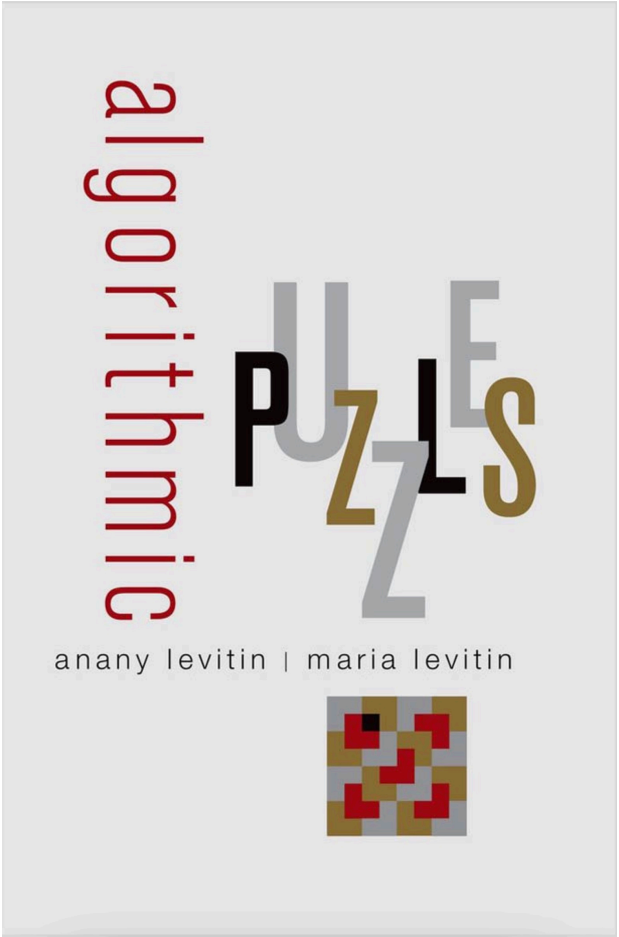
# Sixth Chapter

## Problem Solving

# The 8.5 strategies for problem solving

1. Exhaustive Search
2. Backtracking / *Branch and bound*
3. Decrease-and-Conquer
4. Divide-and-Conquer
5. Transform and Conquer
6. Greedy Approach
7. Iterative Improvement
8. Dynamic Programming



# Exhaustive Search

## Principle

- Try all candidate solutions to a problem (*aka Brute Force*
- Stop when a solution is found

## Usage

- Often a good starting point !
- Easy to implement (quick and dirty)


Albrecht Dürer, Melancholia (1514) *source: wikipedia.org*

# Exhaustive Search

## Principle

- Try all candidate solutions to a problem (*aka Brute Force*
- Stop when a solution is found

## Usage

- Often a good starting point !
- Easy to implement (quick and dirty)

## Limits

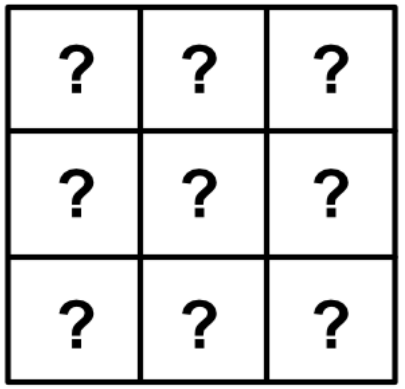
- Not all problems have a limited set of solution candidates
- Combinatorial explosion
- Works on limited (sometimes **very**) limited datasets


Albrecht Dürer, Melancholia (1514) *source: wikipedia.org*

# 3x3 Magic Square Exhaustive Search

## Goal

Fill the 3x3 table with nine distinct integers from 1 to 9 so that the sum of each row, column and full diagonals are equal.



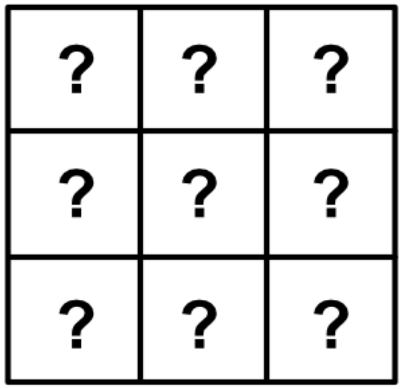

Solutions:

```
[2, 7, 6, 9, 5, 1, 4, 3, 8]
[2, 9, 4, 7, 5, 3, 6, 1, 8]
[4, 3, 8, 9, 5, 1, 2, 7, 6]
[4, 9, 2, 3, 5, 7, 8, 1, 6]
[6, 1, 8, 7, 5, 3, 2, 9, 4]
[6, 7, 2, 1, 5, 9, 8, 3, 4]
[8, 1, 6, 3, 5, 7, 4, 9, 2]
[8, 3, 4, 1, 5, 9, 6, 7, 2]
```

# 3x3 Magic Square Exhaustive Search

## Goal

Fill the 3x3 table with nine distinct integers from 1 to 9 so that the sum of each row, column and full diagonals are equal.




## Exhaustive search

- Fill each cell with all possible values
- Check each solution's validity

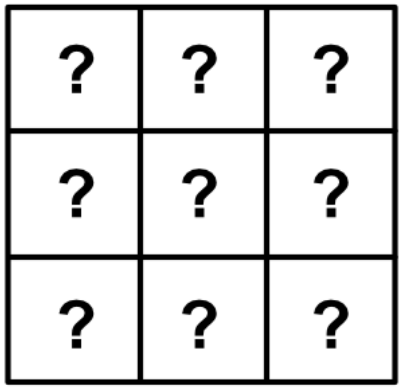
Solutions:

```
[2, 7, 6, 9, 5, 1, 4, 3, 8]
[2, 9, 4, 7, 5, 3, 6, 1, 8]
[4, 3, 8, 9, 5, 1, 2, 7, 6]
[4, 9, 2, 3, 5, 7, 8, 1, 6]
[6, 1, 8, 7, 5, 3, 2, 9, 4]
[6, 7, 2, 1, 5, 9, 8, 3, 4]
[8, 1, 6, 3, 5, 7, 4, 9, 2]
[8, 3, 4, 1, 5, 9, 6, 7, 2]
```

# 3x3 Magic Square Exhaustive Search

## Goal

Fill the 3x3 table with nine distinct integers from 1 to 9 so that the sum of each row, column and full diagonals are equal.




## Exhaustive search

- Fill each cell with all possible values
- Check each solution's validity

## Cost

- $9^9 = 387.420.489$ combinations
- Working with distinct values only: $9! = 362.880$ combinations

Solutions:

```
[2, 7, 6, 9, 5, 1, 4, 3, 8]
[2, 9, 4, 7, 5, 3, 6, 1, 8]
[4, 3, 8, 9, 5, 1, 2, 7, 6]
[4, 9, 2, 3, 5, 7, 8, 1, 6]
[6, 1, 8, 7, 5, 3, 2, 9, 4]
[6, 7, 2, 1, 5, 9, 8, 3, 4]
[8, 1, 6, 3, 5, 7, 4, 9, 2]
[8, 3, 4, 1, 5, 9, 6, 7, 2]
```

# Magic Square Exhaustive Search

- Magic square exists for any $n >= 3$ !!!
- Combinatorial explosion occurs fast!

## Some basic counting:

- $n * n$ square holds $n^2$ values
- the sum of each line and each diagonal is $n * (n^2 + 1)/2$
- This can be used to avoid useless searches

## Homework

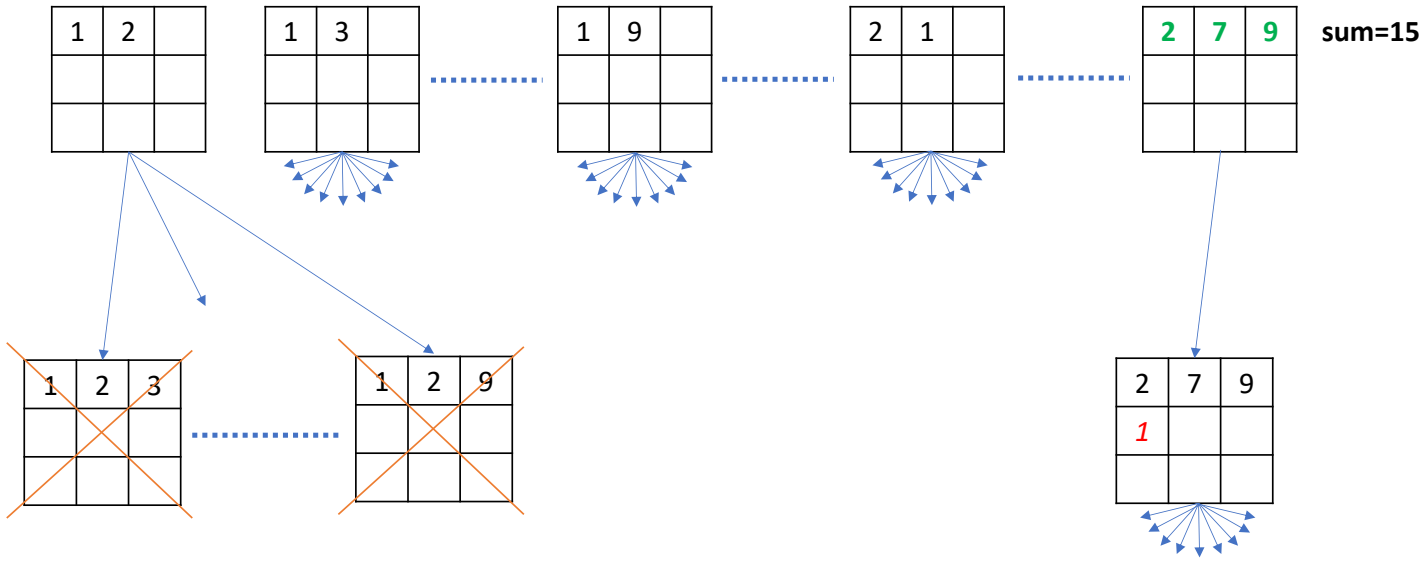
- Prove the sum property
- Code a simple 3x3 Magic Square computation in Python and count the steps
- Implement a generalized Magic Square builder using exhaustive search with $n$ as the only parameter of your function

# Backtracking

## Principle

- State-space exploration (depth-first)
- Build candidate solutions as in exhaustive search but ...
- Add early "cutoff" porperties [Knuth F5B2019]
- Undo wrong choices early to avoid useless explorations

## The Magic Square example



# Backtracking

## Typical operation

```
solution = [None] * size

    def solution(position):
        for value in values:
            solution[position] = value
            if safe_up_to(solution, position):
                if position >= size-1 or solution(position+1):
                    return solution
        return None

    return extend_solution(0)
```

*source: wcs.lmu.edu*

# Backtracking

## Typical operation

```
solution = [None] * size

    def solution(position):
        for value in values:
            solution[position] = value
            if safe_up_to(solution, position):
                if position >= size-1 or solution(position+1):
                    return solution
        return None

    return extend_solution(0)
```

*source: wcs.lmu.edu*

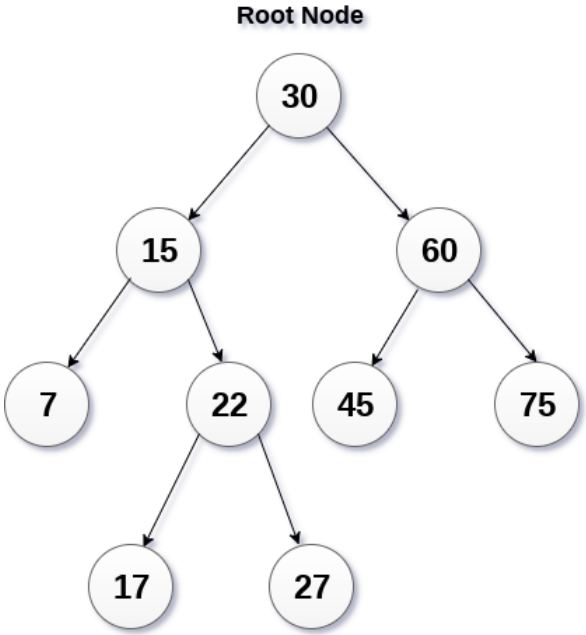
## Usage

- N-Queen (seen later in the lecture)n, Sudoku (usual suspect :-)
- knapsack problem (seen in a TD)
- Traveler Salesman Problem (seen in Operational Research Lecture)
- Text Segmentation (*IAMOUTOFOFFICERIGHTNOW*)
- Longest Subsequences ...

# Decrease and conquer

## Principle

Find a link between a solution to a problem and a solution to its smaller instance.




Binary Search Tree

*source: javapoint.com*

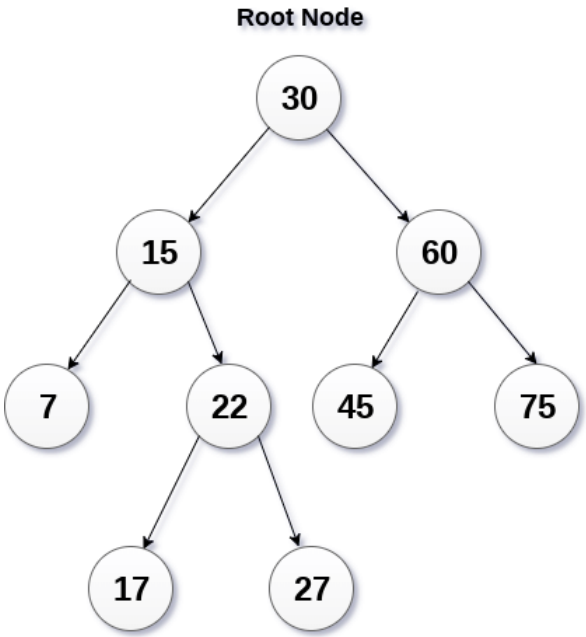
# Decrease and conquer

## Principle

Find a link between a solution to a problem and a solution to its smaller instance.

## Operations

- Operate over decreasing instances
- compute the solution over a trivial case
- if needed, build the solution up from the trivial case




Binary Search Tree

*source: javapoint.com*

# Decrease and conquer

## Principle

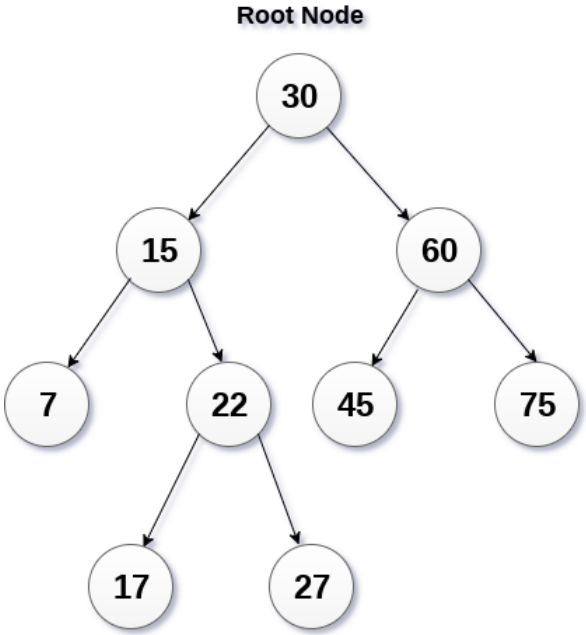
Find a link between a solution to a problem and a solution to its smaller instance.

## Operations

- Operate over decreasing instances
- compute the solution over a trivial case
- if needed, build the solution up from the trivial case

## Usage

- Parsing algorithms
- Counting
- Binary search
- Numerical Computing (root/bisection finding)
- ...




Binary Search Tree

*source: javapoint.com*

# Decrease and conquer

## Principle

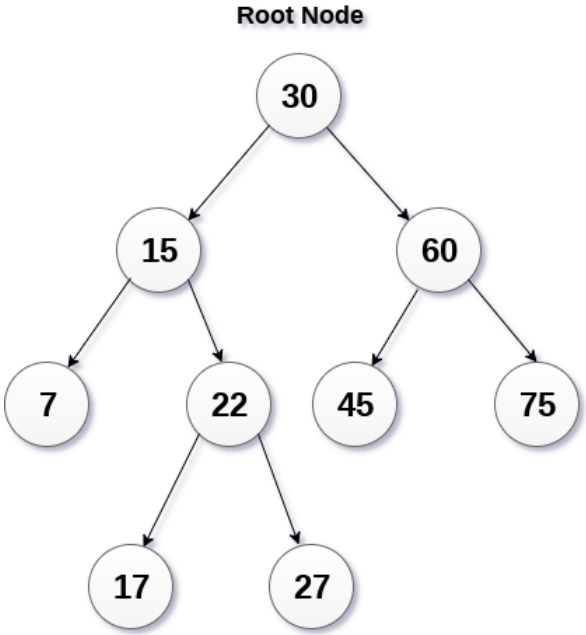
Find a link between a solution to a problem and a solution to its smaller instance.

## Operations

- Operate over decreasing instances
- compute the solution over a trivial case
- if needed, build the solution up from the trivial case

## Usage

- Parsing algorithms
- Counting
- Binary search
- Numerical Computing (root/bisection finding)
- ...




Binary Search Tree

*source: javapoint.com*

## Support mechanisms

- Recursion (seen earlier in the lecture)

# Decrease and conquer

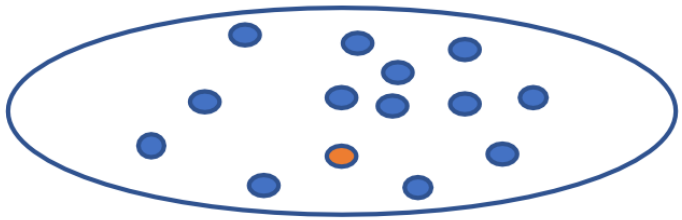
## The Celebrity problem

A celebrity in a crowd is defined as :

- a person who knows no one else
- is known by all

The only question allowed to a person is:

- do you know x?



# Decrease and conquer

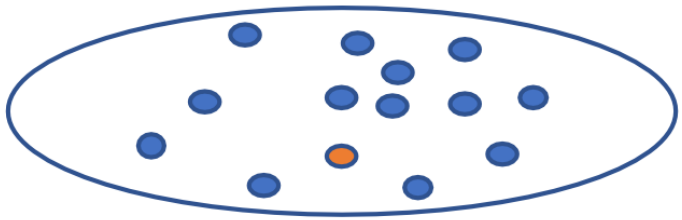
## The Celebrity problem

A celebrity in a crowd is defined as :

- a person who knows no one else
- is known by all

The only question allowed to a person is:

- do you know x?



Find the celebrity!

# Decrease and conquer

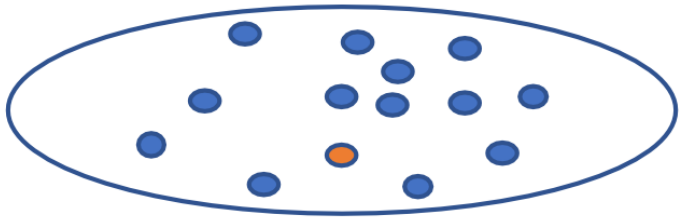
## The Celebrity problem

A celebrity in a crowd is defined as :

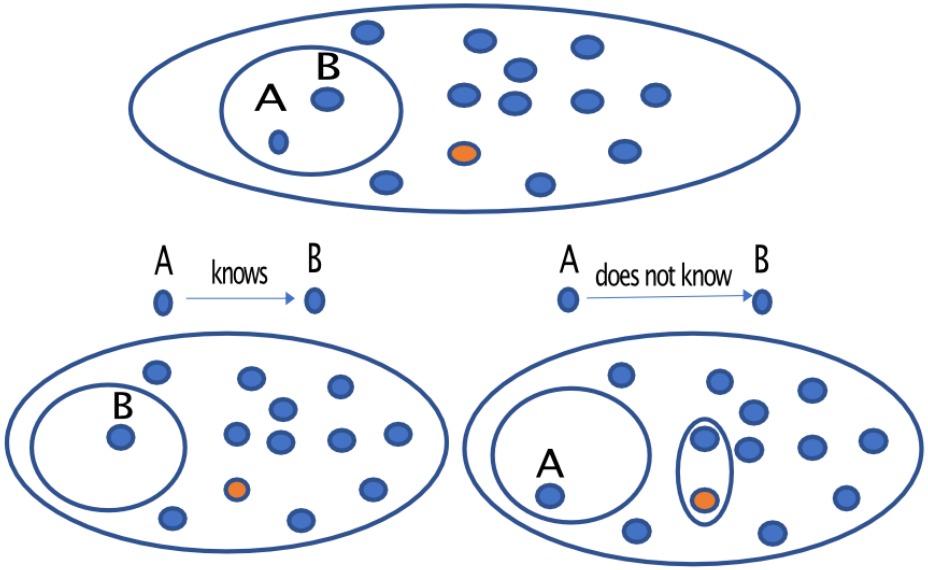
- a person who knows no one else
- is known by all

The only question allowed to a person is:

- do you know x?



## Elements of the solution

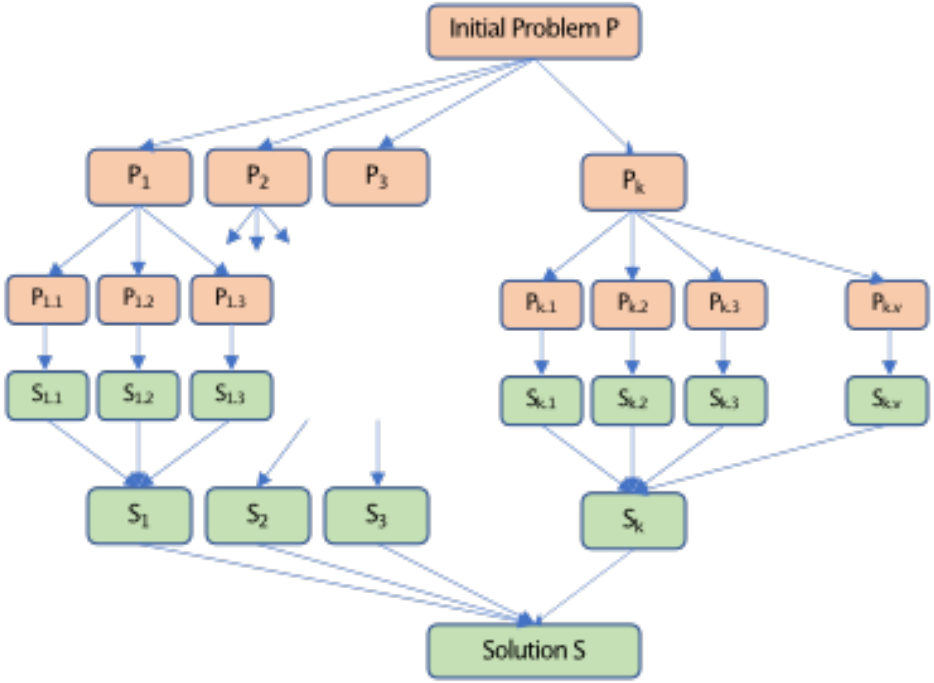



Find the celebrity!

# Divide and conquer

## Principle

Find a link between a solution to a problem and a solution to a **set of smaller instances**.



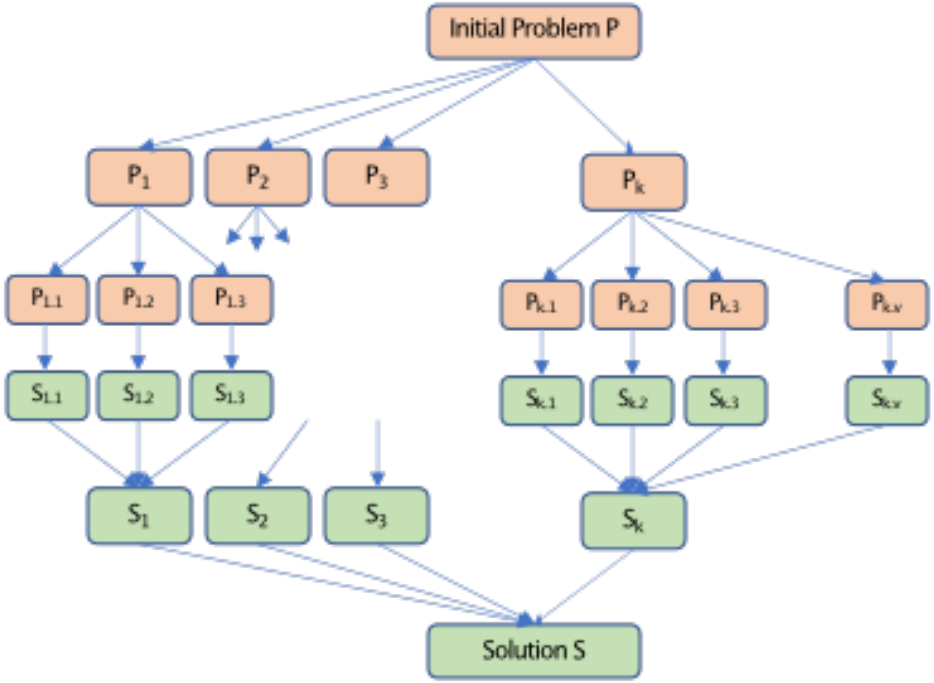
# Divide and conquer

## Principle

Find a link between a solution to a problem and a solution to a **set of smaller instances**.

## Operations

- **Divide**: Partition the problem into sub-problems
- **Conquer**: Solve the sub-problems
- **Combine**: the sub-problems solutions



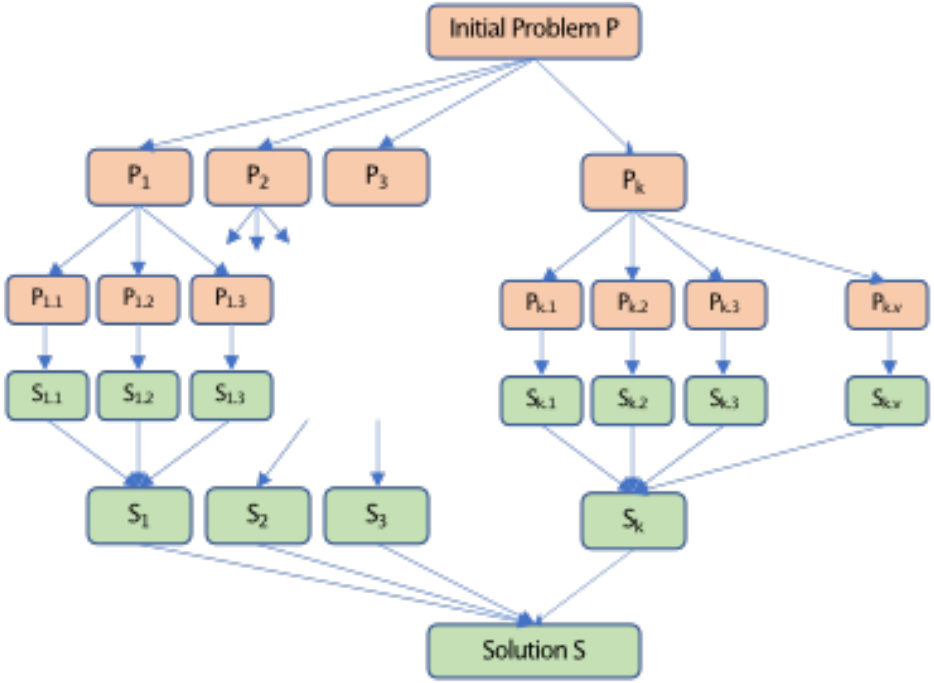
# Divide and conquer

## Principle

Find a link between a solution to a problem and a solution to a **set of smaller instances**.

## Operations

- **Divide**: Partition the problem into sub-problems
- **Conquer**: Solve the sub-problems
- **Combine**: the sub-problems solutions




## Usage

- 
- many many others . . .

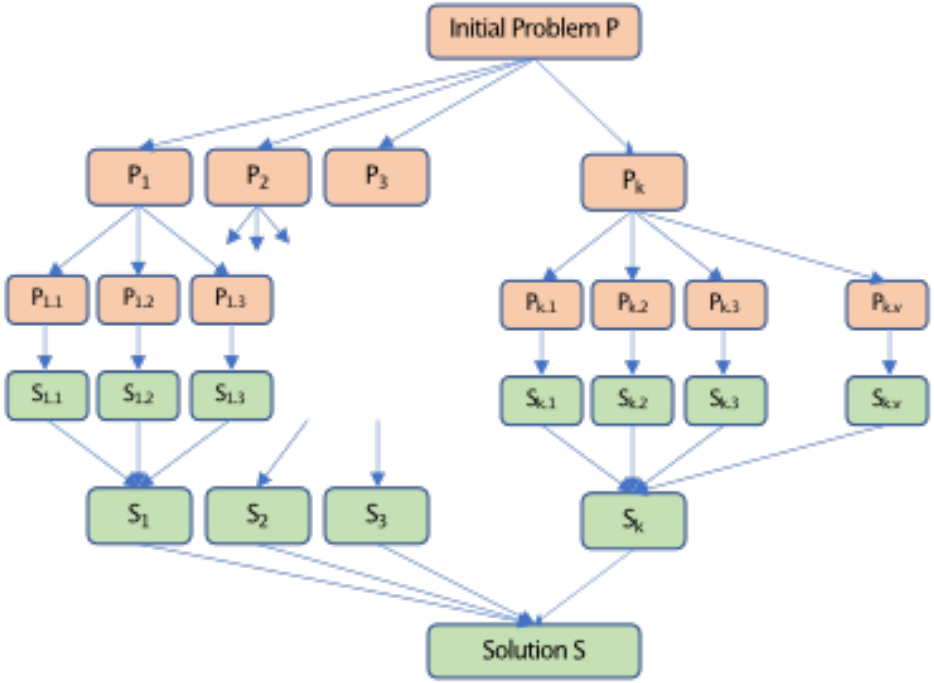
# Divide and conquer

## Principle

Find a link between a solution to a problem and a solution to a **set of smaller instances**.

## Operations

- **Divide**: Partition the problem into sub-problems
- **Conquer**: Solve the sub-problems
- **Combine**: the sub-problems solutions




## Usage

- 
- many many others . . .

## Support mechanisms

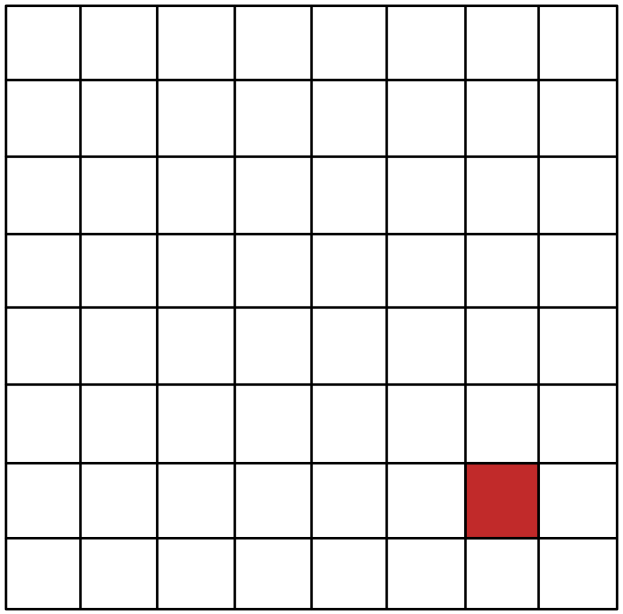
- Recursion (seen later in the course)

# Divide and conquer - A Graphical Puzzle Example

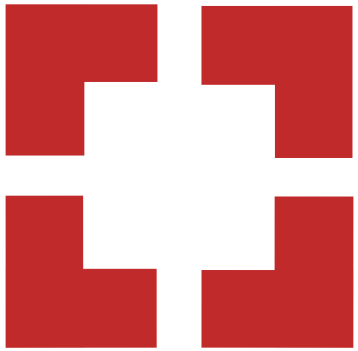
## Tromino Puzzle

Given a $2^n x 2^n$ square missing one square and L-shaped trominoes, fill the square without overlaps !

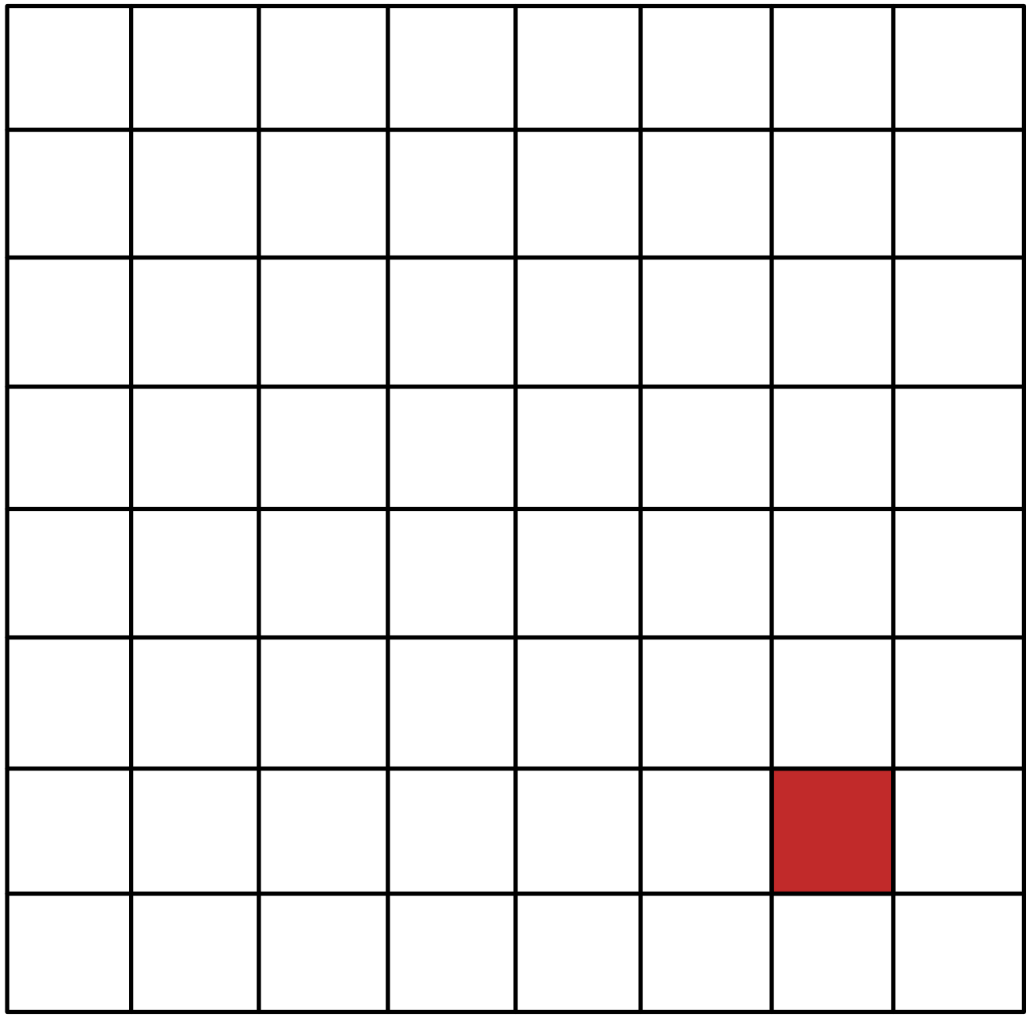
## Deck



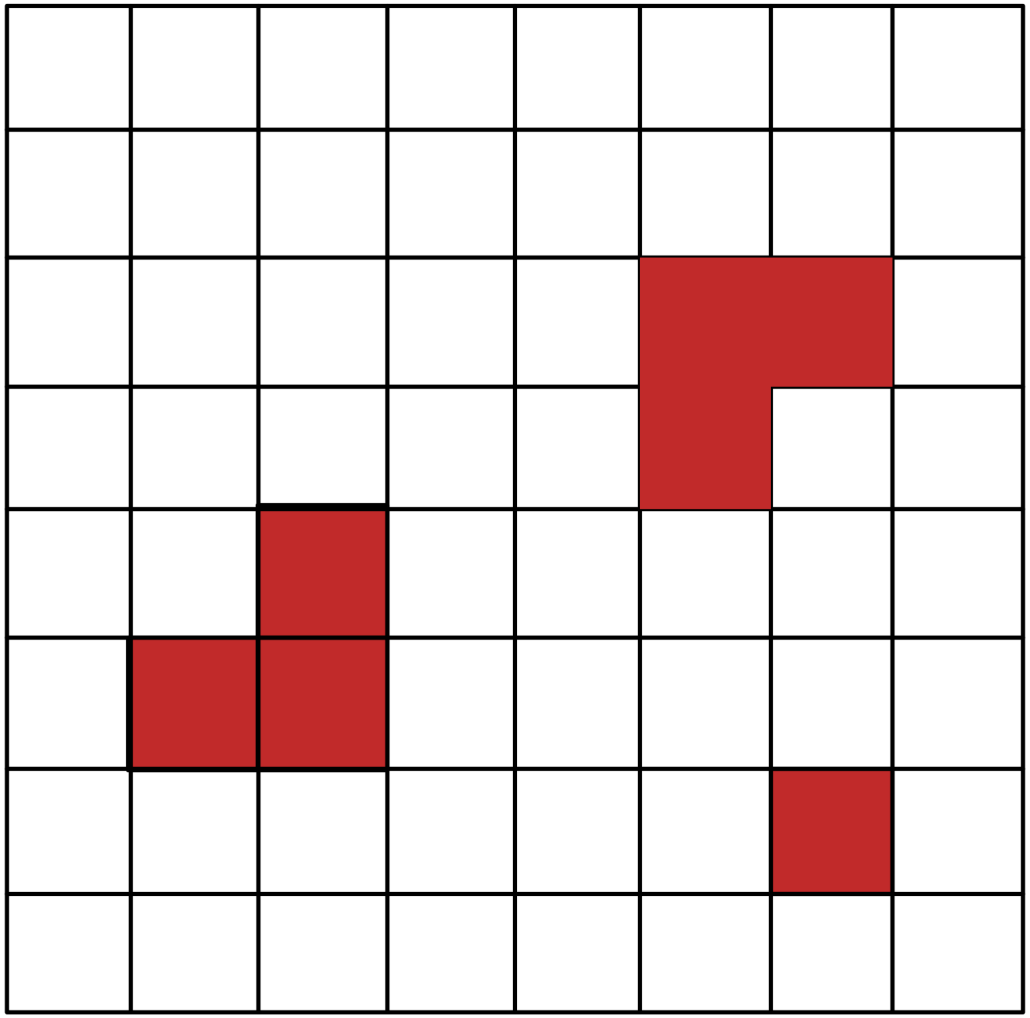
## Shapes



# Divide and conquer - A Graphical Puzzle Example



# Divide and conquer - A Graphical Puzzle Example



# Divide and conquer - A Graphical Puzzle Example

Divide:

- Choose smaller squares

Conquer:

- to fill the decks
- to maintain sub-problem properties

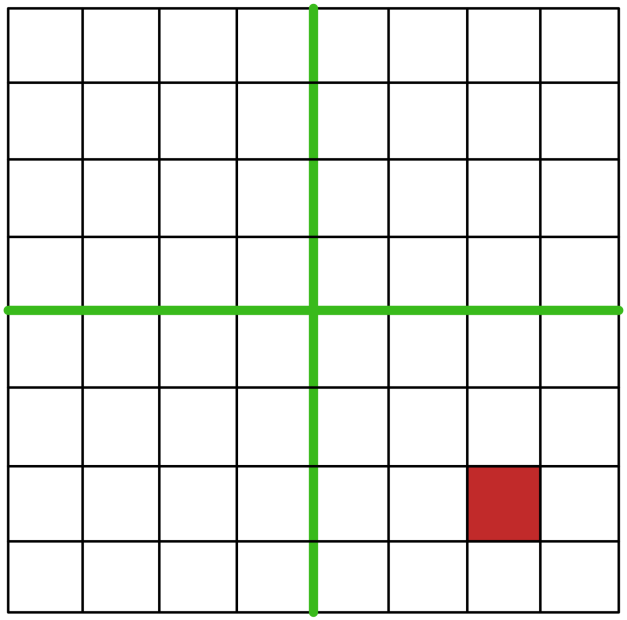
# Divide and conquer - A Graphical Puzzle Example

Divide:

- Choose smaller squares

Conquer:

- to fill the decks
- to maintain sub-problem properties



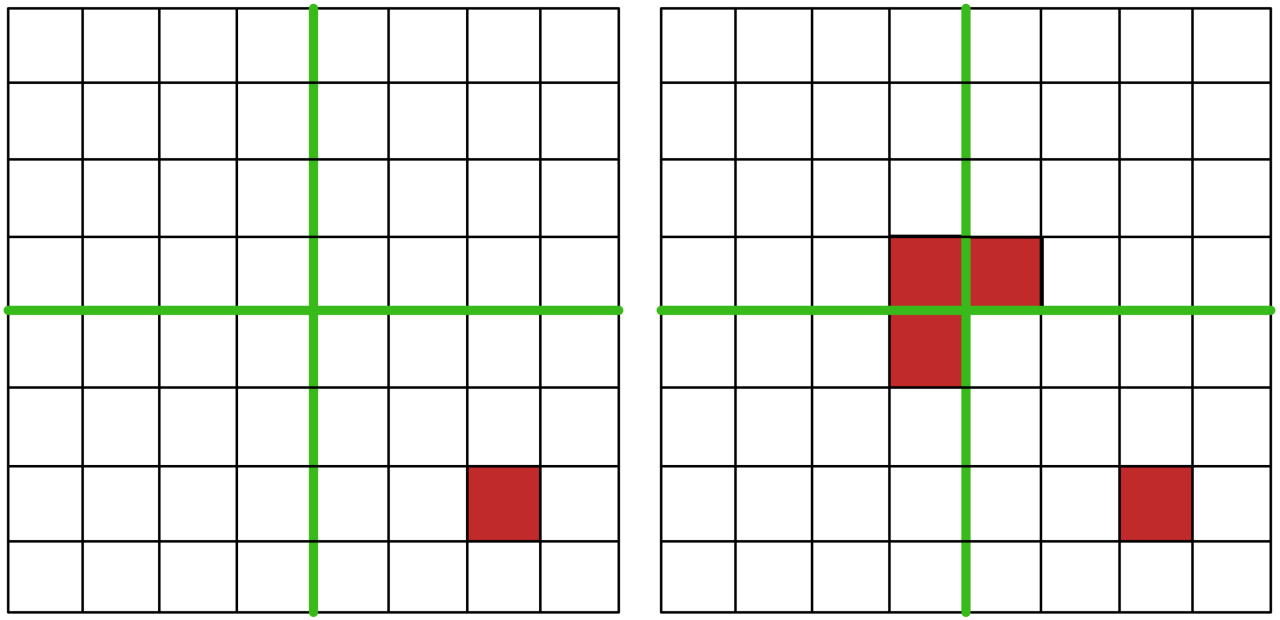
# Divide and conquer - A Graphical Puzzle Example

Divide:

- Choose smaller squares

Conquer:

- to fill the decks
- to maintain sub-problem properties



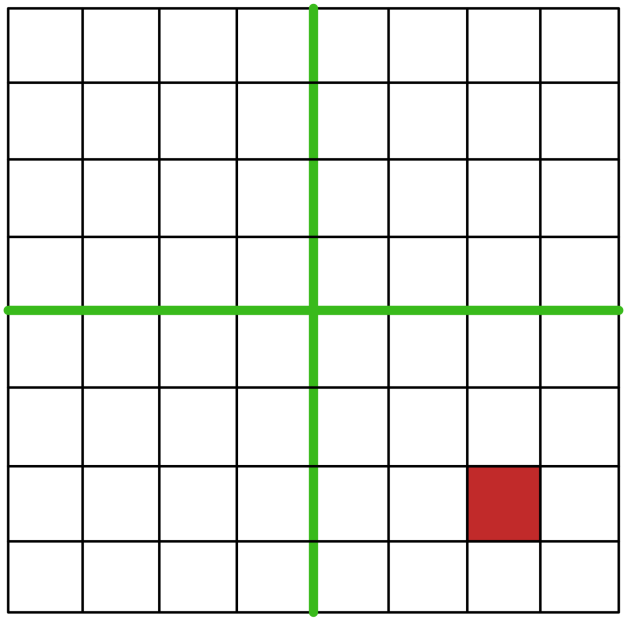
# Divide and conquer - A Graphical Puzzle Example

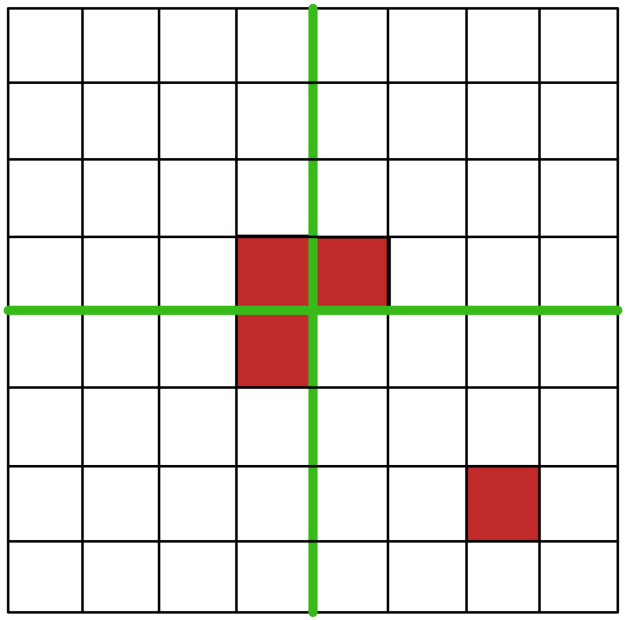
Divide:

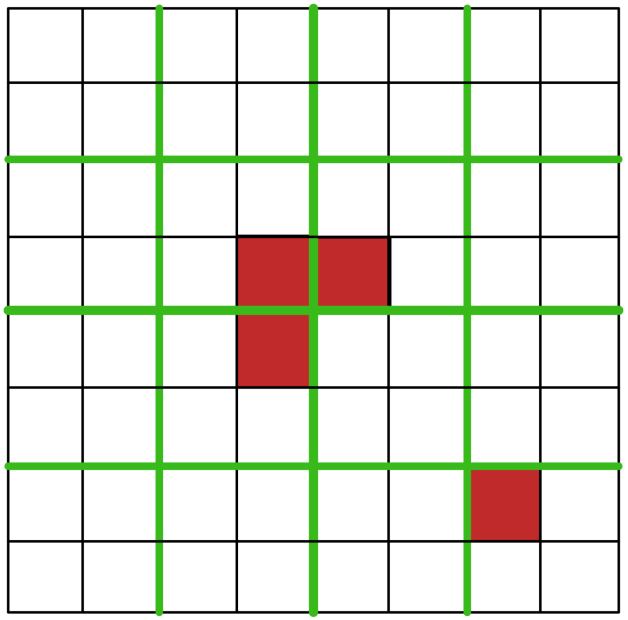
- Choose smaller squares

Conquer:

- to fill the decks
- to maintain sub-problem properties







# Transform and conquer

## Principle

Transform the problem in another more manageable problem and solve it!

# Transform and conquer

## Principle

Transform the problem in another more manageable problem and solve it!

## Operations

- **Transform**: model and map he problem in another instance of the problem or an anoher problem
- **Conquer**: Solve the target instance
- **Interpret**: the results of the mapped-problem into a result of the original problem

# Transform and conquer

## Principle

Transform the problem in another more manageable problem and solve it!

## Operations

- **Transform**: model and map he problem in another instance of the problem or an anoher problem
- **Conquer**: Solve the target instance
- **Interpret**: the results of the mapped-problem into a result of the original problem

## Transformation strategies

- instance simplification (simpler, more manageable)
- representation change (different instance of the same problem))
- problem reduction to an existing solved problem
- Any combination of them!

# Transform and conquer - Instance Simplification

## Principle

Simplify the problem to a simpler instance of the same problem !

# Transform and conquer - Instance Simplification

## Principle

Simplify the problem to a simpler instance of the same problem !

## Typical instance simplification

- Data Presorting

# Transform and conquer - Instance Simplification

## Principle

Simplify the problem to a simpler instance of the same problem !

## Typical instance simplification

- Data Presorting

## Unsorted uniqueness check

```
def uniquenessSearchUnsorted(pList):
    for i in range(0, len(pList)-1):
          for j in range (i+1, len(pList)):
            if pList[i] == pList[j]:
                return False
    return True
```

# Transform and conquer - Instance Simplification

## Principle

Simplify the problem to a simpler instance of the same problem !

## Typical instance simplification

- Data Presorting

## Unsorted uniqueness check

```
def uniquenessSearchUnsorted(pList):
    for i in range(0, len(pList)-1):
          for j in range (i+1, len(pList)):
            if pList[i] == pList[j]:
                return False
    return True
```

## Sorted uniqueness check

```
def uniquenessTestSorted(pList):
    for i in range(0,len(pList)-1):
        if pList[i]== pList[i+1]:
            return False
    return True
```

# Transform and conquer - Instance Simplification

## Principle

Simplify the problem to a simpler instance of the same problem !

## Typical instance simplification

- Data Presorting

## Uniquess ckeck in a list

- Basic algorithm : $O(n^2)$
- Pre-sorted algorithm : $O(n * log(n))$

## Unsorted uniqueness check

```
def uniquenessSearchUnsorted(pList):
    for i in range(0, len(pList)-1):
          for j in range (i+1, len(pList)):
            if pList[i] == pList[j]:
                return False
    return True
```

## Sorted uniqueness check

```
def uniquenessTestSorted(pList):
    for i in range(0,len(pList)-1):
        if pList[i]== pList[i+1]:
            return False
    return True
```

# Transform and conquer - Combination

## DAG

Simplify the problem definition

# Transform and conquer - Combination

## DAG

Simplify the problem definition

## Anagram

- **Transform**: model and map he problem in another
- **Conquer**: Solve the mapped problem
- **Interpret**: the results of the mapped-problem into a result of the original problem

# Transform and conquer - Combination

## DAG

Simplify the problem definition

## Anagram

- **Transform**: model and map he problem in another
- **Conquer**: Solve the mapped problem
- **Interpret**: the results of the mapped-problem into a result of the original problem

# Transform and conquer - Combination

## DAG

Simplify the problem definition

## Anagram

- **Transform**: model and map he problem in another
- **Conquer**: Solve the mapped problem
- **Interpret**: the results of the mapped-problem into a result of the original problem

## Support mechanisms

- Sorting algorithms
- Binary representations
- Graph representations
- Hashing functions
- ...

# Transform and conquer - Problem Reduction

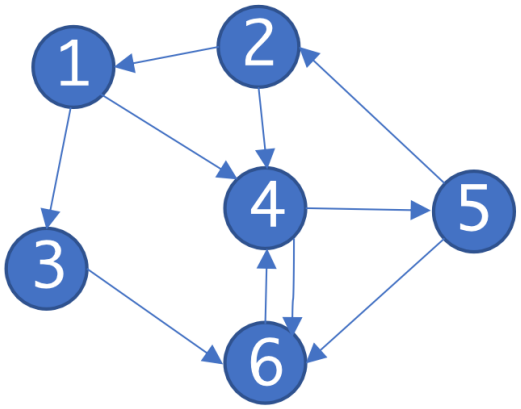

## Supporting technologies

- All mathematical models and field mapping (e.g. analytical geometry, algebra, ...)
- Graph theory and field mapping (e.g. matrix representation of graphs)
- Linear programming
- Optimisation problems
- ...

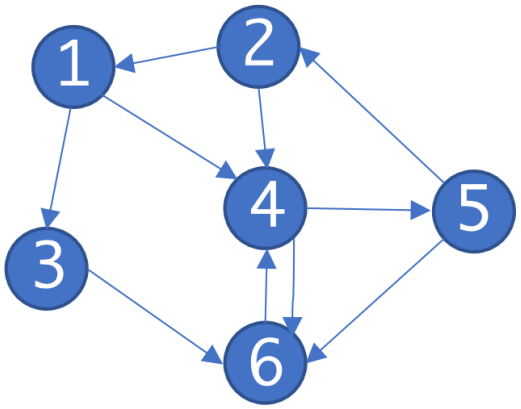
# Transform and conquer - Problem Reduction

## Paths lengths in a Directed Graph

Compute the number of paths of length $k$ in a given directed graph !



# Transform and conquer - Problem Reduction




## Paths lengths in a Directed Graph

Compute the number of paths of length $k$ in a given directed graph !

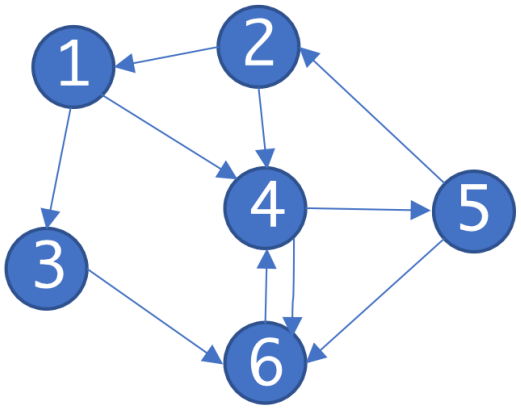
## Model it as a linear algebra problem

- Graph model = Adjacency Matrix $A$
- Compute $A^k$

## Interpretation

$A^k[i,j]$ = number of paths of length $k$ from node $i$ to node$j$

# Transform and conquer - Problem Reduction




## Paths lengths in a Directed Graph

Compute the number of paths of length $k$ in a given directed graph !

## Model it as a linear algebra problem

- Graph model = Adjacency Matrix $A$
- Compute $A^k$

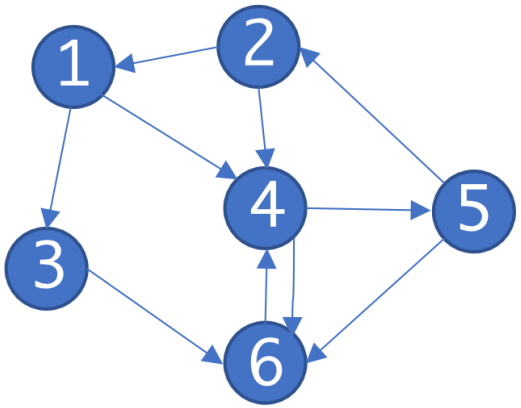
$A^1$

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

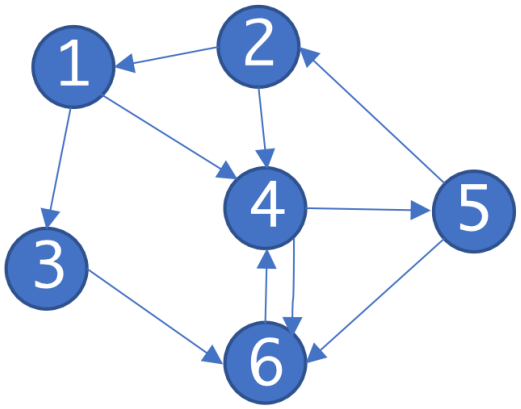
## Interpretation

$A^k[i,j]$ = number of paths of length $k$ from node $i$ to node$j$

# Transform and conquer - Problem Reduction



# Transform and conquer - Problem Reduction




## Interpretation

$A^k[i,j]$ = number of paths of length $k$ from node $i$ to node$j$

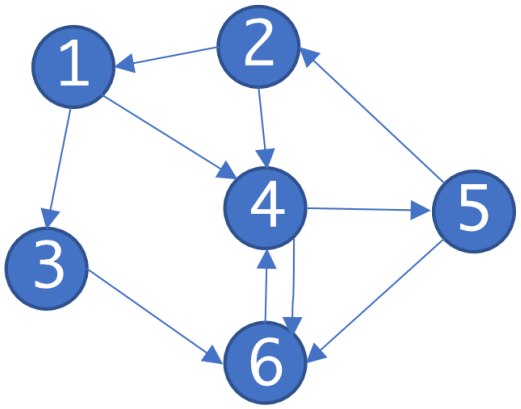
$A^2$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

# Transform and conquer - Problem Reduction

## Interpretation

$A^k[i,j]$ = number of paths of length $k$ from node $i$ to node$j$




$A^2$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

$A^3$

$$\begin{pmatrix} 0 & 1 & 0 & 2 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 3 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 2 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

# Greedy Algorithms

# Greedy Algorithms

## Principle

- At each step of the algorithm, make the **Best local choice**
- **Never undo a previous decision**
- **Best = most rewarding towards reaching the solution**
- **Each step leads to a smaller problem**

## Usage

- Mainly **optimization problems**
- **Useful if approximate solution is sufficient**
- **Easy to implement**

# Greedy Algorithms

## Principle

- At each step of the algorithm, make the **Best local choice**
- **Never undo a previous decision**
- **Best = most rewarding towards reaching the solution**
- **Each step leads to a smaller problem**

## Limits

- Does not always lead to an optimal solution
- Strong mathematical basis to prove the quality of the approach on a given problem
  - prove that the solution is a real solution
  - prove that the solution is optimal

## Usage

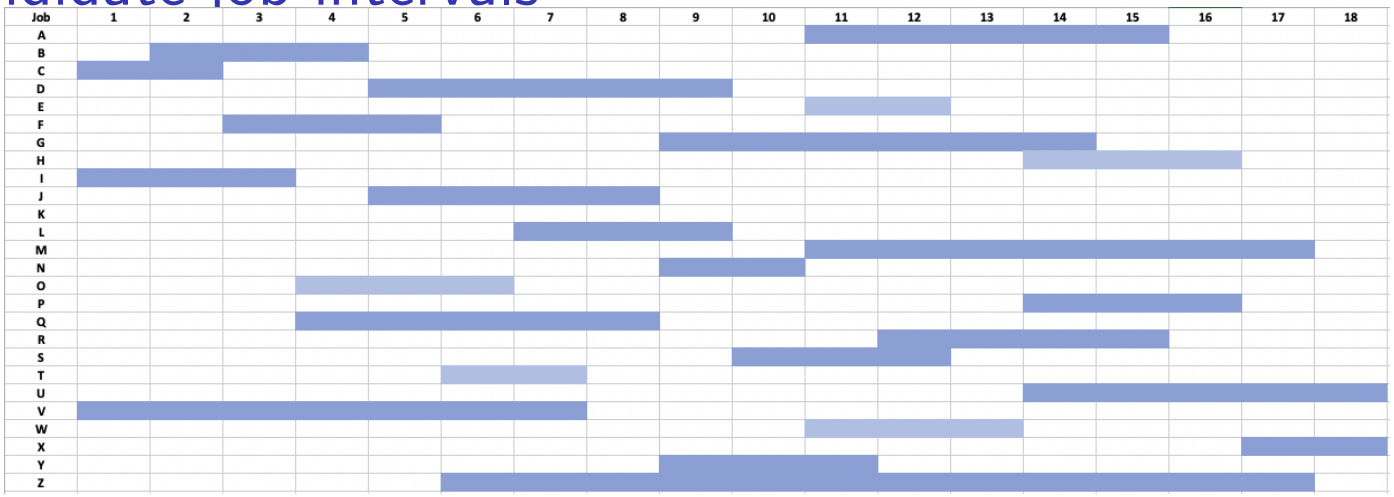
- Mainly **optimization problems**
- **Useful if approximate solution is sufficient**
- **Easy to implement**

# The Interval Scheduling Problem

## Problem formulation

Given $n$ jobs, provide a schedule that maximizes the number of performed jobs in a given time frame!

## List of candidate job intervals




## Solution

```
[(1, 2), (3, 5), (6, 7), (9, 10), (11, 12), (14, 16), (17, 18)]
```

# The Interval Scheduling Problem

## Implementation

```
def greedy(pList):
    pList.sort(key=lambda tup: tup[1])
    selectedJobs = list()
    lastJobEnd = 0
    for job in pList:
        if job[0]>= lastJobEnd:
            selectedJobs.append(job)
            lastJobEnd=job[1]
    return selectedJobs
print(greedy([(11,15),(2,4),(1,2),(5,9),(11,12),(3,5),(9,14),(14,16),(1,3),
              (5,8),(7,9),(11,17),(9,10),(4,6),(4,8),(12,15),(10,13),
               (6,7),(14,18),(1,7),(11,13),(17,18),(9,11),(6,17)]))
```

## Complexity and spirit of the proof

- Complexity ? (You guess)
- Proof of optimality done by contradiction
- if a better solution exists, it means another job was selected at some time by the optimal solution
- but this job ended later than our selected one, so we still are better

# Memoization !

# Chapter

# Conclusion on TOP

1. Practical and Theoretical Foundations of Programming
   - CS vs. SE; Abstraction for complex algorithms; Algorithmic efficiency.
2. Iterative Sorting Algorithms
   - Specification; Selection, Insertion and Bubble sorts.
3. Recursion
   - Principles; Practice; Recursive sorts; Non-recursive Form; Backtracking.
4. Software Correction
   - Introduction; Specifying Systems.
5. Testing Software
   - Testing techniques; Testing strategies; Pytest; Design By Contract.

Problem solving techniques (although not all)