Lecture #3
Recursion (1)

Algorithm

JBNU

Jinhong Jung

In This Lecture

❑ Concept of recursion

- What is recursion? Why do we need recursion?

❑ How to design and analyze recursion

- Design by **divide and conqure**
- Correctness analysis by **mathematical induction**

❑ Recursive complexity

- How is $T(n)$ of a recursive algorithm represented?

Outline

- ❑ Recursion
- ❑ How to Design Recursion
- ❑ Correctness Analysis of Recursion
- ❑ Recursive Complexity

Recursion

- We say "Something is recursive" when it is defined in terms of itself

Recursion In Math & CS

Recurrence relation in Mathematics

- Equation that is recursively defined by itself

Recursive function in CS

- Function that is recursively defined by itself

$$a_n = \begin{cases} n \times a_{n-1}, & n > 1 \\ 1, & n = 1 \end{cases}$$

Recurrence relation

```
def f(n):
    if n == 1:
        return 1
    else:
        return n * f(n - 1)
```

Recursive function

- They are the same intrinsically under the concept of recursion

Why Do We Need Recursion?

❑ Numerous operations are compactly represented in just a few words (i.e., it improves readability!)

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

⚠ Do not misunderstand!

- Not saying recursion is always proper for every problem
  - It's effective when your problem has a recursive property
- Not saying a recursive function is always computationally efficient and optimized

Definition of Recursive Function

- A function is recursive when it is defined by
  - 1) Simple base case(s)
    - **Terminating scenario** that doesn't use recursion to produce an answer
      - If there is no base case, the function will run forever, incurring a stack overflow error
  - 2) Recursive step
    - Rules that **reduces** all other cases towards the base by calling itself

```
def function(n):
    if n == 1: # base case (example)
        do something
    else:      # recursive step
        do something with function(k)
        where k is reduced toward the base case
        (e.g., k = n-1, n/2, etc.)
```

Example: Factorial

- **Factorial:** $n! = 1\times 2\times \cdots \times (n-1)\times n$

- **Recurrence relation of $n!$**

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

Base case

Recursion step

- **Recursive function of $n!$**

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

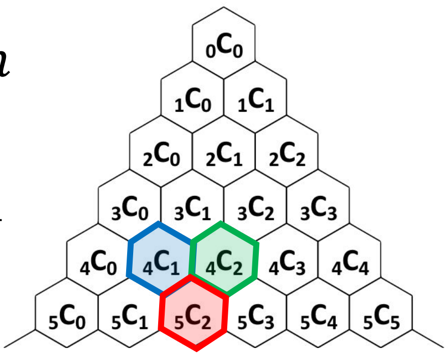
Example: Binomial Coefficient

- Binomial coefficient of n & k (where $0 \le k \le n$)

$$ {}_{n}C_{k} = \binom{n}{k} = \frac{n!}{k!\,(n-k)!} $$

- Recurrence relation

$$ \binom{n}{k} = \begin{cases} 1 & k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & 1 \le k \le n-1 \end{cases} $$



- Recursive function

```
def bin-coeff(n, k):
    if k == 0 or k == n:  return 1
    else:  return bin-coeff(n-1, k-1) + bin-coeff(n-1, k)
```

Outline

- ❑ Recursion
- ❑ How to Design Recursion
- ❑ Correctness Analysis of Recursion
- ❑ Recursive Complexity

How To Design Recursion? (1)

❑ Problem: Exponentiation (or power)

- **Input**: base number a and exponent n
- **Output**: a^n

❑ Let's design the problem in a recursive way!

- One strategy is Divide & Conquer
  - Divide the problem into several (smaller) sub-problems
  - Conquer them separately
  - Aggregate the answers of the sub-problems if necessary

$$a^n = \underbrace{a \times a \times \cdots \times a}_{n-1} \times \underbrace{a}_{1} = a^{n-1} \times a^1$$

How To Design Recursion? (2)

- Let's define the recurrence relation for the problem

$$\overbrace{a^n}^{\texttt{power}(a,n)} = \underbrace{a \times a \times \cdots \times a}_{a^{n-1} = \texttt{power}(a,n-1)} \times \overbrace{a}^{a^1 = \texttt{power}(a,1)}$$

  - **Assume** that a function called $\texttt{power}(a,n)$ computes a^n
    - Base case: return 1 if $n = 0$
    - Recursive step: return $\texttt{power}(a, n-1) \times a$ if $n > 0$

```
def power(a, n):
    if n == 0:  return 1
    else:       return power(a, n-1) * a
```

Outline

- ❑ Recursion
- ❑ How to Design Recursion
- ❑ **Correctness Analysis of Recursion**
- ❑ Recursive Complexity

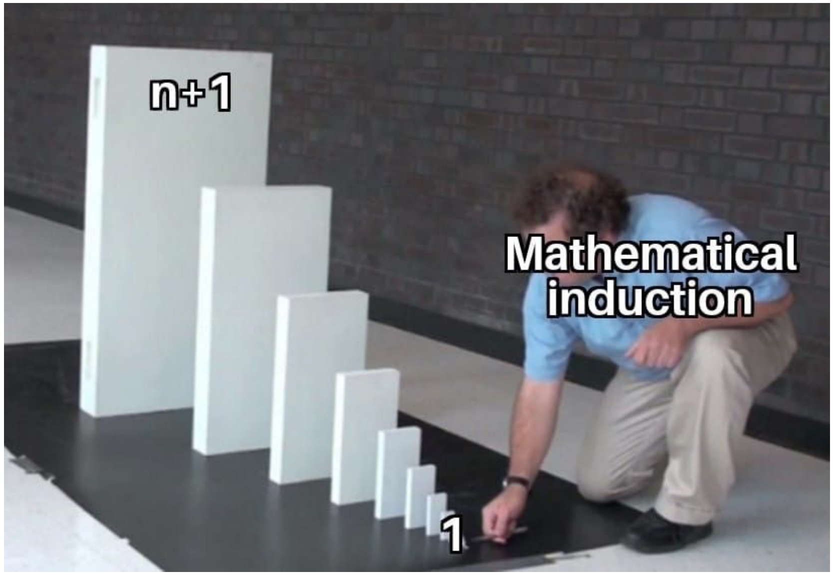
How To Prove Its Correctness?

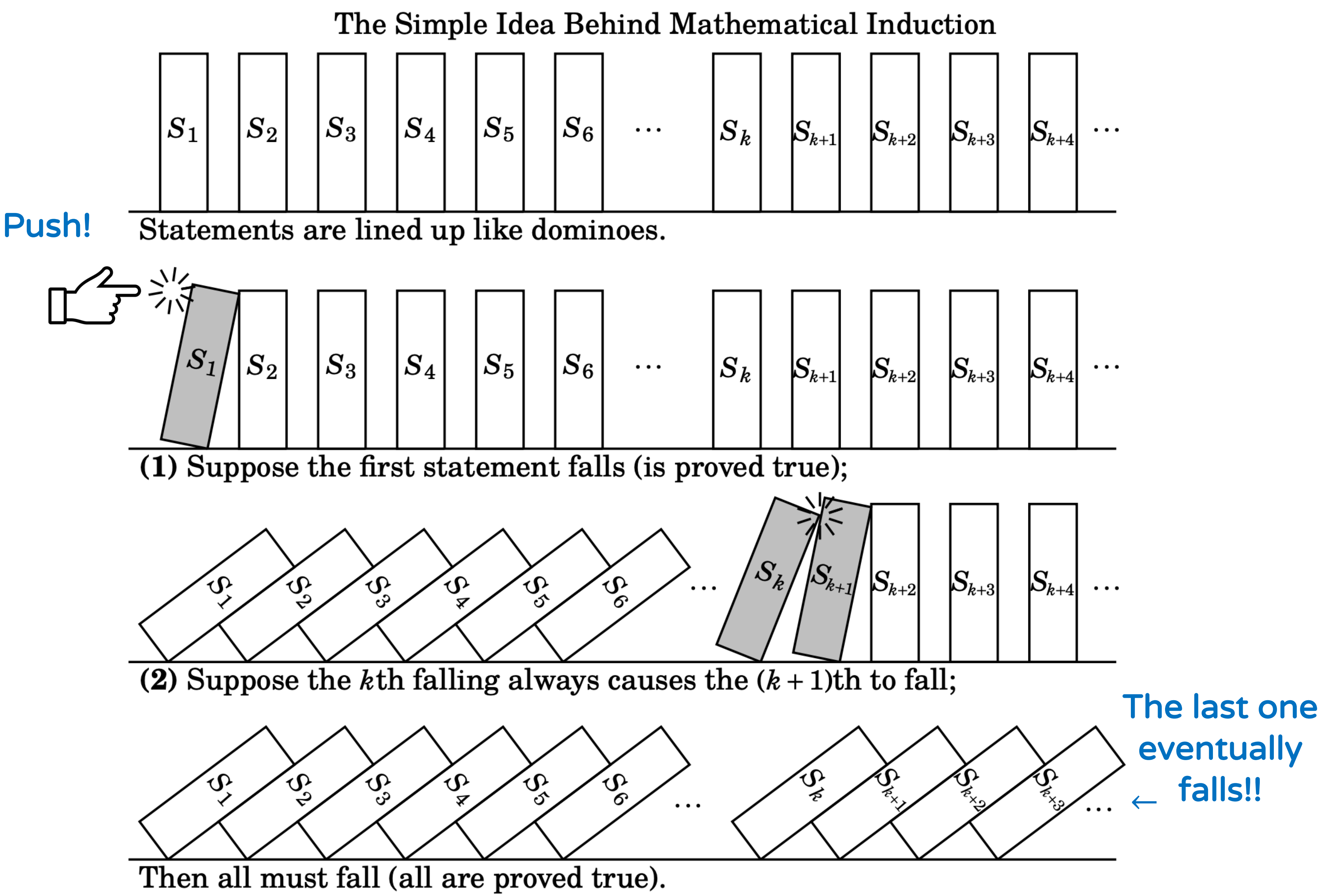
- Prove the correctness using Mathematical Induction

- Main idea of mathematical induction
  - **If the previous domino falls, show the next domino also falls!**
  - All remaining dominoes are expected to be fallen when the above is revealed!

⚠️ **Do not misunderstand!**

- Mathematical induction **is not** inductive reasoning.
  - It is rather deductive reasoning!





Reference: http://www.people.vcu.edu/~rhammack/BookOfProof/Main.pdf#page=192

Mathematical Induction

- **Claim.** S_n holds for every natural number n.
- 1) Base case(s)
  - Prove that S_n holds when n is base case(s).
- 2) Inductive step
  - **Previous case**: Assume that S_k is true for $n = k$.
  - **Next case**: Does S_{k+1} also hold for $n = k + 1$?
    - Prove it also holds for $k + 1$ based on the assumption at k.
    - The increment does not need to be 1.
      - Any increment such as +2 and ×2 is possible (it depends on problems).
- Then, say "S_n **holds for every** n **by induction!**"

Example: Power (1)

- S_n : power(a, n) correctly computes a^n for all $n \in \mathbb{N}_0$.

```
def power(a, n):
    if n == 0:   return 1
    else:        return power(a, n-1) * a
```

- Proof by induction
  - 1) Base case
    - The base case is $n = 0$, and in this case, power(a, n) always returns 1.
    - That is, $a^0 = 1$ is obviously true for any a.
    - Thus, S_0 is true for the base case!

Example: Power (2)

- S_n : power(a, n) correctly computes a^n for all $n \in \mathbb{N}$.

```
def power(a, n):
    if n == 0:    return 1
    else:         return power(a, n-1) * a
```

- **Proof by induction**
  - 2) Inductive step
    - **Previous case**: assume the claim holds for k (i.e., assume S_k is true).
    - **Next case**: does S_{k+1} also hold?

$$\text{power}(a, k+1) = \underbrace{\text{power}(a, k)\times a = a^k}_{\text{true by } S_k}\times a = a^{k+1}$$

    - Thus, S_n holds for every n by induction!

Example: Inequality

- $S_n : 2^n > n + 4$ for $n \geq 3$.
  - Base case: $n = 3$ and $2^3 = 8 > 3 + 4 = 7$; thus, it holds.
  - Inductive step
    - **Previous case:** assume S_k holds for $n = k$ (where $k > 3$).

$$S_k : \; 2^k > k + 4 \text{ is true (assumed)}$$

    - **Next case:** Dose S_{k+1} also hold?

$$S_{k+1} : \; 2^{k+1} > k + 5 \Leftrightarrow 2\times 2^k - k - 5 > 0 \qquad \Leftarrow \text{Is it really true?}$$

true by S_k

$$2\times \boxed{2^k} - k - 5 > 2\times \boxed{(k+4)} - k - 5 = k + 3 > 0 \qquad (\because k > 3)$$

    - Thus, S_n holds for $n \geq 3$ by induction!

Strong Induction

- **We use strong induction when**
  - It is hard to specify a previous case (e.g., not k), or
  - Multiple previous cases are needed for proving.
- **Claim.** S_n holds for every $n \geq a$.

b = # of base cases

- 1) Base case: Prove $S_a, S_{a+1}, \cdots, S_{a+b}$ holds.
- 2) Inductive step
  - **Previous case**: Assume that all S_i hold for $a \leq i \leq k$
  - **Next case**: Does S_{k+1} also hold?
    - Prove the claim using one or more assumptions S_i **in the range** (that's why it's called strong)

Example: Strong Induction

- $S_n : a_n = a_{n-2} + 2a_{n-1}$ is odd for every $n \geq 1$ given $a_1 = 1$ and $a_2 = 3$.
- 1) Base cases
  - S_1 and S_2 are trivially true because a_1 and a_2 are odd.
- 2) Inductive step
  - **Previous**: Assume S_i holds, i.e., a_i are odd for $1 \leq i \leq k$
  - **Next:** is S_{k+1} also true? $\Rightarrow$ is $a_{k+1} = a_{k-1} + 2a_k$ odd?
    - Use two assumptions on S_{k-1} and S_k.
    - The sum of odd and even leads to odd, implying a_{k+1} is odd too!
- By induction, a_n is odd for $n \geq 1$ [Q.E.D]

Outline

- ❑ Recursion
- ❑ How to Design Recursion
- ❑ Correctness Analysis of Recursion
- ❑ Recursive Complexity

Complexity of Recursive Function

❑ What is the time complexity of the below algorithm?

```
def power(a, n):
    if n == 0:
        return 1
    else:
        return power(a, n-1) * a
```

- $T(n)$ of the recursive algorithm is also described **recursively**.
- We need to solve the recurrence relation of $T(n)$ to obtain its closed solution.

Example: Power (1)

- **Let $T(n)$ denote the time complexity of** `power`(a, n)
  - Base case: $T(n) = C$ because it just returns $1 \in O(1)$ time
  - Recursive step: $T(n) = T(n-1) + C$
    - 1) It recursively calls `power`(a, n-1) which requires $T(n-1)$ time
    - 2) After then, the result is multiplied by a requiring $O(1)$ time

```python
def power(a, n):
    if n == 0:
        return 1
    else:
        return power(a, n-1) * a
```

$$T(n) = \begin{cases} C & n = 0 \\ T(n-1) + C & n > 0 \end{cases}$$

Q. What is its closed solution?

Example: Power (2)

- Let's divide the problem as follows:

If n is even: $$a^n = (a \times a) \times (a \times a) \times \cdots \times (a \times a) = (a^2)^{\frac{n}{2}}$$

If n is odd: $$a^n = a \times (a \times a) \times (a \times a) \times \cdots \times (a \times a) = a \times (a^2)^{\frac{n-1}{2}}$$

```
def power(a, n):
    if n == 0:
        return 1
    else if n is even:
        return power(a * a, n/2)
    else:
        return a * power(a * a, (n-1)/2)
```

$$T(n) = \begin{cases} C & n = 0 \\ T(n/2) + C & n \text{ is even} \\ T\left(\frac{n-1}{2}\right) + C & n \text{ is odd} \end{cases}$$

Q. What is its closed solution?

What You Need To Know

- Concept of recursion
  - Something is recursive when it is defined in terms of itself
- How to design and analyze recursion
  - **Divide & conquer:** Divide the problem, **conquer** its sub-problems, and **aggregate** their answers
  - **Mathematical induction:** if the previous domino falls, show the next domino also falls!
- Recursive complexity
  - The time complexiy $T(n)$ of a recursive algorithm is also recursive. ⇒ How to solve it?

In Next Lecture

- How to obtain the closed solution of a recursive time complexity?
  - Method 1) Subsitute method
  - Method 2) Mathematical induction
  - Method 3) Master theorem

Thank You