# CS 2500: Algorithms

## Lecture 4: Introduction to Recursion

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

August 29, 2024

# Logistics

- Weekly assignment 1 released (due Sep 3 at 11.59 PM).
- Quick assignment 1 released (due Aug 29 at 11.59 PM)
- **No quick assignment today!**
- Come to recitations and office hours to get help on assignments.

# Introduction to Recursion

**What is recursion?**

- Recursion is a technique where a function calls itself to solve a smaller instance of the same problem.
- It's particularly powerful for problems that can be broken down into simpler, identical subproblems.
- Common examples include factorial computation, Fibonacci sequence, and algorithms like GCD.

**Key Idea:** Reduce a problem to one or more smaller instances of the same problem, and solve it using the same approach.

# Real-world Example: RSA Encryption

**RSA Encryption and Decryption**

- RSA is a widely used public-key cryptosystem for secure data transmission.
- Encryption: Given a message $M$, compute the ciphertext $C$ as:

$$C = M^e \mod n$$

- Decryption: To recover $M$, compute:

$$M = C^d \mod n$$

- Here, $e$ is the public key exponent, and $d$ is the private key exponent.

**Last class: How well can we implement this?**

# Real-world Example: RSA Key Generation

**RSA Key Generation and GCD**

- In RSA, choosing a public key exponent $e$ requires it to be coprime with $\phi(n)$, where
  - $n = p \times q$
  - $\phi(n) = (p-1) \times (q-1)$.
  - $p$ and $q$ are large prime numbers.
- To ensure $e$ and $\phi(n)$ are coprime, we need to compute the greatest common divisor (GCD) of $e$ and $\phi(n)$:

$$\gcd(e, \phi(n))$$

- If $\gcd(e, \phi(n)) = 1$, $e$ is a valid public key exponent.
- This process can be implemented efficiently using the Euclidean algorithm.

**Question: How can we compute the GCD using recursion?**

# Iterative GCD Algorithm

**Algorithm IterativeGCD(a, b)**

```
1: while b ≠ 0 do
2:     r ← a mod b
3:     a ← b
4:     b ← r
5: end while
6: return a
```

**Example:**

- Start with $a = 252$ and $b = 105$.
- Iteration 1: $r = 252 \mod 105 = 42$, update $a = 105$, $b = 42$.
- Iteration 2: $r = 105 \mod 42 = 21$, update $a = 42$, $b = 21$.
- Iteration 3: $r = 42 \mod 21 = 0$, update $a = 21$, $b = 0$.
- Result: GCD is 21.

# Identifying Base Case and Recursive Step

- **Base Case**: The loop terminates when $b = 0$. This becomes our base case in the recursive version.
- **Recursive Step**: In each iteration, $a$ and $b$ are updated. In the recursive version, this update becomes the recursive call.
- **Conversion Strategy**: Replace the loop with a recursive function that reduces the problem size in each call.

# Converting Iteration to Recursion

**Iterative to Recursive Conversion:**

- **Iterative Loop**: Continues until $b = 0$.
- **Recursive Call**: Each iteration corresponds to a recursive call where $a$ and $b$ are updated.
- **Termination**: The loop's termination condition $b = 0$ becomes the base case for recursion.

# Recursive GCD Algorithm

**Algorithm RecursiveGCD(a, b)**

```
1: if b = 0 then                                   ▷ Base Case
2:     return a
3: else                                            ▷ Recursive Step
4:     return RecursiveGCD(b, a mod b)
5: end if
```

**Example:**

- Start with $\gcd(252, 105)$.
- Compute $252 \mod 105 = 42$, so $\gcd(252, 105) = \gcd(105, 42)$.
- Compute $105 \mod 42 = 21$, so $\gcd(105, 42) = \gcd(42, 21)$.
- Compute $42 \mod 21 = 0$, so $\gcd(42, 21) = 21$.
- Result: GCD is 21.

# Side-by-Side Comparison

## Iterative GCD

**Algorithm** IterativeGCD(a, b)

```
1: while b ≠ 0 do
2:     r ← a mod b
3:     a ← b
4:     b ← r
5: end while
6: return a
```

## Recursive GCD

**Algorithm** RecursiveGCD(a, b)

```
1: if b = 0 then ▷ Base Case
2:     return a
3: else ▷ Recursive Step
4:     return RecursiveGCD(b, a mod b)
5: end if
```

# Real-world Example: RSA Encryption

**RSA Encryption and Decryption**

- RSA is a widely used public-key cryptosystem for secure data transmission.
- Encryption: Given a message $M$, compute the ciphertext $C$ as:

$$C = M^e \mod n$$

- Decryption: To recover $M$, compute:

$$M = C^d \mod n$$

- Here, $e$ is the public key exponent, and $d$ is the private key exponent.

**Last class: Iterative algorithm for computing $x^n$. Complexity: $\Theta(\log n)$**

# Recursive Algorithm for $b^n \mod m$

- The recursive algorithm for computing $b^n \mod m$ can be based on the relationship:

$$b^n \mod m = (b \times (b^{n-1} \mod m)) \mod m$$

- Initial condition: $b^0 \mod m = 1$.

# How the Algorithm Works

**Base Case:**

- The base case occurs when $n = 0$.
- By definition, any number raised to the power of 0 is 1 (i.e., $b^0 = 1$).
- So, $b^0 \mod m = 1$.
- This provides a stopping condition for the recursion.

# How the Algorithm Works

**Recursive Case:**

- For $n > 0$, the problem can be reduced by recognizing that $b^n$ can be expressed as $b \times b^{n-1}$.
- Instead of computing $b^n$ directly, compute $b^{n-1} \mod m$ recursively.
- Then multiply the result by $b$ and take the result modulo $m$ again:

$$b^n \mod m = (b \times (b^{n-1} \mod m)) \mod m$$

- This breaks down the problem into smaller multiplications, each time reducing the exponent by 1.

# Recursive Algorithm

**Algorithm** RecursiveModExp(b, n, m)

1: **if** $n = 0$ **then** ▷ Base Case
2: **return** 1
3: **else** ▷ Recursive Step
4: $y \leftarrow \text{RecursiveModExp}(b, n-1, m)$
5: **return** $(b \times y) \mod m$
6: **end if**

# Why This Works

**Modular Arithmetic Property:**

- The equation

$$(x \times y) \mod m = [(x \mod m) \times (y \mod m)] \mod m$$

  ensures that intermediate results remain within manageable bounds.
- This prevents overflow and makes the algorithm efficient even for large exponents.

# Why This Works (continued)

**Recursive Decomposition:**

- By recursively reducing the exponent $n$ by 1 at each step, the algorithm gradually breaks down the problem into smaller, easily solvable pieces.
- This approach mirrors how mathematical induction works: solve the problem for a base case, then assume it works for a smaller problem, and show that it works for the next step.

# Efficiency

- Although this recursive approach is straightforward, it is not the most efficient way to compute large powers modulo $m$.
- The time complexity is $\Theta(n)$ due to the $n$ recursive calls.
- However, it clearly illustrates the concept of breaking down the problem using recursion.

**Question: Can we do better than $\Theta(n)$?**

# Efficient Recursive Algorithm for $b^n \mod m$

We can devise a much more efficient recursive algorithm based on the observation that:

- when $n$ is even:

$$b^n \mod m = \left(b^{n/2} \mod m\right)^2 \mod m$$

- when $n$ is odd:

$$b^n \mod m = \left[\left(b^{\lfloor n/2 \rfloor} \mod m\right)^2 \mod m \cdot b\right] \mod m$$

**Constraints:**

- $b$, $n$, and $m$ are integers
- $m \geq 2$, $n \geq 0$, $1 \leq b < m$

# Case 1: $n$ is Even

- Express $n$ as $n = 2k$ for some integer $k$.
- Then:

$$b^n = b^{2k} = \left(b^k\right)^2$$

- Apply modulo $m$ on both sides:

$$b^n \mod m = \left(\left(b^k\right)^2\right) \mod m$$

# Case 1: $n$ is Even

- Use the property of modular arithmetic:

$$(x \times y) \mod m = [(x \mod m) \times (y \mod m)] \mod m$$

- Applying it to the equation:

$$b^n \mod m = \left[\left(b^k \mod m\right) \times \left(b^k \mod m\right)\right] \mod m$$

- Simplifying gives:

$$b^n \mod m = \left(b^{n/2} \mod m\right)^2 \mod m$$

- Therefore, the equality holds for even $n$.

# Case 2: $n$ is Odd

1. **Express $n$ as $n = 2k + 1$:**
   - Since $n$ is odd, we can express it as $n = 2k + 1$, where $k = \left\lfloor \frac{n}{2} \right\rfloor$.
   - This gives us:

$$b^n = b^{2k+1} = b \times b^{2k}$$

# Case 2: $n$ is Odd

**2. Apply the Property of Exponents:**

- We know that:

$$b^{2k} = \left(b^k\right)^2$$

- So, substitute this into the previous expression:

$$b^n = b \times \left(b^k\right)^2$$

# Case 2: $n$ is Odd

**3. Apply Modulo $m$ to Both Sides:**

- Now, take modulo $m$ on both sides:

$$b^n \mod m = \left[ b \times \left( b^k \right)^2 \right] \mod m$$

- According to the properties of modular arithmetic, you can apply the modulo to each term:

$$b^n \mod m = \left[ \left( \left( b^k \right)^2 \mod m \right) \times (b \mod m) \right] \mod m$$

# Case 2: $n$ is Odd

**4. Simplify the Expression:**

- Recognize that $\left(b^k\right)^2 \mod m$ can be expressed as $\left(b^k \mod m\right)^2 \mod m$.
- Also, if $b < m$, then $b \mod m = b$. Therefore, the expression simplifies to:

$$b^n \mod m = \left[\left(b^{\lfloor \frac{n}{2} \rfloor} \mod m\right)^2 \mod m \times b\right] \mod m$$

# Time Complexity Analysis

The recursive algorithm for computing $b^n \mod m$ works as follows:

- **Base Case:** If $n = 0$, return 1.
- **Recursive Case:**
  - If $n$ is even:

$$b^n \mod m = \left(b^{n/2} \mod m\right)^2 \mod m$$

  - If $n$ is odd:

$$b^n \mod m = \left[\left(b^{\lfloor n/2 \rfloor} \mod m\right)^2 \mod m \times b\right] \mod m$$

# Time Complexity Analysis

- The algorithm reduces the exponent $n$ by half at each recursive step.
- The depth of the recursion tree is approximately $\log_2(n)$.
- Key operations at each step:
  - Squaring the result of a recursive call.
  - Multiplication and modulo operation.

# Time Complexity Analysis

- **Number of Recursive Calls:**
  - The problem size reduces from $n$ to $n/2$ at each step.
  - Recursion depth is $\log_2(n)$. **[Homework: Why?]**
- **Work Done at Each Step:**
  - Constant amount of work ($O(1)$) involving multiplication and modulo operation.

# Time Complexity Analysis

- **Total Work:**
  - Recursion depth: $\log_2(n)$.
  - Work at each level: $O(1)$.
- **Overall Time Complexity:**

$$f(n) = \Theta(\log n)$$

- The algorithm is efficient with logarithmic time complexity, ideal for cryptographic applications.

# Efficient Recursive Algorithm for $b^n \mod m$

**Algorithm** mpower(b, n, m)

1: **if** $n = 0$ **then**
2: **return** 1
3: **else if** $n$ **is even then**
4: **return** $\text{mpower}(b, n/2, m)^2 \mod m$
5: **else**
6: **return** $\left(\text{mpower}(b, \lfloor n/2 \rfloor, m)^2 \mod m \times b\right) \mod m$
7: **end if**

# Recursive Linear Search

The goal is to search for the first occurrence of $x$ in the sequence $a_1, a_2, \ldots, a_n$ using a recursive approach.

- At the $i$th step, compare $x$ with $a_i$.
- If $x = a_i$, return the index $i$.
- Otherwise, reduce the search to the sequence $a_{i+1}, \ldots, a_j$.
- The algorithm returns 0 if $x$ is not found after all elements are examined.

# Recursive Linear Search

**Algorithm search(a,i,j,x)**

```
1: if a[i] = x then
2:     return i
3: else if i = j then
4:     return 0
5: else
6:     return search(i + 1, j, x)
7: end if
```

# Recursive Binary Search

The goal is to locate $x$ in the sequence $a_1, a_2, \ldots, a_n$ of integers in increasing order using a recursive binary search approach.

- Compare $x$ with the middle term $a_{\lfloor (i+j)/2 \rfloor}$.
- If $x = a_m$, return the location $m$.
- If $x < a_m$, search the first half of the sequence.
- If $x > a_m$, search the second half of the sequence.
- Return 0 if $x$ is not found in the sequence.

# Recursive Binary Search

**Algorithm RecBinSearch(a,i,j,x)**

1: $m \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
2: **if** $x = a[m]$ **then**
3: **return** $m$
4: **else if** $x < a[m]$ **and** $i < m$ **then**
5: **return** RecBinSearch$(a, i, m-1, x)$
6: **else if** $x > a[m]$ **and** $j > m$ **then**
7: **return** RecBinSearch$(a, m+1, j, x)$
8: **else**
9: **return** 0
10: **end if**

# Iterative Factorial

**Algorithm IterativeFactorial(n)**

1: $result \leftarrow 1$
2: **for** $i \leftarrow 2$ **to** $n$ **do**
3: $\quad result \leftarrow result \times i$
4: **end for**
5: **return** *result*

# Recursive Factorial

The factorial of a non-negative integer $n$ is defined recursively as:

- Base Case: $0! = 1$
- Recursive Case: $n! = n \times (n-1)!$ for $n > 0$

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$

# Recursive Factorial

**Algorithm** RecursiveFactorial(n)

```
1: if n = 0 or n = 1 then                          ▷ Base Case
2:     return 1
3: else                                            ▷ Recursive Step
4:     return n × RecursiveFactorial(n − 1)
5: end if
```

# Tracing Recursive Calls: Factorial Example

- Recursion involves a function calling itself to solve smaller subproblems.
- Tracing recursive calls helps us understand the flow of execution in a recursive algorithm.
- We'll walk through the recursive calls for the factorial function as an example.

# Tracing the Recursive Calls for $n = 3$

Let's trace the recursive calls when calculating factorial(3):

- Call: factorial(3)
- Call: factorial(2)
- Call: factorial(1)
- Call: factorial(0)
- Return: 1 (from factorial(0))
- Return: $1 \times 1 = 1$ (from factorial(1))
- Return: $2 \times 1 = 2$ (from factorial(2))
- Return: $3 \times 2 = 6$ (from factorial(3))

# Visualizing the Call Stack for factorial(3)

- The recursive calls build up the call stack as follows:
  1. factorial(3)
  2. factorial(2)
  3. factorial(1)
  4. factorial(0)
- As the base case is reached, the stack unwinds, returning values:
  1. factorial(0) $= 1$
  2. factorial(1) $= 1 \times 1 = 1$
  3. factorial(2) $= 2 \times 1 = 2$
  4. factorial(3) $= 3 \times 2 = 6$

# Common Pitfalls

- Writing recursive functions can be challenging due to the need for careful structuring.
- Common mistakes can lead to issues such as infinite recursion, incorrect results, or stack overflow.
- Understanding these pitfalls helps in writing correct and efficient recursive functions.

# Pitfall 1: Missing or Incorrect Base Case

- **Base Case**: The base case is the condition under which the recursion stops.
- **Common Mistake**: Forgetting to include a base case or having an incorrect base case.
- **Consequence**: Leads to infinite recursion, as the function keeps calling itself indefinitely.
- **Example**:
  - **Incorrect**:

$$\text{factorial}(n) = n \times \text{factorial}(n-1) \quad \text{(No base case)}$$

  - **Correct**:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$

# Pitfall 2: Incorrect Recursive Case

- **Recursive Case**: The recursive case defines how the problem is broken down into smaller subproblems.
- **Common Mistake**: Incorrectly structuring the recursive case or using the wrong recursive logic.
- **Consequence**: Leads to incorrect results, even if the base case is correct.
- **Example**:
  - **Task**: Find the maximum element in a list.
  - **Incorrect Recursive Case**:

$$\max(L) = \max(\max(L[1:]), L[0])$$

    **Issue**: The comparison should be between the first element and the maximum of the rest, but the logic incorrectly assumes it is between the maximum of all elements and the first element, leading to incorrect results.
  - **Correct Recursive Case**:

$$\max(L) = \begin{cases} L[0] & \text{if } \text{len}(L) = 1 \\ \max(L[0], \max(L[1:])) & \text{if } \text{len}(L) > 1 \end{cases}$$

# Pitfall 3: Infinite Recursion Due to Improper Decrement/Increment

- **Improper Decrement/Increment**: Incorrectly modifying the argument in the recursive call, leading to no progress towards the base case.
- **Common Mistake**: Failing to properly decrement or increment, causing the function to never reach the base case.
- **Consequence**: Infinite recursion and potential stack overflow.
- **Example**:
  - **Incorrect**:

$$\text{countdown}(n) = \text{countdown}(n) \quad \text{(No decrement)}$$

  - **Correct**:

$$\text{countdown}(n) = \begin{cases} \text{Done} & \text{if } n = 0 \\ \text{countdown}(n-1) & \text{if } n > 0 \end{cases}$$

# Pitfall 4: Overlapping Subproblems Leading to Inefficiency

- **Overlapping Subproblems**: Solving the same subproblem multiple times in the recursive calls.
- **Common Mistake**: Not storing the results of subproblems (e.g., using memoization).
- **Consequence**: Leads to exponential time complexity and poor performance.
- **Example**:
  - **Fibonacci**: Without memoization, the Fibonacci sequence recalculates the same values repeatedly.
  - **Improvement**: Use a dictionary to store results and avoid redundant calculations.

# Pitfall 5: Lack of Understanding of Recursion Depth

- **Recursion Depth**: Understanding the limitations of recursion depth in a given environment.
- **Common Mistake**: Writing deep recursive functions without considering the maximum recursion depth.
- **Consequence**: Stack overflow or maximum recursion depth exceeded errors.
- **Solution**:
  - Optimize with iterative solutions where possible.
  - Increase recursion depth limit in environments where necessary.

# Common Pitfalls

**Conclusion:**

- Avoiding these common pitfalls requires careful planning and a solid understanding of recursion.
- Always ensure base cases are correctly defined and reachable.
- Be mindful of the efficiency and limitations of recursive solutions.