1.3 Algorithm design principles: modularity, abstraction

Modularity and abstraction are fundamental algorithm and software design principles used to manage complexity and improve the maintainability, reusability, and clarity of a system.

Modularity

Modularity is the principle of dividing a complex problem or system into smaller, independent, and self-contained parts called modules. Each module is designed to handle a specific functionality or concern.

- **Decomposition**: The process of breaking down a large system into smaller, manageable subproblems.
- **Loose Coupling**: Modules should have minimal interconnections or dependencies on each other. Changes within one module should have little to no impact on others.
- **High Cohesion**: The elements within a single module should have a unified purpose and work together to achieve a common goal.
- **Benefits**: Modularity simplifies development, allows for parallel work among different teams, makes testing and debugging easier, and promotes code reuse.

Example (Without Modularity)

```
arr = [1, 2, 3, 4]
total = 0
for i in arr:
    total += i
print(total)
```

Example (With Modularity)

```
def calculate_sum(arr):

    total = 0

    for i in arr:

        total += i

    return total


data = [1, 2, 3, 4]

result = calculate_sum(data)

print(result)
```

Advantages of Modularity

- Easy to **debug and test**
- Improves **code readability**
- Enables **code reuse**
- Simplifies **maintenance**
- Supports **team development**

Real-World Example

- Banking system:
    - Login module
    - Transaction module
    - Balance check module

Abstraction

Abstraction is the principle of hiding complex implementation details and showing only the essential features or relevant information to the user or other parts of the system. It focuses on *what* a component does rather than *how* it does it.

- **Information Hiding**: Protecting the internal state and implementation logic of an object or module from the outside world.
- **Interfaces**: Abstraction is often achieved through well-defined interfaces (like function signatures or abstract data types) that specify the public functionality, acting as a contract for interaction.
- **Levels of Abstraction**: A solution can be viewed at different levels, starting with a high-level, broad description and progressively adding detail at lower levels.
- **Benefits**: Abstraction simplifies the mental model required to use a component, manages complexity by disregarding irrelevant details, and allows the underlying implementation to be modified without affecting client code.

Example of Abstraction

```
def sort_array(arr):
    return sorted(arr)
```

- User knows:
    - Function sorts the array
- User does NOT need to know:
    - Sorting algorithm used internally

Example (Abstraction Using Function Interface)

```
def area_of_circle(radius):
    return 3.14 * radius * radius
```

- Internal formula is hidden
- User only calls the function

Advantages of Abstraction

- Reduces **complexity**
- Improves **security**
- Enhances **flexibility**
- Makes algorithms **easy to understand**

Difference Between Modularity and Abstraction

Modularity	Abstraction
Divides algorithm into parts	Hides internal details
Focuses on structure	Focuses on interface
Uses functions/modules	Uses abstract functions/classes
Improves maintainability	Improves usability