UNIVERSITÄT BERN

# (Data Structure) Algorithms

Lecturer: **Nataliia Stulova**
Teaching assistant: **Mohammadreza Hazirprasand**

MASTER IN COMPUTER SCIENCE
OF THE UNIVERSITIES OF BERN, NEUCHÂTEL AND FRIBOURG

Software Skills Lab

## From the data structure point of view

## Categories

From the data structure point of view, following are some important categories of algorithms:

- **Search** − search an item in a data structure.
- **Sort** − sort items in a certain order.
- **Insert** − insert item in a data structure.
- **Update** − update an existing item in a data structure.
- **Delete** − delete an existing item from a data structure.

Next slides cover some common approaches to designing algorithms

## More general approach to algorithms

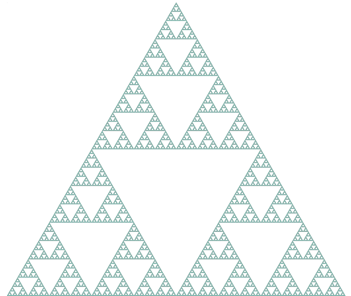
## Recursion

“to understand recursion you need to understand recursion”

In mathematics: an abstraction being defined through itself.

In computer science:

- a data structure defined so (list, tree,...)
- an algorithm solving a problem where the solution depends on solutions to smaller instances of the same problem (think of methods that call themselves)



# Recursion

“to understand recursion you need to understand recursion”

Structure:

- **base** case (minimal solution)
- **recursive** case (growing the solution)

Example: factorial computation

4! = 4*3*2*1 = 24
5! = 5*4*3*2*1 = 5*4!

```
int factorial(int n){
  if (n == 0) return 1;
  else return(n * factorial(n-1));
}
```

## Divide and Conquer

**Optimal Substructure Property**: A problem follows optimal substructure property if the optimal solution for the problem can be formed on the basis of the optimal solution to its subproblems

Divide and conquer approach works for solving problems which:

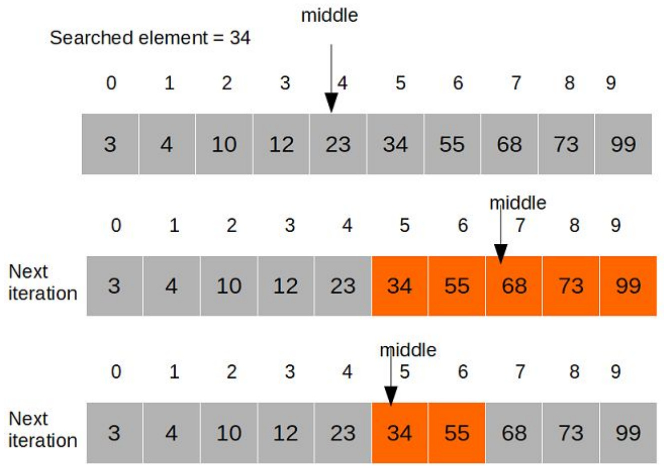
- have optimal substructure property
- can be solved by combining optimal solutions to *non-overlapping* sub-problems

# Divide and Conquer

**Optimal Substructure Property**: A problem follows optimal substructure property if the optimal solution for the problem can be formed on the basis of the optimal solution to its subproblems

Example: binary search for an element with a value **N** in a sorted array

- **divide**: split array in the middle
- if middle < N, repeat for left split
- else repeat for right split

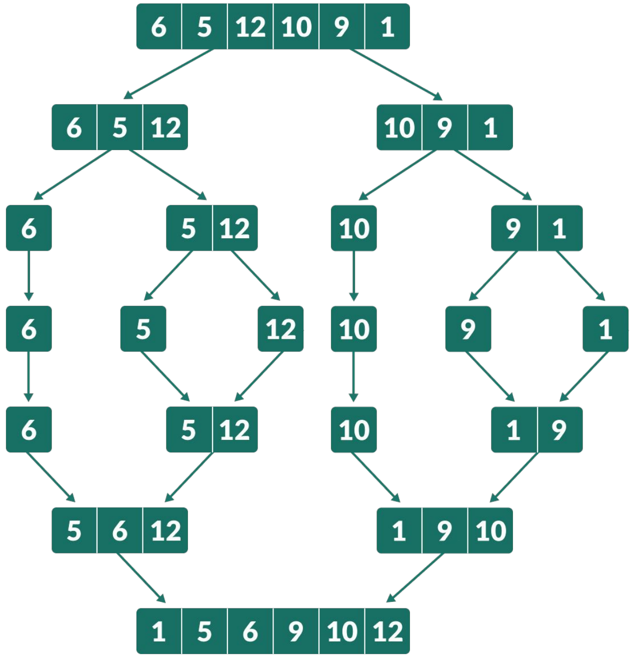


# Divide and Conquer

**Optimal Substructure Property**: A problem follows optimal substructure property if the optimal solution for the problem can be formed on the basis of the optimal solution to its subproblems

Example: mergesort:

- **divide** the unsorted list into n sublists, each containing one element
- repeatedly merge sublists to produce new sorted sublist



## Greedy algorithms

**Optimal Substructure Property**: A problem follows optimal substructure property if the optimal solution for the problem can be formed on the basis of the optimal solution to its subproblems

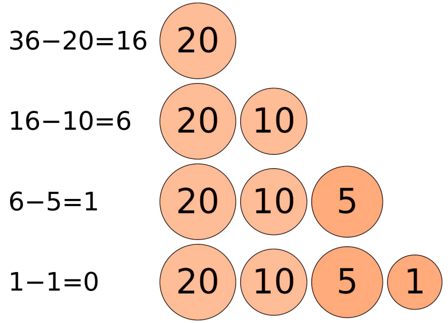
Greedy algorithms work for solving problems which:

- have optimal substructure property
- have **Greedy Choice Property**: global optimum can be reached by selecting the local optimums.

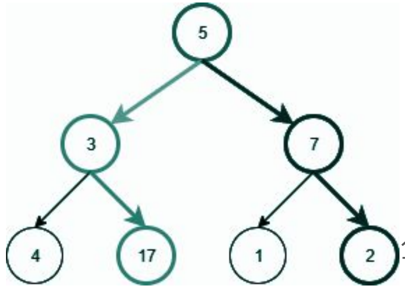
# Greedy algorithms

**Optimal Substructure Property**: A problem follows optimal substructure property if the optimal solution for the problem can be formed on the basis of the optimal solution to its subproblems

*Problem where both greedy choice and optimal substructure property hold: determine minimum number of coins to give while making change*



*Example where greedy choice property does not hold: finding a path with maximal weight in a binary tree*



# Dynamic Programming

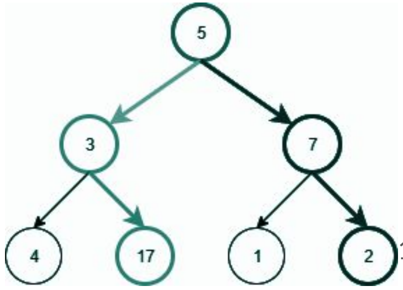
**Optimal Substructure Property**: A problem follows optimal substructure property if the optimal solution for the problem can be formed on the basis of the optimal solution to its subproblems

Divide and conquer approach works for solving problems which:

- have optimal substructure property
- can be solved by combining optimal solutions to *overlapping* sub-problems

Contrast with divide-and-conquer: sub-problems are overlapping, so parts of solution already computed can be reused

*finding a path with maximal weight in a binary tree can be done with DP by memorizing path weights*



## Practice

## Exercise 1

**recursion**

- read an integer number N from input
- print all even numbers from 2 to N without using a loop

**I/O**

- Stream IO to read the input
- Output: Stream IO to print

**Tests**

**-** for algorithm correctness

## Exercise 2

**divide and conquer: binary search**

- read an array of integer numbers from a text file
- sort it
- ask user several times to guess if a certain number is in the array
- if the user guesses correctly, return the position of the number, using binary search
- if the user guesses wrong, return -1

**I/O**

- Input: File IO for reading data
- Output: Stream IO to print

**Tests**

**-** for algorithm correctness

## Exercise 3

**greedy algorithms**

- read two same-size arrays of integers
- write a function to get the minimal sum of their dot product

E.g., if a = {1,3,7}, b = {4,0,5}, then

min_product = 5*1 + 4*3 + 0*7 = 17

think of coin changing problem :-)

**I/O**

- Stream IO to read the input
- Output: Stream IO to print

**Tests**

**-** for algorithm correctness