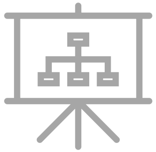
METROPOLITAN
TIRANA
UNIVERSITY

Course: Data Structures and Algorithms

Algorithm Analysis and Problem Solving



Evis Plaku

Why do we even need algorithms?

METROPOLITAN TIRANA UNIVERSITY

Because even your

favorite app's

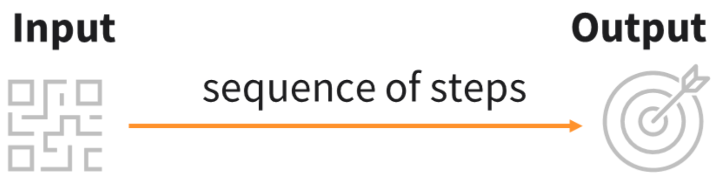
'**loading**' screen

has a story to tell...

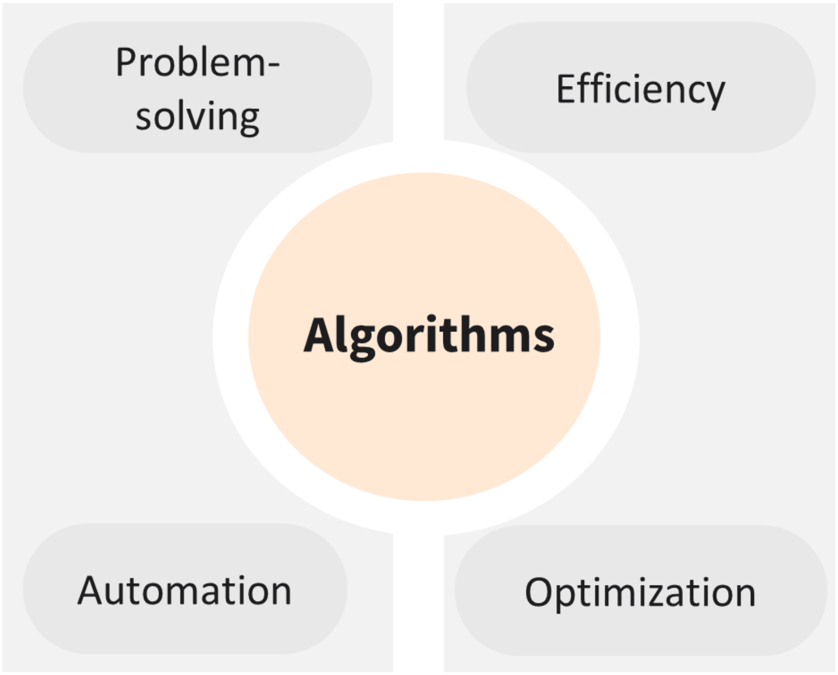
I blame "the algorithm" behind this

Algorithm Fundamentals

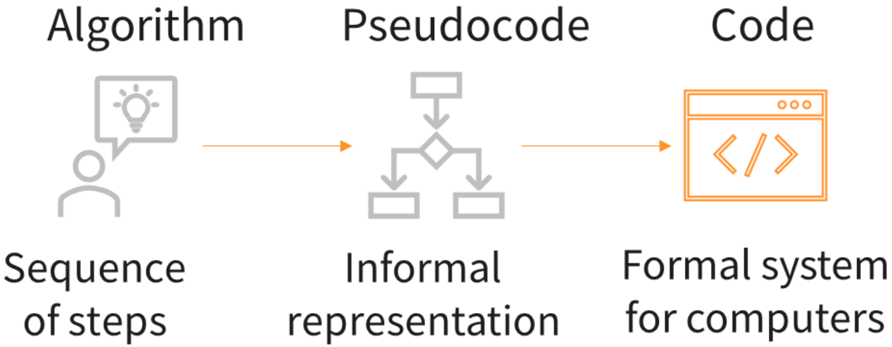
Importance of algorithms



- **Step-by-step instructions to solve tasks efficiently**
- Definite: clear and unambiguous
- Termination: ends in finite steps



Programming the algorithms



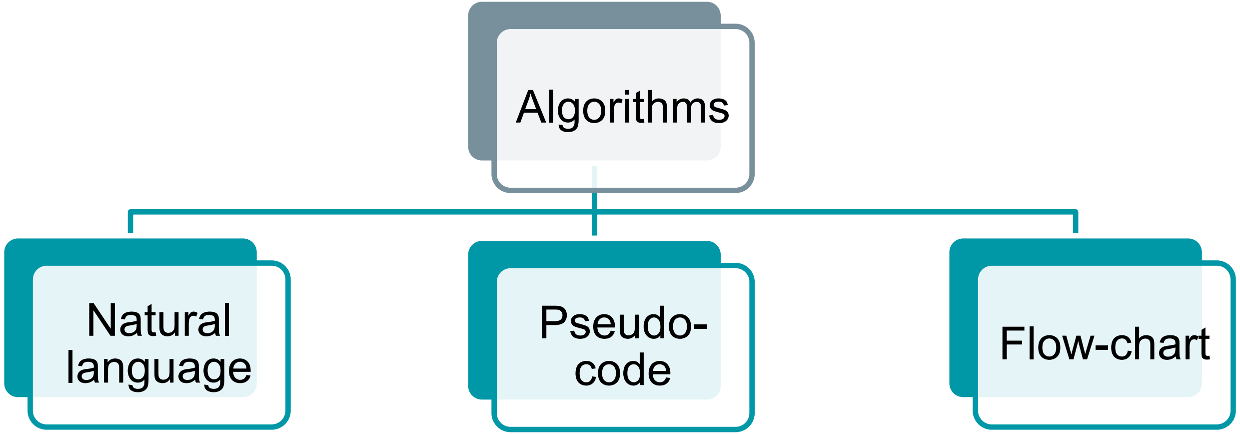
- Programming: convert algorithms to machine-readable instructions
- Language types: machine, assembly, low and high-level languages

How can we choose the most effective algorithm for a task?

- Consider time and space complexity
- Time concerns the number of algorithmic steps
- Space efficiency relates to memory usage

Detailing and abstraction

Clear algorithm design increases programming efficiency



Plain language

Code-like representation

Visualized process step

Example: finding greatest common divisor

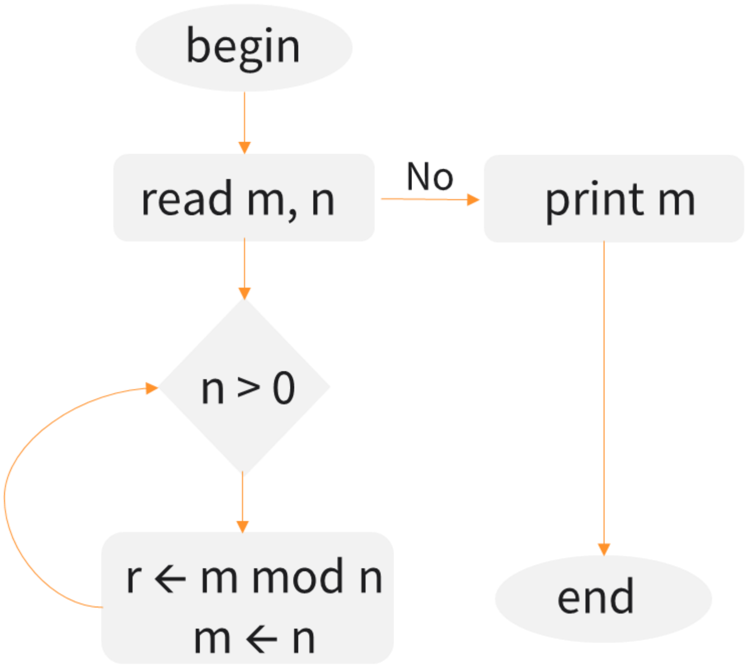
Plain language

```
1. Input two numbers.
2. Determine the maximum (m) and minimum (n).
3. If n is zero, output m.
4. Otherwise, find the remainder (r) of m divided by n.
4. Set m = n and n = r.
5. Repeat from step 3.
```

Pseudocode

```
begin
    read a, b
    m ← maximum(a, b)
    n ← minimum(a, b)
    while (n ≠ 0)
        r ← m mod n
        m ← n
        n ← r
    endwhile
    return m
end
```

Flowchart



Problem solving and algorithm design

A clear, structured approach ensures correct and efficient solutions to problems

1. **Understand the problem**
   Clarify the problem, inputs, outputs, constraints, and requirements
2. **Plan the solution**
   Break the problem into smaller steps or subproblems
3. **Solve the problem**
   Design an algorithm using appropriate techniques
4. **Test and optimize**
   Validate the solution and improve efficiency

Evaluating algorithms

METROPOLITAN TIRANA UNIVERSITY

Algorithm selection is about balancing efficiency, not just optimizing one factor

- Analyze how algorithm runtime grows with input size (Big-O notation)
- Measure memory usage as input size increases (Big-O space complexity)
- Optimizing time may increase space usage, or vice versa

Select based on problem constraints: faster solution or lower memory usage?

Practical aspects on evaluating algorithms

METROPOLITAN TIRANA UNIVERSITY

Time measurement helps to understand algorithm performance under different conditions

- Measure the time before execution
- **Run the algorithm**
- Measure the time after execution

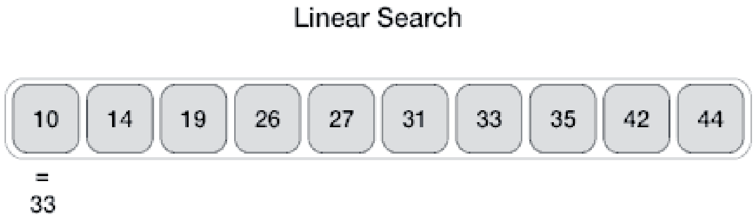
```
START
  startTime = GetCurrentTime()
  result = RunAlgorithm(inputData)
  endTime = GetCurrentTime()
  elapsedTime = endTime - startTime
END
```

- Run multiple tests with varying input sizes and conditions to gather meaningful results. Average time over multiple runs for more reliable data

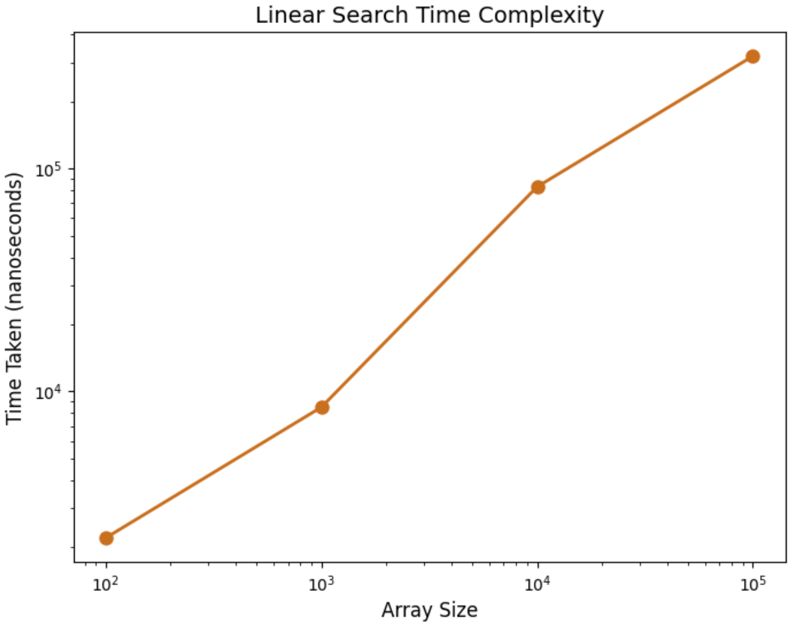
Basic algorithm example: linear search

METROPOLITAN TIRANA UNIVERSITY

Search for an element in a list by checking each element sequentially




```java
public static int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
       if (arr[i] == target) {
           return i;
        }
    }
    return -1;
}
```

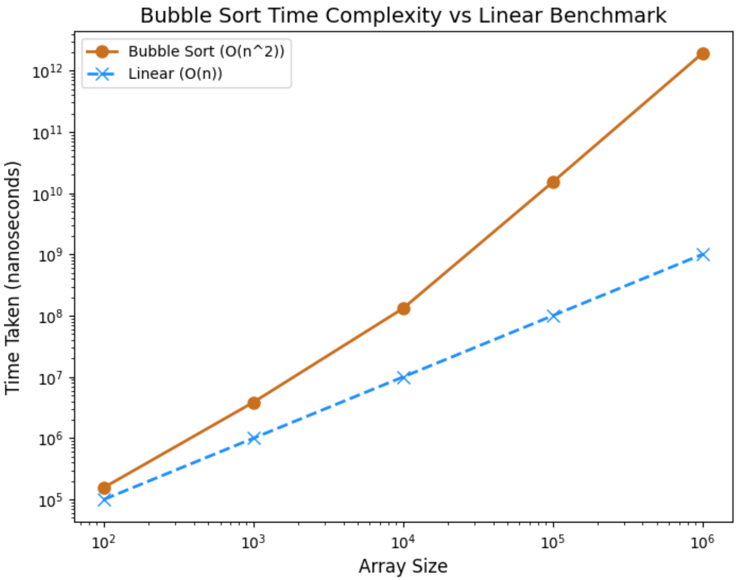


Basic algorithm example: bubble sort

Bubble Sort repeatedly swaps adjacent elements until the array is sorted

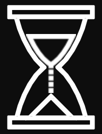
8 5 3 1 4 7 9

```java
public static void bubbleSort(int[] arr) {
   for (int i = 0; i < arr.length - 1; i++) {
       for (int j = 0; j < arr.length - i - 1; j++) {
           if (arr[j] > arr[j + 1]) {
               int temp = arr[j];
               arr[j] = arr[j + 1];
               arr[j + 1] = temp;
           }
       }
   }
}
```



Understanding Algorithm Efficiency

Time and Space Complexity



Time and Space Complexity

Understanding efficiency helps improve algorithm performance and scalability

- **Optimize performance**: ensure fast execution for large inputs
- **Manage resources**: limit memory usage and storage needs
- **Scalability:** handle increasing input sizes effectively

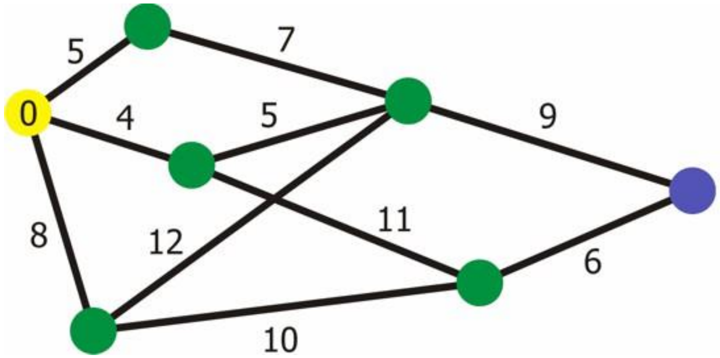


Illustration: a graph algorithm

Time and Space Complexity

METROPOLITAN TIRANA UNIVERSITY

Time complexity	Space complexity
• Time to run algorithm • Based on input size • Behavior for large inputs • Big O notation	• Memory required • Accounts for all storage • Memory usage growth • Big O notation

Optimizing one often compromises the other

Balance is key

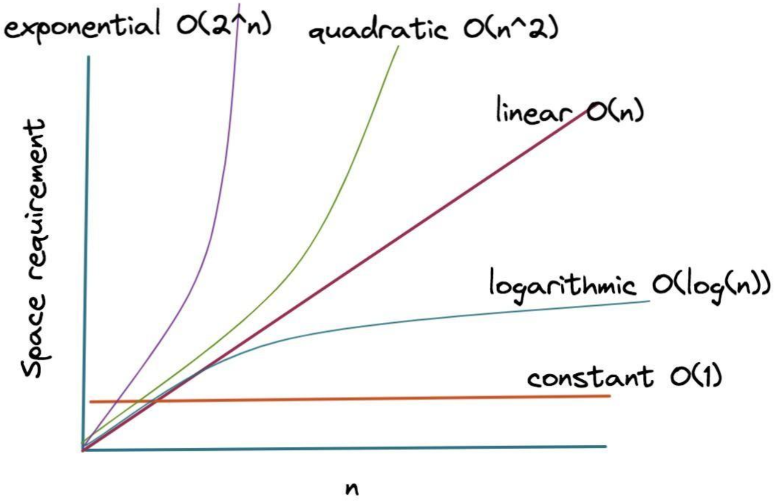
Big O Notation

A mathematical notation to describe algorithm performance

- Describes time/space complexity growth

- Usually focuses on upper-bound behavior

- Helps compare algorithms regardless of hardware/environment



Big O Mathematical Formulation

$$O(f(n) = \{g(n) : \exists c > 0, n_0 \geq 0\}$$

$$such\ that\ for\ all\ n \geq n_0,\ |g(n)| \leq c \cdot f(n))$$

- $f(n)$ algorithm's time or space complexity
- $g(n)$ function that describes the growth rate
- c is a constant factor, and n_0 is the input size where the behavior starts to hold true

expresses how the algorithm's resource usage grows as the input size (n) increases

Best, Average and Worst Time Complexity

- **Best Time (Ω notation)**: minimum performance for optimal inputs

$$T_{best} = \Omega(f(n))$$

- **Average Time (Θ notation)**: expected performance over typical inputs

$$T_{average} = \Theta(f(n))$$

- **Worst Time (O notation)**: maximum performance for worst-case inputs

$$T_{worst} = O(f(n))$$

☞ These notations capture the range of possible execution times

Understanding Complexity through Examples

METROPOLITAN TIRANA UNIVERSITY

Linear search

```
public int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) {
            return i;  // Found
        }
    }
    return -1;         // Not found
}
```

Checking if an element is part of an array

- Element is the first one.

$$T_{best} = \Omega(1)$$

- Target is somewhere *in middle*

$$T_{average} = \Omega(n)$$

- Target is the last element (or not present)

$$T_{worst} = O(n)$$

Understanding Complexity through Examples

METROPOLITAN TIRANA UNIVERSITY

Finding maximum element in a list

```java
public int findMax(int[] arr) {
    int max = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

Best case = Average Case = Worst Case

$$\Omega(n) = \Theta(n) = O(n)$$

- All elements must be checked to ensure maximum is found

- What if array is already sorted?

Common Big O Time Complexities

- $O(1)$	Constant Time	Checking if number is even Accessing an element in an array
- $O(\log n)$	Logarithmic Time	Looking up a word in a dictionary Finding a contact in a sorted phonebook
- $O(n)$	Linear Time	Counting occurrences of words in a book Checking if a list contains a value
- $O(n \log n)$	Linearithmic time	Sorting a list of names Merging large log files by timestamp

Common Big O Time Complexities

- $O(n^2)$	Quadratic Time	Comparing all students for similarity Checking for duplicate names in list
- $O(2^n)$	Exponential Time	Generating all subsets of a set Solving Towers of Hanoi
- $O(n!)$	Factorial time	Finding all possible seat arrangements Brute force to solve traveling salesman

Avoid **quadratic, exponential, and factorial** solutions for large inputs!

Algorithm Design Techniques

Fundamental algorithm design techniques

Multiple techniques exist to tackle diverse problem types and ensure optimal solutions

- Algorithm techniques
  - Brute force
  - Divide and conquer
  - Recursion
  - Greedy
  - Dynamic programming

Brute force approach

A straightforward problem-solving method that involves trying all possible solutions to find the correct one

When to apply

- Small problem size
- Solution space is manageable
- Sophisticated algorithms non available or necessary

+	−
- Easy to implement	- Slow and expensive
- Guaranteed solution	- Impractical for large input size
- No assumptions	- Inefficient for complex problems

Brute force approach example: linear search

```
function linearSearch(arr, target):
    for i = 0 to length(arr) - 1:
        if arr[i] == target:
            return i
    return -1
```

- Best case: $O(1)$
- Worst and average case: $O(n)$

- **Exhaustive search**: checks every element
- **Simple & direct**: no optimizations or shortcuts
- **Guaranteed solution**: finds the target if it exists
- **Inefficient**: slow for large datasets

Divide and conquer strategy

Break problem into subproblems,
solve independently, combine results

Divide

Break the problem into smaller subproblems

Conquer

Solve each subproblem recursively

Combine

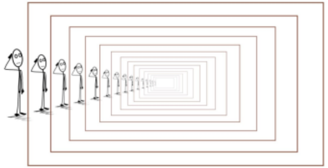
Combine the solutions of the subproblems to form the final solution

Divide and conquer example: merge sort

- **Divide:** split the array into two halves
- **Conquer**: recursively sort each half
- **Combine**: merge the sorted halves into a single sorted array

Recursively divide the array, sort each half, then merge them

Recursion



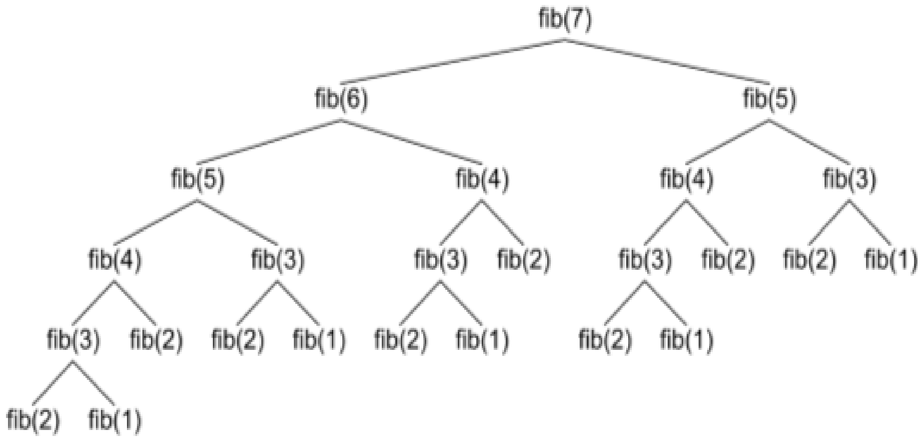
Technique where a function calls itself to solve smaller instances of the same problem

- **Base case:** simple, direct solution for smallest problem
- **Recursive step**: break problem into smaller subproblems
- Self-call: function calls itself with smaller input

Recursion is a key technique used in Divide and Conquer to solve subproblems recursively

Recursion example: Fibonacci numbers

```
function fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

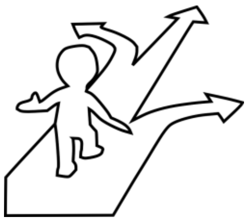


- **Fibonacci series:** each number is the sum of the two preceding ones

- Time Complexity: $O(2^n)$ due to redundant recursive calls

Recursive functions can be elegant but may lead to high time complexity and inefficiency

Greedy algorithms



Approach where locally optimal choices lead to a globally optimal solution

- **Greedy property:** making the best choice at each step ensures an optimal solution

- A problem can be broken into subproblems, and <u>optimal solutions to subproblems lead to an overall optimal solution</u>

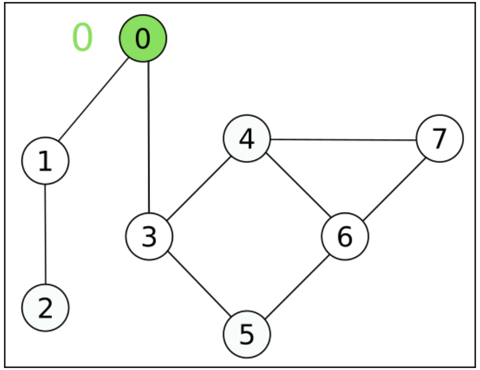
Use for optimization problems where local choices lead to global solutions, like scheduling and networking

Greedy algorithms typical use cases

METROPOLITAN TIRANA UNIVERSITY

Graph problems

Finds the shortest path from a single source to all other nodes



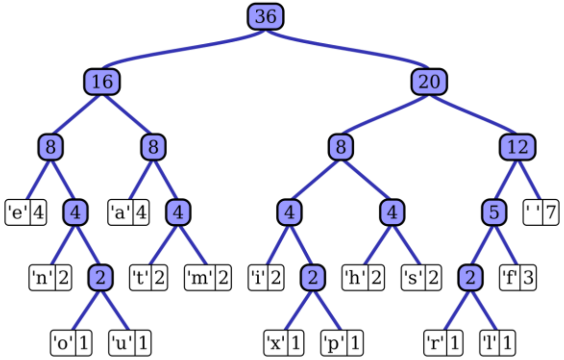
Scheduling and optimization

Choose the maximum number of non-overlapping activities

ACTIVITY	START TIME	END TIME
A1	1	3
A2	2	5
A3	3	4
A4	4	7
A5	7	10
A6	8	9

Resource allocation

Huffman coding: creates optimal prefix-free encoding for data compression



Greedy algorithm example: activity selection

METROPOLITAN TIRANA UNIVERSITY

Select the maximum number of non-overlapping activities given their start and end times

- **Greedy Choice:** always pick the activity that finishes the earliest
- Time Complexity: $O(n \log n)$ due to sorting, followed by $O(n)$ selection
- Real-World Applications: scheduling tasks, meeting room allocation, interview scheduling

```
FUNCTION MaxActivities(activities):
    SORT activities by end time in ascending order
    count ← 1
    lastEnd ← activities[0].end

    FOR i FROM 1 TO length(activities) - 1:
        IF activities[i].start ≥ lastEnd:
            count ← count + 1
            lastEnd ← activities[i].end
    RETURN count
```

Dynamic programming

Solve problems by breaking them into subproblems and reusing results

- Optimal solution is built from optimal subproblem solutions

- Overlapping subproblems: **subproblems are solved multiple times, so store results.**

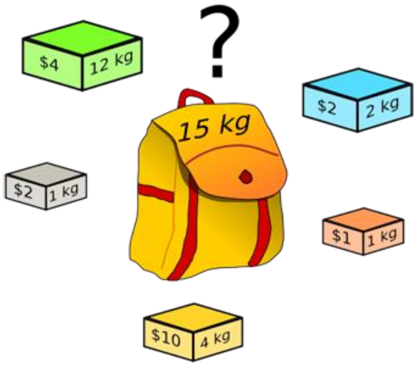
Dynamic programming is used for optimization, decision-making, resource allocation, problem decomposition

Dynamic programming major use cases

METROPOLITAN TIRANA UNIVERSITY

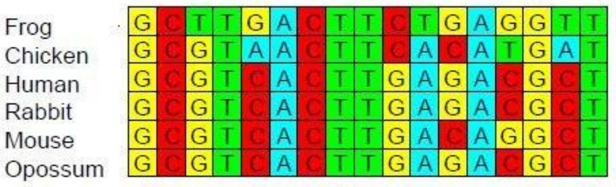
Optimization

Knapsack problem by maximizing profit within constraints



Sequence alignment

Finding the best match between two sequences (e.g., DNA, text)



Shortest path

Computing the shortest path in weighted graphs (e.g., Dijkstra's algorithm)

Backtracking

A general algorithm for finding solutions to problems incrementally, by exploring all possibilities

- Build solutions step by step, abandoning partial solutions that fail constraints (backtrack)

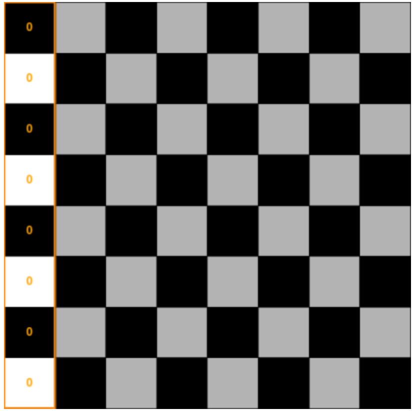
- Use cases: solving combinatorial problems like puzzles, pathfinding, and constraint satisfaction

Backtracking tries all possibilities step by step, undoing choices when they don't work

Backtracking major use cases

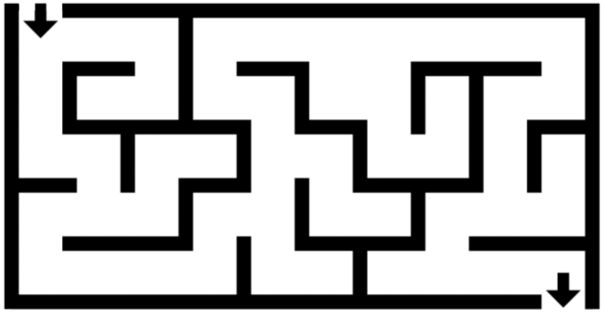
Solving puzzles

Exploring all possible moves in puzzles like Sudoku and N-Queens to find a solution



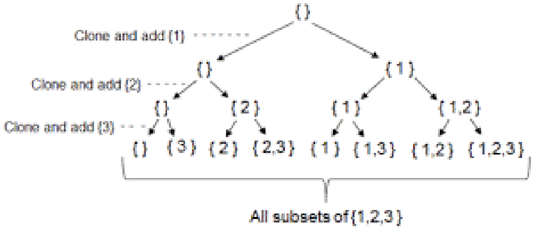
Pathfinding

Finding possible paths in mazes or graphs by trying different routes and backtracking



Subset generation

Generating all possible subsets or combinations, like in the subset sum problem



Summary of algorithm design techniques

METROPOLITAN TIRANA UNIVERSITY

Optimization and problem solving

- **Greedy:** fast solutions by making local optimal choices
- **Dynamic programming:** optimizes by breaking problems into overlapping subproblems

Divide and conquer

- **Divide and conquer:** divides problems into smaller subproblems, solving them independently
- **Recursion:** break problems into smaller, similar subproblems

Exhaustive search

- **Brute force:** tries all solutions without optimization, best for small problems
- **Backtracking:** explore all solutions, backtrack when constraints are violated

Algorithm Complexity Classes

What are complexity classes

Categories of problems based on
how their time/space requirements grow

- P: Problems solvable in polynomial time (efficient)
- NP: Problems whose solutions can be verified in polynomial time
- NP-Complete: Hardest problems in NP, no known fast solution
- NP-Hard: Problems at least as hard as NP problems, may not be in NP

Helps determine if a problem can be solved efficiently

Class P

Problems that can be solved in time proportional to a polynomial function of input size

- Sorting: arranging data in a specific order (e.g., Merge Sort, Quick Sort)
- Searching: finding an element in a list (e.g., Binary Search, Linear Search)
- Shortest Path: finding the shortest path in a graph (e.g., Dijkstra's Algorithm)

Easy to solve.

Feasible even with large inputs

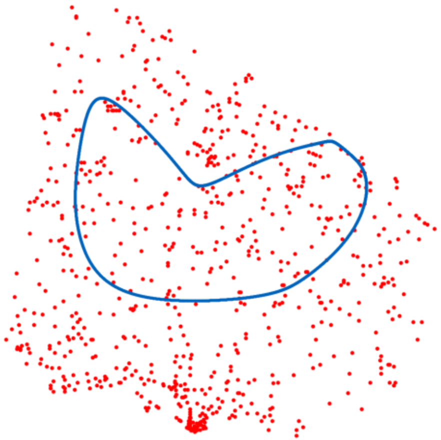
crucial for real-world applications and scalability

Class NP

Problems where, if given a solution, we can verify its correctness in polynomial time

- Significance: solutions may be hard to find, but easy to check

- Example: **knapsack problem** (finding the most valuable combination of items without exceeding weight limit)



Traveling Salesman Problem: finding the shortest possible route that visits each city once

The NP-hard and NP-complete problems

METROPOLITAN TIRANA UNIVERSITY

NP – hard

- Problems at least as hard as NP but may not be in NP
- No known polynomial-time solution; may not even have verifiable solutions in polynomial time

NP – complete

- Problems that are both in NP and NP-Hard
- *If one NP-Complete problem is solved in polynomial time, all NP problems can be solved in polynomial time*

$$P \subseteq NP \subseteq NP\ \text{hard} \subseteq NP\ complete$$

Understanding these classes helps in determining problem feasibility

Many real-world optimization problems fall into these categories

Key Takeaways

- **Algorithm design strategies**: Brute Force, Divide & Conquer, Recursion, Greedy, Dynamic Programming, and Backtracking solve different types of problems
- **Efficiency matters**: Polynomial-time (P) problems are feasible; NP and NP-Hard problems are much harder to solve
- **NP vs. P**: P problems can be solved efficiently; NP problems can be verified quickly but may not be solvable efficiently
- Computational Limits: NP-Complete problems connect NP and NP-Hard; solving one efficiently could solve all NP problems

Helpful Resources on Algorithms & Complexity

METROPOLITAN TIRANA UNIVERSITY

- 📝 GeekForGeeks Algorithm design techniques
  Concise explanations and examples

- 📖 P versus NP in simple plain English

- 📝 Khan's academy comprehensive list of important algorithms

- 🎥 Example of divide and conquer strategy

Quote of the Week

METROPOLITAN
TIRANA
UNIVERSITY

The right algorithm can
save hours of computing…
or years of waiting