**Week 1: Data Structures Course Material**

---

Authors: Refat Othman and Diaeddin Rimawi

---

# Lecture 1: Recursion and Algorithm Analysis

## Part 1: Revision on Recursion

### What is Recursion?

- A function that calls itself (directly or indirectly).
- Useful for solving problems that can be broken into similar subproblems.

### Key Characteristics:

- Has one or more **base cases**.
- Other cases must call the function recursively with reduced input.

### Example 1: Recursive Multiplication Using Addition

This function simulates multiplication by using repeated addition. It multiplies two positive integers `a` and `b` by adding `a` to itself `b` times.

### Mathematical representation:

- Base case: $\text{multiply}(a, 1) = a$
- Recursive case: $\text{multiply}(a, b) = a + \text{multiply}(a, b - 1)$

```
def multiply(a, b):
    if b == 1:
        return a
    return a + multiply(a, b - 1)
```

### Example 2: Factorial

- Mathematical definition:
  - 0! = 1
  - n! = n * (n - 1)!

### Recursive implementation:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

**Iterative implementation:**

```
def factorial_iterative(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
```

**Example 3: Fibonacci**

This function computes the Fibonacci sequence, where each number is the sum of the two preceding ones. It is used to model natural phenomena like rabbit populations and tree branching.

**Mathematical representation:**

- Base cases: $\text{fib}(0) = 1$, $\text{fib}(1) = 1$
- Recursive case: $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$

**Naive recursive implementation:**

```
def fibonacci(n):
    if n == 0 or n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

**Iterative implementation (efficient):**

```
def fibonacci_iterative(n):
    t1, t2 = 0, 1
    for _ in range(n):
        print(t1, end=" ")
        t1, t2 = t2, t1 + t2
```

## Part 2: Introduction to Algorithm Analysis

**Why Analyze Algorithms?**

- To compare efficiency independent of machine and compiler.
- Time and space complexity are two major metrics. Nowadays, memory is relatively abundant and inexpensive, so developers often prioritize optimizing time complexity (performance) over minimizing space usage.

**Types of Analysis:**

- **Worst-case**: guarantees upper bound
- **Best-case**: too optimistic
- **Average-case**: uses statistical assumptions

**How to Analyze?**

- Count the **basic operations** (e.g., comparisons, assignments).
- Express time as a function of input size `n`.

**Basic Operation Counting Example:**

```
sum = 0
for i in range(1, n + 1):
    sum += i
```

- Time complexity: O(n) since the loop runs n times

**Loop Rules Summary:**

**Simple loop example (O(n))**

```
import time

def simple_loop(n):
    start = time.time()
    total = 0
    for i in range(n):
        total += i
    end = time.time()
    print("Total:", total)
    print("Time taken:", end - start, "seconds")

simple_loop(1000000)
```

**Triangular nested loop example (O(n^2), non-uniform inner loop)**

This loop has the outer loop run `n` times and the inner loop runs `i` times for each `i` from `1` to `n`, so the total number of iterations is the sum of the first `n` integers: $\frac{n(n+1)}{2}$.

```
import time

def triangular_nested_loop(n):
    start = time.time()
    total = 0
    for i in range(1, n + 1):
        for j in range(1, i + 1):
            total += 1
```

```
    end = time.time()
    print("Total operations:", total)
    print("Time taken:", end - start, "seconds")

triangular_nested_loop(10000)
```

**Constant time summation (O(1))**

This uses the formula $\frac{n(n+1)}{2}$ to compute the sum in constant time.

```
import time

def constant_time_sum(n):
    start = time.time()
    total = n * (n + 1) // 2
    end = time.time()
    print("Total:", total)
    print("Time taken:", end - start, "seconds")

constant_time_sum(1000000)
```

**Nested loop example (O(n^2))**

```
import time

def nested_loop(n):
    start = time.time()
    count = 0
    for i in range(n):
        for j in range(n):
            count += 1
    end = time.time()
    print("Count:", count)
    print("Time taken:", end - start, "seconds")

nested_loop(1000)
```

**Consecutive statements example**

```
import time

def consecutive_statements(n):
    start = time.time()
    total = 0
    for i in range(n):
        total += i  # O(n)
    for j in range(n):
```

```
            total += j  # O(n)
        end = time.time()
        print("Total:", total)
        print("Time taken:", end - start, "seconds")

    consecutive_statements(1000000)
```

**Conditional example**

```
    import time

    def conditional_example(n):
        start = time.time()
        total = 0
        for i in range(n):
            if i % 2 == 0:
                total += i * 2
            else:
                total += i + 1
        end = time.time()
        print("Total:", total)
        print("Time taken:", end - start, "seconds")

    conditional_example(1000000)
```

---

# Lecture 2: Asymptotic Analysis

## Part 1: Motivation

- Measures algorithm growth rate for large input sizes.
- Ignores constants and lower-order terms.
- Allows developers to estimate performance before writing the actual code, aiding in design decisions.
- Helps in choosing the most suitable algorithm when working with large datasets or resource-constrained environments.

## Part 2: Big-O, Big-Omega, and Big-Theta

**Big-O Notation (Upper Bound)**

Definition:

> f(n) is in O(g(n)) if there are positive constants c and n0 such that: $0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$

Example:

- f(n) = 5n + 6
- Prove O(n): choose c=6, n0=1 → valid

**Big-Omega (Lower Bound)**

> f(n) is in $\Omega(g(n))$ if $f(n) \geq c \cdot g(n)$ for some $c > 0$, $n \geq n_0$

**Big-Theta (Tight Bound)**

> f(n) is in $\Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$

## Common Orders of Growth

- O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(n^3) < O(2^n)

## Part 3: Analyzing Examples

This section shows how to analyze the time complexity of algorithms by counting how many steps (basic operations) they execute as the input size `n` grows. For each example below, we provide a detailed explanation of how the time complexity is computed.

**Example 1: Constant Time - O(1)**

Accessing an element in an array is done in constant time regardless of the size of the input.

```
def get_first_element(arr):
    return arr[0]  # Always takes one step
```

**Explanation**: This function executes a single operation (array access), independent of `n`. So, time complexity is O(1).

**Example 2: Logarithmic Time - O(log n)**

Binary search splits the problem in half each time, reducing the size logarithmically.

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

**Explanation**: Each call halves the problem size. Let's say we start with an array of size `n`. After one step, the size becomes `n/2`, then `n/4`, then `n/8`, and so on. After `k` steps, the array size becomes `n / 2^k`. We stop when this value equals 1. That gives us the equation `n / 2^k = 1`, which simplifies to `2^k = n`. Taking logarithms on both sides gives $k = \log_2(n)$. Therefore, the number of steps required is proportional to log n, hence the time complexity is O(log n).

**Example 3: Linear Time - O(n)**

Linear search checks every item in the array until the target is found.

```
def linear_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
```

**Explanation**: In the worst case, we iterate over the entire array of size `n`, performing one comparison per element. Time complexity is O(n).

**Example 4: Linearithmic Time - O(n log n)**

A common example of O(n log n) time complexity occurs when we iterate over a list and perform a logarithmic operation for each element.

```
import math

def linearithmic_example(arr):
    total = 0
    for x in arr:
        i = len(arr)
        while i > 1:
            i = i // 2
            total += 1
    return total
```

**Explanation**: The outer loop runs `n` times. The inner loop executes $\log_2(n)$ times per iteration. Hence, total operations = n * log n → O(n log n).

**Example 5: Quadratic Time - $O(n^2)$**

Nested loops over the same input size.

```
def count_pairs(arr):
    count = 0
    for i in range(len(arr)):
        for j in range(len(arr)):
```

```
            count += 1
    return count
```

**Explanation**: For each of the `n` iterations of the outer loop, the inner loop runs `n` times, leading to `n * n = n²` operations. Time complexity is $O(n^2)$.

**Example 6: Cubic Time - $O(n^3)$**

Triple nested loop example.

```
def count_triplets(n):
    count = 0
    for i in range(n):
        for j in range(n):
            for k in range(n):
                count += 1
    return count
```

**Explanation**: Three nested loops each run `n` times → total iterations = `n * n * n = n³`. Time complexity is $O(n^3)$.

**Example 7: Exponential Time - $O(2^n)$**

Recursive Fibonacci is an example of exponential time.

```
def fib(n):
    if n <= 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

**Explanation**: Each call results in two more calls. The call tree has a branching factor of 2 and depth of `n`, leading to roughly $2^n$ calls. Time complexity is $O(2^n)$.

### Tips:

- Drop constants: O(2n) → O(n)
- Keep dominant term: O(n^2 + n) → O(n^2)

---

## Self-Check Exercises

1. Write and trace `f(x) = f(x-1) + 2` with `f(0) = 0`
2. Compare recursive vs iterative Fibonacci runtime
3. Prove that 2n^2 + 1 ∈ O(n^2)

---

Let me know what you'd like to add, refine, or illustrate with diagrams or code snippets!