ALGORITHMS – COMTEK3,CCT3 & ESD3

Taste of Algorithmic Concepts

Ramoni Adeogun

Associate Professor & Head AI for Communications
Wireless Communication Networks Section (WCN)
Department of Electronic Systems
Email: ra@es.aau.dk

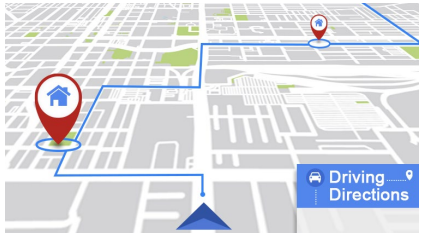
Outline

- Introduction to Algorithms
  - Background and definition
  - Representation & requirements
  - Classification of algorithms
  - Design, analysis, and testing of algorithms
- Algorithmic problem solving & problem specification
- Correctness of algorithms
- Algorithmic efficiency and complexity
- Algorithm design principles
- Summary and key takeaways

Algorithms: What are they about?

There are problems that we solve in everyday life, e.g:

- **drive** from Aalborg to Berlin
- **cook** Frikadeller
- **apply** for Computer Technology Bachelor at Aalborg University



For each of these problems, there are

- **instructions,**
- **recipes, and**
- **procedures,**

which describe the complex operation in terms of elementary **operations**, **control structures** and **conditions**

Algorithms

Some example of problems involving numbers, strings and mathematical objects:

- given **two numbers**, determine their **sum, product, ratio,** …
- given **two numbers**, compute their **lowest common factor**
- given a sequence of strings,
  find an alphabetically sorted permutation of the sequence
- for two arithmetic expressions, determine whether they are equivalent
- for a given house on a map, find the closest bus station

The instructions/procedures and recipes and for solving such problems are referred to as algorithms.

What do recipes have in common with algorithms?
How are they different?

Algorithms: Definition

- An **Algorithm**:
  - is a **finite** sequence of **unambiguous** steps for solving a **well-defined** problem, i.e., for obtaining a desired output for any valid input in a **finite amount of time**
  - is often given either by **several paragraphs in carefully written English** or using **pseudocode**. Such algorithmic description is (largely) "**implementation independent**"
  - can be implemented as (part of) a program using some programming language

- Many engineering and/or computing problems rely on solutions to a small number of *fundamental problems*

- **Resource requirements and limitations** is also often important- and may differ from application to application

- **Consequence**:
  - It is generally useful to have knowledge about several algorithms for the same problem because there will be situations in which each is a better choice than the others

Algorithm Requirements

The requirements for algorithms can be divided into:

- **Data structure and algorithm design requirements:**
  - **Efficiency:** It has to be solvable before we grow too old, or a deadline hits us
  - **Definiteness:** Its steps must be defined precisely.
  - **Correctness**: It must produce the correct output values for each set of input values.
  - **Finiteness**: It must produce the desired output after a finite number of steps for any input
  - **Effectiveness**: It must be possible to perform each step exactly and in a finite amount of time.
  - **Generality:** applicable for all problems of the desired form, not just for a particular set of input values
- **Implementation requirements and/or goals:**
  - **Robustness:** Easy to understand and maintainable
  - **Reusability and adaptability:**

Algorithms: Relationship with Data Structure and Program

- A **data type** is defined by
  - **Data values** and **their representation**
  - Operations defined on the data values

- An **abstract data type** is:
  - a specification of requirements for a data type
  - it does not include a specific implementation but may include conditions that data values must satisfy

- A **data structure** provides a way to represent, store, and/or organize data values as specified by an abstract data type
  - Together with **algorithms** for an abstract data type's operations, this provides an implementation-dependent description of a data type.

- We will study several simple and complex data types, along with data structures and algorithms for their operations in this course.

From Problems to Algorithms

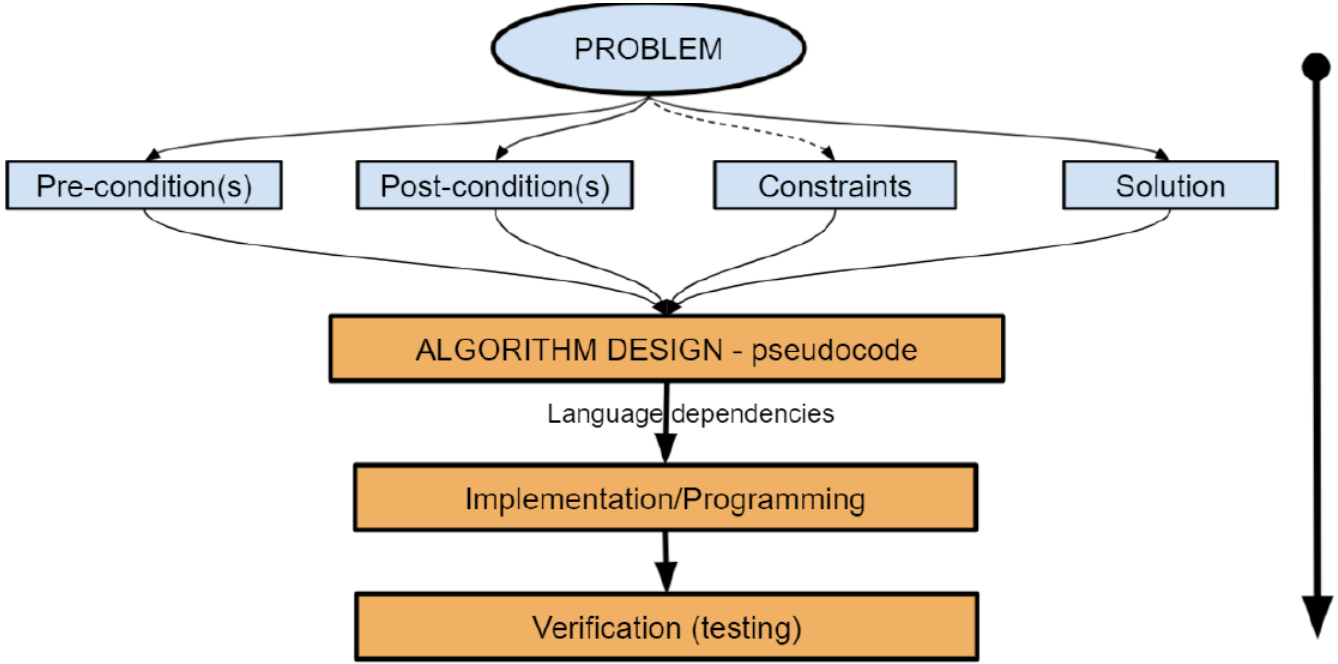
- A problem can be solved in several different ways, therefore there may be many **correct** algorithms for the same problem.

- Which algorithm is best for a given problem largely depends on:
  - the **size of the input**,
  - the actual configuration of the input,
  - the **data structures** used,
  - the computer architecture, and
  - the **algorithmic design**

- What is a correct algorithm?
  - An algorithm is said to be correct if, for every input instance, it halts with the correct output.
  - An incorrect algorithm might not halt on some input instances, or it might halt with an incorrect answer.

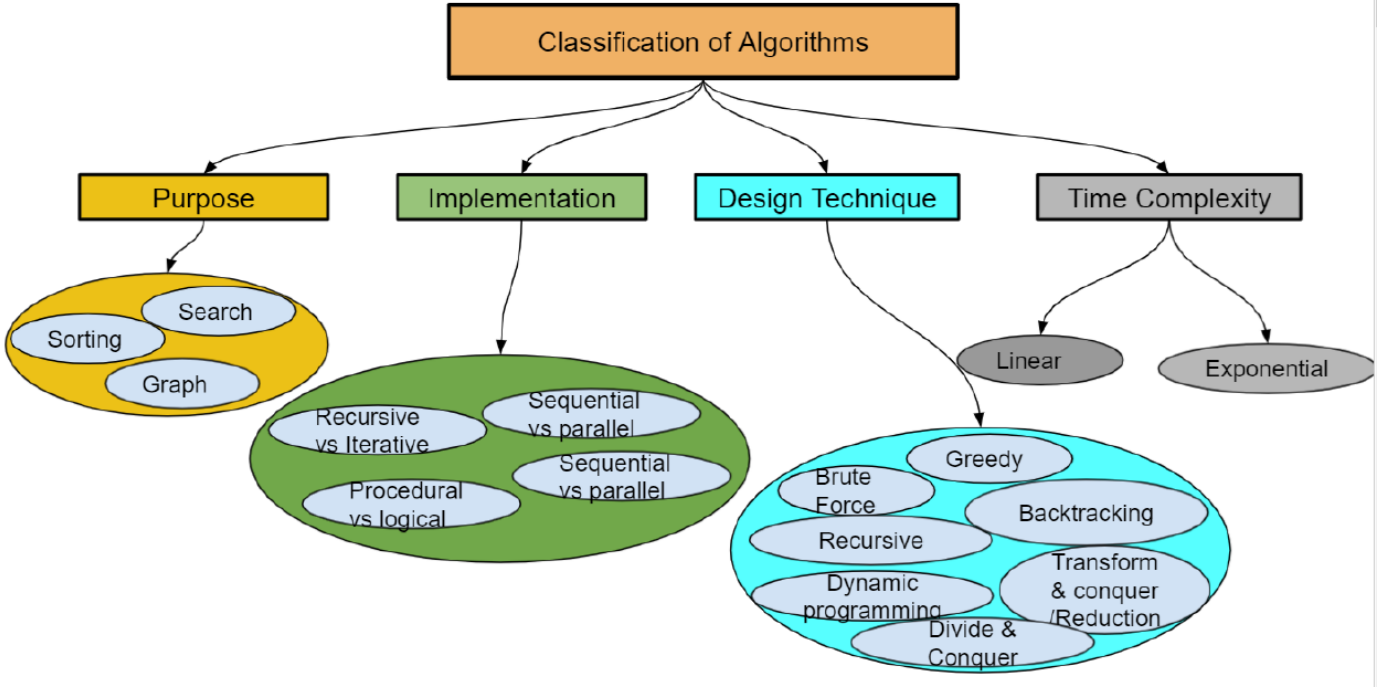
Algorithmic Problem-Solving Process

What are the steps for solving an algorithmic problem?



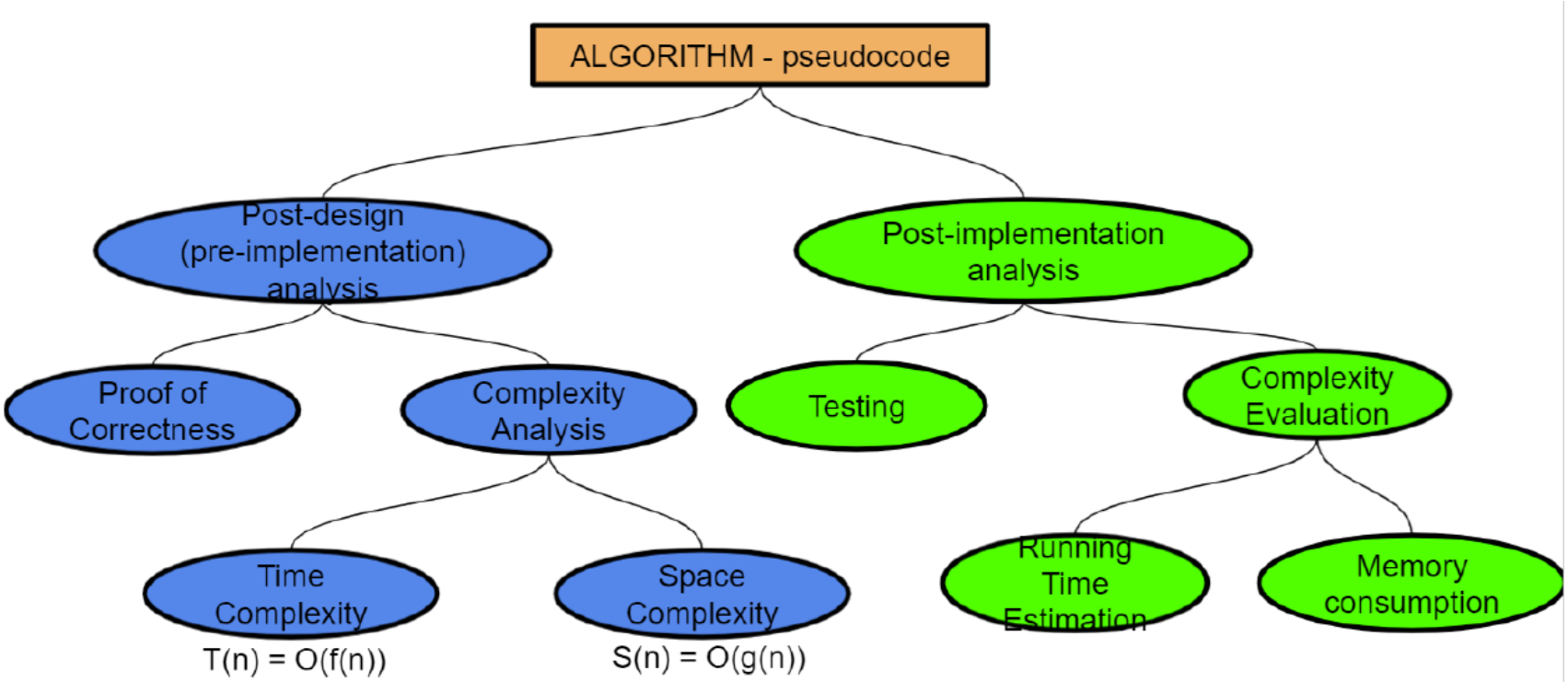
Classification of Algorithms

How do we classify algorithms?



AALBORG UNIVERSITET

Design, Analysis and Testing of Algorithms



AALBORG UNIVERSITET

Example: Specification of a "Median Finding" Problem

The Problem:
Given an array $A = [A[0], \cdots, A[n-1]]$ of n numbers and an index, $i (1 \leq i \leq n)$, find the ith smallest element of A.

Precondition P_1:

- n: a positive integer
- A: an integer array of length n, with entries $A[0], A[1], \cdots, A[n-1]$
- i: An integer index such that $1 \leq i \leq n$

Postcondition, Q_1:

- Output is a number v such that $v = A[i-1]$

Constraints: Inputs (and other variables) have not changed

Special Cases:

- $i = 1$: finding the minimum of A
- $i = n$: finding the maximum of A
- n is odd and $i = (n+1)/2$: finding the median of A
- n is even and $i = (n+1)/2$ or $i = (n+1)/2$: finding the median of A

Example: Specification of a "Search" Problem

Precondition P_1:

- n: a positive integer
- A: an array of length n, with entries $A[0], A[1], \cdots, A[n-1]$
- v: An integer found in the array (i.e., such that $A[i-1] = v$ for at least one integer i between 1 and $n-1$)

Precondition P_2 :

- 1 and 2 as in P_1 and
- v: An integer not found in the array (i.e., $A[i-1] \neq v$ for every i between 1 and $n-1$)

Postcondition, Q_1:

- Output is the integer i such that $1 \leq i \leq n$, $A[j] \neq v$ for every integer j such that $1 \leq j \leq i$, and such that $A[i-1] = v$

Postcondition, Q2:

- A notFoundException is thrown

Constraints:

- Inputs (and other variables) have not changed

Example: Specification of a "Search" Problem

A problem can be specified by multiple precondition-postcondition pairs

$$(P_1, Q_1);\ (P_2, Q_2); \cdots;\ (P_m, Q_m)$$

so long it is not possible for more than one of the preconditions

$$P_1, P_2, \cdots, P_m$$

to be satisfied at the same time.

For example, if P_1, Q_1, P_2, and Q_2 are as defined in the last two slides, then the precondition-postcondition pairs

$$(P_1, Q_1);\ (P_2, Q_2)$$

could specify a "search" problem in which the inputs are expected to be a positive integer n, an integer array, A with length n, and an integer, v.

Instance of an Algorithmic Problem

- Consider **the Search Problem** described earlier, the following is a valid **input instance** and the corresponding **output**

- **Precondition/Input:**
  - A sequence of integers ⟨2, 8, 10, 9, 15, 20, 14⟩**, and**
  - An integer, ⟨20⟩
- **Postcondition/Output:**
  - an integer index , ⟨5⟩

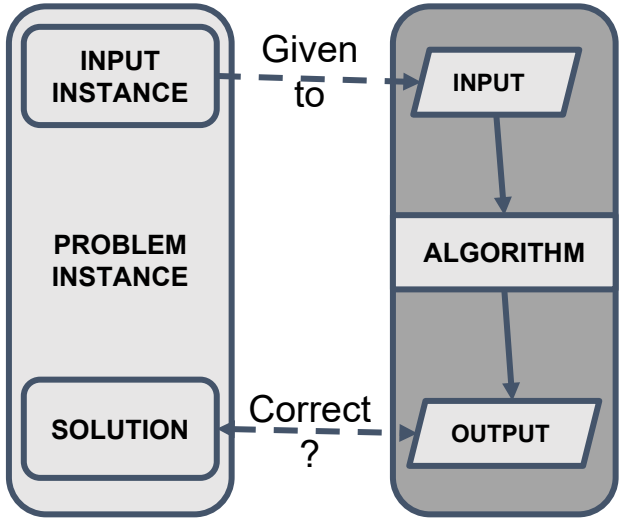
Note: An ***instance*** of a problem consists of the input (satisfying constraints imposed in the problem statement and/or preconditions) needed to obtain a solution to the problem.

Correctness of Algorithms

- Suppose that a problem is specified by a single precondition-postcondition pair (P,Q). An algorithm (that is supposed to solve this problem) is correct if it satisfies the following conditions:
  - inputs satisfy the given precondition, P and
  - the algorithm is executed

  then
  - the algorithm eventually halts, and the given postcondition, Q is satisfied on termination.

- In general, a distinction between *partial correctness* (which requires that **if an answer is returned, it will be correct**) and *total correctness* (which additionally **require that the algorithm terminates**) is made.



Correctness of Algorithms

Now suppose that a problem is specified by m (with $m \geq 2$) precondition-postcondition pairs

$$(P_1, Q_1); (P_2, Q_2); \cdots ; (P_m, Q_m)$$

where it is impossible for more than one of the conditions to be satisfied at the same time.

An algorithm for solving this problem is correct if the following is true for every integer i between 1 and m: if

- inputs satisfy the given precondition, P_i and
- the algorithm is executed

then

- the algorithm eventually halts, and the given postcondition, Q_i is satisfied on termination.

Consequently, an algorithm for this problem is correct if and only if it is a correct solution for each of the m problems that are each specified by the single precondition-postcondition pair, (P_i, Q_i), for i between 1 and m.

Why are Proofs of Algorithm Correctness Useful?

Who Generates Proofs of Correctness?

- Engineers and/or scientists designing new algorithms (especially when the algorithms are not obvious).
- Other people need to see evidence that the new algorithm(s) really works as desired.

Who Uses Proofs of Correctness?

- Anyone involved with coding, testing, and/or maintenance of the software implementing a non-trivial algorithm needs to know why (or how) the algorithm solves the problem it is supposed to solve correctly.

Algorithm Efficiency and Complexity

Now that our algorithm has been shown to be **correct**, we also need to consider its **efficiency**!.

But how do we measure efficiency? The following are all (somewhat) important:

- **Running Time**: We do not want to wait a life-time for an algorithm to execute.
- **Data Storage Space**: While time is somewhat unconstrained, any computer can run out of memory.
- **Memory Used by the Code**: This is an issue if a program is to be stored on low-memory devices
- **Implementation Time**: Programmers must be paid to code the algorithm and development process has deadlines!

In this course, we will concentrate on analysis of **running time** and **space**.

But How Do We Measure Time Complexity?

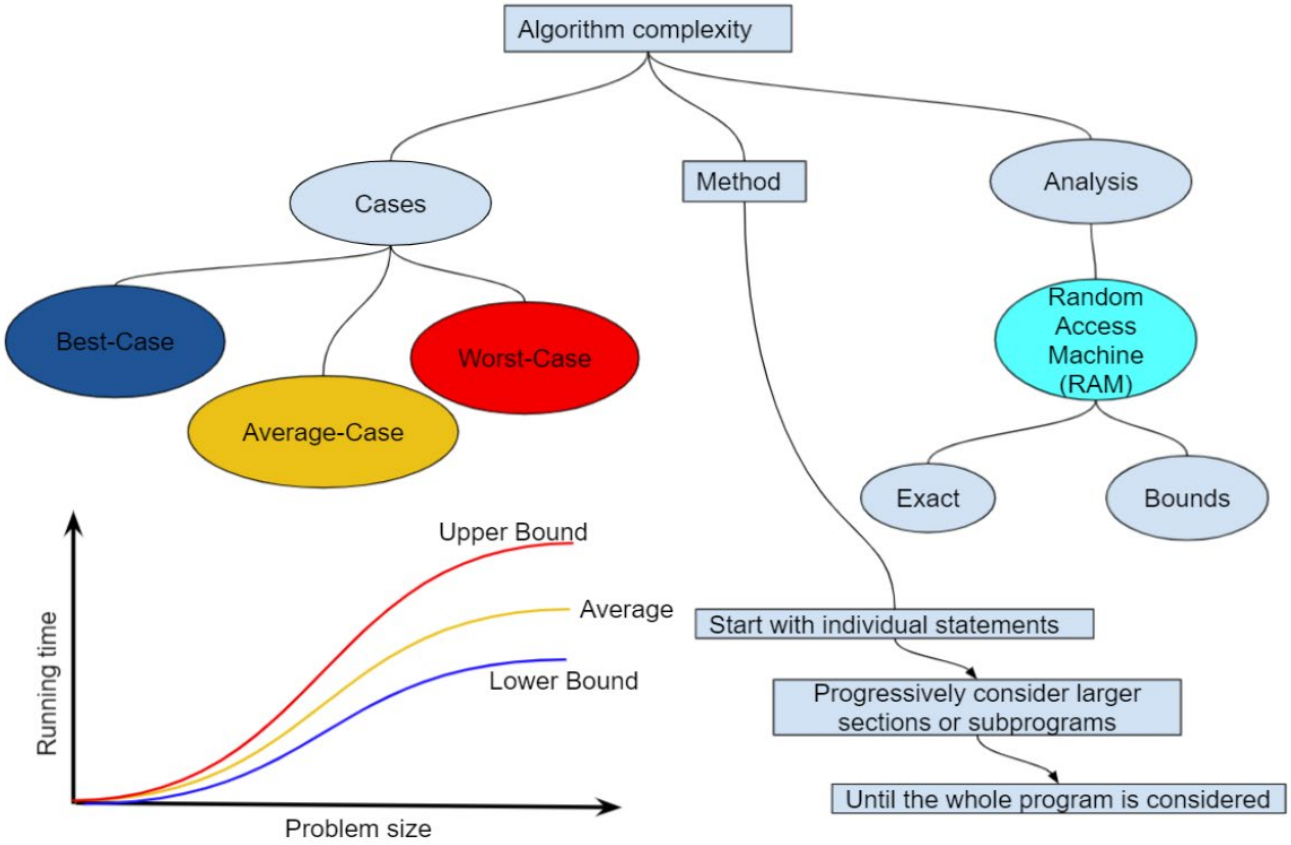
How can we compare the time complexity of algorithms or programs?

- **Run the code and time its execution**.
  Challenge: Running time is influenced by many factors:
  - Hardware: How fast is the CPU? How many are there?
  - Compiler and System Software: Which Operating System?
  - Multiple User Activities at the same time
  - Size of input data
  - Programming style

- **Analyze the code or algorithm**: Only influenced by choice of data but can be quite difficult!!

In general, we try to combine complexity analysis with running times.

Algorithm Complexity



The RAM Model

- We study complexity based on a simplified machine model, the RAM (Random Access Machine):
  - accessing and manipulating data takes a (small) constant amount of time
- Among the **instructions** (each taking constant time), **we usually choose one type of instruction as a characteristic operation that is counted**:
  - arithmetic (add, subtract, multiply, etc.)
  - data movement (assign)
  - control flow (branch, subroutine call, return)
  - comparison
- **Data types**: integers, floats and characters

Role of Complexity Analysis

- Complexity analysis is a vital tool for **making informed decisions about algorithm selection, optimization, and system design**.
- Complexity analysis assists in **selecting the most suitable algorithm for a specific problem**.
  - We can prioritize algorithms that offer better time or space efficiency, depending on the application's requirements.
- Complexity analysis **guides algorithm optimization efforts**.
  - Balancing time and space complexity often involves trade-offs; improving one aspect might lead to a decline in the other.
- In real-world applications, input sizes can vary significantly.
  - Complexity analysis helps **predict how an algorithm will perform as data scales**
- Complexity analysis highlights bottlenecks, prompting the search for more efficient algorithms or data structures.

Algorithm design principles

Several paradigms exist for the design and analysis of algorithms, some examples are:

- **Greedy:**
  - picking of apparent local optimal solution to a subproblem => disregard “global optimality” consideration at each step
- **Divide-and-conquer:**
  - **Breaking down** of problem into subproblems that are easier to solve, **solving** them and combining the solutions
    - Divide -> conquer -> combine
- **Brute-force:**
  - keep it algorithm simple and straightforward based on the problem statement, etc
  - disregard any special structure of the problem or possible smart ideas
  - Examples
    - Finding maximum by comparing each element with a temporary maximum
- **Dynamic programming:**
  - Reuse of earlier results
  - Similar to divide-and-conquer, but are typically faster due to reuse of results

Summary and Key Takeaways

- To solve an algorithmic problem:
  1. Clearly define the problem → preconditions, postconditions, constraints
  2. Specify the algorithm → pseudocode
  3. Proof that the algorithm is correct → it terminates and produces the correct result
  4. Analyze the algorithm efficiency → exact and/or asymptotic complexity
  5. Select a programming language and implement
  6. Perform post-implementation program analysis → testing and running time estimation

- Design technique for an algorithm depends on the problem, e.g.,
  - Divide and Conquer → divide, solve (recursively) and combine
  - Dynamic programming
  - Greedy
  - Brute force, etc.