Lecture #2
Recursion (1)

Algorithm

JBNU Spring 2021
Jinhong Jung

In Previous Lecture

Efficiency of algorithm

- When it solves the problems within resource contrains

How to measure the efficiency?

- Empirical measurement (directly run a program)
- Theoretical measurement (do complexity analysis)

Best, average, and worst cases

- Should do analysis for **the worst case** at least

Asymptotic analysis

- Express & group various complexities in simply notations
  - While considering the large size of input at the same time
- Using Big-O, Omega, and Theta bounds
  - At least, get Big-O bound for worst case as tight as possible

In This Lecture

Concept of recursion

- What is recursion?
- Why do we need recursion?

How to design and analyze recursion

- Divide and conqure
- Mathematical induction

Outline

Concept of recursion and recurrence

How to design and analyze recursion

Concept Of Recursion

Recursion (재귀)

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

Recursion In Math & CS

Recurrence relation (점화식) in Mathematics

- Equation that is recursively defined by itself

Recursive function (재귀함수) 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 Recursion?

Q. Why do we need recursion?

- A: Can simply describe an algorithm into several terms which are easily understood by most people
  - An infinite number of computations can be described by a finite & simple recursive form without explicit repetitions (such as for loop)

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

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

Formal Definition of Recursion

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 case 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 (n = 1)
        (e.g., k = n-1, n/2, etc.)
```

Example: Factorial (1)

Factorial of n

$$n! = 1\times2\times\cdots\times(n-1)\times n$$

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

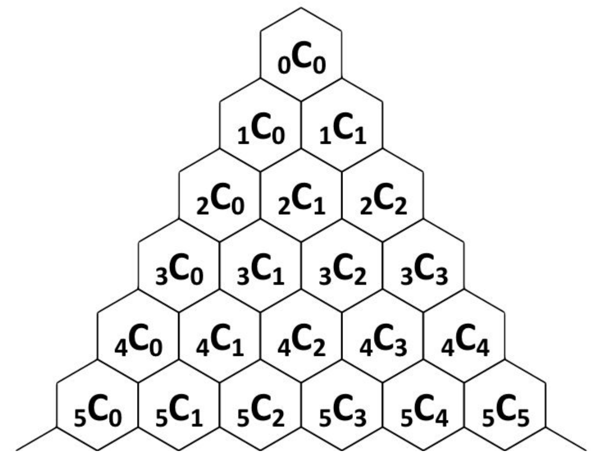
Recursive function of $n!$

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

Example: Binomial Coefficient

Binomial coefficient of n & $0 \leq k \leq n$

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



Pascal's Triangle

Recurrence relation of $\binom{n}{k}$

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

Recursive function of $\binom{n}{k}$

```
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

Concept of recursion and recurrence

How to design and analyze recursion ☜

How To Design Recursion? (1)

Problem: Exponentiation (or power)

- **Input**: base number a and exponent n
- **Output**: to calculate 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 results 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

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

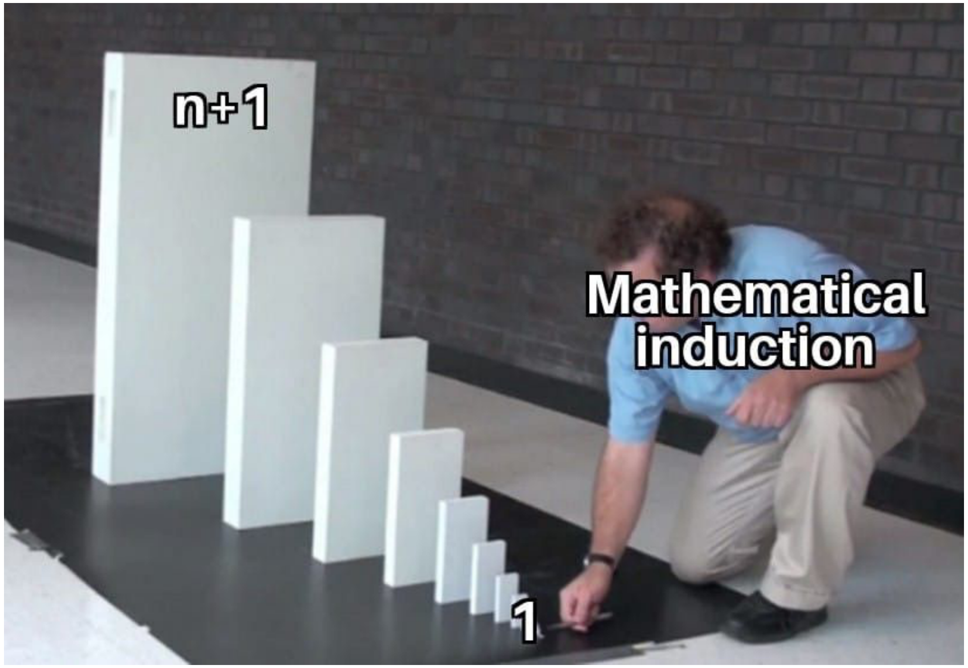
- **Assume** that a function called $power(a, n)$ computes a^n
  - Base case: the function should return 1 if $n = 0$
  - Recursive step: the function should return $power(a, n-1) \times a$ if $n > 0$

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

How To Prove Its Correctness?

Q. How can we guarantee that the designed $power$ function correctly computes its output?

- A: Prove it using Mathematical Induction **(수학적 귀납법)**
  - It's also shortly called 'proof by induction'
- Recursion is highly related to mathematical induction!!!



Mathematical Induction

Claim. $P(n)$ holds for every natural number n

1) Base case(s)

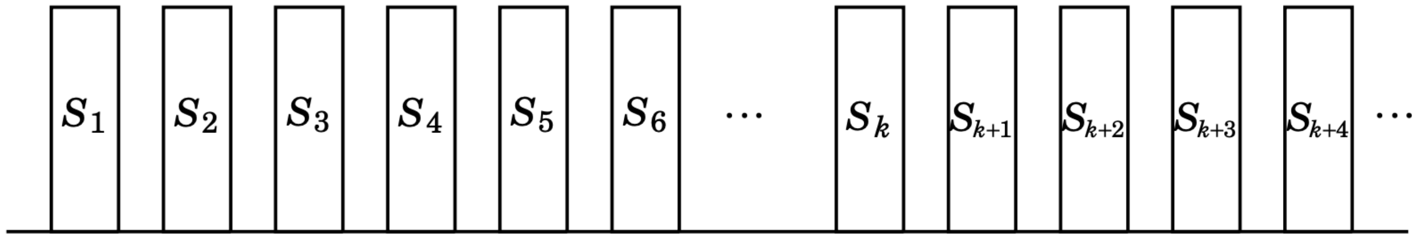
- Prove that $P(n)$ holds when n is base case(s)

2) Inductive step

- Previous case: Assume that the claim is true for $n = k - 1$
- Next case: Does the claim also hold for $n = k$?
  - Prove it must also hold for k based on the assumption at $k - 1$
  - The increment does not need to be 1
    - Any increment such as +2 and ×2 is possible (it depends on problems)

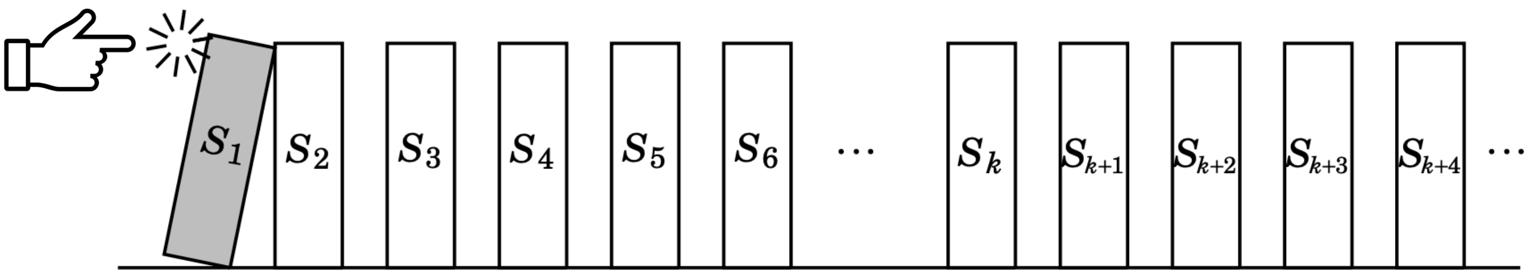
By mathematical induction, $P(n)$ holds for every n

The Simple Idea Behind Mathematical Induction

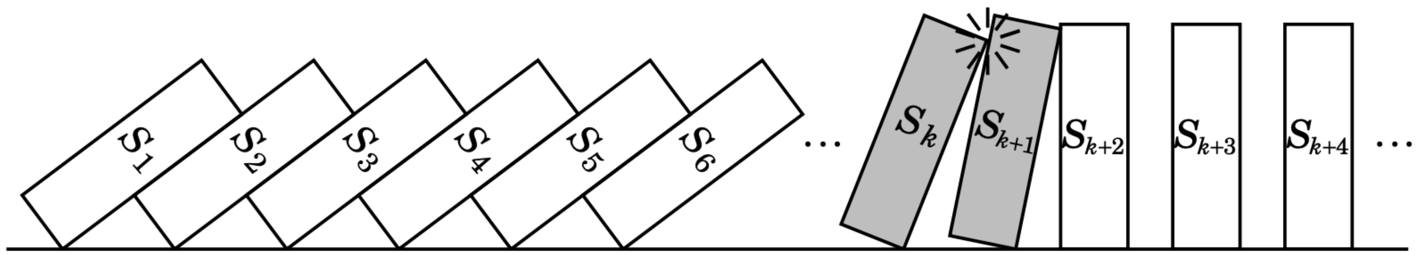


Statements are lined up like dominoes.

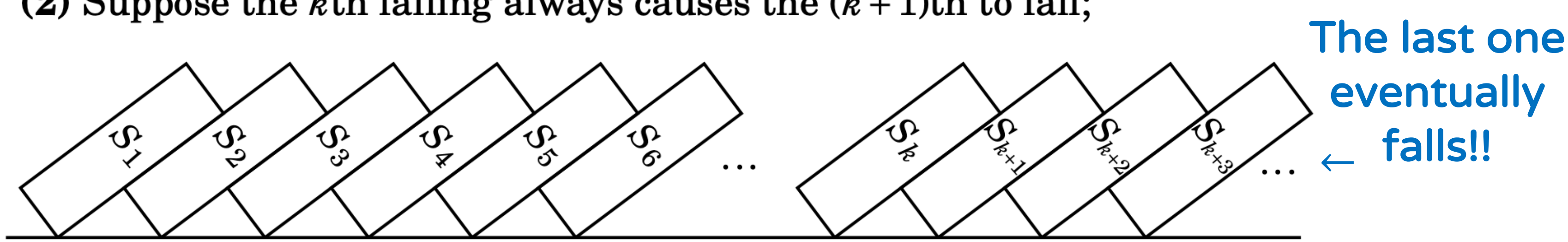
Push!



(1) Suppose the first statement falls (is proved true);



(2) Suppose the kth falling always causes the $(k+1)$th to fall;



Then all must fall (all are proved true).

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

Example For Power

Claim. The function $power(a, n)$ correctly computes a^n for natural number n

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

Proof by induction

- Base case
  - The base case is $n = 0$, and in this case, $power(a, n)$ always returns 1
  - Thus, the claim holds for the base case
- Inductive step
  - Previous case: assume the claim holds for $k - 1$

    $power(a, k - 1)$ computes a^{k-1} correctly (assumed)

  - Next case: does the claim also hold for k?

    Is it true? → $power(a, k)$

    $= power(a, k - 1)\times a = a^{k-1}\times a = a^k$

- By mathematical induction, the claim is true [Q.E.D.]

Example For Inequality

Claim. $P(n)$: $2^n > n + 4$ for $n \geq 3$

- **Base case**
  - $n = 3$ is the base case, and $2^3 = 8 > 3 + 4 = 7$; thus, it holds
- **Inductive step**
  - Previous case: assume the claim holds for $k - 1$

    $2^{k-1} > k + 3$ is correct (assumed)

  - Next case: Dose the claim also hold for k?

    $$2^k > k + 4$$
    $$\Leftrightarrow 2^k - k - 4 > 0$$
    $$\Leftrightarrow 2 \times 2^{k-1} - k - 4 > 0 \quad \leftarrow \text{Is it true?}$$

    - From the assumption $2^{k-1} > k + 3$,

    $$2 \times 2^{k-1} - k - 4 > 2(k + 3) - k - 4 = k + 2 > 0$$

    - Note that since this case is byond the base case (i.e., $k > 3$), $k + 2 > 0$

What You Need To Know

Concept of recursion

- When it is defined in terms of itself, it is called recursion
  - Base case(s) and recursive step
- Why do we need recursion? ⇒ Can simply describe an algorithm into several terms

How to design and analyze recursion

- Divide and conqure
  - Divide the problem into several (smaller) sub-problems
  - Conquer them separately & aggregate the results if necessary
- Mathematical induction
  - If $k-1$-th domino falls, then k-th domino falls surely
  - Prove base cases and inductive step

In Next Lecture

You might ask like

- My algorithm is working perfectly! But it is recursive & very complicated, and I don’t know if it’s fast or not. How can I analyze its time complexity?

Let’s analyze a recursive & complicated complexity

- Using substitute method
- Using mathmetical induction
- Using master theorem

Thank You