BABEŞ-BOLYAI UNIVERSITY
Faculty of Mathematics and Computer Science

Algorithms and Programming

Lecture 10 – Problem solving methods (I)

Mihai Ioan Popescu

Course content

Big image in programming

- Introduction in the software development process
- Procedural programming
- Modular programming
- Abstract data types
- Software development principles
- Testing and debugging

Detailed image in programming

- Recursion
- Complexity of algorithms
- Search and sorting algorithms
- **Problem solving methods**
  - **Generate and test, Backtracking**
  - **Divide et impera**

Last time

- Search
  - Sequential seach
  - Binary search

- Sort
  - Selection sort
  - Insert sort
  - Bubble sort
  - Quick sort

Today

- Problem solving methods
    - Types

    - Techniques
        - Exact methods
        - Heuristic methods

    - Algorithms
        - Backtracking
        - Divide and conquer

Problem solving methods

- Strategies for solving difficult problems
- General algorithms that can be applied to solve certain type of problem (the problem needs to satisfy certain required criteria)

- Problem characteristics
  - Structure
  - Number of solutions
  - Search, optimization, simulation, etc

Problem types

- **By structure**
  - Problems that can be divided in sub-problems
    e.g. search for an element in a list
  - Problems that can not be divided in sub-problems
    e.g. place queens on a chessboard
- **By number of solutions**
  - Problems with a single solution
    e.g. sort a list
  - Problems with several solutions
    e.g. generate permutations
- **By solving possibilities**
  - Problems that can be deterministically solved
    e.g. compute the sin or the square root of a number
  - Problems that can be solved stochastically (heuristics)
    e.g. Real-world problems such as vehicle routing optimization
    Need to *search* for a solution

Problem types

- **By run time complexity**
  - Problems from class P – can be solved in polynomial time (n^2, n^3,...)
    e.g. sorting problems
  - Problems from class NP – can be solved in non-deterministic polynomial time (but for deterministic time we would have: n!, 2^n,...)
    e.g. the shortest path in a graph of cities
- **By scope**
  - Search / optimization problems
    e.g. planning, scheduling, resource allocation
  - Modeling problems
    e.g. forecasting, classification, prediction
  - Simulation
    e.g. economic game theory

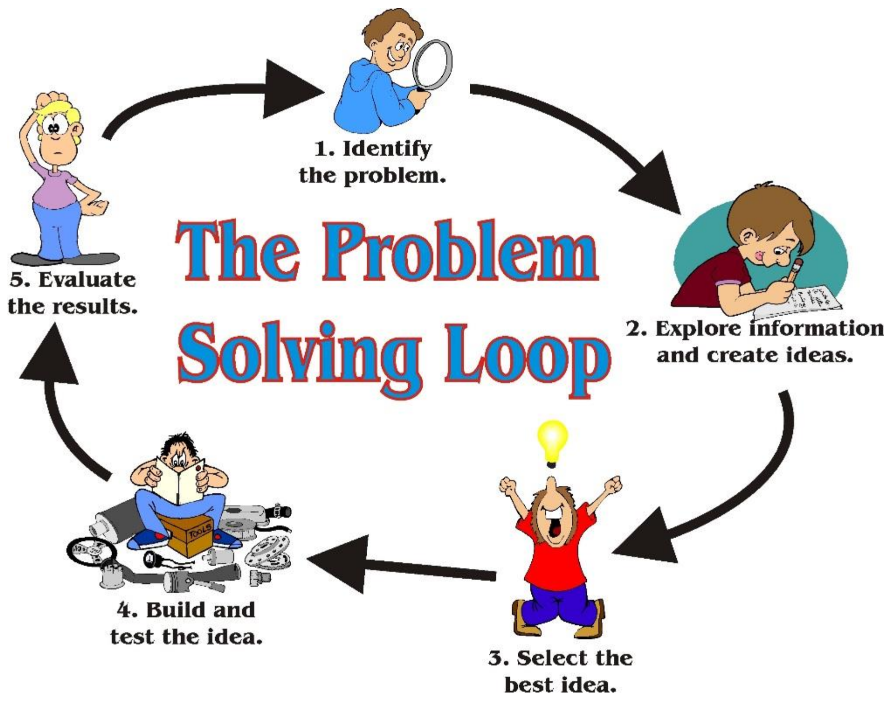
Problem solving

- Identification of a solution
  - Computer science – search process
  - Engineering and mathematics – optimization process

- How?
  - Representation of (partial) solutions – points in the search space
  - Design of search operators – transform a possible solution in a new solution

The problem solving loop

- Problem definition
- Problem analysis
- Choose problem solving technique
    - **Search**
    - Knowledge representation
    - Abstraction



Problem solving steps

- Choose a problem solving technique
  - Solve using rules (and a control strategy) to move in the search space until a path from the initial state to the final one is identified
  - Solve using search
    - Sistematically analyse states in order to identify:
      - A path from initial state to the final one
      - An optimal state
      - Search space – all possible states and the operators that allow moving from a state to another
    - How to choose the search strategy?
      - Computational complexity (run time and space)
      - Completeness – the algorithm always ends and finds a solution if one exists
      - Optimality – the algorithm finds the optimal solution

Problem solving by search

- Many search strategies – how to choose one?
  - Computational complexity
    - Performance depends on:
      - Time needed to run the algorithm
      - Space (memory) needed for the run
      - Size of the input data
      - Computer speed
      - Processor quality

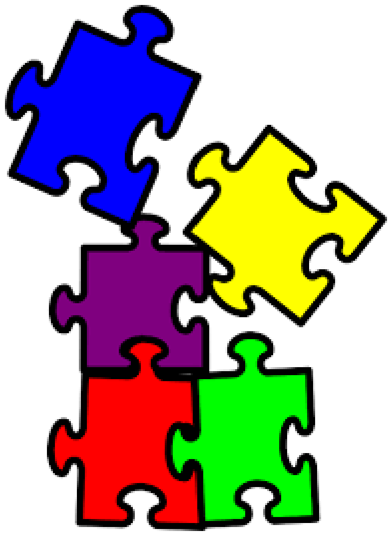
Internal factors

External factors

  - Measured using complexity – ***Computational Efficiency***
    - **Space** – memory needed to identify the solution
      - ***S(n)*** – quantity of memory used by the best algorithm A which solves a decision problem f with input data of size n
    - **Time** – time needed to identify the solution
      - ***T(n)*** – running time (number of steps) used by the best algorithm A for a decision problem f with input data of size n

Problem solving by search

- Solving problems by search can mean:
  - Build the solution step by step
  - Identify the potential optimal solution



Problem solving by search

- Solving problems by search using standard methods

  - Exact methods
    - **Generate and test**
      - **Backtracking**
    - **Divide and conquer**
    - Dynamic programming

  - Heuristic methods
    - Greedy method

Generate and test

- Basic idea
  - Generate a possible solution and verify if it’s correct
  - Trial and error
  - Exhaustive search
- Mechanism
  - Generate: determine all possible solutions
  - Test: search solutions that are correct (satisfy some conditions)
- When to use it?
  - Problems that can have multiple solutions
  - Problems with restrictions (solutions need to satisfy some conditions)

Generate and test

- Algorithm

```
#D = D(D1) = D(D1(D2))...
def generate_test(D):
    while (True):
        sol = generate_solution()
        if (test(sol) == True):
            return sol
```

1. Generate a possible solution
2. Test is solution is correct
3. Quit if a solution is found, return to step 1 otherwise

➢ This is not backtracking

Generate and test

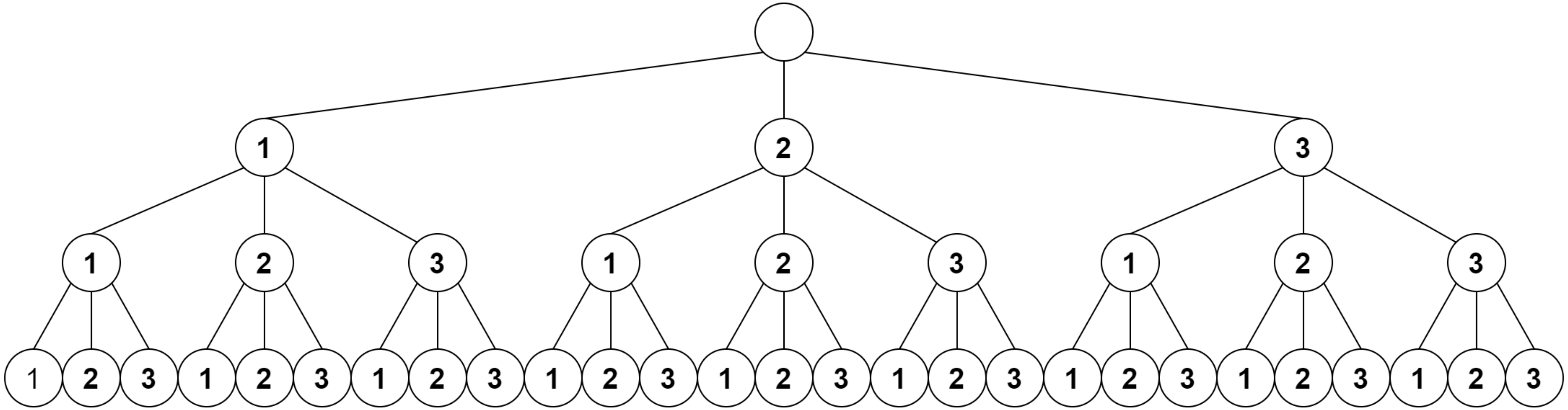
- Example: generate permutations with n=3 elements

```python
def permut3():
    for i in range(1,4):
        for j in range(1,4):
            for k in range(1,4):
                # generate
                possibleSolution = [i,j,k]
                #test
                if i!=j and j!=k and k!=i:
                    yield possibleSolution

def callPermut3():
    for p in permut3():
        print(p)

callPermut3()
```

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

Generate and test

- Example: generate permutations with n=3 elements
- Complexity
  - Number of possible solutions: 3^3 (which is n^n)



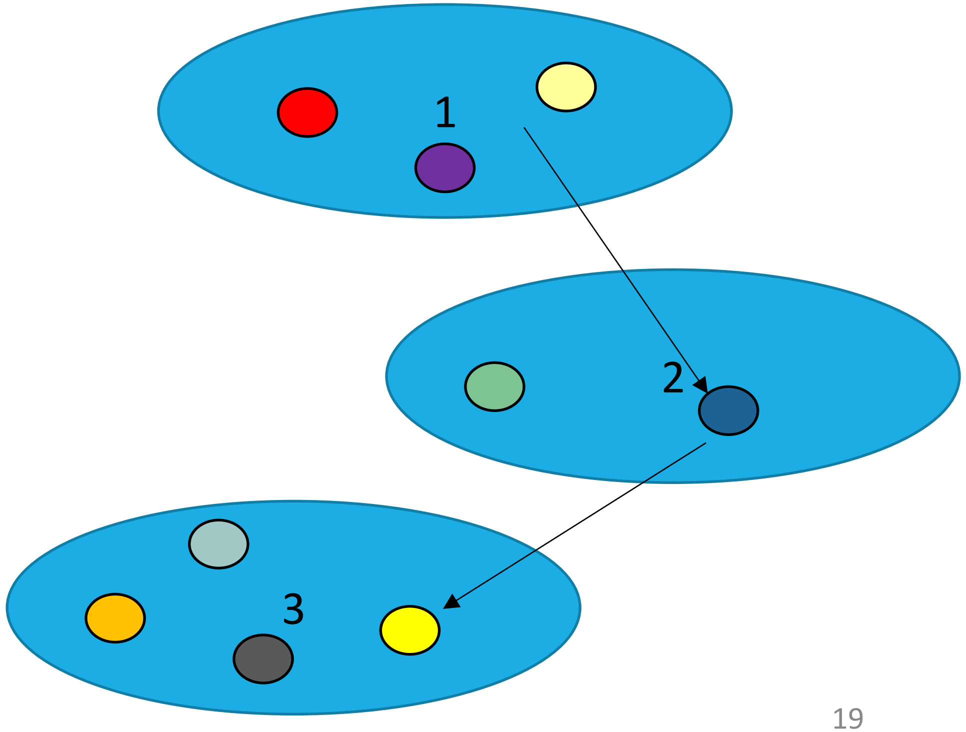
Generate and test

- Possible improvements
  - Do not explore all possible solutions
    - Example: when i= 1 there is no point to verify j=1 and k=1 because this can not lead to a possible solution
  - Build (partially) correct solutions
    - That satisfy certain conditions

```
#D = D(D1) = D(D1(D2))...
def generate_test(D):
    while (True):
        sol = generate_solution_cond()
        if (test(sol) == True):
            return sol
```

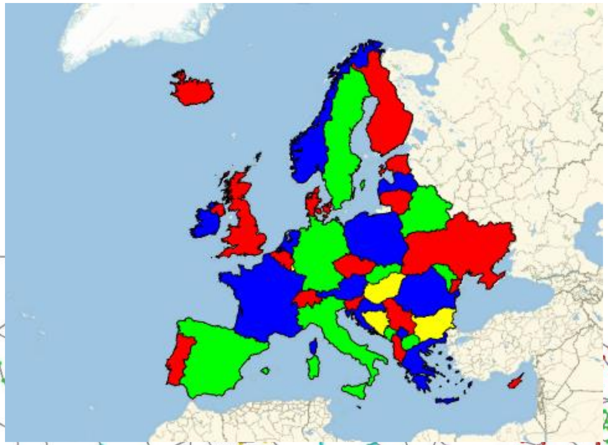
Backtracking

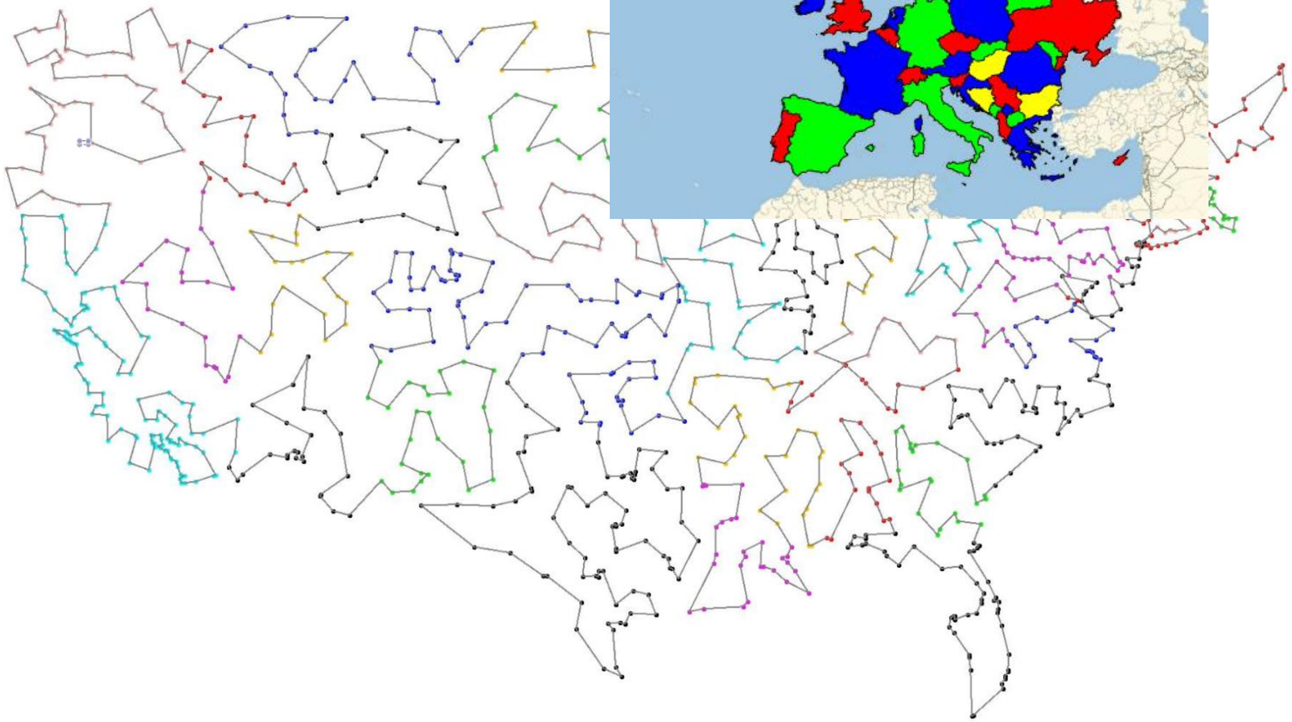
- Brute-force technique for finding solutions, with the main characteristic that it has the ability to undo – *backtrack* – when a potential solution is not valid
- Basic idea:
  - Try every possibility to see if it's a solution
    - unless we already know it's not valid
  - Sequence of choices
    - Once a choice is selected....another choice
    - If bad choice => backtrack
    - Until the solution is perfectly valid



Backtracking

- Problems with many candidate solutions
- Many of these solutions do not satisfy the given constraints
- Examples of problems:
  - N-Queens Problem
  - Sudoku
  - K-colouring maps of (n regions)
  - Traveling Salesperson Problem





Backtracking

- Search space of a solution **s** is ***S*** (definition domain)
- A solution is formed of several elements *s[0], s[1], s[2],...*
- *init:* function that generates an empty value for the definition domain of the solution
- *getNext:* function that returns the next element from the definition domain
- *isConsistent:* function that verifies if a (partial) solution is consistent
- *isSolution:* function that verifies if a (partial) solution is a final (complete) solution of the problem

Backtracking: Iterative version

Generate permutations with n elements

```python
def init():
    return 0

def getNext(sol, pos):
    return sol[pos] + 1

def isConsistent(sol):
    isCons = True
    i = 0
    while (i<len(sol)-1) and (isCons==True):
        if (sol[i] == sol[len(sol) - 1]):
            isCons = False
        else:
            i = i + 1
    return isCons

def isSolution(solution, n):
    return len(solution) == n
```

```python
def permut_back(n):
    k = 0; solution = []
    initValue = init()
    solution.append(initValue)
    while (k >= 0):
        isSelected = False
        while (isSelected==False) and (solution[k]<n):
            solution[k] = getNext(solution, k)
            isSelected = isConsistent(solution)
        if (isSelected == True):
            if (isSolution(solution,n) == True):
                yield solution
            else:
                k = k + 1
                solution.append(init())
        else:
            del(solution[k])
            k = k - 1

def callPermut():
    for p in permut_back(3):
        print(p)

callPermut()
```

Backtracking: Recursive version

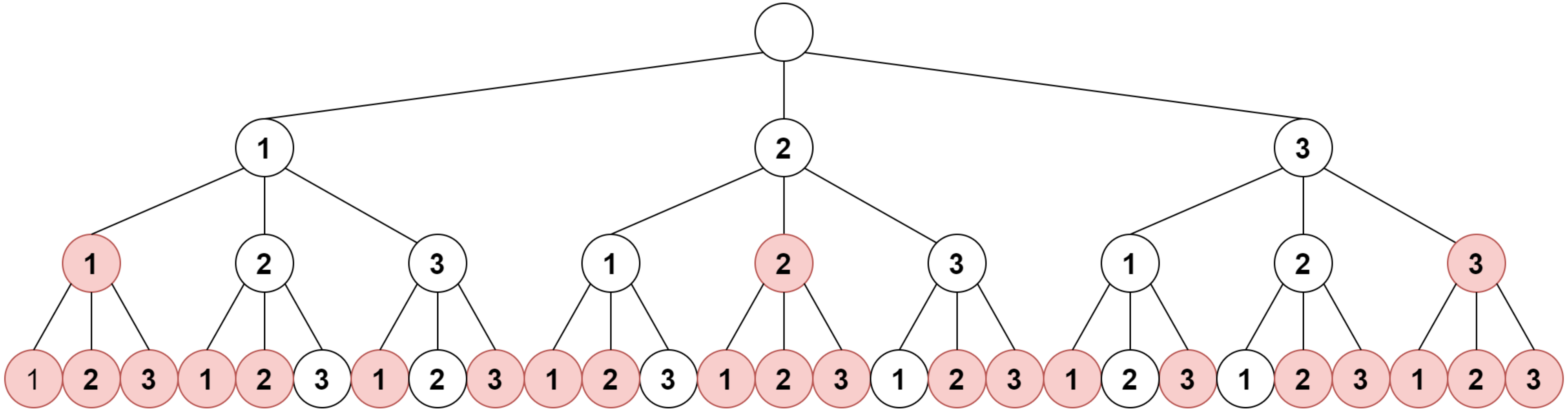
Generate permutations with n elements

```python
def init():
    return 0

def getNext(sol, pos):
    return sol[pos] + 1

def isConsistent(sol):
    isCons = True
    i = 0
    while (i<len(sol)-1) and (isCons==True):
        if (sol[i] == sol[len(sol) - 1]):
            isCons = False
        else:
            i = i + 1
    return isCons

def isSolution(solution, n):
    return len(solution) == n
```

```python
def permut_back_rec(n, solution):
    initValue = init()
    solution.append(initValue)
    elem = getNext(solution, len(solution) - 1)
    while (elem <= n):
        solution[len(solution) - 1] = elem
        if (isConsistent(solution) == True):
            if (isSolution(solution, n) == True):
                yield solution
            else:
                yield from permut_back_rec(n, solution[:])
        elem = getNext(solution, len(solution) - 1)

def callPermutRec():
    for p in permut_back_rec(3, []):
        print(p)

callPermutRec()
```

Backtracking

- Nodes explored for generating permutations with n=3 elements

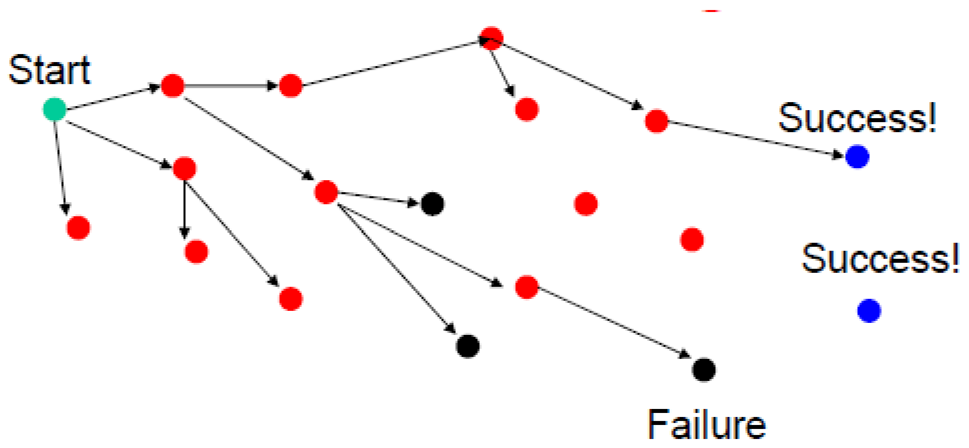


Recap: How to use backtracking

- Represent the solution as a vector: *s[0], s[1], s[2],...*
- Define what a valid solution candidate is

(filter out candidates that

will not lead to a solution)

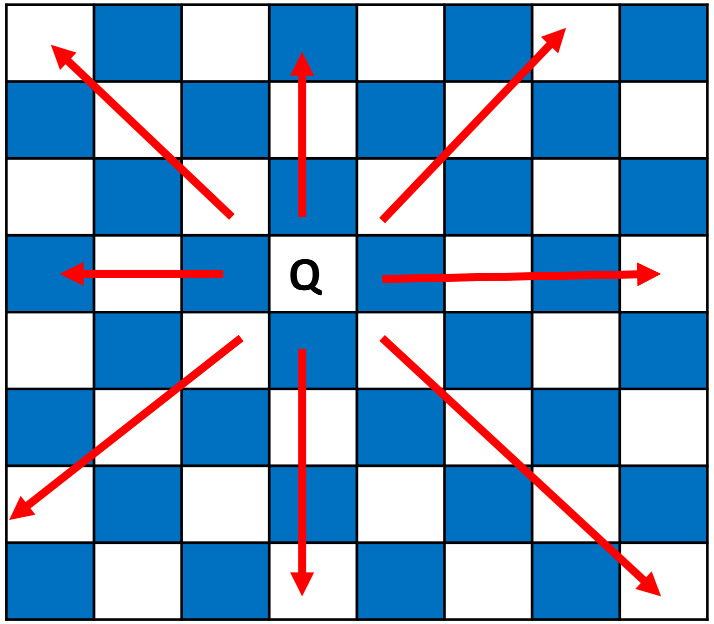


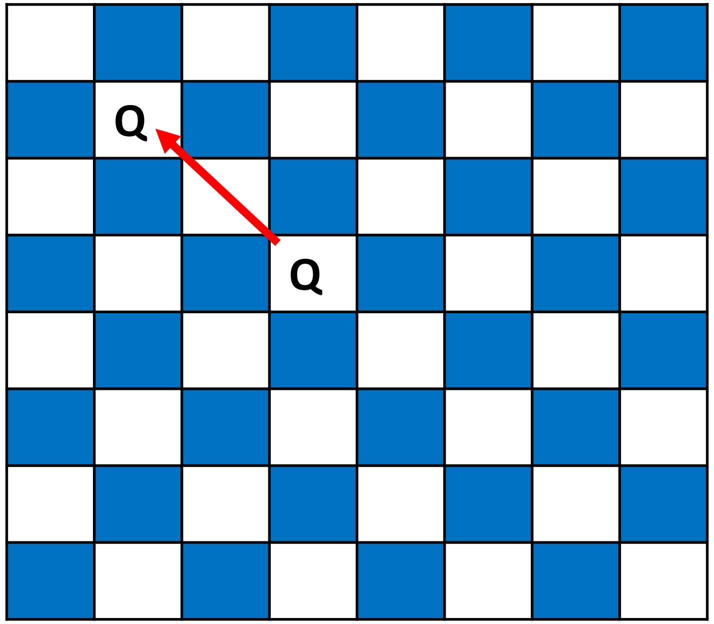
Remember:

- Problem space: states (nodes) and actions (paths that lead to new states)
- If a node leads to failure go back and try other alternatives

Backtracking: Example
8 queens

- 8 queens
  - Classic backtracking problem
  - Place 8 queens on an 8x8 chessboard so that no queen can attack another





Backtracking: Example

8 queens

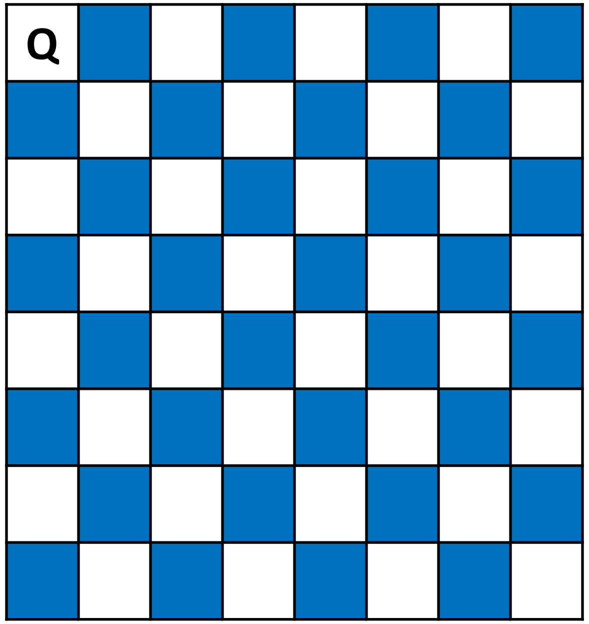
- 8x8 chessboard => 64 locations
  - After placing one queen => 63 locations to choose from
  - ....
  - 64*63*62*61*60*59*58*57 = 178,462,987,637,760 possibilities

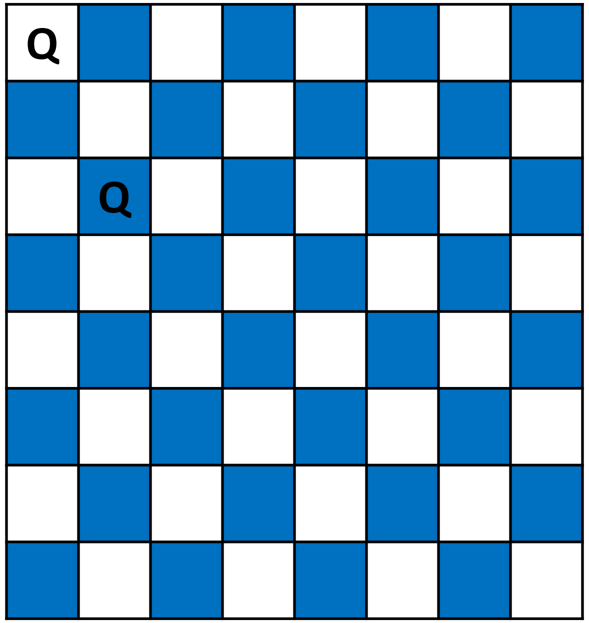
- However:
  - A valid solution has:
    - exactly 1 queen in each row and exactly 1 queen in each column
  - Explore 1 queen per column (not per cell)
  - Possibilities reduced to 8^8 = 16,777,216

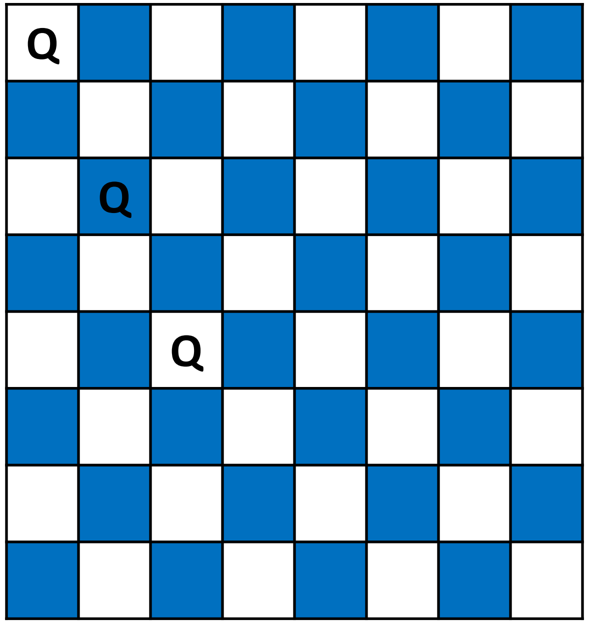
Backtracking: Example

8 queens

- Make a choice for first column
- The second choice is affected by the first choice, etc







Backtracking: Example

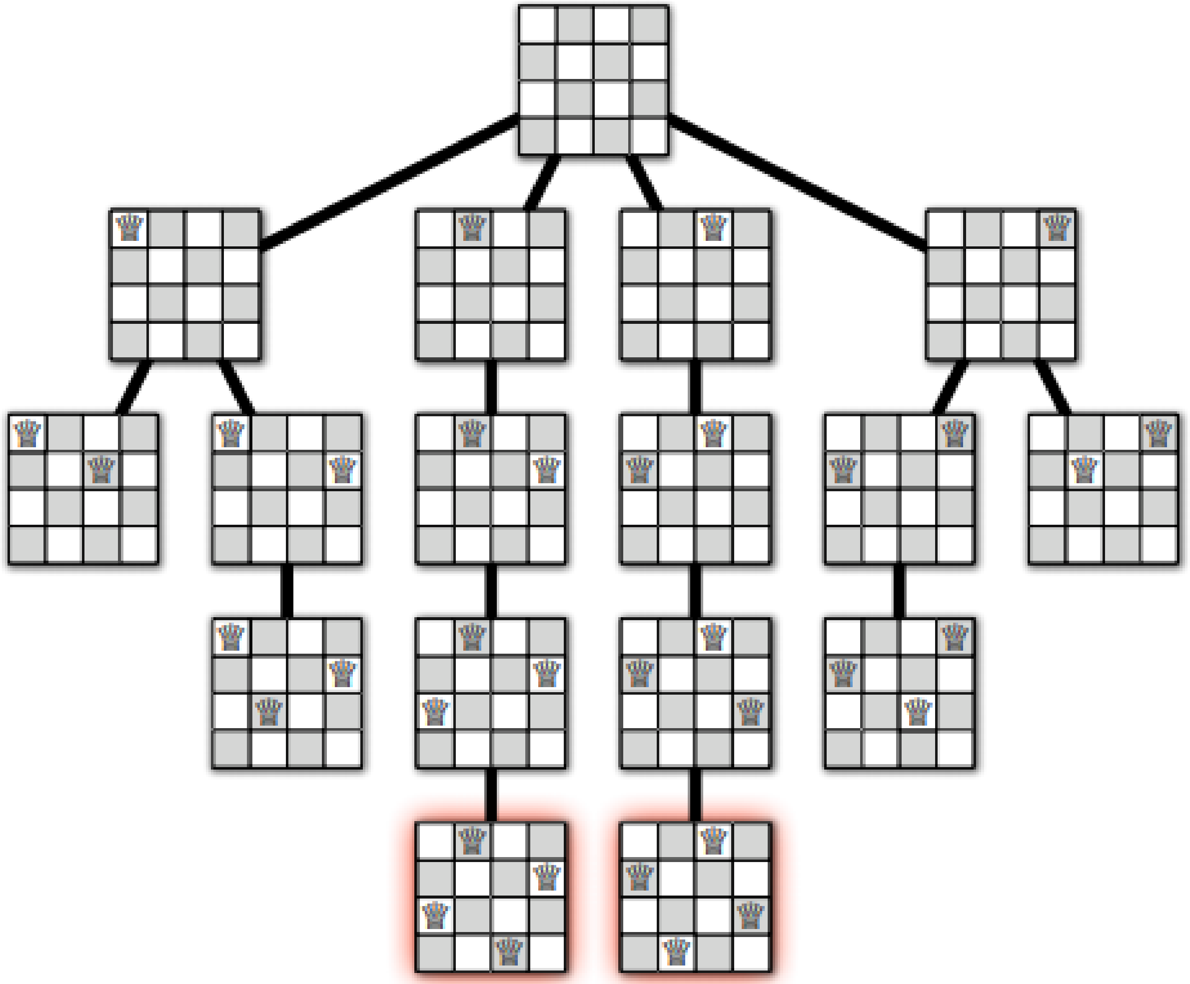
N queens

```python
def initSolution():
    solution = [[0 for i in range(n)] for j in range(n)]
    return solution

def printSolution(solution):
    for row in solution:
        print(row)
```

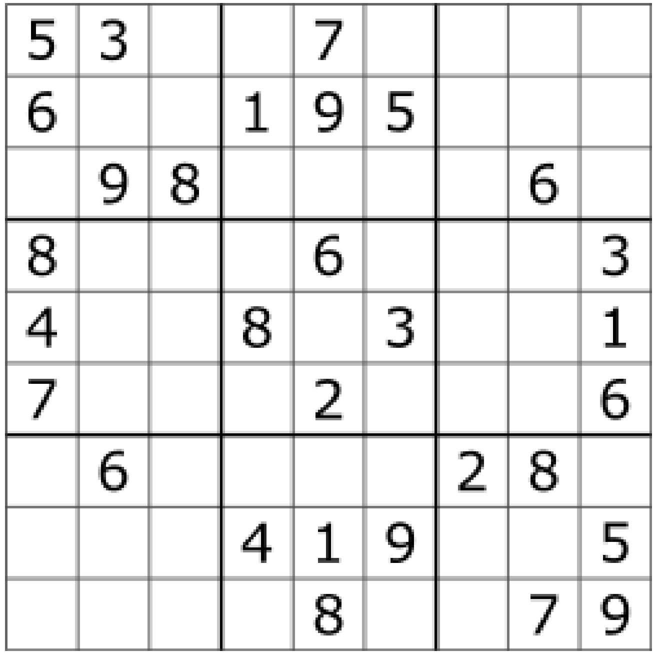
```python
def isConsistent(solution, row, column):
    # check the row
    for j in range(column):
        if solution[row][j] == 1:
            return False

    # check the first diagonal to left (up)
    for i,j in zip(range(row,-1,-1), range(column,-1,-1)):
        if solution[i][j] == 1:
            return False

    # check the second diagonal to left (down)
    i = row + 1
    j = column - 1
    while (i < len(solution)) and (j >= 0):
        if solution[i][j] == 1:
            return False
        i = i + 1
        j = j - 1

    return True
```

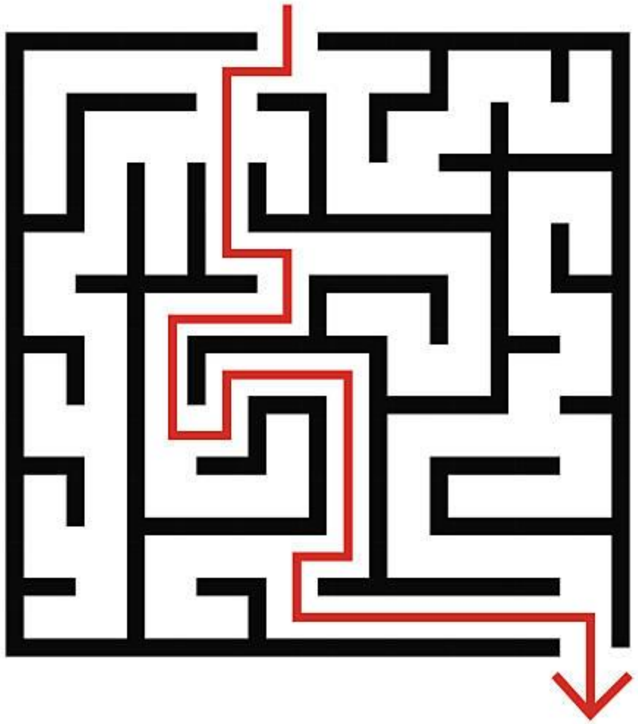
```python
def solveProblem(solution, column):
    if column >= n:
        print("COMPLETE solution:")
        printSolution(solution)
        return True

    for i in range(n):
        if isConsistent(solution, i, column):
            solution[i][column] = 1
            print("Partial correct solution:")
            printSolution(solution)
            if solveProblem(solution, column + 1) == True:
                return True
            else:
                solution[i][column] = 0
    return False

n=8
solveProblem(sol, 0)
```

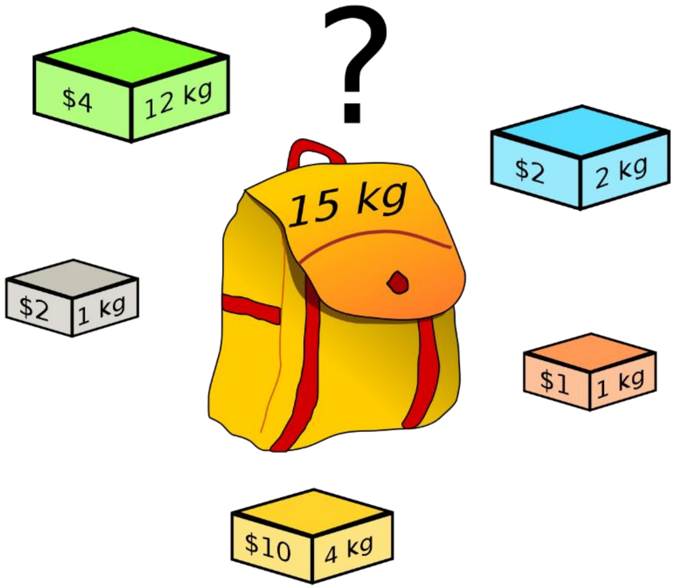
Backtracking Example



Backtracking: other examples







Divide et impera – Divide and conquer

- Basic idea
  - Divide the problem in several independent sub-problems similar to the initial problem but smaller in size and determine the final solution by combining sub-solutions
- Mechanism
  - **Divide**: breaking the problem in sub-problems
  - **Conquer**: solve the sub-problems
  - **Combine**: combine sub-solutions to obtain final solution
- When it can be used
  - A problem ***P*** with the input data ***D*** can be solved by solving the same problem ***P*** but with input data ***d***, where ***d* < *D***

Divide et impera – Divide and conquer

- Algorithm

```
#D = d1 U d2 U d3...U dn
def div_imp(D):
    if (size(D) < lim):
        return rez
    rez1 = div_imp(d1)
    rez2 = div_imp(d2)
    ...
    rezn = div_imp(dn)
    return combine(rez1, rez2, ..., rezn)
```

Divide et impera – Divide and conquer

- Example: find the maximum of a list
  - Size of problem = n
  - First version
    - Size of sub-problem 1 = n-1
    - Size of sub-problem 2 = n-2
    - ...
    - *meaning:*
      - D = l = [l1,l2,..,ln]
      - d1=[l2,..,ln]
      - d2=[l3,..,ln]
      - ...
  - *O(n)*

```python
def findMax(l):
    '''
    Descr: finds the maximum elem of a list
    Input: a list
    Output: the maximum elem of list
    '''
    if (len(l) == 1):
        return l[0]
    max = findMax(l[1:])
    if (max > l[0]):
        return max
    else:
        return l[0]

def test_findMax():
    assert findMax([2,5,3,6,1]) == 6
    assert findMax([12,5,3,2,1]) == 12
    assert findMax([2,5,3,6,11]) == 11

test_findMax()
```

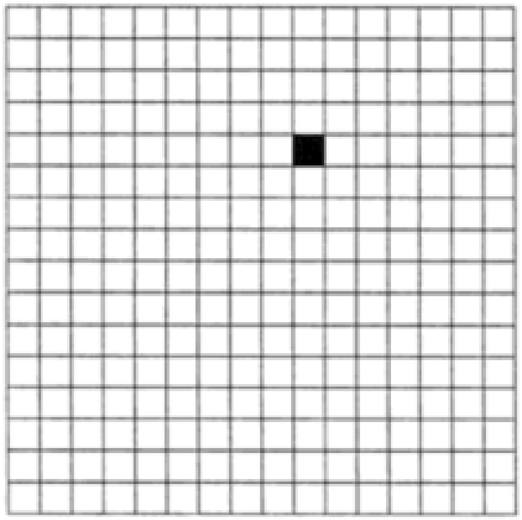
Divide et impera – Divide and conquer

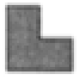
- Example: find the maximum of a list
  - Size of problem = n
  - Second version
    - Size of sub-problem 1 = n/2
    - Size of sub-problem 2 = n/2
    - *meaning:*
      - D = l = [l1,l2,..,ln]
      - d1=[l2,..,ln/2]
      - d2=[ln/2+1,..,ln]
  - *O(n)*

```python
def findMax_v2(l):
    '''
    Descr: finds the maximum elem of a list
    Data: a list
    Res: the maximal elem of list
    '''
    if (len(l) == 1):
        return l[0]
    middle = len(l) // 2
    max_left = findMax_v2(l[0:middle])
    max_right = findMax_v2(l[middle:len(l)])
    if (max_left < max_right):
        return max_right
    else:
        return max_left

def test_findMax_v2():
    assert findMax_v2([2,5,3,6,1]) == 6
    assert findMax_v2([12,5,3,2,1]) == 12
    assert findMax_v2([2,5,3,6,11]) == 11

test_findMax_v2()
```

Divide et impera – Example

- Consider a chessboard of size 2^m (with 2^m X 2^m cells) that contains a hole (one random cell is removed)
- We have several shapes L
- Objective: cover the chessboard with L shapes (any orientation)





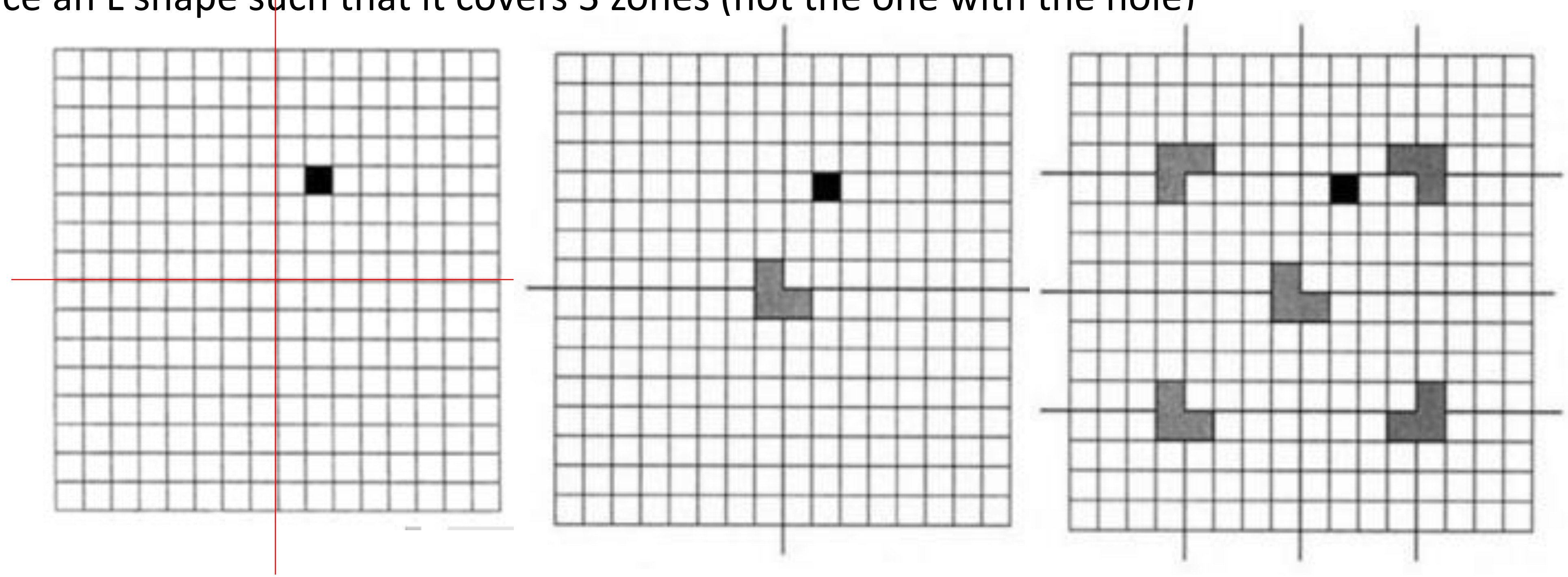
m=4 => chessboard16 X 16

✓ Search space: possible arrangements of L shapes on the board

✓ D&C is an ideal method

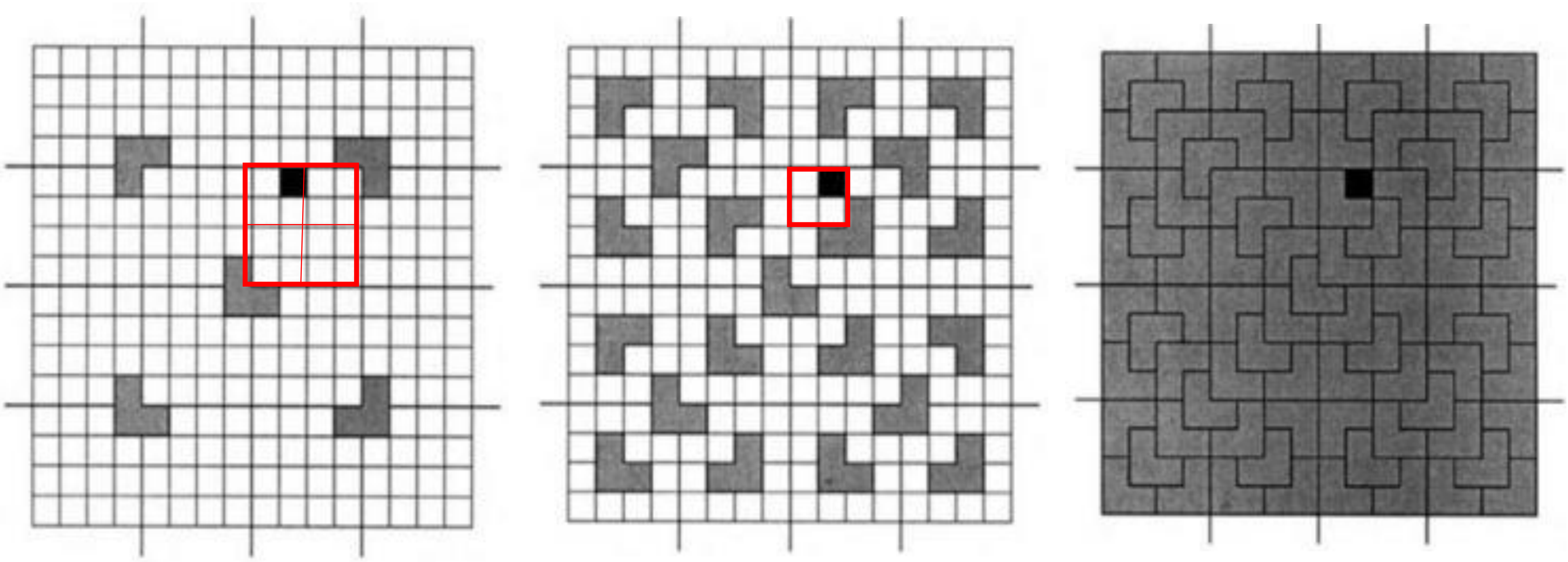
Divide et impera – Example

- Divide the chessboard in 4 equal zones
- Only one will contain the hole
- Place an L shape such that it covers 3 zones (not the one with the hole)



Divide et impera – Example

- Each square has a single black cell



Final solution

Recap today

- Problem solving methods
- Generate and test
  - Exhaustive
  - Backtracking
- Divide and conquer

Next time

- Algorithms
  - Dynamic programming
  - Greedy method

Reading materials and useful links

1. The Python Programming Language - https://www.python.org/
2. The Python Standard Library - https://docs.python.org/3/library/index.html
3. The Python Tutorial - https://docs.python.org/3/tutorial/
4. M. Frentiu, H.F. Pop, Fundamentals of Programming, Cluj University Press, 2006.
5. MIT OpenCourseWare, Introduction to Computer Science and Programming in Python, https://ocw.mit.edu, 2016.
6. K. Beck, Test Driven Development: By Example. Addison-Wesley Longman, 2002. http://en.wikipedia.org/wiki/Test-driven_development
7. M. Fowler, Refactoring. Improving the Design of Existing Code, Addison-Wesley, 1999. http://refactoring.com/catalog/index.html