Lecture #4
Recursion (2)

Algorithm

JBNU

Jinhong Jung

In This Lecture

- How to obtain the closed solution of a recursive time complexity?

```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?

- Method 1) Subsitute method
- Method 2) Mathematical induction
- Method 3) Master theorem

Outline

- ❑ Repeated Substitution
- ❑ Mathematical Induction
- ❑ Master Theorem

Repeated Substitution (1)

❑ Basic idea of repeated substitution

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

❑ Example: $T(n) = T(n-1) + C$

- $T(1) = C$ as its base case

$T(n-1) = T(n-2) + C$

substitute

$$\begin{aligned} T(n) &= \boxed{T(n-1)} + C \\ &= T(n-2) + 2C \\ &= T(n-3) + 3C \\ &= \cdots \\ &= T(1) + (n-1)C = Cn = \Theta(n) \end{aligned}$$

Repeated Substitution (2)

- **Example:** $T(n) \leq 2T\left(\frac{n}{2}\right) + n$
  - $T(1) = C$ as its base case

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

Assume $n = 2^k \rightarrow$

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

Assumption on $n = 2^k$

Details

- $n = 2^k$ is often assumed for computational ease.
  - **Fact**: $n \le 2^k \le 2n$ (see the appendix)
  - If $T(n) = O(n^r)$, then $T(2n) = O\left((2n)^r\right) = O(2^r n^r) = O(n^r)$.
    - i.e., $T(n) = T(2n)$
  - Suppose $T(n)$ is monotonically increasing as $n \to \infty$.
    - e.g., $T(n)$ is polynomial & its leading coefficient is positive.
    - $n \le 2^k \le 2n \Leftrightarrow T(n) \le T\left(2^k\right) \le T(2n)$
    - $\Leftrightarrow T(n) = T\left(2^k\right) = T(2n)$ because $T(n) = T(2n)$.
  - Thus, it's okay that we assume the size n of input is 2^k.

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

- ❑ Repeated Substitution
- ❑ Mathematical Induction
- ❑ Master Theorem

Mathematical Induction (1)

❑ Basic idea of mathematical induction

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

❑ Example

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

Mathematical Induction (2)

❑ Inductive step

- **Previous case**: assume it's true for $\frac{k}{2} \Rightarrow T\left(\frac{k}{2}\right) \le c\left(\frac{k}{2}\right)\log\frac{k}{2}$
- **Next case**: does the claim hold for $n = k$?

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

← Use the assumption

$$\le 2\boxed{c\left(\frac{k}{2}\right)\log\frac{k}{2}} + k = ck\log k - ck\log 2 + k$$

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

Also true for k → $\le ck\log k$

- Note that there is always c satisfying $T(n) \le cn\log n$ for any n.
- Implying $T(n) = O(n\log n)$.

No Consideration of Base Case

- Don't need to consider the part of base case when proving a recursive complexity by induction.

- Why?
  - Suppose we should show $T(n) \leq cf(n)$.
  - Then, we show $T(a) \leq cf(a)$ for constant a as a base case.
  - In general, $T(a)$ and $f(a)$ return positive numbers.
  - In other words, there is always c satisfying $T(a) \leq cf(a)$.

- Thus, it's okay that we only consider the inductive step for such proving.

Examples (1) – Wrong Version

- **Claim:** $T(n) \le 2T(n/2) + 1$ and it's $O(n)$.
  - 1) Estimate $T(n) \le cn$
  - 2) Inductive step
    - **Previous case**: assume the claim holds for $n = \frac{k}{2}$

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

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

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

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

Examples (1) – Correct Version

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

Examples (2)

- **Claim:** $T(n) \leq 2T\left(\frac{n}{2} + 10\right) + n$ and it's $O(n \log n)$.
- **Proof by strong induction**
  - 1) Estimate $T(n) \leq cn \log n$
  - 2) Inductive step
    - **Previous cases**: Assume all $T(i) \leq ci \log i$ are true for $n_0 \leq i < k$
    - **Next case**: is $T(k) \leq ck \log k$ true too?

$$T(k) \leq 2T\left(\frac{k}{2} + 10\right) + k$$

← Pick $i = \frac{k}{2} + 10 < k \Rightarrow k > 20$; and then use $T(i)$ for the derivation.

$$\leq 2c\left(\frac{k}{2} + 10\right) \log\left(\frac{k}{2} + 10\right) + k$$

$$\leq \cdots$$

← See the derivation in the textbook (66p).

$$\leq ck \log k \,.$$

← this holds for $k > 20$.

Mathematical Induction

❑ Pros

- For a complicated function, it's easier than the repeated substitution method.

❑ 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.

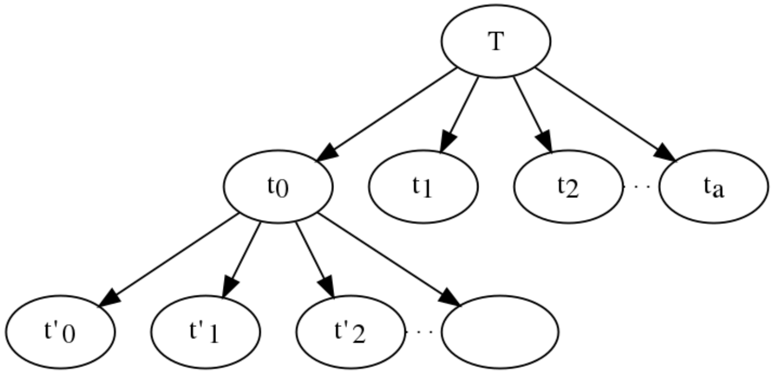
Outline

- ❑ 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 $k$
        solve the input directly without recursion
    else:
        create $a$ sub-problems, each having size $n/b$
        call procedure recursively on each sub-problem
        aggregate the results from the sub-problems
```



Solution tree

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

$f(n)$: remaining cost to divide the problem and aggregate the results

Master Theorem

- **Easily find Θ if $T(n)$ is represented as follows:**

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

  - 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 divide/aggregate

- There is an exact version of Master Theorem.
  - But not discussed in this lecture since it's complicated.
  - **Instead, let's check its approximate (easier) version.**

Master Theorem [Approx. Version]

- $T(n) = aT(n/b) + f(n)$ **is bounded as follows:**
  - $h(n) = n^{\log_b a}$ is the cost for solving all of problems of size 1.
    - See the appendix for details.
  - **Case 1)** $\lim_{n\to\infty} \frac{f(n)}{h(n)} = 0 \Rightarrow T(n) = \Theta(h(n))$
  - **Case 2)** $\lim_{n\to\infty} \frac{f(n)}{h(n)} = \infty$ & $af\left(\frac{n}{b}\right) < f(n) \Rightarrow T(n) = \Theta(f(n))$
  - **Case 3)** $\frac{f(n)}{h(n)} = \Theta(1) \Rightarrow T(n) = \Theta(h(n)\log_2 n)$

Proving the Master theorem is beyond this course (it's graduate level)

Examples (1)

- **Case 1**: $T(n) = 2T\left(\frac{n}{3}\right) + c$
  - $a = 2$ and $b = 3$.
  - $h(n) = n^{\log_3 2}$and $f(n) = c$.

$$\lim_{n\to\infty} \frac{f(n)}{h(n)} = \lim_{n\to\infty} \frac{c}{n^{\log_3 2}} = 0$$

  - Thus, $T(n) = \Theta\big(h(n)\big) = \Theta\big(n^{\log_3 2}\big)$.

Examples (2)

- **Case 2**: $T(n) = 2T\left(\frac{n}{4}\right) + n$
  - $a = 2$ and $b = 4$.
  - $h(n) = n^{\log_4 2} = \sqrt{n}$ and $f(n) = n$.

$$\lim_{n\to\infty} \frac{f(n)}{h(n)} = \lim_{n\to\infty} \frac{n}{\sqrt{n}} = \infty$$

$$af\left(\frac{n}{b}\right) < f(n) \Rightarrow 2\frac{n}{4} = \frac{n}{2} < n$$

  - Thus, $T(n) = \Theta\big(f(n)\big) = \Theta(n)$.

Examples (3)

- **Case 3**: $T(n) = 2T\left(\frac{n}{2}\right) + n$
  - $a = 2$ and $b = 2$.
  - $h(n) = n^{\log_2 2} = n$ and $f(n) = n$.

$$\frac{f(n)}{h(n)} = 1 = \Theta(1)$$

  - Thus, $T(n) = \Theta(h(n)\log_2 n) = \Theta(n \log_2 n)$.

Master Theorem + Variable Trick

- 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. Version]

❑ Pros

- Easy to 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).
- Hard to apply it when the function does not follow the form.
  - Variable trick can be helpful.

What You Need To Know

❑ Repeated substitution

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

❑ Mathmetical induction

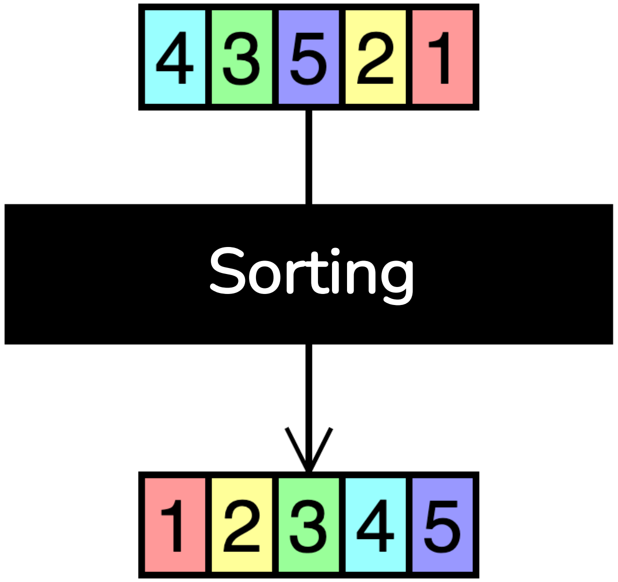
- Estimate the closed solution of a recursive complexity, and then prove it by induction.

❑ Master theorem

- Can solve any function in form of $T(n) = aT(n/b) + f(n)$.

In Next Lecture

- Sorting problem and basic sorting algorithms
  - Selection Sort
  - Bubble Sort
  - Insertion Sort



Thank You

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

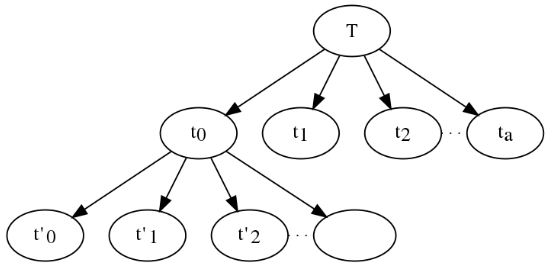
- **Claim: there exists positive integer k s.t. $n \leq 2^k \leq 2n$**
  - Note that $n = 2^{\log_2 n}$ for a natural number n,

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

Appendix: Interpretation of MT (1)

❑ Master Theorem [approximate version]

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



Solution tree

- Suppose $h(n) = n^{\log_b a}$.
  - ⇒ # of leaf nodes in the solution tree.
  - ⇒ # of problems where the input size is 1.
  - ⇒ 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 $n \rightarrow \frac{n}{b} \rightarrow \cdots \rightarrow \frac{n}{b^k}$; when $\frac{n}{b^k} = 1$, it reaches at a leaf
- The number of leaf nodes at level k is $a^k = a^{\log_b n} = n^{\log_b a}$
  - $a^{\log_b n} = x \Leftrightarrow \log_b n \times \log_b a = \log_b x \Leftrightarrow \log_b n^{\log_b a} = \log_b x$

Appendix: Interpretation of MT (2)

Master Theorem [approximate version]

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

- **Case 1)** $\lim_{n\to\infty} \frac{f(n)}{h(n)} = 0 \Rightarrow T(n) = \Theta\big(h(n)\big).$
  - Condition 1) $\lim_{n\to\infty} \frac{f(n)}{h(n)} = 0$ 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 & combining the results at the top level
  - Then, $h(n)$ determines the complexity $T(n)$.
    - The condition is called "the solution tree is leaf-heavy."

Appendix: Interpretation of MT (3)

❑ Master Theorem [approximate version]

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

- **Case 2)** $\lim_{n\to\infty} \frac{f(n)}{h(n)} = \infty$ & $af\left(\frac{n}{b}\right) < f(n) \Rightarrow T(n) = \Theta\big(f(n)\big)$.
  - Condition 1) $\lim_{n\to\infty} \frac{f(n)}{h(n)} = \infty$ 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."

Appendix: Interpretation of MT (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 $T(n)$.
  - It is hard to interpret $\log n$ in this case because $\log n$ is attached during the derivation.