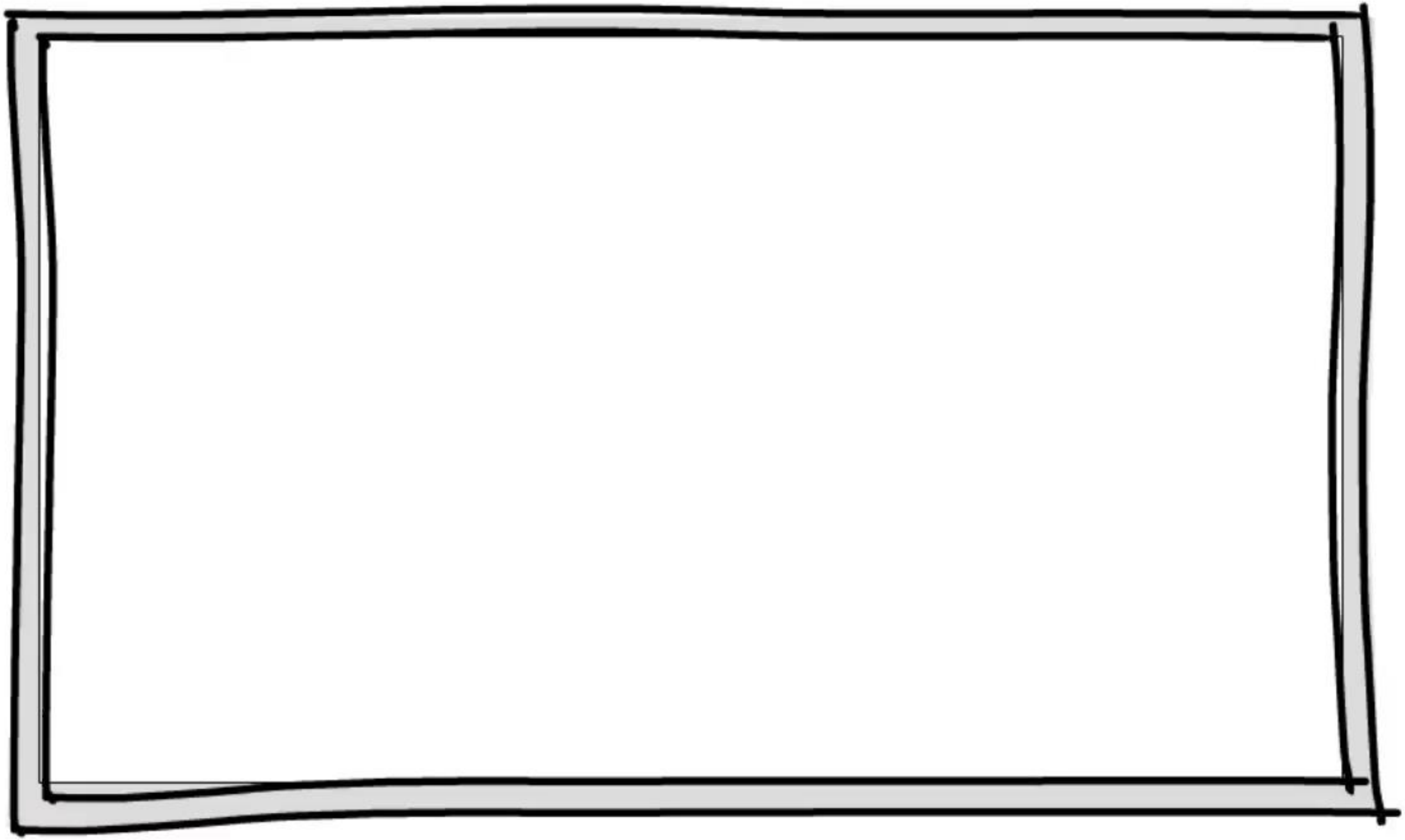
Backtracking

ALGORITHMICS

Vicente García Díaz
garciavicente@uniovi.es




Contents

BACKTRACKING

Basic concepts

Brute force to solve a problem

- **Brute force** or **exhaustive search should not** be used unless absolutely necessary

- For example, this simple game: *Magic Square*
  - There is a n x n table with n^2 distinct integers from 1 to n^2. The aim is to place the numbers so that the sum of the numbers in each row, column and corner-to-corner diagonal has to be the same

Brute force to solve a problem (II)

- How many ways are there to fill such a table?

?	?	?
?	?	?
?	?	?

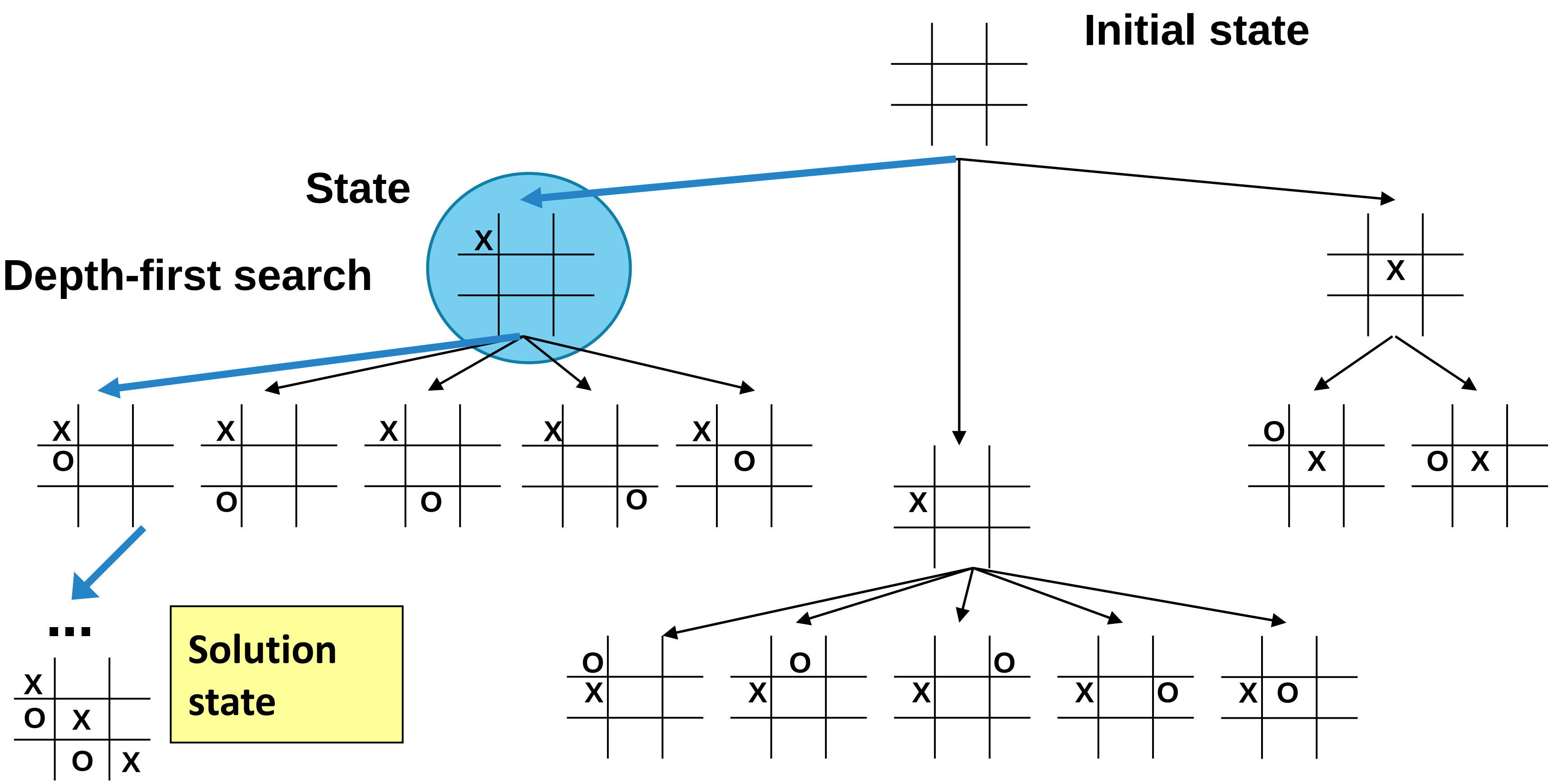
4	9	2
3	5	7
8	1	6

- There are 9 ways to place 1, 8 ways to place 2, and so until the last number 9 that is placed in the only unoccupied cell of the table
- If n = 3 ➔ 9! = 9*8*7*6*5*4*3*2*1 = 362880 ways to arrange the 9 numbers
- If n = 5 ➔ 25! ≈ 1.5 * 10^{25} ways to arrange the 25 numbers!!!

Problems represented as graphs

- Many problems can be represented as **abstract graphs**
- **Nodes** represent the state of the problem
- **Edges** represent valid changes
- To solve the problem, we must find a node or a path in the associated graph

Tree of states for the *3 in a row* game



Implicit graph

- If the graph contains a large or infinite number of nodes, it is impossible to explicitly construct it and use a conventional technique for searching in graphs
- We do not create the graph → **implicit graph**
- We have a description of its nodes and edges → **constructing parts of the graph only when the path is processed**
- We **save computation time** if the path is successful before we build the entire graph
- **Economy of space is obtained**, since nodes can be discarded after examination

Backtracking

- The implicit graph used for backtracking is usually depicted as a **tree**, called **tree of states**
- With the technique of backtracking, we perform a **depth-first search** of the tree
- The objective is to find the nodes (states) that are **solutions** to the problem
- To that end, an **exhaustive and systematic search** in the solution space of the problem is performed

Backtracking (II)

- The **solution** of a backtracking problem can be expressed as a vector
  - $(x_1, x_2, \ldots, x_n)$
- At each moment, the algorithm will stay at some level k, with a **partial solution**
  - $(x_1, \ldots, x_k)$
- If we can add a new item to the solution x_k, we generate it and we move to the level $k+1$

Backtracking (III)

- The path is **successful** if we can completely define a solution

- In this case, the algorithm may stop (if all we need is a solution to the problem)
  - → **first solution**

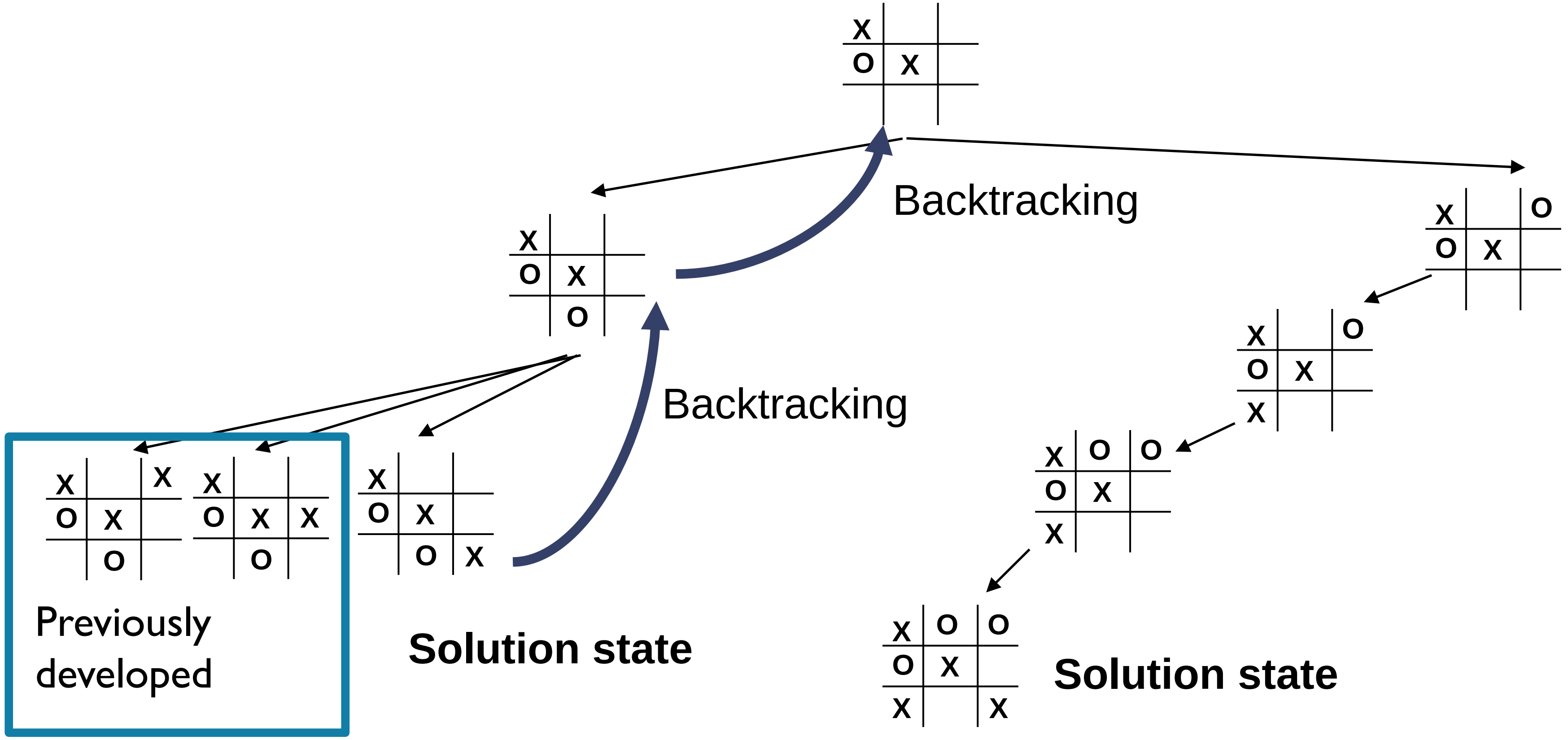
- Or keep looking for alternative solutions (if we want to examine all the possibilities)
  - → **all the solutions**

Backtracking (IV)

- The path is **not successful** if at any stage the partial solution created so far cannot be completed

- In such a case, the path **comes back**
  - Removing the elements that were added in each stage
  - When it returns to a node that has one or more children not yet explored, it continues the path through these nodes

Depth-first seach. Backtracking



Pseudocode. First solution and exit

```
boolean found = false
found = backtracking(initialState, …)
if (!found) "THERE IS NO SOLUTION"

boolean backtracking(State state, …)
  if (isSolution(state))
      print(state) //make things with the solution
      found = true
  else  //for all the j children of state
      if (!found)
           backtracking(j, …)
```

Pseudocode. All the solutions / Best solution / Worst solution

```
int count = 0
backtracking(initialState, …)
if (count == 0) "THERE IS NO SOLUTION"

int backtracking(State state, …)
   if (isSolution(state))
        print(state) //make things with the solution
        //here you could calculate if it is the best or worst solution
        count++
   //we put the else ONLY when below a solution state there cannot appear more
   solutions. If there can appear more solutions, we should remove the else
   else  //for all the j children of state
        backtracking(j, …)
```

Complexity analysis

- The execution time depends on the number of nodes generated and the time required for each node
- Usually, the time for each node is constant
- Assuming that a solution is of the form: $(x_1, x_2, \ldots, x_n)$, in the worst case there will be generated all the possible combinations for each x_i

Complexity analysis (II)

- If the number of possible values for each x_i is m (tree degree), then we will generate:
  - m_1 nodes at level 1
  - $m_1 \cdot m_2$ nodes at level 2
  - $m_1 \cdot m_2 \cdot \ldots \cdot m_n$ nodes at level n

- For a problem m = 2 (degree 2). The number of generated nodes:
  - $T(n) = 2 + 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 2 \rightarrow \mathbf{O(2^n)}$

- For a problem that the degree of the tree is reduced by one at each level:
  - $T(n) = n + n \cdot (n-1) + n \cdot (n-1) \cdot (n-2) + \ldots + n! \in \mathbf{O(n!)}$

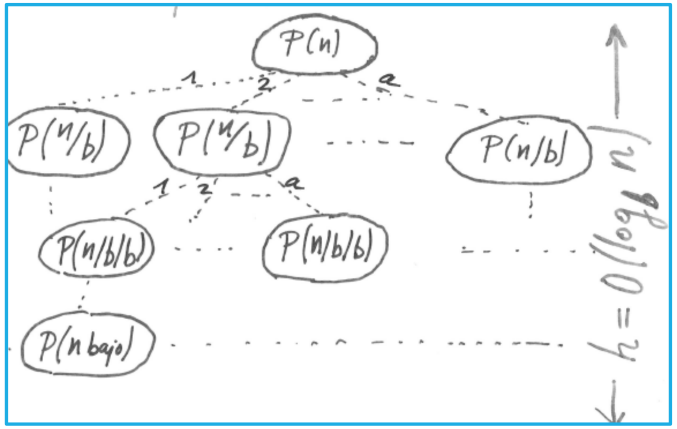
- In general, **factorial or exponential complexities**

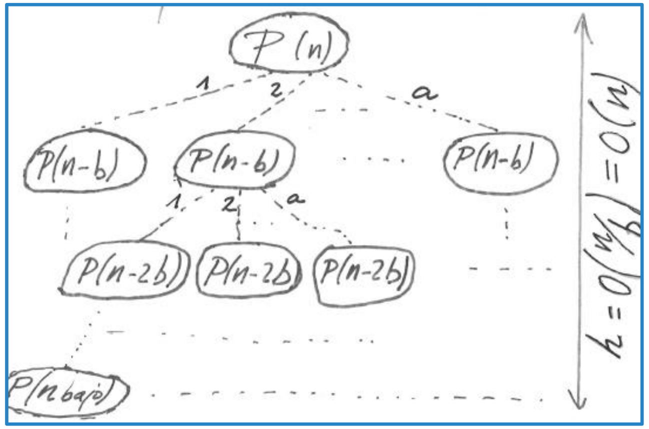
Memory consumption

- The memory consumption (M_{stack})
  - $M_{stack} = O(h) * O(f(n)) = O(h * f(n))$
  - h → height of the tree of calls
  - f(n) → waste of stack of each recursive call

- Assuming that each method has a stack waste of O(1)
  - $M_{stack} = O(h * 1) = O(h)$

- It is usually:
  - O(logn)
  - O(n)

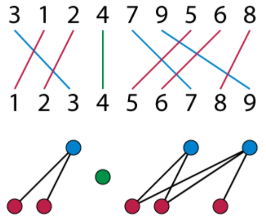




BACKTRACKING

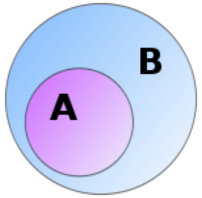
Examples of use

Permutations of elements



- We have some numbers (e.g., v = {0, 1, 2, 3})
- We need al algorithm that gives us all the permutations:
  - 0,1,2,3
  - 0,1,3,2
  - 0,2,3,1
  - 0,2,1,3
  - …
- Hints:
  - We can save all the solutions in the same array (e.g. `sol[]`)
  - We need to mark the elements that are already used in each of the solutions
  - A possible method: `backtracking(int level)`
    - At the beginning `level = 0` because we have nothing
    - At the end of each solution `level = n` because we have created a permutation

Subsets of a given sum



- We need a program that, given a set consisting of `n` different positive integers, computes all subsets which sum a given value `c`
- For example, with this numbers {1,2,3,4,5}, and looking for the sum 10
  - SUBSET THAT SUMS 10 = 2+3+5
  - SUBSET THAT SUMS 10 = 1+4+5
  - SUBSET THAT SUMS 10 = 1+2+3+4
- Hints
  - We don't need an array for the solutions (e.g. `sol[]`). A `sum` variable is enough
  - At the end of the tree of calls (when `level == n`) we also need to check if `sum == c` (if not, it is not a valid solution)
  - From any position of the array we have only 2 options (in the previous case we had `n-1` options)
    - Consider the item as part of the solution
    - Don't consider the item as part of the solution

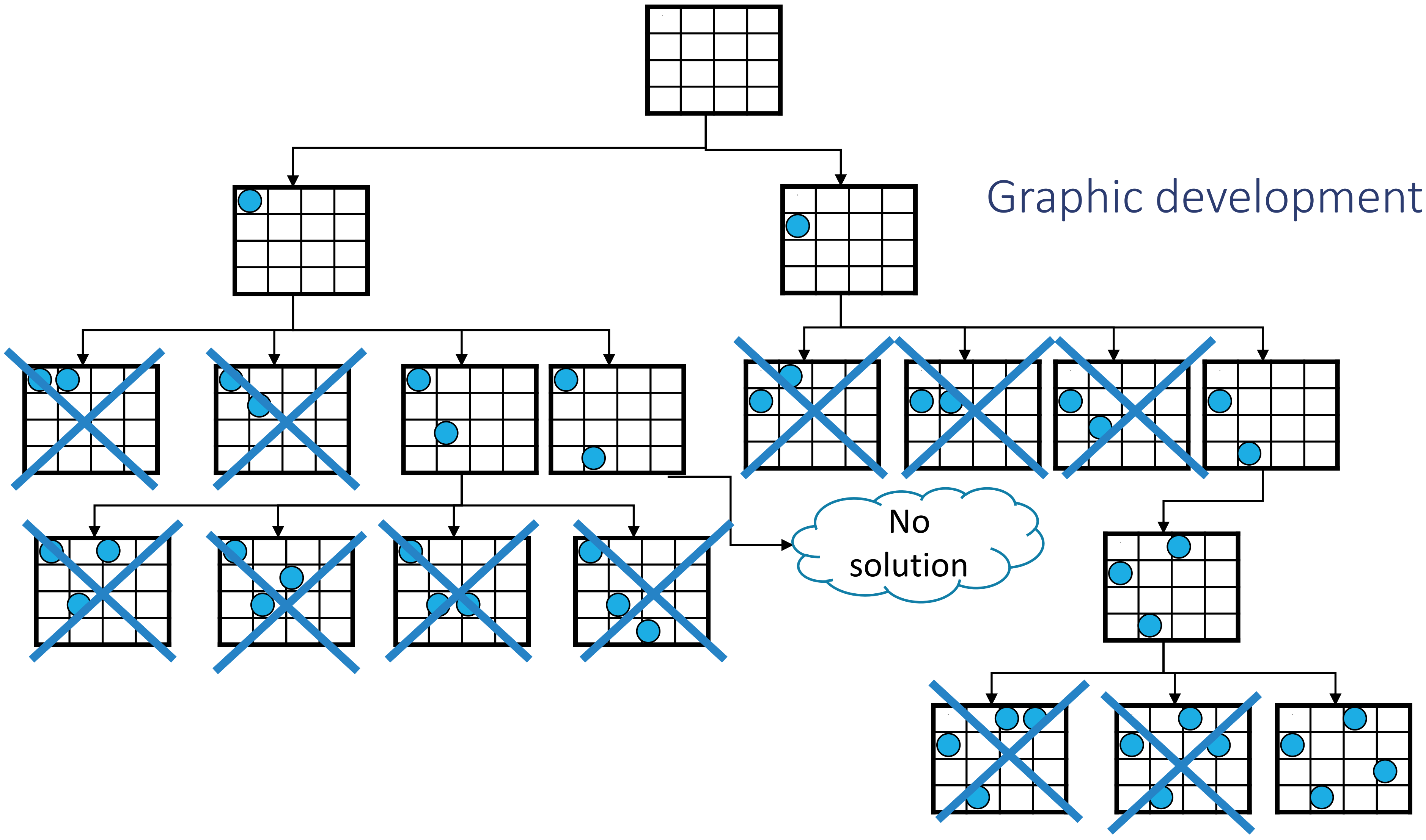
The problem of the n queens

- The aim is to place n queens on a chessboard without any queen can eat another queen
  - http://www.hbmeyer.de/backtrack/achtdamen/eight.htm#up

- A queen can move any number of cells horizontally, vertically or diagonally

- The difficulties in solving the problem are:
  - To find a suitable board representation
  - To always know if a movement is possible or not

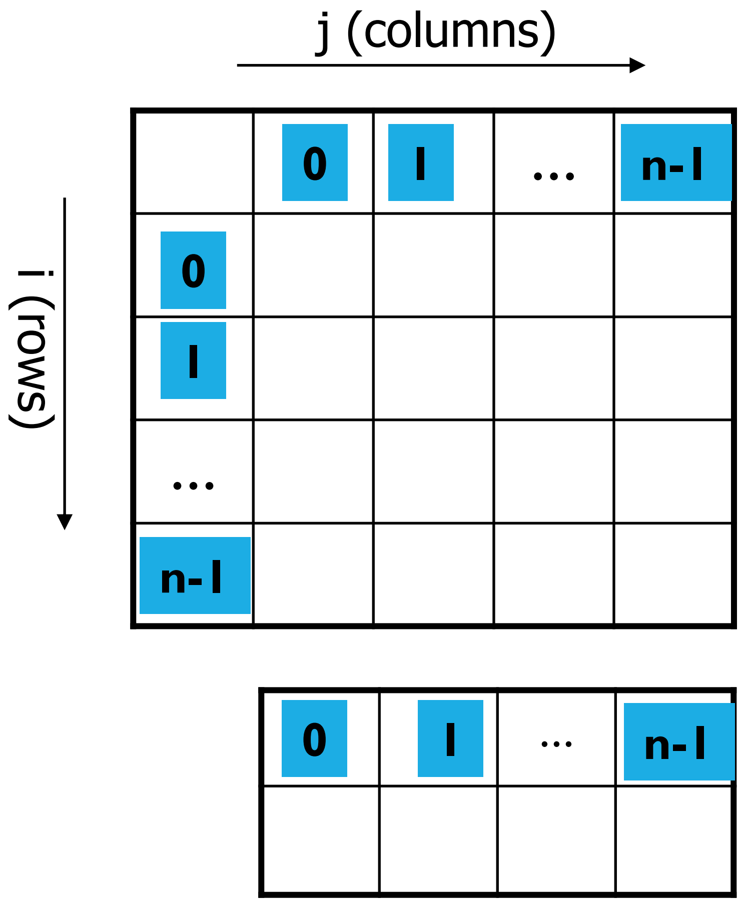
Graphic development



Solution!

Strategy

- **Board representation**
  - To represent the board, we can use a one-dimensional ad-hoc structure
    - `int[] sol = new int[n];`
    - If `sol[j] = i` → there is a queen in column `j` and row `i`



Strategy (II)

➢ Representing valid positions

- array [0..n-1] of boolean → `boolean[] a = new boolean[n]`
  - Rows
  - if a[i]= true → There is a queen in row `i`
  - if a[i]= false → There is **not** a queen in row `i`

- array [0..2*n-2] of boolean → `boolean[] b= new boolean[2*n-1]`
  - Diagonals 45° meet (i+j)=const
  - b[i+j]=true → There is queen in diagonal `i+j`
  - b[i+j]=false → There is **not** queen in diagonal `i+j`

- array[0..2*n-2] of boolean → `boolean[] c= new boolean[2*n-1]`
  - Diagonals 135° meet (i-j)=const
  - c[i-j+(n-1)]=true → There is queen in diagonal `i-j`
  - c[i-j+(n-1)]=false → There is **not** queen in diagonal `i-j`

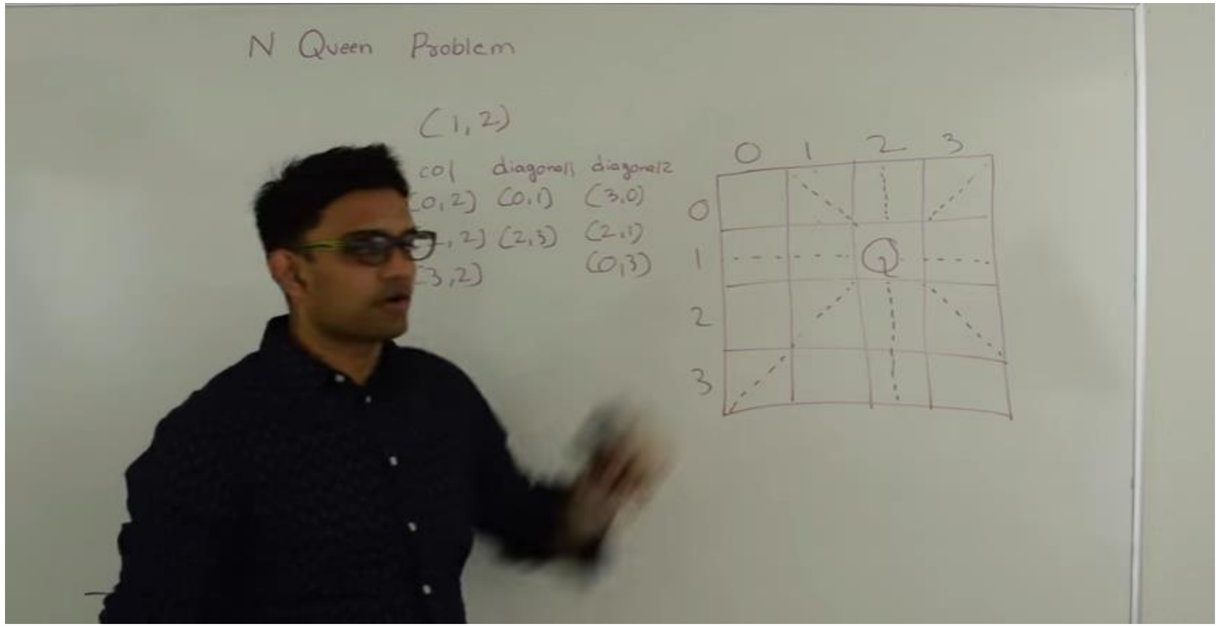
First solution

```
void backtracking(int j) {
    if (j==n) {  //we have already placed the n queens
        found = true;
        System.out.println("SOLUTION FOUND"); //print it
    }
    else
        for (int i=0;i<n;i++)
            if (!a[i] && !b[i+j] && !c[i-j+n-1] && !found){
                sol[j] = i; //a queen in column j and row i
                a[i] = true; //row used
                b[i+j] = true; //diagonal i+j used
                c[i-j+n-1] = true; //diagonal i-j used
                backtracking(j+1);
                sol[j] = -1; //we leave it as it was
                a[i] = false;
                b[i+j] = false;
                c[i-j+n-1] = false;
            } //if
    } //method
```

All the solutions

```
void backtracking(int j) {
    if (j==n) {  //we have already placed the n queens
        count++;
        System.out.println("SOLUTION FOUND"); //print it
    }
    else
        for (int i=0;i<n;i++)
            if (!a[i] && !b[i+j] && !c[i-j+n-1]){
                    sol[j] = i; //a queen in column j and row i
                    a[i] = true; //row used
                    b[i+j] = true; //diagonal i+j used
                    c[i-j+n-1] = true; //diagonal i-j used
                    backtracking(j+1);
                    sol[j] = -1; //we leave it as it was
                    a[i] = false;
                    b[i+j] = false;
                    c[i-j+n-1] = false;
            } //if
    } //method
```

backtracking()?

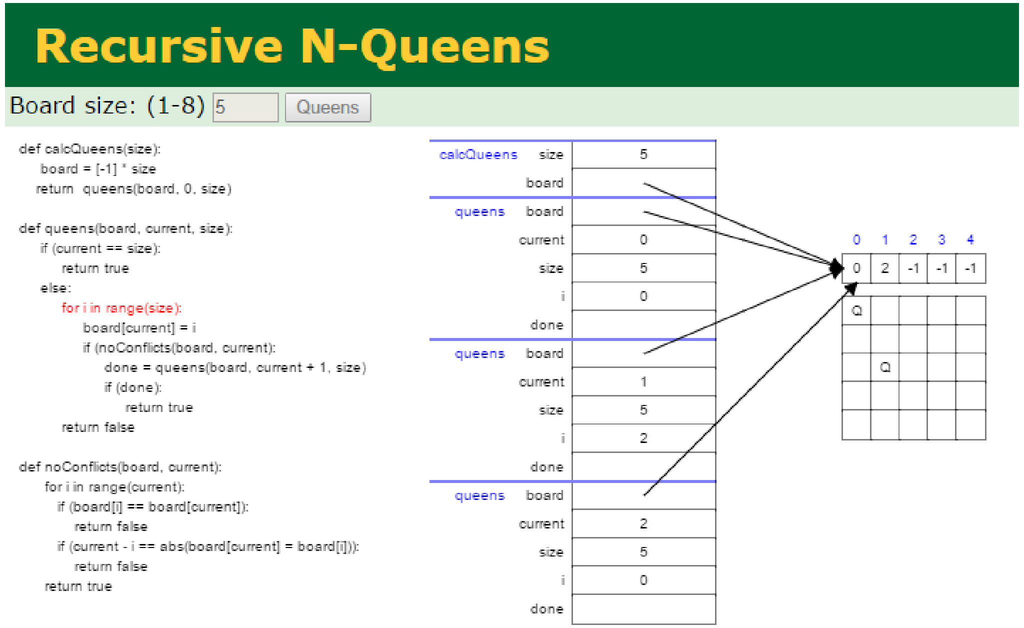


More at:

- N Queen Problem Using Backtracking Algorithm
  https://www.youtube.com/watch?v=xouin83ebxE

visualize it

- http://www.cs.usfca.edu/~galles/visualization/RecQueens.html



The horse jumping problem

- It consists in fully going through a chessboard based on the movements performed by the horse

- The cells that a horse can access from a given cell are shown in the following tables

	3		2	
4				1
		*		
5				8
	6		7	

1		7			
8				6	
	2				
	9				5
		3			
		10		4	

Strategy

- `board` → array representing the board
- `int [][] board= new int [n][n]`
  - `board[x][y]=0` the cell (x y) has not been visited
  - `board[x][y]=i` the cell (x, y) has been visited in the *i*th order
- `i` → number of movement of the horse
- `x, y` → coordinates of the last movement of the horse
- Initial cell `[0][1]`

	0	1	2	3	4	5	6	n
0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
n	0	0	0	0	0	0	0	0

Strategy (II)

- Rules of variation of the states
  - Possible movements of the horse are represented through two vectors:
    - Horizontal displacement, a
    - Vertical displacement, b
  - The order to choose the horse's movements has been established arbitrarily

	3º		2º	
4º				1º
		*		
5º				8º
	6º		7º	

a	-1	-2	-2	-1	1	2	2	1
b	2	1	-1	-2	-2	-1	1	2

Strategy (III)

- **Application of the scheme: “first solution”**
  - Recursive call, we pass as parameters the number of movement and the current coordinates
    - `backtracking(int i, int x, int y)` ← Parameters
  - Loop to select a state
    - `k` index ranging between `0` and `NUM_MOVEMENT` of the horse
    - Target position: `u= x+a[k]; v= y+b[k];`
    - `backtracking(i+1,u,v)` ← Call
  - Valid status conditions:
    - Verify that it is within the limits of the board
    - Check that the box has not been accessed
  - Solution condition:
    - `i == n*n` → We completed the board
  - Record new state → `board[u][v]= i;`
  - Remove previosly recorded state → `board[u][v]= 0;`

First solution

```
void backtracking(int i, int x, int y) {
    if (i==n*n+1) { //the horse has finished
        found=true;
        System.out.println("SOLUTION FOUND"); //print it
    }
    else
        for (int k=0;k<=7;k++) { //8 possibilities
            int u = x+a[k]; //target coordinate x
            int v = y+b[k]; //target coordinate y
            if (!found  && u>=0 && u<=n-1 && v>=0 && v<=n-
            1 && board[u][v]==0 ) {
                        board[u][v] = i; //we mark it
                        backtracking(i+1,u,v);
                        board[u][v] = 0; //we unmark it
            }
        } //for
} //method
```

backtracking()? (all the solutions)

Analysis

- `x,y` → coordinates on the board of the position of the horse at a given moment
- `u,v` → target coordinates of the horse

- Complexity analysis:
  - Tree of states (degree): 8
  - Tree of states (height): n^2
  - The maximum complexity assuming all valid states is $\mathbf{O(8^{n^2})}$

The problem of assigning tasks to agents

- We must assign j tasks to i agents, so that each agent performs only one task
- We have a matrix of costs, to know the cost of performing a task j by an agent i
- Goal: Minimizing the sum of the costs for executing the n tasks

j (tasks)

i (agents)

	1	**2**	**…**	**n**
a	11	12	…	40
b	14	15	…	22
…				
n	17	14	…	28

Solution

```
public void backtracking(int worker) {
    if (worker==n) {  //solution condition
        checkIfBestSolution(sol); //if the new solution
    improves the past solutions => we get a new best solution
    }
    else {
        for (int task=0; task<n; task++) {
            if (!assigned(task)) {
                sol[worker] = task;

                backtracking(worker+1);

                sol[worker] = -1; //we leave it as it was
            }
        } //for
    } //else
}
```

checkIfBestSolution()?
assigned()?
printBestSol()?

Meetoo
Real-time polling

On your mobile browser go to
web.meetoo.com
Or
Download Meetoo from your app store
and enter meeting ID

120-620-870

GET IT ON Google Play
Download on the App Store
Available as a Web App

POLL OPEN

If you need to solve a problem and you know how to solve it using a dynamic programming algorithm, but you also know how to solve it using a backtracking-based algorithm, which one you should usually use?

1. Backtracking
2. Dynamic Programming
3. Neither

30

POLL OPEN

For the Magic Square problem, with a size of $n=3$, how many ways to arrange the n^2 numbers are there using brute force?

1 27

2 9

3 81

✓ 4 362.880

5 About 1000

6 3

30

POLL OPEN

Many problems can be represented as abstract graphs. Nodes usually represent valid changes and edges usually represent the state of the problem. Thus, to solve the problem we must find a node or a path in the associated graph

1 True

✓ 2 False

30

POLL OPEN

There are usually two versions of backtracking algorithms. There can be:

"**first solution**" if we only need a solution to the problem

"**all the solutions**" if we need all the solutions, the best or the worst solution

1. True

0%

2. False

0%

30

POLL OPEN

Indicate which ones can be common complexities when working with backtracking

Vote for up to 3 choices

1. ✓ $O(n!)$
2. $O(n^2)$
3. $O(n)$
4. ✓ $O(2^n)$
5. ✓ $O(4^n)$
6. $O(n \log n)$

30

Meeting ID: 120-620-870

Which complexity is the worst?

POLL OPEN

1. $O(n!)$
2. $O(10^n)$
3. $O(n^n)$

30

POLL OPEN

The memory consumption with backtracking depends on the height of the tree of calls and the waste of stack for each recursive call

1. True
2. False

30

POLL OPEN

With backtracking, the height of the tree of states is usually XXX when we have a behavior by division and XXX when we have a behavior by subtraction

1 O(n) and O(logn)

2 O(logn) and O(n)

3 O(logn) and O(n^2)

4 O(n^2) and O(n)

5 O(n) and O(n)

6 O(logn) and O(long)

30

POLL OPEN

Working with backtracking to solve the problem of the n Queens with a board of size n=5, what is the time complexity of the algorithm?

1. O(4!)
2. O(n!)
3. O(2^n)
4. O(5^n)
5. O(n^n)

30

Bibliography