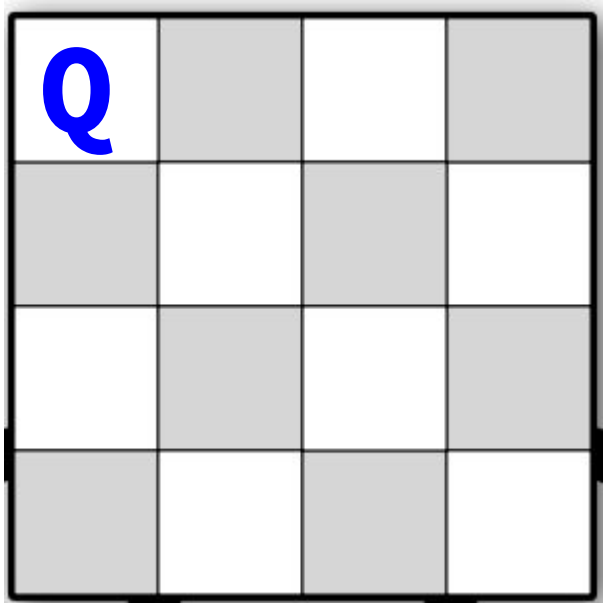
Backtracking Problems: N-Queens



Backtracking Problems: N-Queens



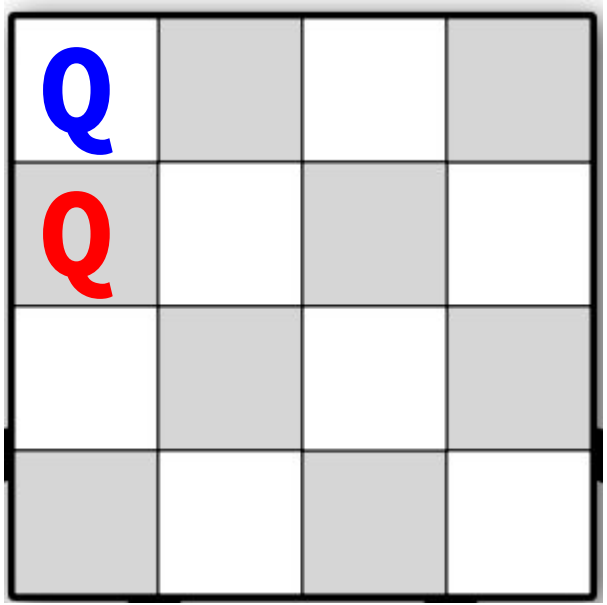
Backtracking Problems: N-Queens



Place queen on the first row
Check if it is a safe move or not.
If it is a safe move then recursively attempt to place queen on the next row

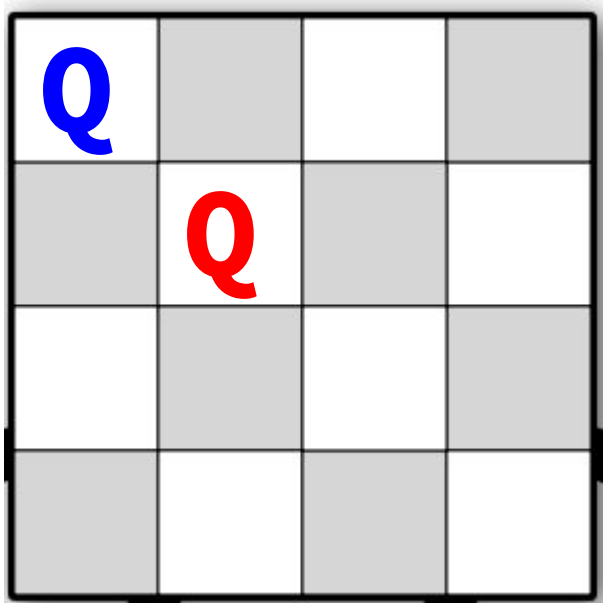
Backtracking Problems: N-Queens

When the new added queen result in invalid solution. We attempt change it is position on the same raw to valid position



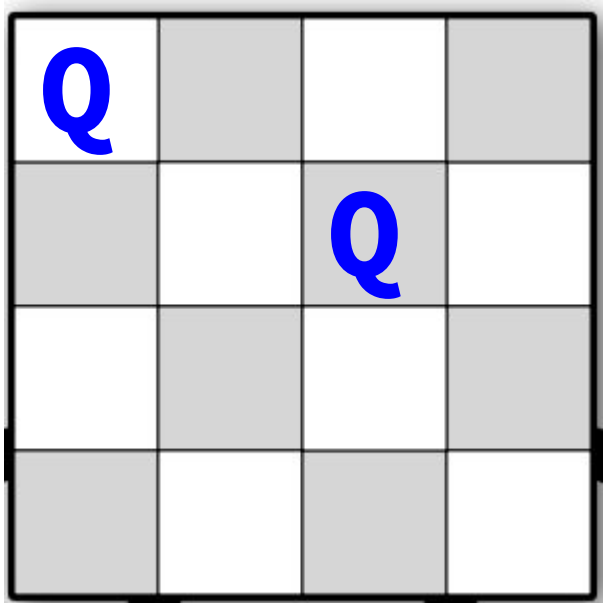
Backtracking Problems: N-Queens

NOT SAFE



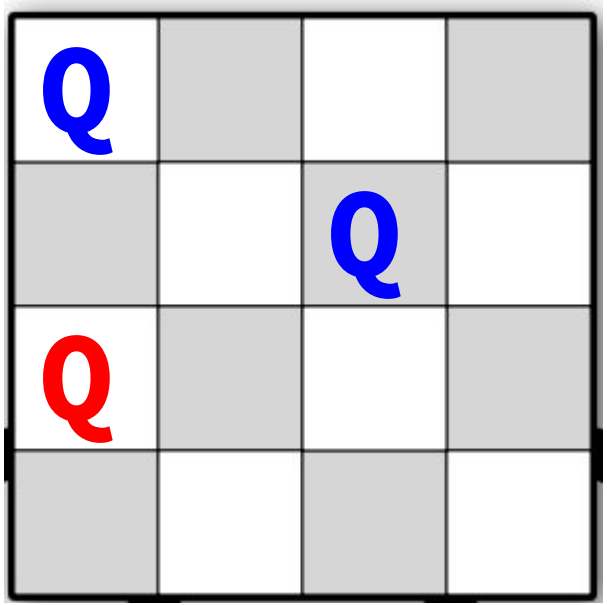
Backtracking Problems: N-Queens

SAFE, recursively move to the next row



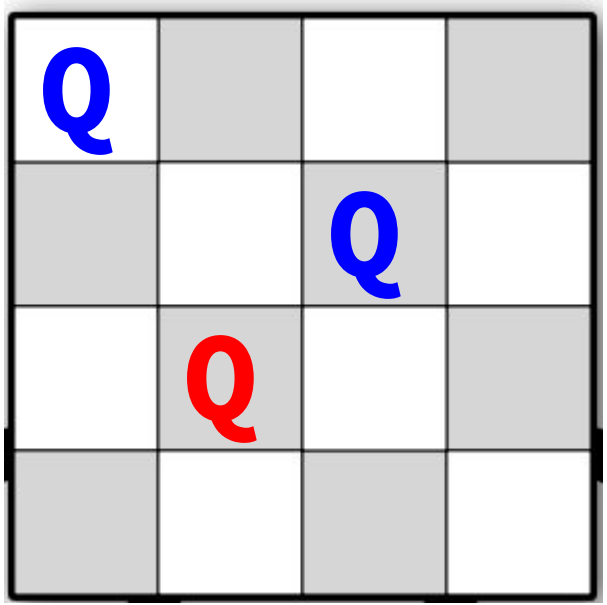
Backtracking Problems: N-Queens

NOT SAFE



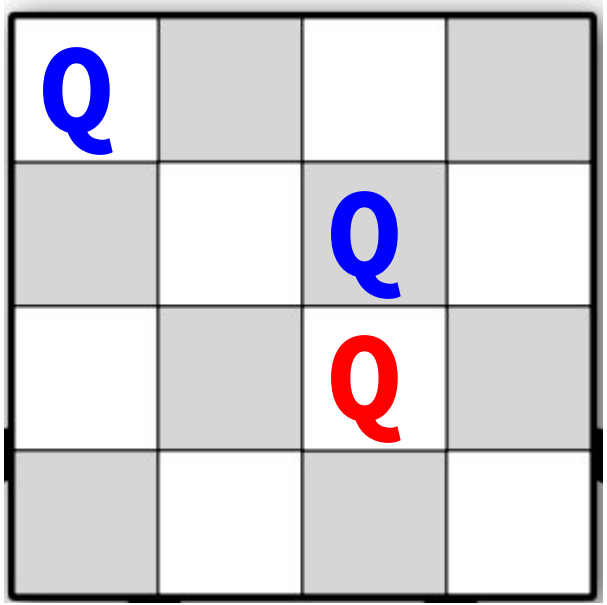
Backtracking Problems: N-Queens

NOT SAFE



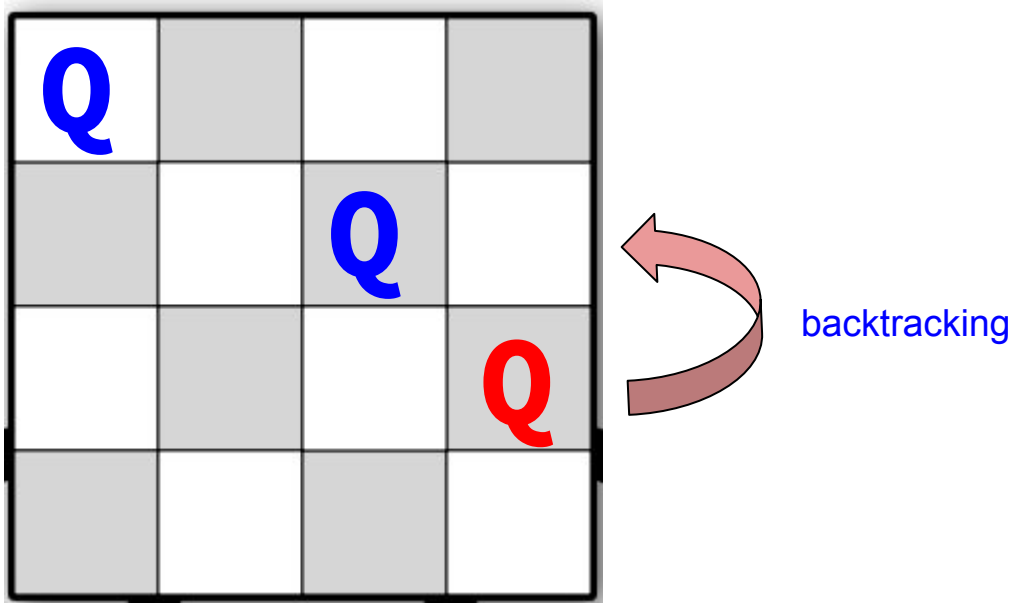
Backtracking Problems: N-Queens

NOT SAFE



Backtracking Problems: N-Queens

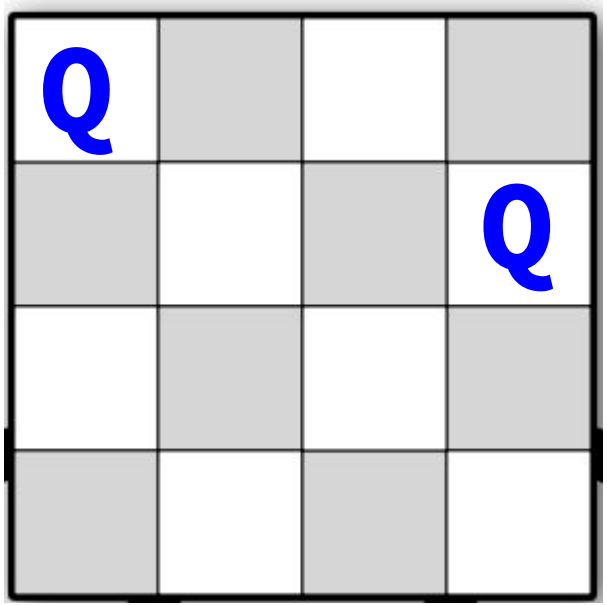
NOT SAFE, we can not move the current queen any more on the same row. We return from the recursive call to the previous row. (backtracking)



Backtracking Problems: N-Queens

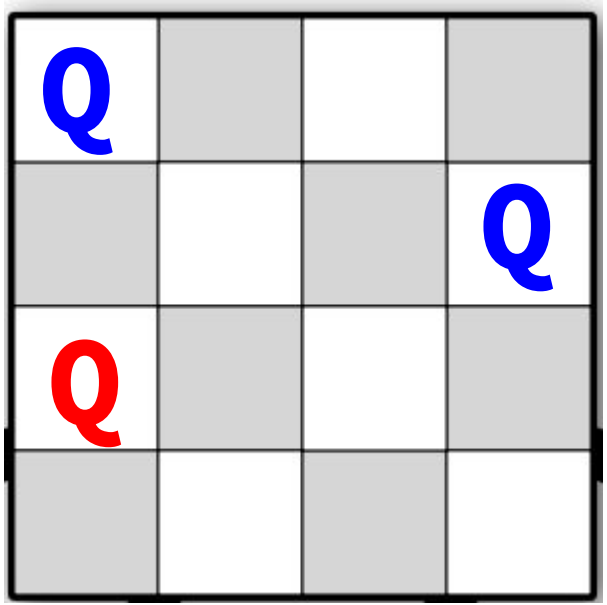
We applied backtracking and move the queen in the current row to the next column position

SAFE, recursively move to the next row



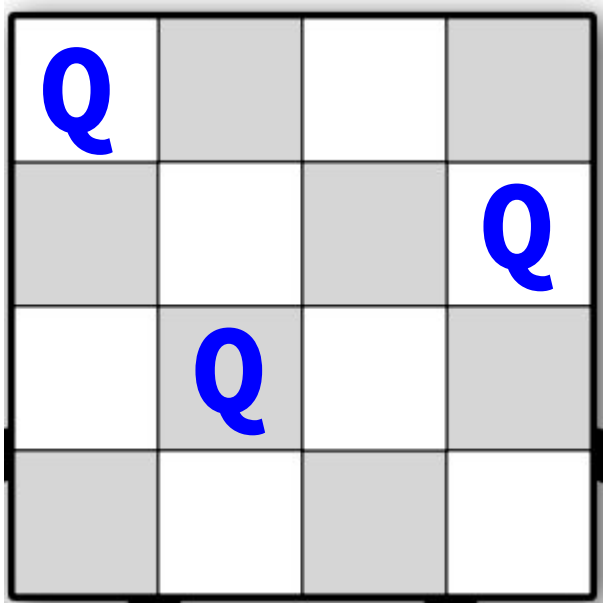
Backtracking Problems: N-Queens

NOT SAFE



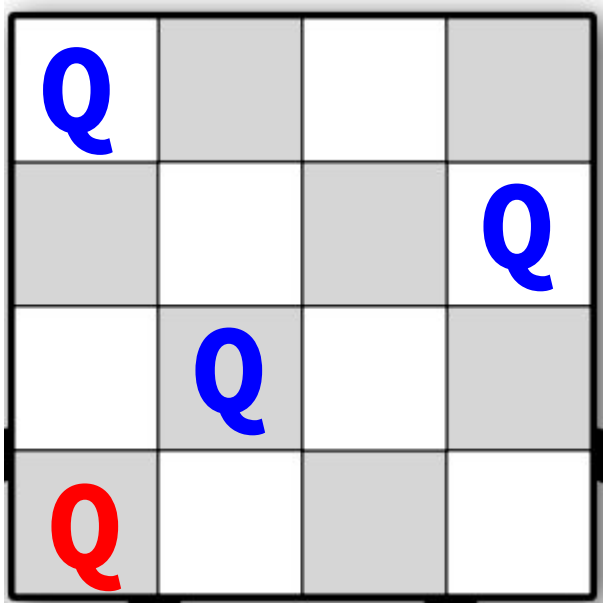
Backtracking Problems: N-Queens

SAFE, recursively move to the next row



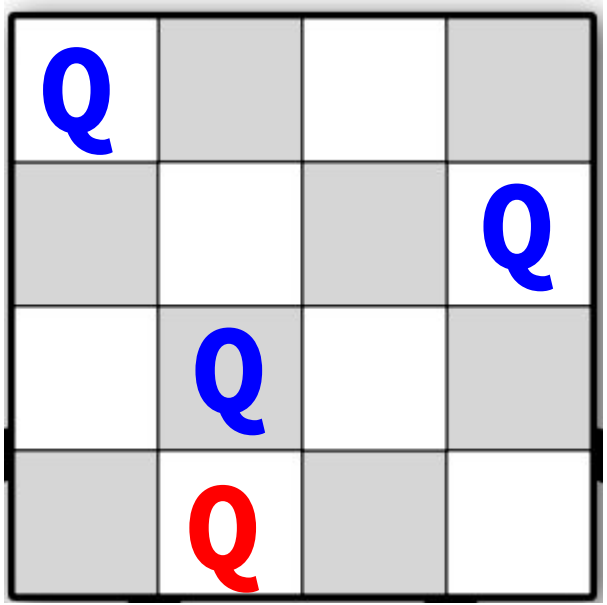
Backtracking Problems: N-Queens

NOT SAFE



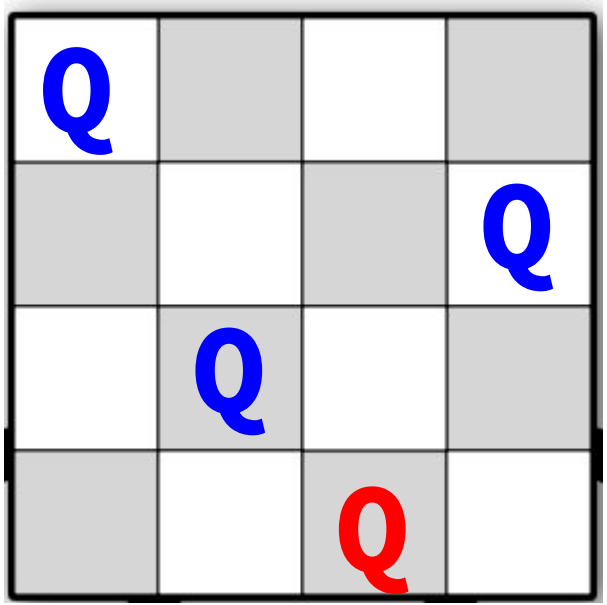
Backtracking Problems: N-Queens

NOT SAFE



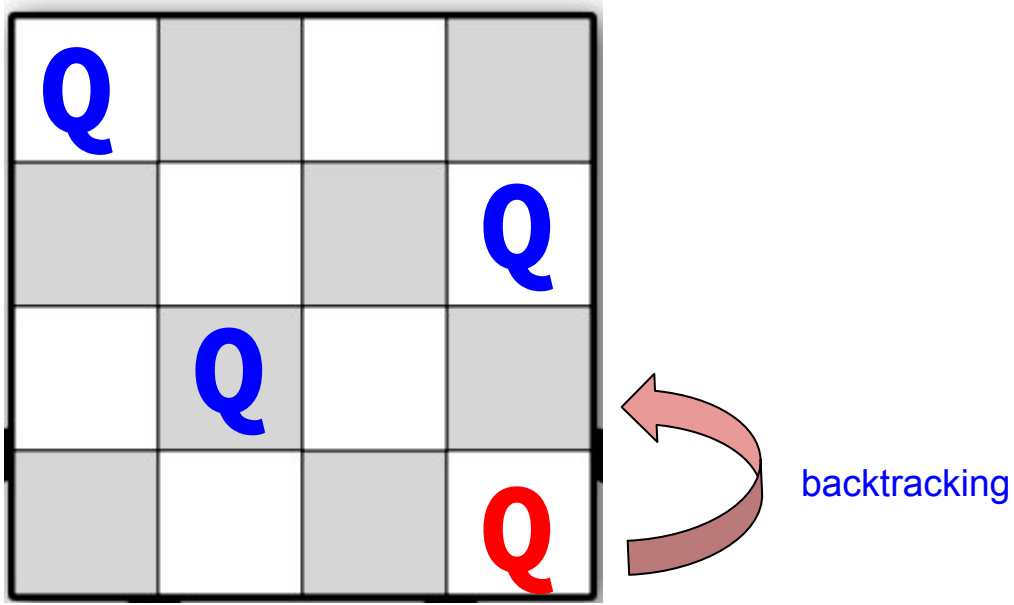
Backtracking Problems: N-Queens

NOT SAFE



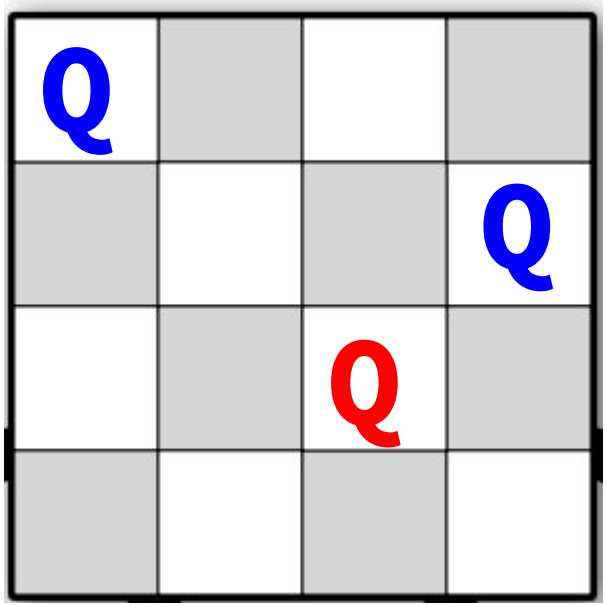
Backtracking Problems: N-Queens

NOT SAFE, we can not move the current queen any more on the same row. We return from the recursive call to the previous row. (backtracking)



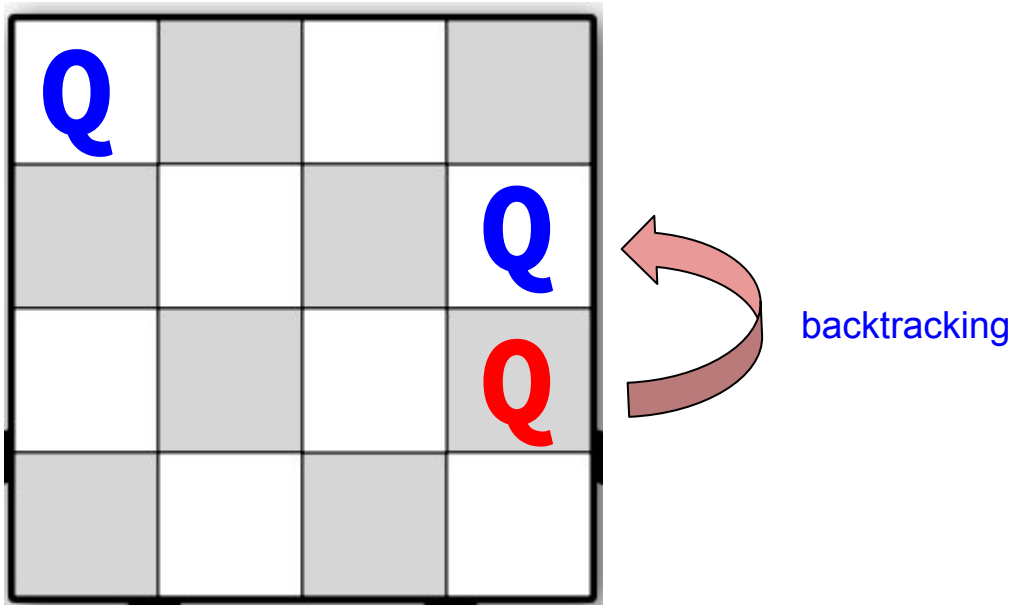
Backtracking Problems: N-Queens

NOT SAFE

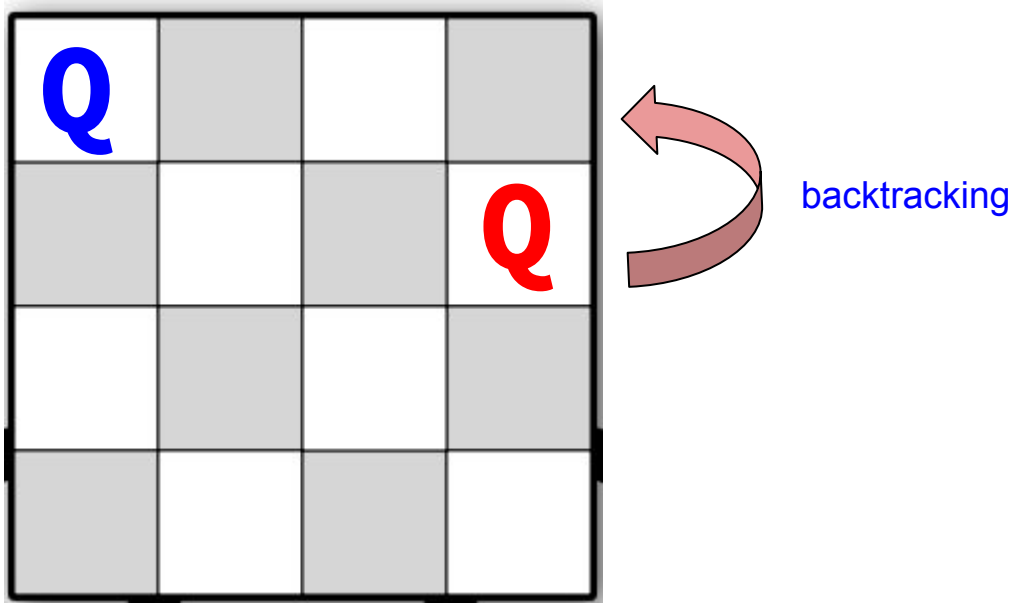


Backtracking Problems: N-Queens

NOT SAFE

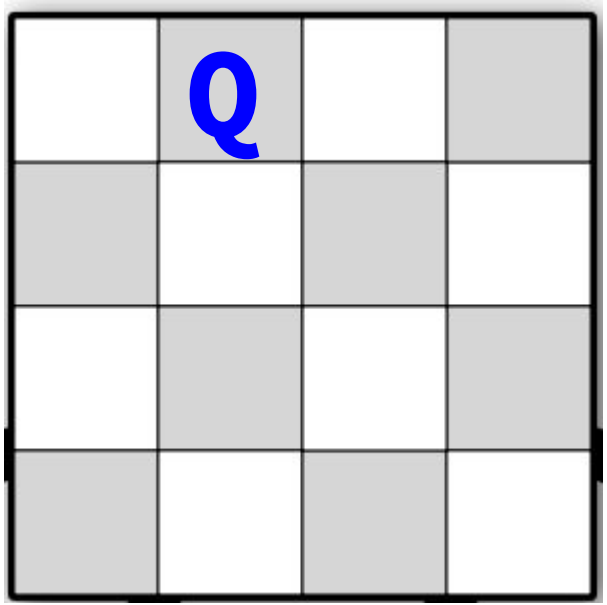


Backtracking Problems: N-Queens



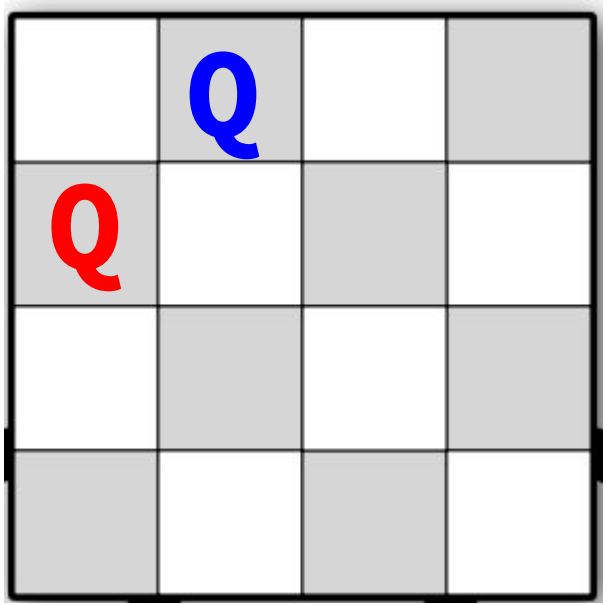
Backtracking Problems: N-Queens

SAFE, recursively move to the next row



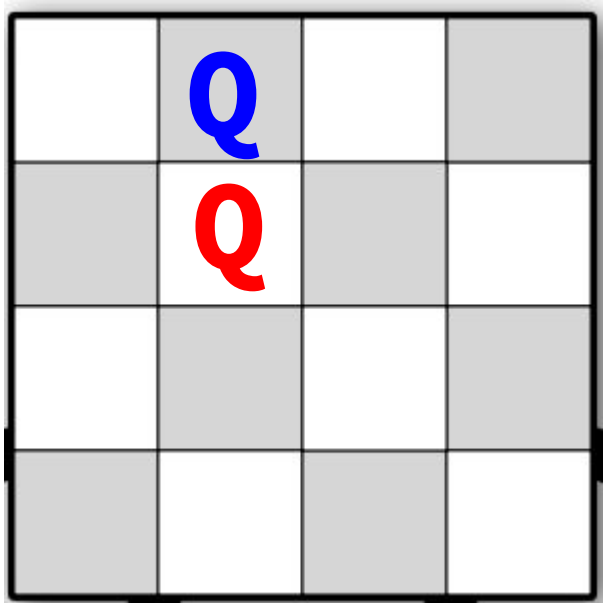
Backtracking Problems: N-Queens

NOT SAFE



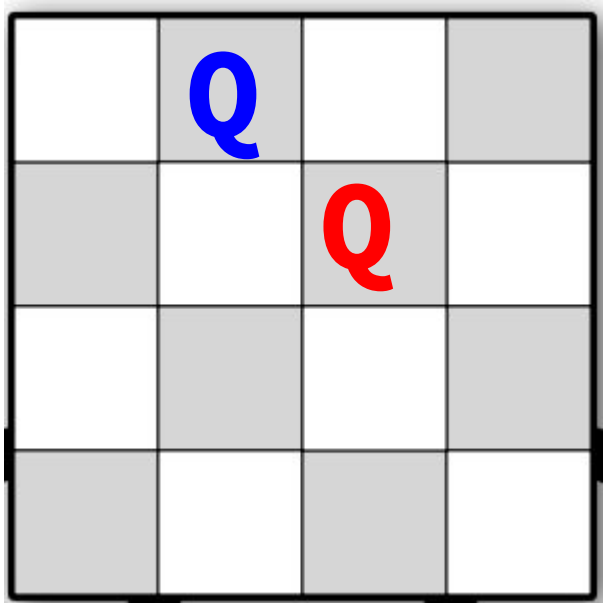
Backtracking Problems: N-Queens

NOT SAFE



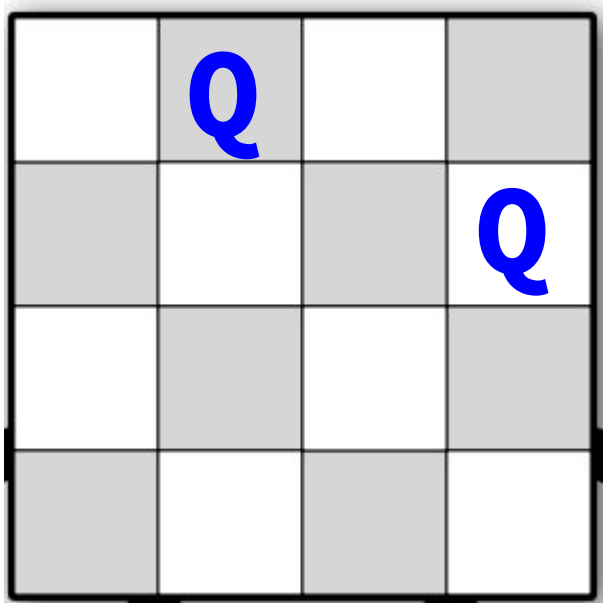
Backtracking Problems: N-Queens

NOT SAFE



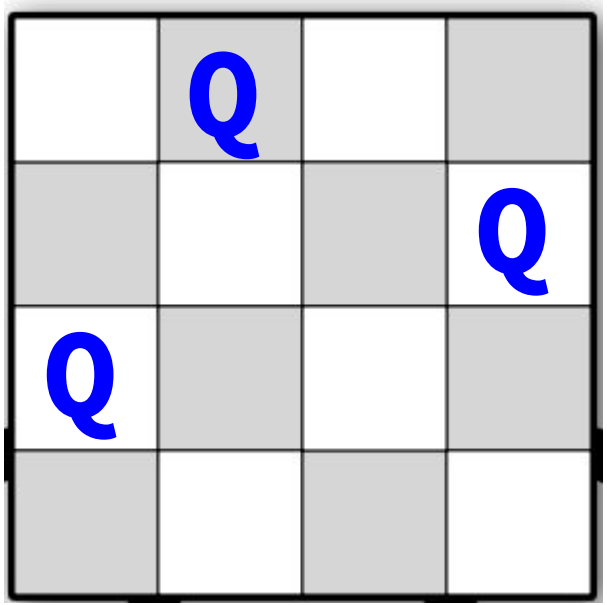
Backtracking Problems: N-Queens

SAFE, recursively move to the next row

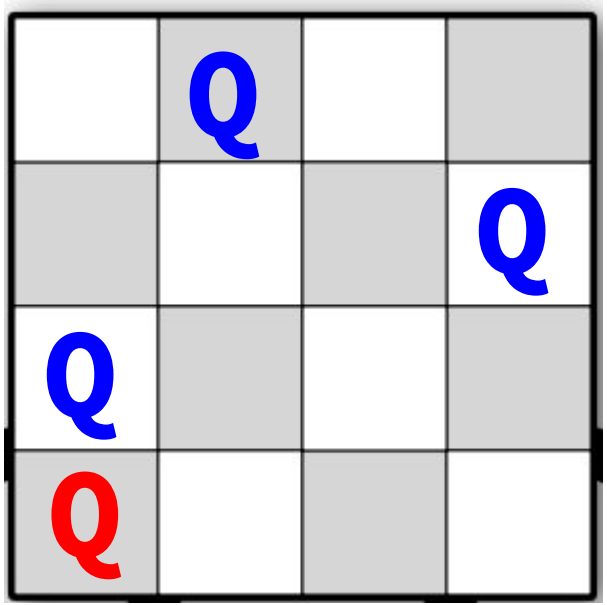


Backtracking Problems: N-Queens

SAFE, recursively move to the next row

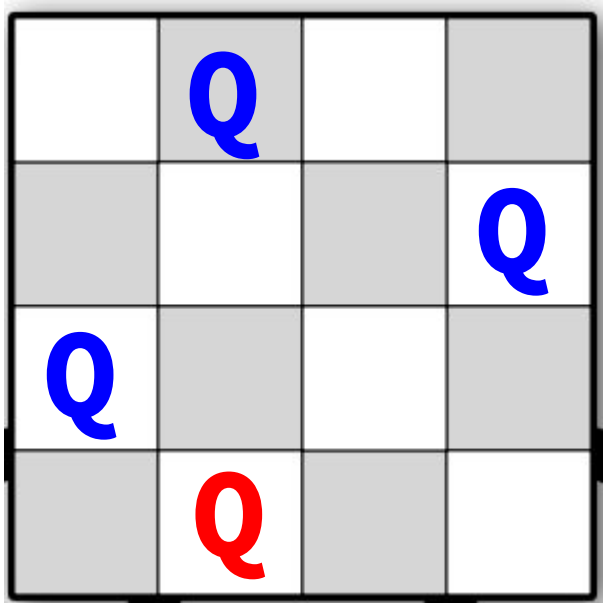


Backtracking Problems: N-Queens



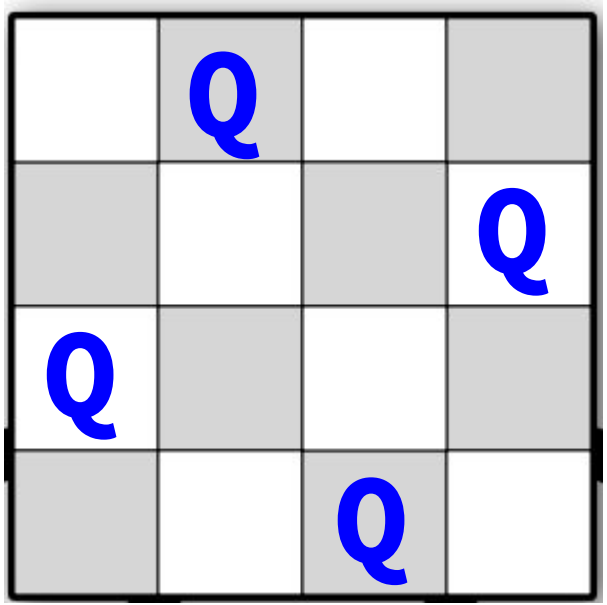
Backtracking Problems: N-Queens

NOT SAFE



Backtracking Problems: N-Queens

SAFE, recursively move to the next row. We will hit the termination condition which is row = n+1

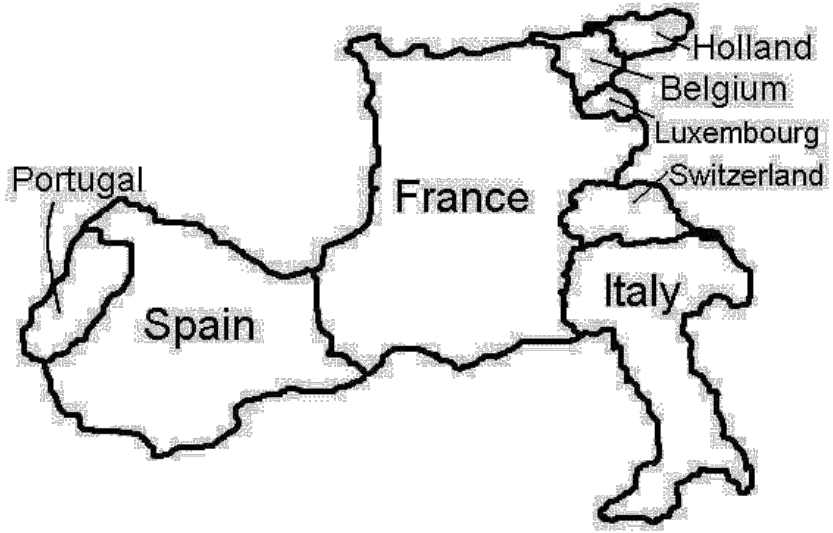


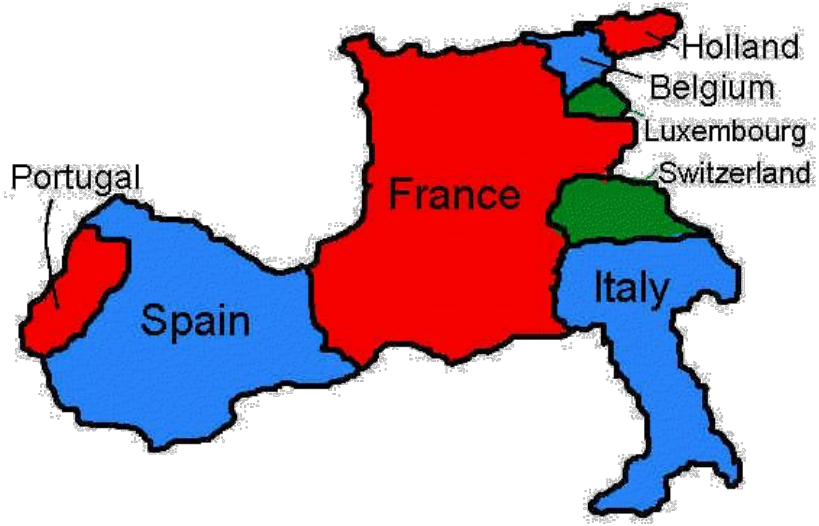
Backtracking Problems: N-Queens

```
1. algorithm NQueens( Q: array of size n, r: index of row)
2.    if r == n +1:
3.        print Q
4.    else:
5.        for j = 1 to n:
6.            legal = True
7.            for i =1 to r-1:
8.                  if(Q[i] == j) or (Q[i] == j + r - i) or(Q[i] == j - r + i):
9.                        legal = False
10.           if legal:
11.               Q[r] = j
12.               NQueens(Q, r+1)
```

Backtracking Problems: Map Coloring

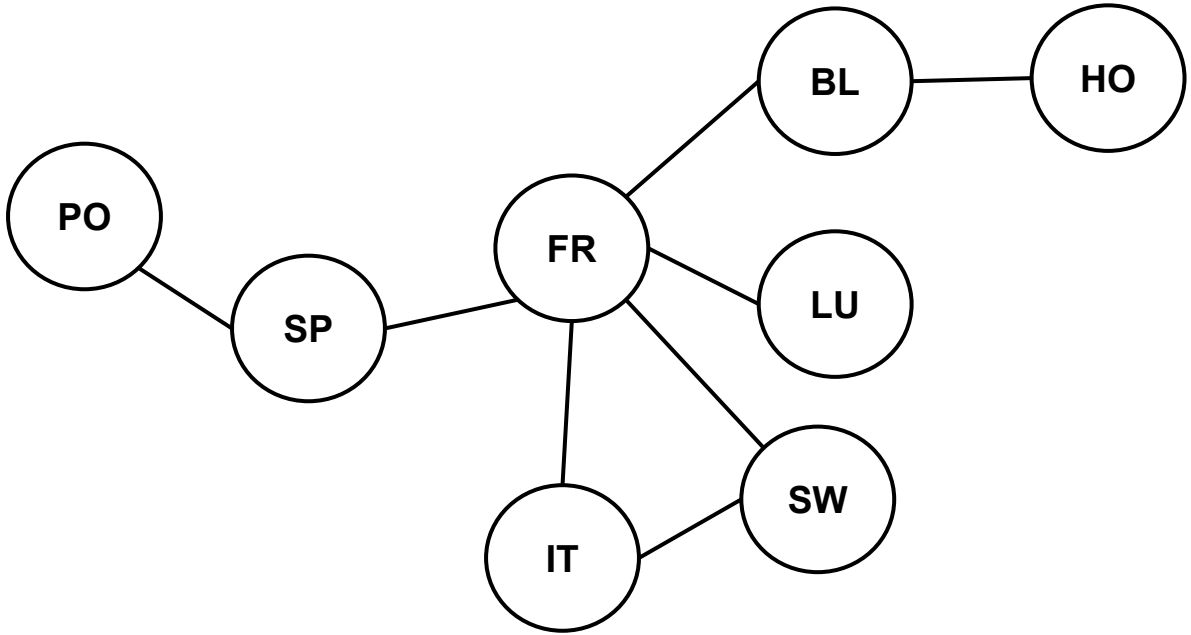
Map Coloring: Given a 2D map of n countries and a set of K colors, color every country differently from its neighbors (countries with shared borders). Use minimum number of colors.





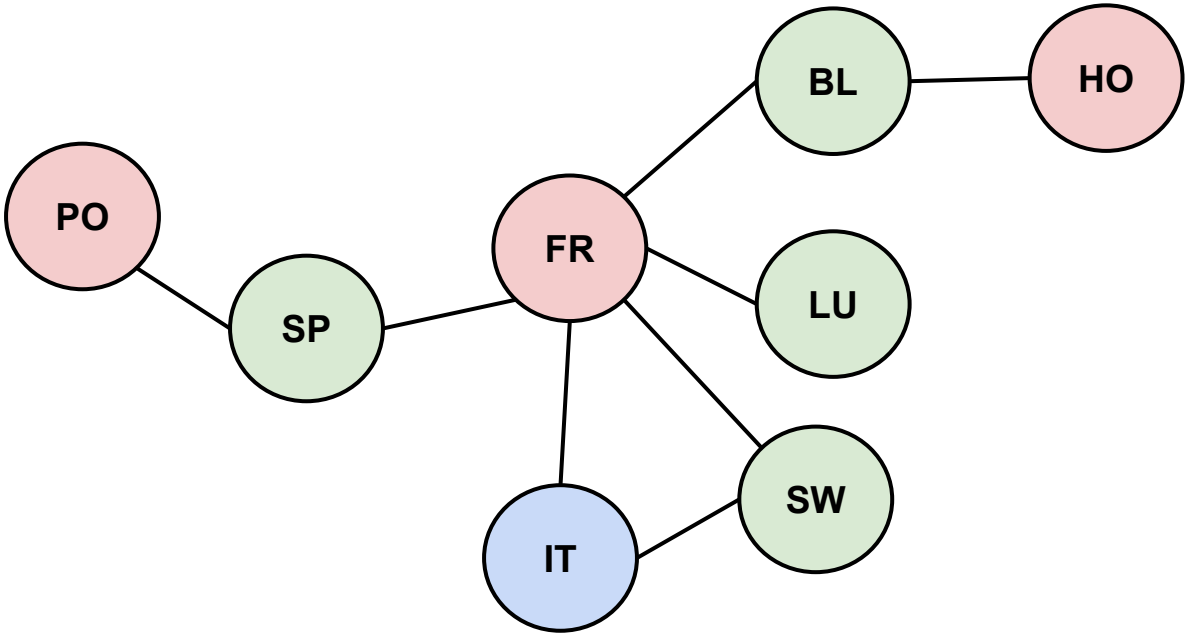
Backtracking Problems: Map Coloring

We will need to represent the problem using graph



Backtracking Problems: Map Coloring

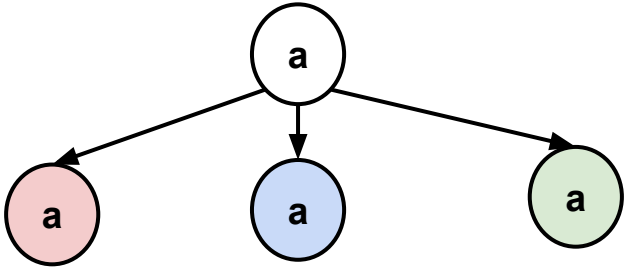
We will need to represent the problem using graph

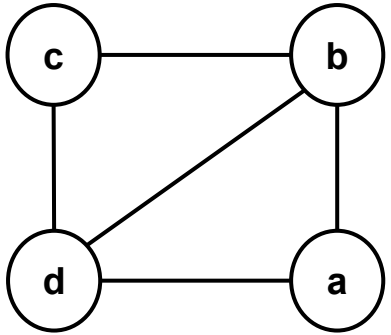


Backtracking Problems: Map Coloring

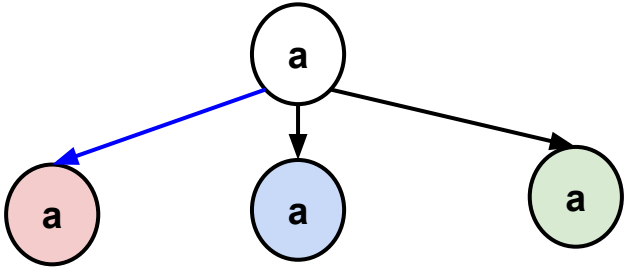
```
1. void MapColor(int k, int [ ] map. int n){
2.     if ((k+1) == n){
3.         print(map);
4.         return;
5.     }
6.     else{
7.         for (int c = 1; c <=m; c++){
8.             if (isSafe(k, c)){
9.                 map[k] = c;
10.                MapColor(k+1, map, n);
11.            }
12.        }
13.    }
14. }
```

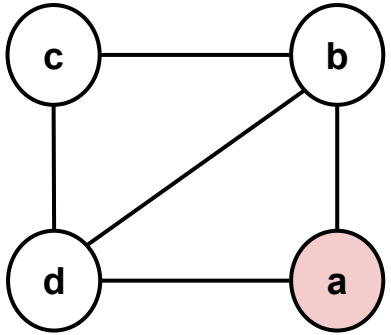
Backtracking Problems: Map Coloring



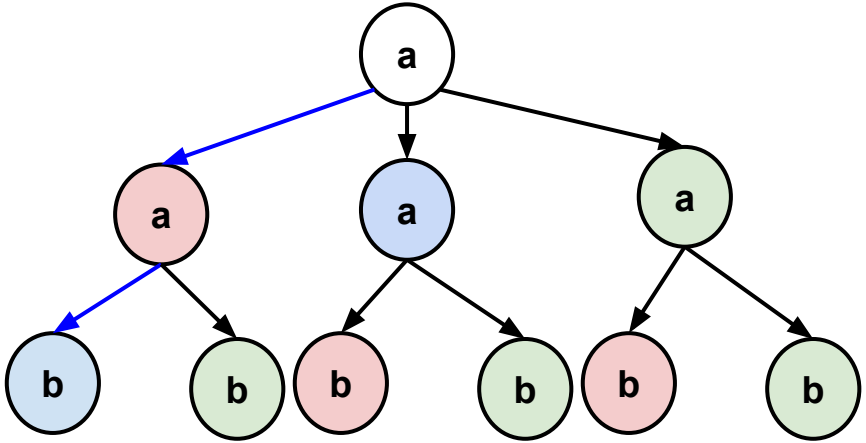


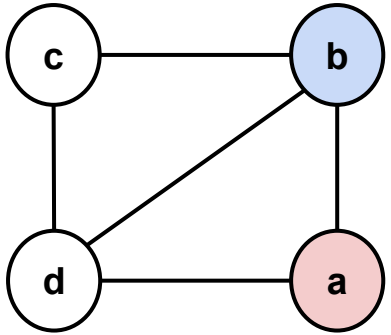
Backtracking Problems: Map Coloring



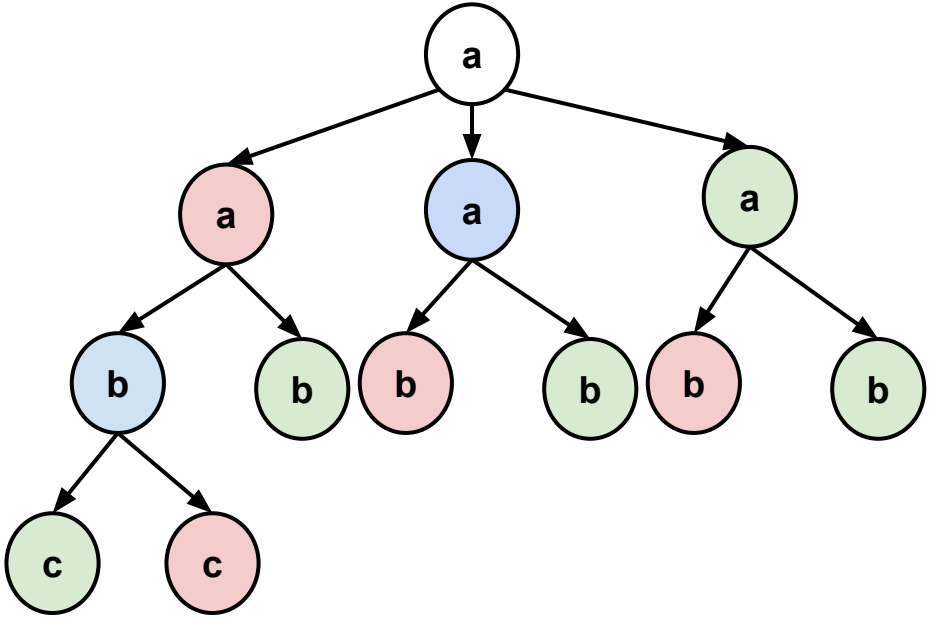


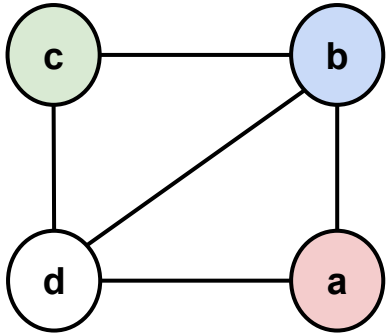
Backtracking Problems: Map Coloring



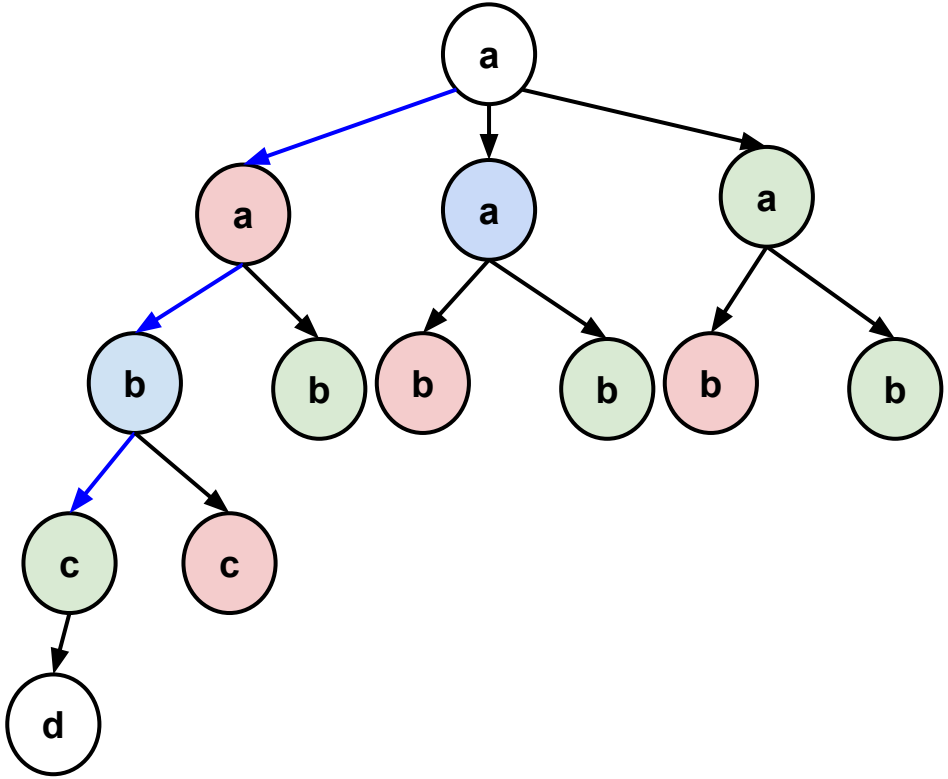


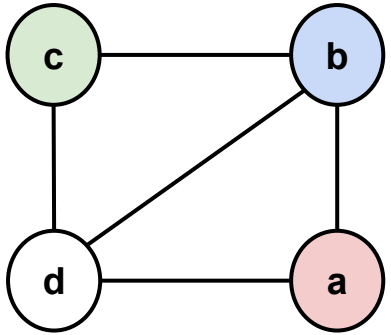
Backtracking Problems: Map Coloring



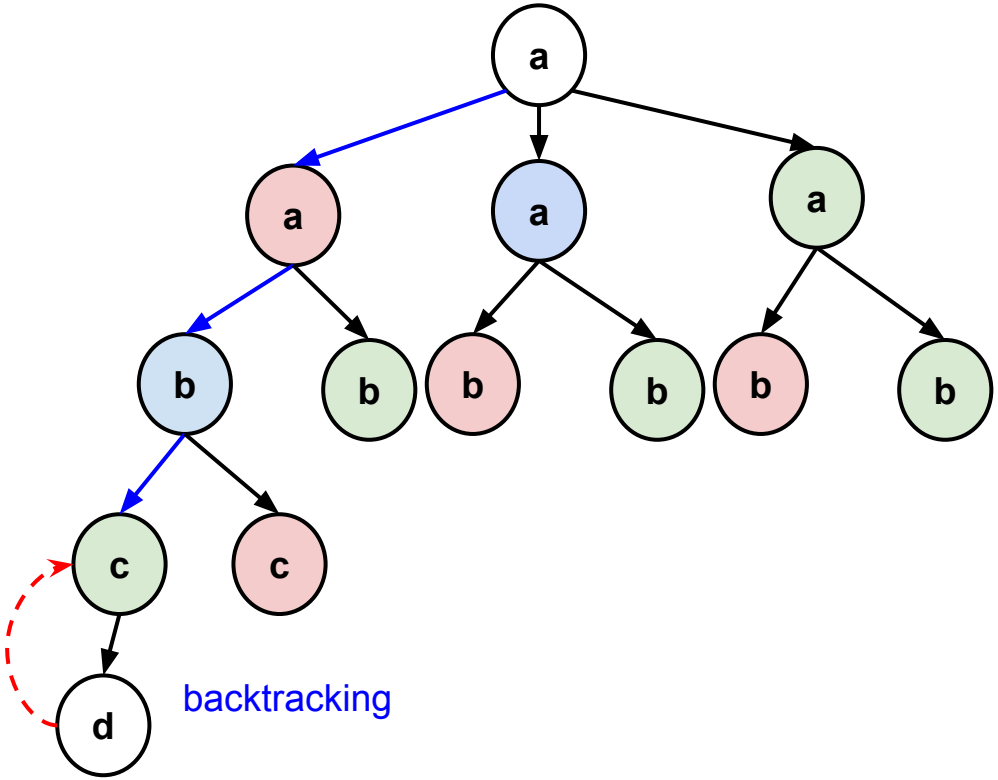


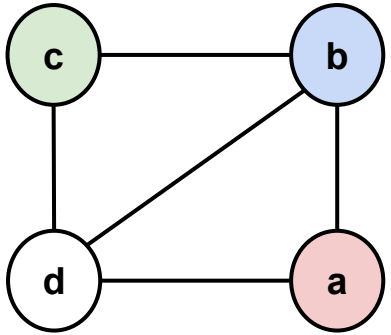
Backtracking Problems: Map Coloring



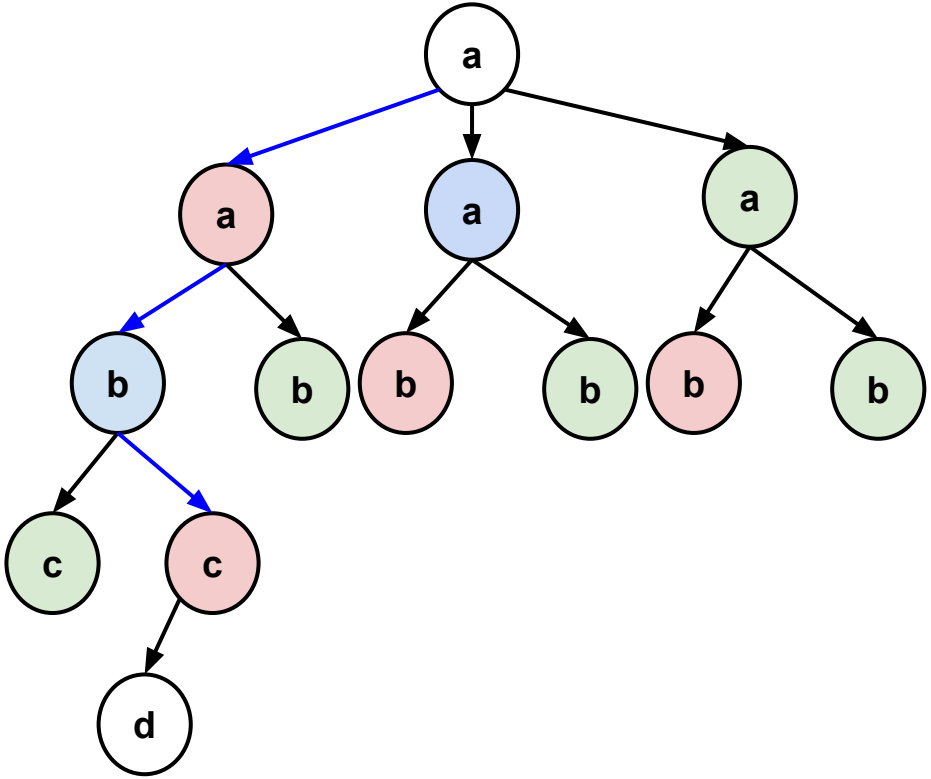


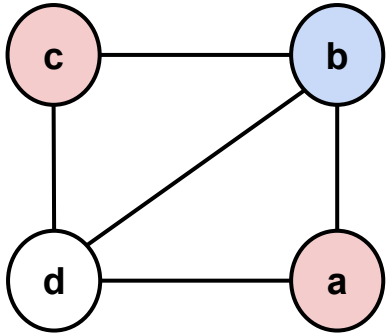
Backtracking Problems: Map Coloring



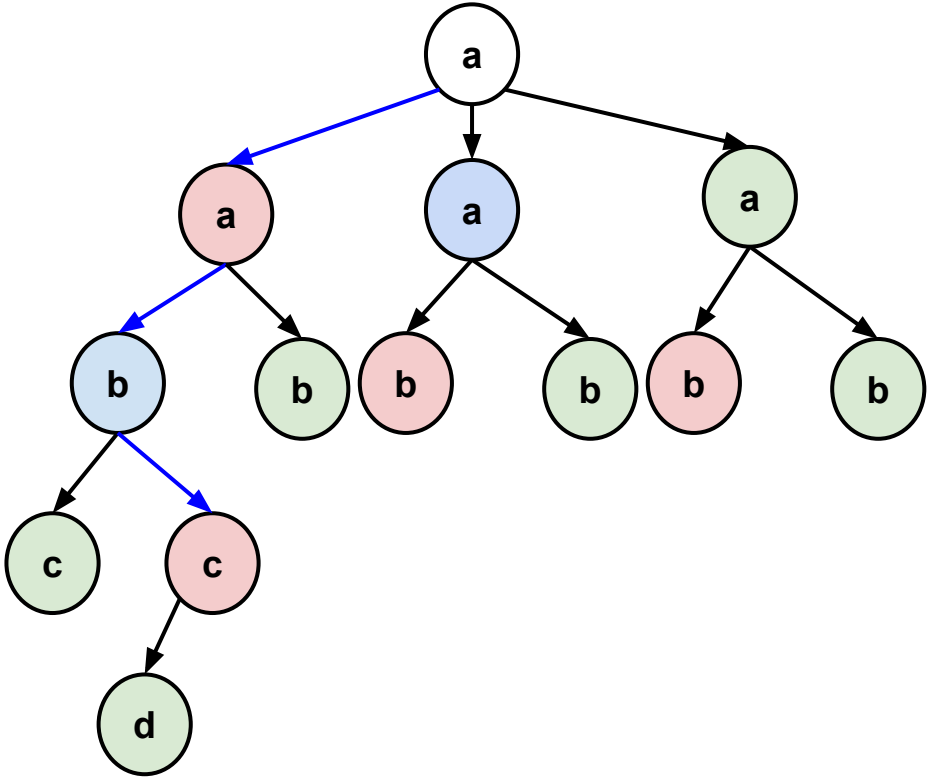


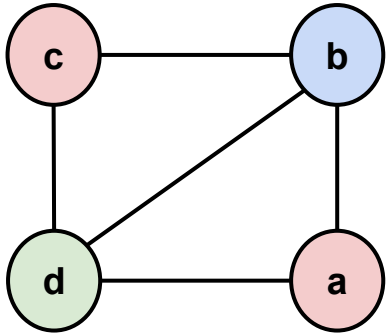
Backtracking Problems: Map Coloring





Backtracking Problems: Map Coloring

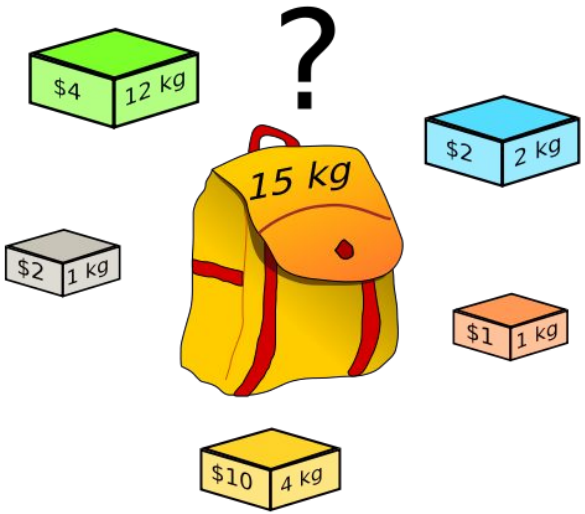




Backtracking Problems: 0-1 Knapsack

Given weights and values of ***n*** items, put these items in a knapsack of capacity ***W*** to get the maximum total value in the knapsack.

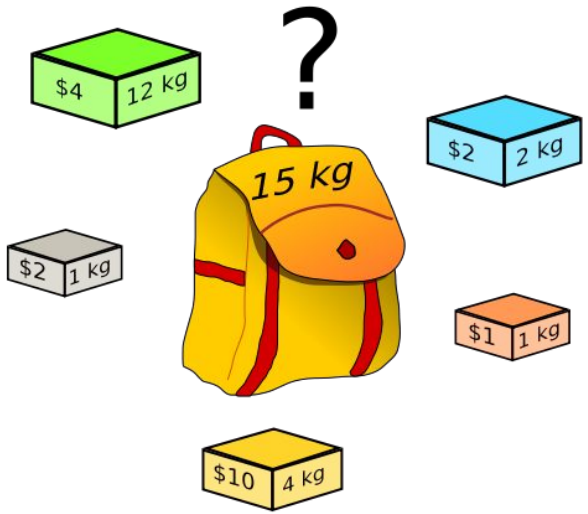
This an example of combinatorial optimization problem.



Backtracking Problems: 0-1 Knapsack

Assume we have the following set of items and a knapsack of capacity = 9. Find the items that maximize the profit with the capacity limit

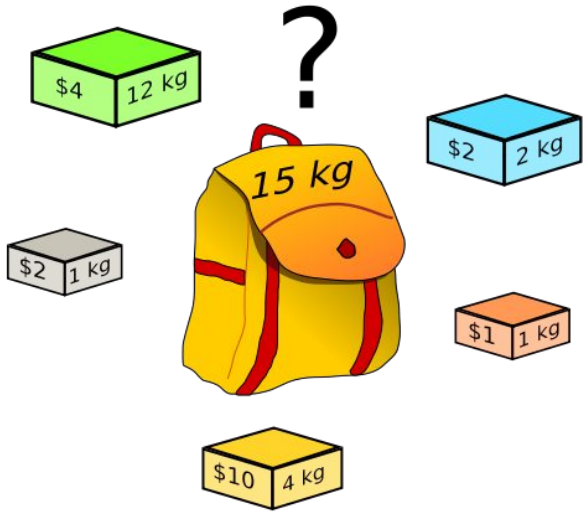
Item	A	B	C	D	E	F	G
Profit	6	5	8	9	6	7	3
Weight	2	3	6	7	5	9	4



0-1 Knapsack Problem: Greedy Solution

How big is the solution (search) space?

Item	A	B	C	D	E	F	G
Profit	6	5	8	9	6	7	3
Weight	2	3	6	7	5	9	4



One solution is [A, B, G] , where the profit = 14 and the weight = 9 (Can you come up with a better solution?)

Self-Assessment

Write an iterative algorithm to solve the n-queen problem. Note using iterative or recursive you will need to perform backtrack step anyway.