HyperionDev

Welcome to this session:
Introduction to Algorithms and Problem Solving Methodologies

The session will start shortly...

Questions? Drop them in the chat.
We'll have dedicated moderators answering questions.

HyperionDev

What is Safeguarding?

Safeguarding refers to actions and measures aimed at protecting the human rights of adults, particularly vulnerable individuals, from abuse, neglect, and harm.

To report a safeguarding concern reach out to us via email:

safeguarding@hyperiondev.com

HyperionDev

Live Lecture Housekeeping:

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
- No question is daft or silly - ask them!
- For all non-academic questions, please submit a query: www.hyperiondev.com/support
- To report a safeguarding concern reach out to us via email: safeguarding@hyperiondev.com
- If you are hearing impaired, please kindly use your computer's function through Google chrome to enable captions.

Learning Outcomes

- **Define** and **explain** the concept of **algorithms** in computer science.
- **Describe** the characteristics of a **good algorithm** (correctness, efficiency, readability).
- **Identify** and **utilise basic problem-solving methodologies** such as divide-and-conquer, brute force, and greedy algorithms.
- **Illustrate the process** of translating a problem into a **step-by-step algorithm.**

Relevance

Think about your morning routine - from the moment you wake up until you arrive at class.

- *How many decisions do you make, and what's your process for making them efficiently?*
- *For example, do you optimize your route to avoid traffic, or plan your breakfast to save time?*

HyperionDev

Relevance

- This is actually an algorithm you've created!
- Could someone share their morning 'algorithm' and explain why they chose that specific sequence of steps?

HyperionDev

Relevance

- What happens when something unexpected disrupts your routine?
- Have you ever tried to optimize or improve your morning sequence?
- How do you handle multiple constraints like time, resources, and preferences?

What is an algorithm?

A. A detailed step-by-step procedure for solving a problem
B. A programming language
C. A hardware component
D. A design pattern

Which of the following is a characteristic of a good algorithm?

A. Complexity and Length
B. Correctness, Efficiency, and Readability
C. Use of Multiple Programming Languages
D. Randomness in Output

HyperionDev

Lecture Overview

- Introduction to Algorithms
- Fundamentals and Characteristics
- Problem-Solving Methods
- Putting it into Practice

Introduction to Algorithms

HyperionDev

What is an Algorithm?

- An **algorithm** is:
  - A precise, step-by-step procedure for solving a problem
  - Comparable to a recipe for computers
- Key Characteristics:
  - Precise: Clear instructions
  - Unambiguous: No room for interpretation
  - Finite: Must terminate
- Examples in Daily Life:
  - Following a cooking recipe
  - Giving directions to a destination

Why are algorithms important?

- **Foundation** of Computer Programming:
  - Core to software development
- **Efficiency** in Problem-Solving:
  - Streamlines complex tasks
- Applications in Various Fields:
  - Data Processing
  - Search Engines
  - Financial Systems
  - Machine Learning

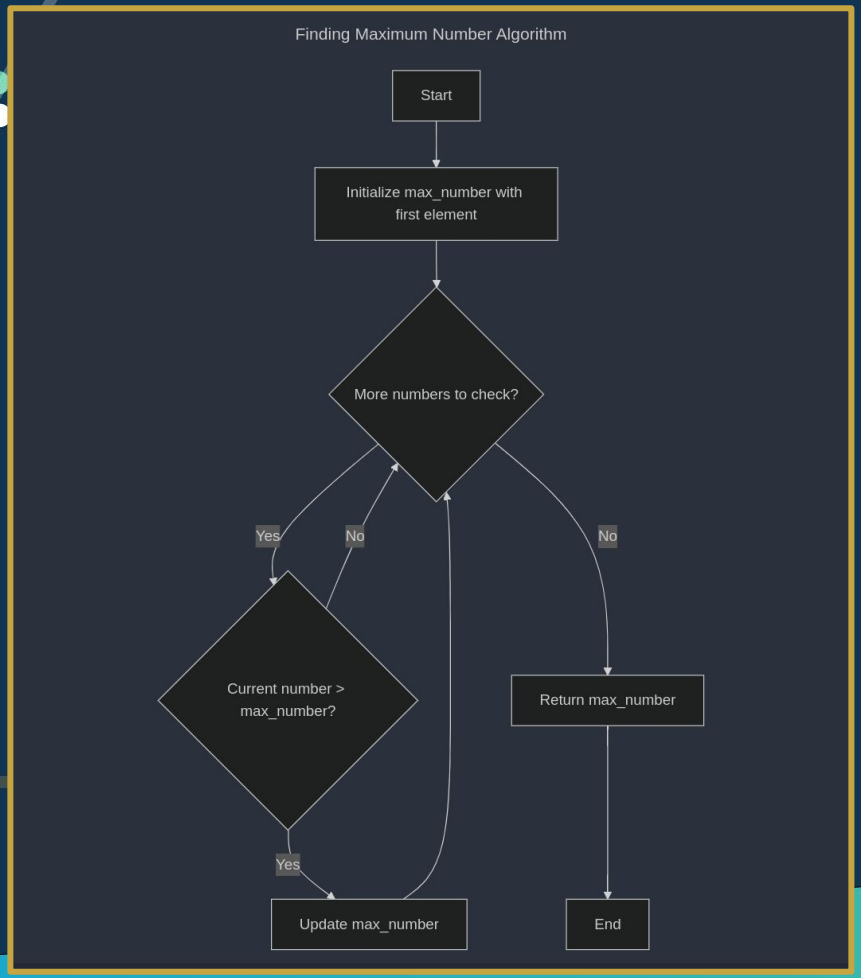
Fundamentals and Characteristics

HyperionDev

Characteristics of Good Algorithms

The Three Pillars:

- Correctness:
  - Produces the correct output for all valid inputs
- Efficiency:
  - Optimizes resource use (time and memory)
- Readability:
  - Clear, understandable, and maintainable code



Example: Finding the Maximum Number

HyperionDev

Example: Finding the Maximum Number

```python
def find_maximum(numbers):
    if not numbers:
        return None
    max_number = numbers[0]
    for number in numbers:
        if number > max_number:
            max_number = number
    return max_number
```

HyperionDev

Algorithm Analysis

- Using find_maximum as an example:
  - Correctness:
    - Works for all valid cases
  - Efficiency:
    - Checks each number once
  - Readability:
    - Clear variable names and logic

Common Problem Solving Methodologies

- **Divide-and-Conquer**
- **Brute Force**
- **Greedy Algorithms**
    - **TBC:** When we come back...

BREAK

HyperionDev

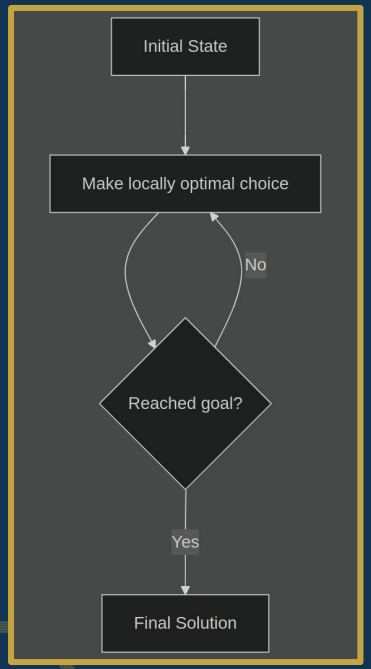
Problem Solving Methods

HyperionDev

Brute Force Method

- **Definition:**
  - Try all possible solutions
- **Advantages:**
  - Simple, guaranteed solution
- **Disadvantages:**
  - Inefficient for large data
- **Best Used For:**
  - Small datasets
  - Validating other algorithms
  - When no optimized solution exists

Greedy Algorithms

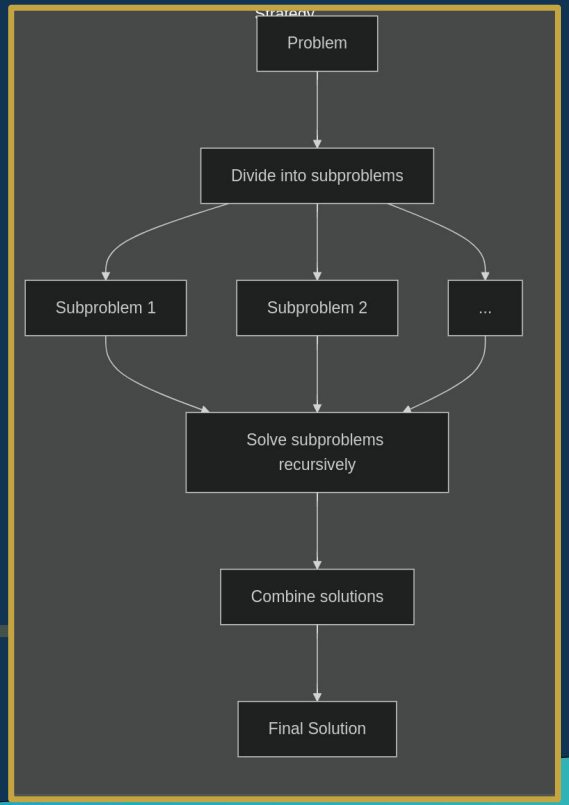


1. **Make the locally optimal choice at each step**
2. **Assume it will lead to the globally optimal solution**

- **Examples:**
  - Coin Change Problem
  - Huffman Coding
  - Activity Selection

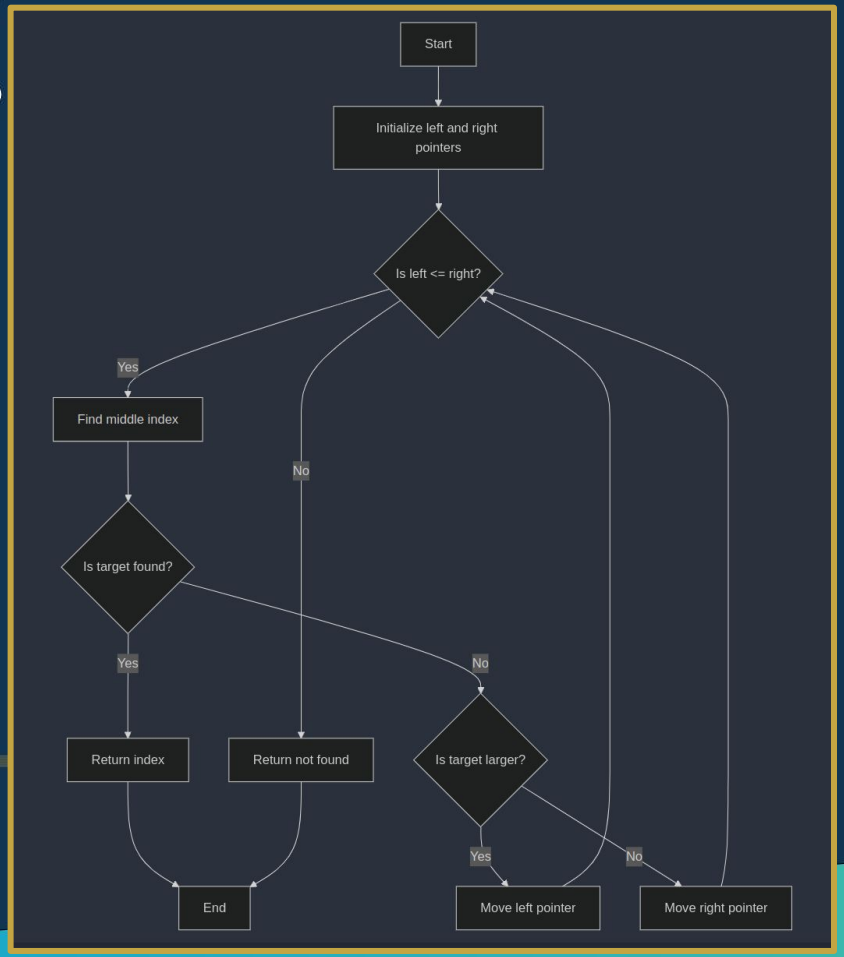
HyperionDev

Divide and Conquer



1. Break the problem into smaller subproblems
2. Solve subproblems recursively
3. Combine results for the final solution

HyperionDev



Binary Search Example

HyperionDev

Binary Search Example

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

HyperionDev

Real-World Application: Route Planning

- **Problem**: Find the shortest path between two points
- **Solutions**:
  - **Brute Force**: Try all routes
  - **Greedy**: Always move toward the destination
  - **Divide-and-Conquer**: Break down the journey into smaller steps

Putting it into Practice

Common Pitfalls in Algorithm Design

- Ignoring **edge cases**
- **Premature** optimization
- **Overcomplicating** the solution
- Failing to **validate inputs**
- Poor **documentation**

Best Practices

- Start with a **simple solution**
- **Test** with various **inputs**
- **Document** your code
- Consider **edge cases**
- **Optimize** only when necessary

Performance Considerations

- When designing algorithms, consider:
  - Time Complexity
  - Space Complexity
  - Input Size
  - Hardware Limitations
  - User Requirements

Steps to Success

- **Understand the Problem**
  - Read the problem carefully.
  - Identify inputs, outputs, constraints, and edge cases.
  - Ask clarifying questions if needed.
- **Plan Before You Code**
  - Use pseudocode or flowcharts to outline the solution.
  - Determine if the problem allows for brute-force solutions or needs optimization.

Steps to Success

- **Implement a Solution**
  - Code step-by-step, following the plan.
  - Test using example cases.
- **Optimise**
  - Identify inefficiencies.
  - Use better data structures or algorithmic paradigms to improve performance.

Summary

- Key Takeaways:
  - Algorithms are step-by-step solutions to problems
  - Good algorithms are correct, efficient, and readable
  - Practice different problem-solving methodologies
- Next Steps:
  - Continue practicing
  - Explore real-world applications

Additional Resources

- Python Documentation:
  - https://python.org
- Algorithm Visualizer:
  - https://visualgo.net
- Practice Problems:
  - https://leetcode.com

Which of these methodologies works by breaking the problem into smaller parts?

A. Brute Force
B. Divide-and-Conquer
C. Greedy Algorithm
D. Randomised Approach

Brute force algorithms are always the most efficient.

A. True
B. False

HyperionDev

Q & A SECTION

Please use this time to ask any questions relating to the topic, should you have any.


Thank you
for attending
HyperionDev