Last Lecture

- Data abstraction
  - Information hiding
  - Data encapsulation
  - how it applies to Abstract data type (ADT)
- Two examples: Indexed set and Complex set
  - the similarities and difference between the two w.r.t. data abstraction
    - how data is protected
    - how implementation is abstracted away from users

What does the following code segment do?

```
public class Fac {
  public static int fac(int n) {
    if (n <= 1) return 1;

    int product = n * fac(n - 1);
    return product;
  }
  public static void main(String[] arguments) {
    try (Scanner scanner = new Scanner(System.in)) {
      int number;
      do {
        System.out.println("Enter a number");
        number = scanner.nextInt();
        System.out.println(fac(number));
      } while (number > 0);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

CPT108 Data Structures and Algorithms

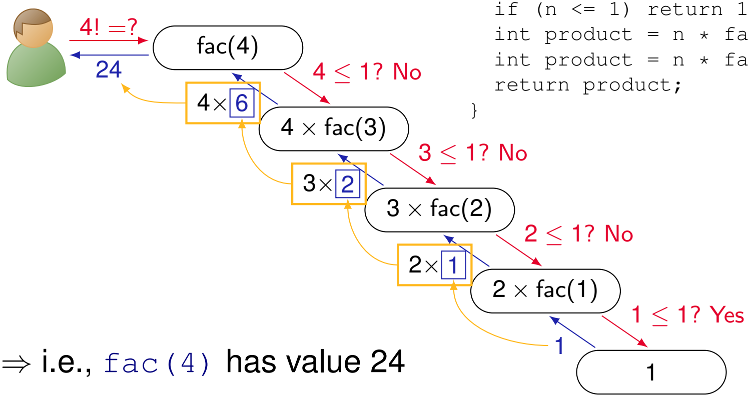
Lecture 8

Recursion

What does the following code segment do? (cont.)

Assuming the number typed is 4.

```
public static int fac(int n) {
  if (n <= 1) return 1;
  int product = n * fac(n - 1);
  int product = n * fac(n - 1);
  return product;
}
```



⇒ i.e., `fac(4)` has value 24

Recursion

- A recursive function is a function that calls itself directly or indirectly
- For example, we are using the following relation:

$$f(n) = \begin{cases} \boxed{n \times f(n-1) \quad \text{if } n > 1} \\ \boxed{1 \qquad\qquad\qquad \text{if } n = 1} \end{cases}$$

Recursion step
$\Rightarrow$ *it takes a number as input, multiply it with that number minus 1 and repeat the same operation down until it reach the stopping case.*

to calculate the factorial of a number, which corresponds to:

```java
public static int fac(int n) {
  if (n <= 1) return 1;

  int product = n * fac(n - 1);
  return product;
}
```

Stopping case
$\Rightarrow$ *return 1 directly!*

Recursion

- A powerful technique in problem solving
  - it helps up to break down large complex problems into smaller ones
- Recursion requires:
  - **Stopping cases** (a.k.a. **exit conditions, base case**) – Simple cases which have straightforward solution.
    - Solve the problem directly.
  - **Recursive step** – The problem can be reproduced to identical but less complex subproblem(s).
    - Reduce the problem to identical but smaller problem(s).
  - **Eventuality** – The problem can eventually be reduced to the stopping cases through the recursive step, i.e., a check for termination is need.

Some Examples

Reverse a character string

- Base case:
  - If `length(string)==1` then `result=string`
- Recursive step:
  - Let `first` be the string's first character, `substring` be the string with `first` removed
  - `result=append( Reverse(substring), first)`

```
public String reverse(String str) {
  if (str.length() == 1) return str;
  return reverse(str.substring(1)) + str.charAt(0);
}
```

Some Examples (cont.)

Find root using bisection method: `FindRoot(lower, upper)`

- Base case:
  - If `|lower−upper|`$<\epsilon$ then `root=(upper+lower)/2`
- Recursive step:
  - `midpt = (upper+lower)/2`
  - If `f(midpt)` and `f(lower)` are of different sign then `root=FindRoot(lower,midpt)`
  - If `f(midpt)` and `f(upper)` are of different sign then `root=FindRoot(midpt,upper)`

```
public double findRoot(double lower, double upper) {
  if (Math.abs(upper - lower) < threshold) return (upper + lower) / 2;

  double midpt = (upper + lower) / 2;
  if (differentSign(f(midpt), f(lower))) {
    return findRoot(lower, midpt);
  } else if (differentSign(f(midpt), f(upper))) {
    return findRoot(midpt, upper);
  }
}
```

Exercises

Write a recursion function that calculate the value of x^n.

Exercises (cont.)

Write a recursion function that counts the number of zero digits in a non-negative integer

Exercises (cont.)

Write a recursion function to determine how many factors m are part of n. For example, if $n = 48$ and $m = 4$, then the result is 2 $(48 = 4 \times 4 \times 2)$.

Proof of Correctness

By mathematical induction on some measure of problem size.

- The *stopping case* is correct
  - The base instance P_1 is `true`
- The *recursion step* is correct
  - If P_k is `true`, then P_{k+1} is `true`

Induction problem size measure:

- Compute factorial: $O(n)$
- Reverse string: $O($`length(string)`$)$
- Bisection method: $O(\log n)$
- …

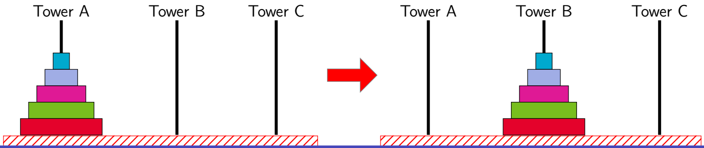
A not so trivial example: Towers of Hanoi

There are three towers (A, B, C) ad five disks ($D1$, $D2$, $D3$, $D4$, $D5$). $D1$ is the smallest disk, $D2$ is the second smallest, and so forth. $D5$ is the largest disk.

Originally all five disks are stacked in tower A according to their size (i.e., $D1$ on top while $D5$ at the bottom).

Move the disks from Tower A to Tower B, subject to:

- Only move one disk at a time and this disk must be the top disk of a tower
- A larger disk can never be placed on top of a smaller one



Towers of Hanoi (cont.)

Problem analysis and recursive algorithm:

- Move the N smallest disks for Tower X to Tower Y
- There always has an alternate Tower Z for use
- Only smaller disks can put on top of a disk

Simple problem instance (stopping case):

- $N = 1$
- Solution: Move the disk form Tower X to Tower Y
- Correct because the disk is the smallest

Problem decomposition (recursion step):

1. Move the top smallest $N - 1$ disks from Tower X to Tower Z
2. Move the N^{th} disk from Tower X to Tower Y
3. Move the top smallest $N - 1$ disks from Tower Z to Tower Y

Towers of Hanoi (cont.)

Implementation

```java
public class SimpleTowerOfHanoi {

  private static String[] TOWERS = new String[] { "A", "B", "C" };

  public void moveDisks(int nDisk, String from, String to, String intermediate) {
    if (nDisk < 2) { // stopping case
      System.out.printf("Move Disk \"%s\" from Tower %s to Tower %s\n", nDisk, from, to);
    } else { // recursion step
      moveDisks(nDisk - 1, from, intermediate, to);
      System.out.printf("Move Disk \"%s\" from Tower %s to Tower %s\n", nDisk, from, to);
      moveDisks(nDisk - 1, intermediate, to, from);
    }
  }

  // initial call to recursive function
  public void run(int nDisks) {
    moveDisks(nDisks, TOWERS[0], TOWERS[1], TOWERS[2]);
  }

  public static void main(String[] arguments) {
    SimpleTowerOfHanoi hanoi = new SimpleTowerOfHanoi();
    hanoi.run(3);
  }
```

Towers of Hanoi (cont.)

Simulation results

when $N = 2$:

```
-- -----------------------
-- Step  0: Initial setting
-- -----------------------
Tower A: Disk(2) Disk(1)
Tower B:
Tower C:
-- -----------------------------------------------
-- Step  1: Move "Disk(1)" from Tower A to Tower C
-- -----------------------------------------------
Move "Disk(1)" from Tower A to Tower C
Tower A: Disk(2)
Tower B:
Tower C: Disk(1)
-- -----------------------------------------------
-- Step  2: Move "Disk(2)" from Tower A to Tower B
-- -----------------------------------------------
Move "Disk(2)" from Tower A to Tower B
Tower A:
Tower B: Disk(2)
Tower C: Disk(1)
-- -----------------------------------------------
-- Step  3: Move "Disk(1)" from Tower C to Tower B
-- -----------------------------------------------
Move "Disk(1)" from Tower C to Tower B
Tower A:
Tower B: Disk(2) Disk(1)
Tower C:
```

Towers of Hanoi (cont.)

Simulation results

when $N = 3$:

```
-- -----------------------
-- Step  0: Initial setting
-- -----------------------
Tower A: Disk(3) Disk(2) Disk(1)
Tower B:
Tower C:
-- ----------------------------------------------
-- Step  1: Move "Disk(1)" from Tower A to Tower B
-- ----------------------------------------------
Move "Disk(1)" from Tower A to Tower B
Tower A: Disk(3) Disk(2)
Tower B: Disk(1)
Tower C:
-- ----------------------------------------------
-- Step  2: Move "Disk(2)" from Tower A to Tower C
-- ----------------------------------------------
Move "Disk(2)" from Tower A to Tower C
Tower A: Disk(3)
Tower B: Disk(1)
Tower C: Disk(2)
-- ----------------------------------------------
-- Step  3: Move "Disk(1)" from Tower B to Tower C
-- ----------------------------------------------
Move "Disk(1)" from Tower B to Tower C
Tower A: Disk(3)
Tower B:
Tower C: Disk(2) Disk(1)
```

```
-- ----------------------------------------------
-- Step  4: Move "Disk(3)" from Tower A to Tower B
-- ----------------------------------------------
Move "Disk(3)" from Tower A to Tower B
Tower A:
Tower B: Disk(3)
Tower C: Disk(2) Disk(1)
-- ----------------------------------------------
-- Step  5: Move "Disk(1)" from Tower C to Tower A
-- ----------------------------------------------
Move "Disk(1)" from Tower C to Tower A
Tower A: Disk(1)
Tower B: Disk(3)
Tower C: Disk(2)
-- ----------------------------------------------
-- Step  6: Move "Disk(2)" from Tower C to Tower B
-- ----------------------------------------------
Move "Disk(2)" from Tower C to Tower B
Tower A: Disk(1)
Tower B: Disk(3) Disk(2)
Tower C:
-- ----------------------------------------------
-- Step  7: Move "Disk(1)" from Tower A to Tower B
-- ----------------------------------------------
Move "Disk(1)" from Tower A to Tower B
Tower A:
Tower B: Disk(3) Disk(2) Disk(1)
Tower C:
```

Towers of Hanoi (cont.)

Simulation results

when $N = 4$:

```
-- ------------------------
-- Step  0: Initial setting
-- ------------------------
Tower A: Disk(4) Disk(3) Disk(2) Disk(1)
Tower B:
Tower C:
-- -----------------------------------------------
-- Step  1: Move "Disk(1)" from Tower A to Tower C
-- -----------------------------------------------
Move "Disk(1)" from Tower A to Tower C
Tower A: Disk(4) Disk(3) Disk(2)
Tower B:
Tower C: Disk(1)
-- -----------------------------------------------
-- Step  2: Move "Disk(2)" from Tower A to Tower B
-- -----------------------------------------------
Move "Disk(2)" from Tower A to Tower B
Tower A: Disk(4) Disk(3)
Tower B: Disk(2)
Tower C: Disk(1)
-- -----------------------------------------------
-- Step  3: Move "Disk(1)" from Tower C to Tower B
-- -----------------------------------------------
Move "Disk(1)" from Tower C to Tower B
Tower A: Disk(4) Disk(3)
Tower B: Disk(2) Disk(1)
Tower C:
```

```
-- -----------------------------------------------
-- Step  4: Move "Disk(3)" from Tower A to Tower C
-- -----------------------------------------------
Move "Disk(3)" from Tower A to Tower C
Tower A: Disk(4)
Tower B: Disk(2) Disk(1)
Tower C: Disk(3)
-- -----------------------------------------------
-- Step  5: Move "Disk(1)" from Tower B to Tower A
-- -----------------------------------------------
Move "Disk(1)" from Tower B to Tower A
Tower A: Disk(4) Disk(1)
Tower B: Disk(2)
Tower C: Disk(3)
.
.
.
-- -----------------------------------------------
-- Step 14: Move "Disk(2)" from Tower A to Tower B
-- -----------------------------------------------
Move "Disk(2)" from Tower A to Tower B
Tower A:
Tower B: Disk(4) Disk(3) Disk(2)
Tower C: Disk(1)
-- -----------------------------------------------
-- Step 15: Move "Disk(1)" from Tower C to Tower B
-- -----------------------------------------------
Move "Disk(1)" from Tower C to Tower B
Tower A:
Tower B: Disk(4) Disk(3) Disk(2) Disk(1)
Tower C:
```

Towers of Hanoi (cont.)

Algorithm complexities

$O(2^n)$ – *Exponential!*

I.e., it can only solve problems when the number of disks (N) is very small

$\Rightarrow$ Indeed a ***hard*** problem though have a ***clean*** solution! 😂

Recursive algorithm

Lesson learned

- Finding the stopping/base case typically is *not* that difficult
- However, it might be difficult in:
  (i) Identifying and formulate the recursive step;
  (ii) Understanding and debugging, especially for complex problem with *multiple* recursive calls and base cases as it is hard to follow the *flow of execution* and identify *logical errors* or edge cases.

Recursion vs Iteration

Recursion

- More powerful than iteration
  - iteration can only solve tail-recursion problems (e.g., $f(n) = a + f(n-1)$)
- Often gives clean and easy-to-understand algorithms

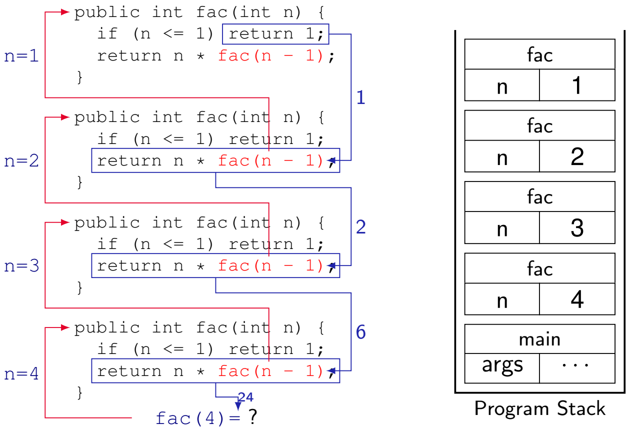
However,

- More expensive in time and space than iteration
  - More time to make function call
  - More stack memory space

Why?

Problem of Recursion

Program stack of the factorial example ($n = 4$)



Program stack for program execution

When a method is called,

- Runtime environment creates *activation record*, a.k.a. *stack frame*, which
  - Save the value for *all* variables for that call in memory!
  - Shows method's state during execution
- For the *activation record* pushed onto the program stack:
  - Top of stack belongs to currently executing method
  - Next record down the stack belongs to the one that called current method
- Therefore, as the activation records keep on accumulating during the execution, more and more memory (on the stack) is consumed.

? *When will the activation records stop accumulating in a recursive process?*

? *What will happen when the computer does not have enough memory?*

Recursion Examples – Fibonacci Sequence

Finding the n^{th} term of the Fibonacci sequence.

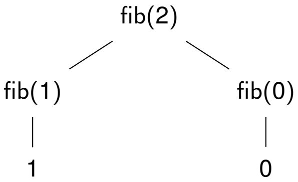
- The Fibonacci series is a series of numbers that starts with 0 and 1, and each subsequent number is the sum of the two preceding numbers, as follow:

  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`
- Recursive definition:
  - Base case:
    - `F(0) = 0`
    - `F(1) = 1`
  - Recursive step:
    - `F(n) = F(n-1) + F(n-2)`, where $F(n)$ is the n^{th} number in the series.

```
public long fibonacci(long n) {
  if (n == 0 || n == 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

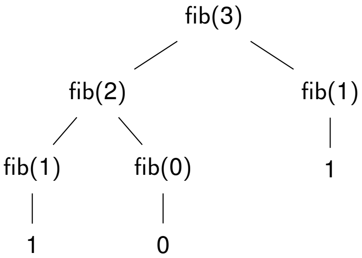
Fibonacci Sequence

When $n = 2$:



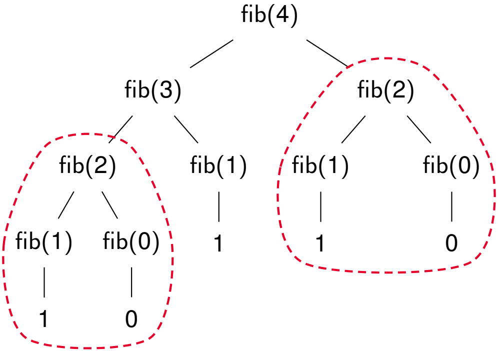
Fibonacci Sequence (cont.)

When $n = 3$:



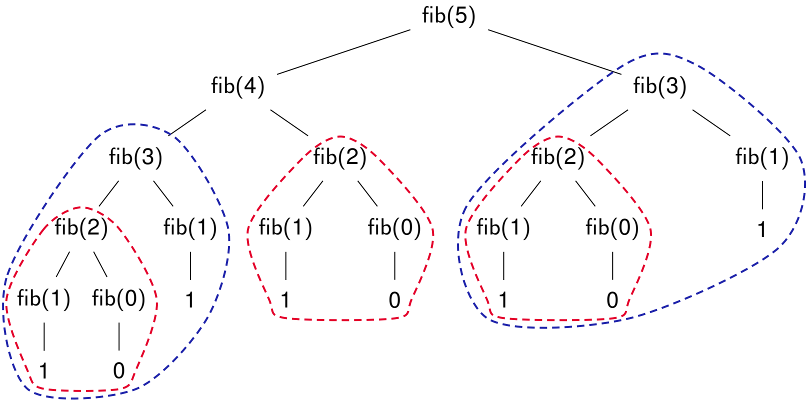
Fibonacci Sequence (cont.)

When $n = 4$:



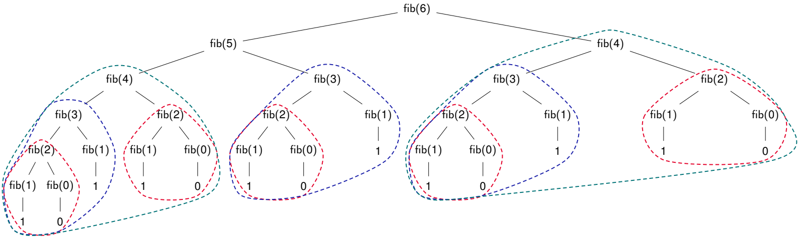
Fibonacci Sequence (cont.)

When $n = 5$:



Fibonacci Sequence (cont.)

When $n = 6$:



Algorithmic complexity: $O(2^n)$ – Exponential

Recursion Example: Combination (C_r^n)

By definition, we have

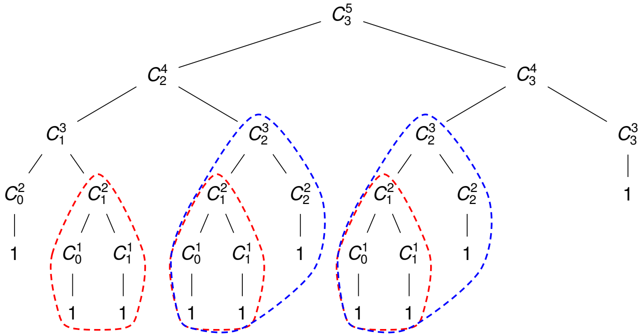
- $C_r^n = \dfrac{n!}{r!(n-r)!}$

Besides,

- $C_r^n = 1$ if $n = 0$ or $n = r$
- $C_r^n = C_r^{n-1} + C_{r-1}^{n-1}$

```
long combin(long n, long r) {
  if (r == 0 || r == n) return 1;
  return combin(n - 1, r - 1) + combin(n - 1, r);
}
```

Recursion Example: Combination (C_r^n) (cont.)



- Algorithmic complexity: $O(2^n)$ – Exponential
- The algorithm fails to store intermediary results, resulting in the repetition of computations for the same formula!

Problems with Recursion

Common problems

Space complexity Recursive solutions often require additional memory for each recursive call, as the *stack grows* with each function invocation, which can lead to high *space complexity*, particular for problems with *large input sizes* or *deep recursion*.

Function call overhead Each recursive function call incurs overhead in terms of *time* and *memory* due to *parameter passing*, *activation records*, and *return results* after execution. This overhead will become more significant and create impact to the performance when a large number of recursive call are made.

Stack Overflow If the recursion depth is too high, it can lead to a *stack overflow*, causing the program to *crash*.

Problems with Recursion (cont.)

Common problems

Redundant computation
: Recursive functions may perform redundant computations by *repeatedly* solving the same subproblems, leading to *unnecessary work* and increased *time complexity*.

Difficulty in understanding and debugging
: As discussed in the Towers of Hanoi example, it can be difficult to follow the *flow of execution* and identify *logical errors* or *edge cases*, making *debugging* more difficult.

Non-optimal time complexity
: Not all recursion can lead to optimal time complexity! Recursive algorithms that do not follow a *divide-and-conquer* or *memorization* approach may result in *exponential* or higher *polynomial* time complexity, making them inefficient for large problem sizes.

$\Rightarrow$ *Replace recursion with iteration by using a stack if possible*

Reading

- Chapter 2 and 4, Cormen (2022)