

# Lecture #3

## Recursion (2)

---

Algorithm

JBNU Spring 2021

Jinhong Jung

# In Previous Lecture

---

## Concept of recursion

- When it is defined in terms of itself, it is called recursion
  - Base case(s) and recursive step
- Recursion can simply describe an algorithm into several terms

## How to design and analyze recursion

- Divide and conquer
  - 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 This Lecture

---

## How to analyze a recursive complexity function

- Repeated substitution
- Mathematical induction
- Master theorem

# Outline

---

Recursive complexity 

Repeated substitution

Mathematical induction

Master theorem

# Complexity of Recursive Alg.

---

## Problem: Exponentiation (or power)

- Input: base number  $a$  and exponent  $n$  (non-negative int.)
- Output: to calculate  $a^n$

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

- Q: What is the time complexity of the above algorithm?
  - It's also represented recursively as recurrence relation
  - We need to solve the recurrence relation to obtain its closed form

# Recursive Complexity

Let  $T(n)$  denote the time complexity of `power(a, n)`

- $T(n)$ : # of operations to solve the problem which size is  $n$ 
  - If  $n$  is a **base case**,
    - It just returns 1 requiring constant time, i.e.,  $T(n) = C$
  - If  $n$  is in a **recursive step**,
    - 1) `power(a, n)` calls `power(a, n-1)` which requires  $T(n-1)$
    - 2) After then, the result is multiplied by  $a$  requiring constant time  $C$
    - i.e.,  $T(n) = T(n-1) + C$

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

problem of size  $n$

sub-problem of size  $n-1$



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

Q. What is its closed form?

# Another Recursive Algorithm

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)
```



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

Q. What is its closed form?

# Fibonacci Number

---

## Problem: Fibonacci number

- **Input:**  $n$  indicates  $n$ -th Fibonacci number, i.e.,  $F_n$
- **Output:** to calculate  $n$ -th Fibonacci number  $F_n$  defined by

$$F_n = F_{n-2} + F_{n-1}$$

- The sum of two consecutive Fibonacci numbers
- $F_0 = 0$  and  $F_1 = 1$  by its definition (base cases)
- Let  $T(n)$  be the time complexity of its recursive algorithm

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



$$T(n) = \begin{cases} C & n \leq 1 \\ T(n-2) + T(n-1) + C & n > 1 \end{cases}$$

Q. What is its closed form?



# Outline

---

Recursive complexity

Repeated Substitution 

Mathematical induction

Master theorem

# Repeated Substitution (1)

---

## Basic idea of repeated substitution


- Repeatedly substitute the complexity function whose input size decreases toward a base case

**Example:**  $T(n) = T(n - 1) + C$

- $T(1) \leq C$  as its base case

$$\begin{aligned} T(n) &= T(n - 1) + C \\ &= T(n - 2) + 2C \\ &= T(n - 3) + 3C \\ &= \dots \\ &= T(1) + (n - 1)C \leq Cn = O(n) \end{aligned}$$

substitute



# Repeated Substitution (2)

---

**Example:**  $T(n) \leq 2T\left(\frac{n}{2}\right) + n$

- $T(1) \leq C$  as its base case

$$\begin{aligned}T(n) &\leq 2T\left(\frac{n}{2}\right) + n \\&\leq 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 2^2T\left(\frac{n}{2^2}\right) + 2n \\&\leq 2^2\left(2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right) + 2n = 2^3T\left(\frac{n}{2^3}\right) + 3n \\&= \dots\end{aligned}$$

$$\begin{aligned}\text{Assume } n = 2^k \rightarrow &\leq 2^kT\left(\frac{n}{2^k}\right) + kn = nT(1) + n \log n \\&\leq nC + n \log n = O(n \log n)\end{aligned}$$

# Assumption on $n = 2^k$

Details

$n = 2^k$  is often assumed for computational ease

- **Fact:** There is always a power of 2 between  $n$  &  $2n$ 
  - i.e.,  $n \leq 2^k \leq 2n$  (see the appendix)
- If  $T(n) = O(n^r)$ ,  $T(2n) = O((2n)^r) = O(2^r n^r) = O(n^r)$ 
  - $T(n)$  and  $T(2n)$  have the same asymptotic behavior, i.e., as  $n \rightarrow \infty$ ,  
 $T(n) = T(2n)$
- Suppose  $T(n)$  is monotonically increasing as  $n \rightarrow \infty$ 
  - e.g.,  $T(n)$  is polynomial & its leading coefficient is positive,
  - $\Rightarrow n \leq 2^k \leq 2n \Leftrightarrow T(n) \leq T(2^k) \leq T(2n)$
  - $\Rightarrow T(n) = T(2^k) = T(2n)$  because  $T(n) = T(2n)$
- Thus, it's okay if we assume the size  $n$  of input is  $2^k$  since they have the same Big-O bound

# Repeated Substitution

---

## Pros

- Intuitive and easy to calculate
- Effective for a recursive complexity in a simple form

## Cons

- Prone to make mistakes
- Require a lot of efforts when we apply the method to a complicated complexity function
  - Try to solve the following problem using repeated substitution

$$T(n) = 3T\left(\frac{n}{4}\right) + \sqrt{n} \times \log n$$

# Outline

---

Recursive complexity

Repeated substitution

Mathematical induction 

Master theorem

# Mathematical Induction (1)

---

## Basic idea of mathematical induction

- Estimate the closed form of a recursive complexity, and then prove it by induction

## Example

- Claim:  $T(n) \leq 2T(n/2) + n$  and its closed form is  $T(n) \leq cn \log n$  for positive  $c$  and large  $n$
- **Base case**
  - If  $n = 2$ , there is always positive  $c$  such that  $T(2) \leq c2 \log 2$
- **Inductive step**
  - **Previous case:** assume the claim holds for  $n = k/2$
  - **Next case:** does the claim hold for  $n = k$ ?

# Mathematical Induction (2)

- Claim:  $T(n) \leq 2T(n/2) + n$  and its closed form is  $T(n) \leq cn \log n$  for positive  $c$  and large  $n$

- **Inductive step**

- Previous case: assume the claim holds for  $\frac{k}{2} \Rightarrow T\left(\frac{k}{2}\right) \leq c\left(\frac{k}{2}\right) \log \frac{k}{2}$
- Next case: does the claim hold for  $n = k$ ?

$$T(k) \leq 2T\left(\frac{k}{2}\right) + k$$

Use the assumption  $\rightarrow$   $\leq 2c\left(\frac{k}{2}\right) \log \frac{k}{2} + k = ck \log k - ck \log 2 + k$

$$= ck \log k + \underbrace{(-c \log 2 + 1)}_{\text{Set } c \text{ such that } -c \log 2 + 1 < 0 \Leftrightarrow c > \frac{1}{\log 2} = 1} k$$

Also true for  $k \rightarrow \leq ck \log k$

- Thus, there is always  $c$  satisfying  $T(n) \leq cn \log n$  for any  $n$
- Meaning  $T(n) = O(n \log n)$



# No Consideration of Base Case

---

Don't need to consider the part of base case

- When solving a recursive complexity by induction
- Why?
  - Suppose we should show  $T(n) = f(n)$
  - Then, we show  $T(a) \leq cf(a)$  for constant  $a$  as a base case
  - Note that  $T(a)$  is also constant and normally,  $f(n)$  returns a positive number (imagine it's a time complexity)
  - Thus, there is always  $c$  satisfying  $T(a) \leq cf(a)$
- Thus, we only consider the inductive step for such proofs

# Other Example – Wrong Version

---

**Claim:**  $T(n) \leq 2T(n/2) + 1$  and it's  $O(n)$

- Estimate  $T(n) \leq cn$

- **Inductive step**

- Previous case: assume the claim holds for  $n = \frac{k}{2}$

$$T\left(\frac{k}{2}\right) \leq c \frac{k}{2}$$

- Next case: does the claim hold for  $n = k$ ?

$$\begin{aligned} T(k) &\leq 2T\left(\frac{k}{2}\right) + 1 \\ &\leq 2c \frac{k}{2} + 1 \\ &= ck + 1 \end{aligned}$$

- Note that we cannot say  $ck + 1 \leq ck$ ; the proving fails

# Other Example – Correct Version

---

**Claim:**  $T(n) \leq 2T(n/2) + 1$  and it's  $O(n)$

- Estimate  $T(n) \leq cn - 2$

- **Inductive step**

- **Previous case:** assume the claim holds for  $n = \frac{k}{2}$

$$T\left(\frac{k}{2}\right) \leq c \frac{k}{2} - 2$$

- **Next case:** does the claim hold for  $n = k$ ?

$$\begin{aligned} T(k) &\leq 2T\left(\frac{k}{2}\right) + 1 \\ &\leq 2c \frac{k}{2} - 4 + 1 \\ &= ck - 3 \leq ck - 2 \end{aligned}$$

- Now the proving is correct since the result has the same form

# Mathematical Induction

---

## Pros

- For a complicated function, it's easier than the repeated substitution method
  - If an effective bound is provided...

## Cons

- Need to estimate “effective” bound
  - Loose bound is meaningless
  - Excessively tight bound will not be proved
- Intuition for such estimate is from experience
  - Need to solve a lot of problems in this way

# Outline

---

Recursive complexity

Repeated substitution

Mathematical induction

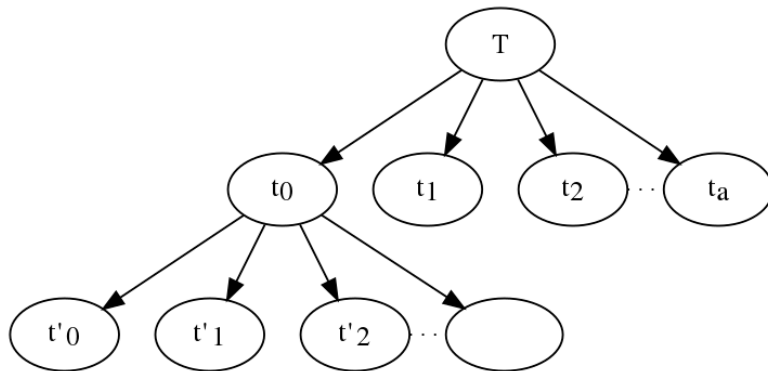
Master theorem



# Generalization of Recursive Alg.

Most recursive algorithms are based on Divide & Conquer as follows

```
def procedure(n):  
    if n ≤ some constant k like 1  
        solve the input directly without recursion  
    else:  
        create a subproblems, each having size n/b  
        call procedure p recursively on each subproblem  
        aggregate the results from the subproblems
```



Solution tree

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$f(n)$ : remaining cost to split the problem and recombining the results

# Master Theorem

---

Master theorem allows many recurrence relations of the following form to be converted  $\Theta$  directly!

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- $n$ : input size for problem
- $a \geq 1$ : number of sub-problems
- $n/b$ : size of input for each sub-problem where  $b \geq 1$
- $f(n)$ : remaining cost (overhead)
  - to split the problem and recombining the results at the top level

There is an exact version of Master Theorem

- But not discussed in this lecture since it's complicated
- Instead, let's check its approximate version

# Master Theorem

---

## Master Theorem [approximate version]

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

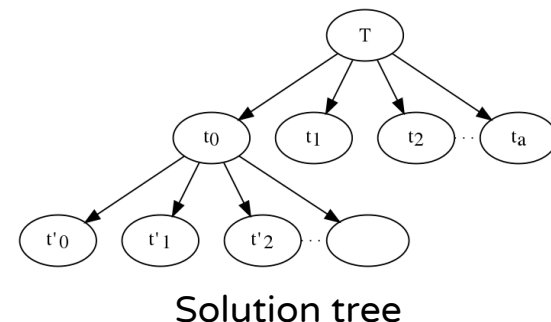
- Suppose  $h(n) = n^{\log_b a}$
- **Case 1)**  $\lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} = 0 \Rightarrow T(n) = \Theta(h(n))$
- **Case 2)**  $\lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} = \infty$  &  $af\left(\frac{n}{b}\right) < f(n) \Rightarrow T(n) = \Theta(f(n))$
- **Case 3)**  $\frac{f(n)}{h(n)} = \Theta(1) \Rightarrow T(n) = \Theta(h(n) \log_2 n)$



# Interpretation (1)

## Master Theorem [approximate version]

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$



- Suppose  $h(n) = n^{\log_b a}$ 
  - $\Rightarrow$  # of leaf nodes in the solution tree
  - $\Rightarrow$  # of problems where the input size is 1
  - $\Rightarrow$  cost for solving all problems where the input size is 1
- The depth of the solution tree is  $k = \log_b n$ 
  - Size changes as  $n \rightarrow \frac{n}{b} \rightarrow \dots \rightarrow \frac{n}{b^k}$ ; when  $\frac{n}{b^k} = 1$ , it reaches at a leaf
- The number of leaf nodes at level  $k$  is  $a^k = a^{\log_b n} = n^{\log_b a}$ 
  - $a^{\log_b n} = x \Leftrightarrow \log_b n \times \log_b a = \log_b x \Leftrightarrow \log_b n^{\log_b a} = \log_b x$

# Interpretation (2)

---

## Master Theorem [approximate version]

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- **Case 1)**  $\lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} = 0 \Rightarrow T(n) = \Theta(h(n))$ 
  - Condition 1)  $\lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} = 0$  meaning  $h(n)$  overwhelms  $f(n)$ 
    - $h(n)$ : cost for solving all problems where the input size is 1
    - $f(n)$ : remaining cost (overhead) to split the problem & recombining the results at the top level
  - Then,  $h(n)$  determines the complexity  $T(n)$ 
    - The condition is called “the solution tree is leaf-heavy”

# Interpretation (3)

---

## Master Theorem [approximate version]

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- **Case 2)**  $\lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} = \infty$  &  $af\left(\frac{n}{b}\right) < f(n) \Rightarrow T(n) = \Theta(f(n))$ 
  - Condition 1)  $\lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} = \infty$  meaning if  $f(n)$  overwhelms  $h(n)$
  - Condition 2)  $af\left(\frac{n}{b}\right) < f(n)$ 
    - $af\left(\frac{n}{b}\right)$ : the sum of remaining costs of all sub-problems at the children level
    - $f(n)$ : remaining cost (overhead) of problem at the root level
    - When the recursion goes to the below level, the overhead cost should decrease!
  - Then,  $f(n)$  determines the complexity  $T(n)$ 
    - These conditions are called “the solution tree is root-heavy”

# Interpretation (4)

---

## Master Theorem [approximate version]

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- **Case 3)**  $\frac{f(n)}{h(n)} = \Theta(1) \Rightarrow T(n) = \Theta(h(n) \log n)$ 
  - Condition 1)  $\frac{f(n)}{h(n)} = \Theta(1)$  meaning if their weights are comparable by a constant
    - Work to split/recombine a problem is comparable to sub-problems
  - Then,  $h(n) \log n$  determines the complexity  $T(n)$

# Examples With Master Theorem

---

Get the closed form of the following complexities

- **Case 1:**  $T(n) = 2T\left(\frac{n}{3}\right) + c$ 
  - $a = 2, b = 3 \Rightarrow h(n) = n^{\log_3 2}$  and  $f(n) = c$
  - $\lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} = \lim_{n \rightarrow \infty} \frac{c}{n^{\log_3 2}} = 0$  thus,  $T(n) = \Theta(n^{\log_3 2})$
- **Case 2:**  $T(n) = 2T\left(\frac{n}{4}\right) + n$ 
  - $a = 2, b = 4 \Rightarrow h(n) = n^{\log_4 2} = \sqrt{n}$  and  $f(n) = n$
  - $\lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} = \lim_{n \rightarrow \infty} \frac{n}{\sqrt{n}} = \infty$  &  $af\left(\frac{n}{b}\right) < f(n) \Rightarrow 2\frac{n}{4} = \frac{n}{2} < n$ ; thus,  $T(n) = \Theta(n)$
- **Case 3:**  $T(n) = 2T\left(\frac{n}{2}\right) + n$ 
  - $a = 2, b = 2 \Rightarrow h(n) = n^{\log_2 2} = n$  and  $f(n) = n$
  - $\frac{f(n)}{h(n)} = 1 = \Theta(1)$ ; thus,  $T(n) = \Theta(n \log_2 n)$

# Other Technique

---

## Changing variables makes an equation simple

- $T(n) = 2T(\sqrt{n}) + \log_2 n$ 
  - Let  $m = \log_2 n \Rightarrow 2^m = n$
- $\Rightarrow T(2^m) = 2T\left(2^{\frac{m}{2}}\right) + m$ 
  - Let  $P(m)$  denote  $T(2^m)$
- $\Rightarrow P(m) = 2P\left(\frac{m}{2}\right) + m$ 
  - By Master theorem,  $P(m) = \Theta(m \log m)$
- Thus,  $T(n) = P(m) = \Theta(m \log m) = \Theta(\log n \log(\log n))$

# Master Theorem [Approx.]

---

## Pros

- Very convenient and can apply it to an arbitrary complexity function in form of  $T(n) = aT(n/b) + f(n)$ 
  - Do not need to calculate or prove something

## Cons

- For some cases, this approximate version cannot be applied
  - In these cases, need to use the exact version (see the textbook)
- Note that not all recurrence relations can be solved with this theorem

# What You Need To Know

---

## Recursive complexity

- Complexity of a recursive algorithm is also represented recursively as recurrence relation

## How to analyze a recursive complexity function

- **Repeated substitution**
  - Repeatedly substitute the complexity function whose input size decreases toward a base case
- **Mathmetical induction**
  - Estimate the closed form of a recursive complexity, and then prove it by induction
- **Master theorem**
  - Can solve any function in form of  $T(n) = aT(n/b) + f(n)$
- For some cases, changing variables makes an equation simple



# In Next Lecture

## Sorting problem

## Basic sorting algorithms

- Selection Sort
- Bubble Sort
- Insertion Sort

	 Insertion	 Selection	 Bubble	 Shell	 Merge	 Heap	 Quick	 Quick3
 Random								
 Nearly Sorted								
 Reversed								
 Few Unique								

Thank You

# Appendix: Proof for $n \leq 2^k \leq 2n$

---

**Claim:** there exists positive integer  $k$  s.t.  $n \leq 2^k \leq 2n$

## Proof

- Note that  $n = 2^{\log_2 n}$  for a natural number  $n$

$$2^{\lfloor \log_2 n \rfloor} \leq n \leq 2^{\lceil \log_2 n \rceil}$$

$$\Rightarrow 2^{\lfloor \log_2 n \rfloor + 1} \leq 2n \leq 2^{\lceil \log_2 n \rceil + 1}$$

- Note that  $2^{\lfloor \log_2 n \rfloor} \leq 2^{\lfloor \log_2 n \rfloor + 1}$ 
  - $\lfloor \log_2 n \rfloor - \lfloor \log_2 n \rfloor - 1 = \begin{cases} -1, & \log_2 n \text{ is integer} \\ 0, & \log_2 n \text{ isn't integer} \end{cases} \leq 0$
- Thus,  $n \leq 2^{\lfloor \log_2 n \rfloor} \leq 2^{\lfloor \log_2 n \rfloor + 1} \leq 2n$
- In other words,  $k = \lfloor \log_2 n \rfloor$  or  $\lfloor \log_2 n \rfloor + 1$ 
  - If  $\log_2 n$  isn't integer,  $\lceil \log_2 n \rceil = \lfloor \log_2 n \rfloor + 1$