CSE3081 Design and Analysis of Algorithms

Dept. of Computer Engineering,
Sogang University

Chapter 4.
Divide-and-Conquer

Divide-and-Conquer

- Base case: if the problem is small enough, you just solve it directly without recursing

- Recursive case: if the problem is not small, you do the following steps:
  - Divide the problem into one or more subproblems that are smaller instances of the same problem
  - Conquer the subproblems by solving them recursively
  - Combine the subproblem solutions to form a solution to the original problem

Recurrences

- To analyze recursive divide-and-conquer algorithms, we use recurrence.

- Recurrence
  - An equation that describes a function in terms of its value on other, typically smaller, arguments.

- $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

Algorithmic Recurrences

- A recurrence $T(n)$ is algorithmic if, for every sufficiently large threshold constant $n_0 > 0$, the following two properties hold:

- For all $n < n_0$, we have $T(n) = \Theta(1)$.
  - Small problems have constant running time.

- For all $n \geq n_0$, every path of recursion terminates in a defined base case within a finite number of recursive invocations.
  - The recursive algorithm terminates.

Conventions

- Often recurrences are stated without base cases.
- If no base case is given, we assume that the recurrence is algorithmic.
- This means that we are free to pick any sufficiently large threshold constant n_0 for the range of base cases where $T(n) = \Theta(1)$.

- Ceilings and floors in divide-and-conquer recurrences do not change the asymptotic solution → they are often dropped.
  - Recurrence for merge sort is $T(n) = T\left(\left\lceil\frac{n}{2}\right\rceil\right) + T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + \Theta(n)$.
  - But they are simplified to $T(n) = 2\mathrm{T}(\frac{\mathrm{n}}{2}) + \Theta(n)$.

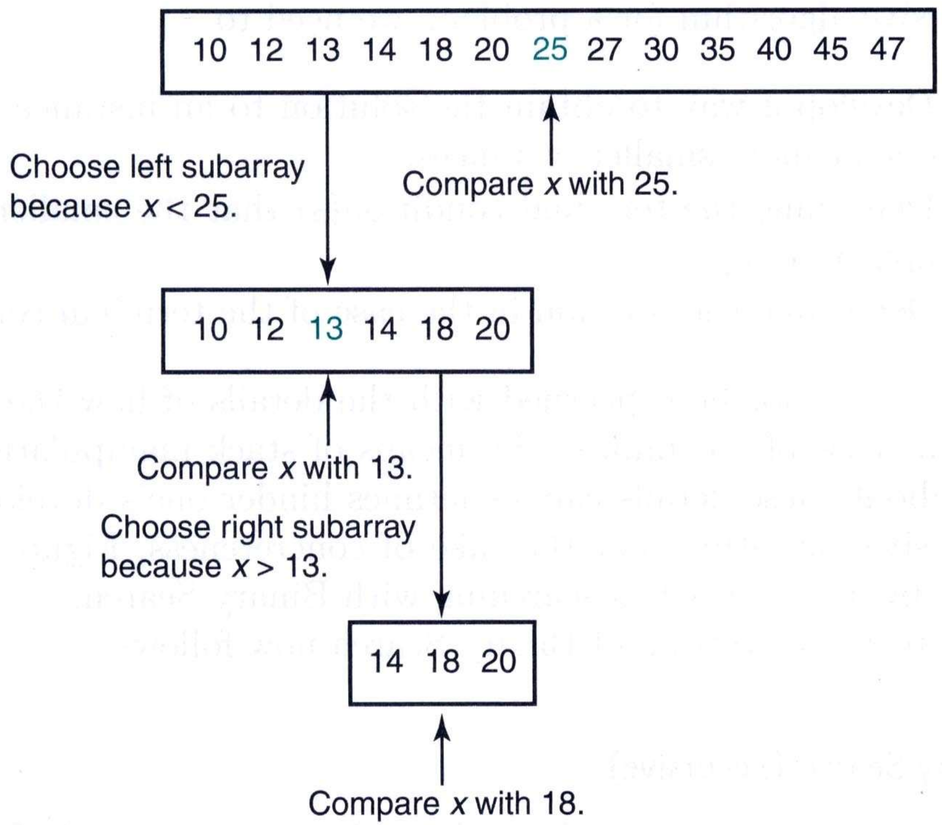
- Sometimes recurrences are inequalities rather than equations.
  - $T(n) \leq 2\mathrm{T}(\frac{\mathrm{n}}{2}) + \Theta(n)$
- Since the recurrence states only an upper bound on $T(n)$, we express the solution using O-notation rather than Θ-notation.

Example: binary search

```
int location(int S[], int low,
                              int high, int x) {
  int mid;

  if (low > high)
    return 0;
  else {
    mid = floor((low+high)/2);
    if (x == S[mid])
      return mid;
    else if (x < S[mid])
      return location(low, mid-1);
    else
      return location(mid+1, high);
  }
}

  int Data[100000];
  …
  result = location(Data, 1, n, key);
  …
```



Example: binary search

- Worst-case time complexity
  - For simplicity, assume n = 2^m (m>0)

$$\boxed{\begin{aligned} T(n) &= T(\tfrac{n}{2}) + c \\ T(1) &= 1 \end{aligned}} \longrightarrow \boxed{T(n) = O(\log n)}$$

  - Also holds for cases where n ≠ 2^m

Example Recurrence Equations

- $T(n) = aT(n-1) + bn$
- $T(n) = T(n/2) + bn \log n$
- $T(n) = aT(n-1) + bn^2$
- $T(n) = aT(n/2) + bn$
- $T(n) = T(n/2) + c \log n$
- $T(n) = T(n/2) + cn$
- $T(n) = 2T(n/2) + cn$
- $T(n) = 2T(n/2) + cn \log n$
- $T(n) = T(n-1) + T(n-2), T(1) = T(2) = 1$

Solving Recurrences

- Substitution Method
  - You guess the form of a bound.
  - Use mathematical induction to prove your guess correct and solve for constants.
- Recursion-Tree Method
  - Model recurrence as a tree whose nodes represent the costs incurred at various levels of recursion.
  - Determine the costs at each level and add them up.
  - Can be helpful in guessing the form of the bound for use in the substitution method.
- Master Theorem
  - If the recurrence is in the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, we can solve the recurrence using the Master theorem.
- Akra-Bazzi Method
  - A general method for solving divide-and-conquer recurrences.
  - Applies to recurrences beyond those solved by the master theorem.

4.1 Multiplying Square Matrices

Matrix Multiplication: Definition

- Two matrices A and B are compatible if the number of columns of A equals the number of rows of B.
  - In general, an expression containing a matrix product AB is always assumed to imply that matrices A and B are compatible.

- If $A = (a_{ik})$ is a $p \times q$ matrix and $B = (b_{kj})$ is a $q \times r$ matrix, then their matrix product $C = AB$ is a $p \times r$ matrix $C = (c_{ij})$, where

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}$$

  - for $i = 1,2, \dots, m$ and $j = 1,2, \dots, p$.

Matrix Multiplication: Problem

- Suppose we have two matrices $A = (a_{ik})$ and $B = (b_{jk})$, and they are both square $n \times n$ matrices.
- The matrix product $C = A \cdot B$ is also an $n \times n$ matrix, where for $i, j = 1,2, \ldots, n$, the (i, j) entry of C is given by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

- For a $n \times n$ matrix, we say the matrix is dense if most of the n^2 entries are not 0.
- We say the matrix is sparse if most of the n^2 entries are 0.

Matrix Multiplication: A Simple Algorithm

- Requires computing n^2 matrix entries, each of which is the sum of n pairwise products of input elements from A and B.

- A straightforward algorithm for matrix multiplication

```
MATRIX-MULTIPLY(A, B, C, n)
for i = 1 to n                      // compute entries in each of n rows
    for j = 1 to n                  // compute n entries in row i
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj   // add in one more term of equation (4.1)
```

- Since we have a triply nested for loops, this algorithm operates in $\Theta(n^3)$ time.

Matrix Multiplication: A Simple Divide-and-Conquer

- To apply divide-and-conquer, we are going to divide the $n \times n$ matrices into four $\frac{n}{2} \times \frac{n}{2}$ submatrices (the divide step).
  - Here we assume that n is an exact power of 2, but this assumption can be relaxed.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

  - We can write the matrix product as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$
$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix}$$

Matrix Multiplication: A Simple Divide-and-Conquer

- We need to calculate the corresponding equations

$$
\begin{aligned}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} , \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} , \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} , \\
C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} .
\end{aligned}
$$

  - This involves eight $\frac{n}{2} \times \frac{n}{2}$ multiplications and four additions of $\frac{n}{2} \times \frac{n}{2}$ submatrices.

Matrix Multiplication: Matrix Partitioning

- First approach
  - Allocate temporary storage to hold A's four submatrices $A_{11}, A_{12}, A_{21}, A_{22}$ and B's four submatrices $B_{11}, B_{12}, B_{21}, B_{22}$.
  - Copy each element in A and B to its corresponding location in the appropriate submatrix.
  - After the recursive conquer step, copy the elements in each of C's four submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ to their corresponding locations in C.
  - This approach takes $\Theta(n^2)$ time, since $3n^2$ elements are copied.
- Second approach
  - We do not copy elements, but use index calculations.
  - A submatrix is specified within a matrix by indicating where within the matrix the submatrix lies without touching any matrix elements.
  - Changes to the submatrix elements update the original matrix.
  - This approach takes $\Theta(1)$ time since, no copying takes place.
- We assume we use the second approach

Matrix Multiplication: The Algorithm

```
MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
if n == 1
// Base case.
    c11 = c11 + a11 · b11
    return
// Divide.
partition A, B, and C into n/2 × n/2 submatrices
    A11, A12, A21, A22; B11, B12, B21, B22;
    and C11, C12, C21, C22; respectively
// Conquer.
MATRIX-MULTIPLY-RECURSIVE(A11, B11, C11, n/2)
MATRIX-MULTIPLY-RECURSIVE(A11, B12, C12, n/2)
MATRIX-MULTIPLY-RECURSIVE(A21, B11, C21, n/2)
MATRIX-MULTIPLY-RECURSIVE(A21, B12, C22, n/2)
MATRIX-MULTIPLY-RECURSIVE(A12, B21, C11, n/2)
MATRIX-MULTIPLY-RECURSIVE(A12, B22, C12, n/2)
MATRIX-MULTIPLY-RECURSIVE(A22, B21, C21, n/2)
MATRIX-MULTIPLY-RECURSIVE(A22, B22, C22, n/2)
```

Matrix Multiplication: Algorithm Analysis

- Let $T(n)$ be the worst-case time to multiply two $n \times n$ matrices using this procedure.
- In the base case, when $n = 1$, just one scalar multiplication and one addition is needed. Thus, $T(1) = \Theta(1)$.
- In the recursive case, we use index calculation based approach to partition the matrices in $\Theta(1)$ time.
- Then, we recursively call the function 8 times.

- Thus, the worst case time complexity of MATRIX-MULTIPLY-RECURSIVE is $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(1)$

- Solving this recurrence relation, we get $T(n) = \Theta(n^3)$.

- The divide-and-conquer algorithm MATRIX-MULTIPLY-RECURSIVE has the <u>same worst-case time complexity</u> as the straightforward MATRIX-MULTIPLY!

4.2 Strassen's Algorithm for Matrix Multiplication

Matrix Multiplication faster than $\Theta(n^3)$

- Strassen's algorithm published in 1969
- runs in $\Theta(n^{\lg 7})$ time. $\Theta(n^{\lg 7}) \approx \Theta(n^{2.81})$
- Instead of performing eight recursive multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices, Strassen's algorithm performs only seven.
- The cost of eliminating one matrix multiplication:
  - several new additions and subtractions of $\frac{n}{2} \times \frac{n}{2}$ matrices, but still only a constant number

Strassen's algorithm: The Idea

- Suppose we have two numbers, x and y. We want to calculate $x^2 - y^2$.
- The straightforward calculation requires two multiplications to square x and y, followed by one subtraction.

- Now, let's recall the following equation.
- $x^2 - y^2 = (x + y)(x - y)$
- Now, we can calculate the same result with a single multiplication and two additions.

- If x and y are scalars the cost of addition(subtraction) and multiplication is not significantly different.
- If x and y are large matrices, the cost of multiplying outweights the cost of adding.

Strassen's Algorithm

- Strassen's algorithm uses a divide-and-conquer strategy.
- Step 1: We divide the input matrices A and B and output matrix C into $\frac{n}{2} \times \frac{n}{2}$ submatrices. This step takes $\Theta(1)$ time, as seen previously.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

- Step 2: We create $\frac{n}{2} \times \frac{n}{2}$ matrices $S_1, S_2, \ldots, S_{10}$, each of which is the sum or difference of two submatrices. This can be done in $\Theta(n^2)$ time.

$$\begin{aligned} S_1 &= B_{12} - B_{22}, & S_6 &= B_{11} + B_{22}, \\ S_2 &= A_{11} + A_{12}, & S_7 &= A_{12} - A_{22}, \\ S_3 &= A_{21} + A_{22}, & S_8 &= B_{21} + B_{22}, \\ S_4 &= B_{21} - B_{11}, & S_9 &= A_{11} - A_{21}, \\ S_5 &= A_{11} + A_{22}, & S_{10} &= B_{11} + B_{12}. \end{aligned}$$

Strassen's Algorithm

- Step 3: We create seven $\frac{n}{2} \times \frac{n}{2}$ matrices $P_1, P_2, \ldots, P_7$ and initialize all entries to zero. This can be done in $\Theta(n^2)$.

- Using the submatrices of input matrices A and B, and $S_1, S_2, \ldots, S_{10}$, we recursively calculate $P_1, P_2, \ldots, P_7$.

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 & (&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22}) , \\
P_2 &= S_2 \cdot B_{22} & (&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22}) , \\
P_3 &= S_3 \cdot B_{11} & (&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11}) , \\
P_4 &= A_{22} \cdot S_4 & (&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11}) , \\
P_5 &= S_5 \cdot S_6 & (&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}) , \\
P_6 &= S_7 \cdot S_8 & (&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}) , \\
P_7 &= S_9 \cdot S_{10} & (&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}) .
\end{aligned}
$$

Strassen's Algorithm

- Step 4: Now we calculate submatrices of C from P matrices.

- $C_{11} = C_{11} + P_5 + P_4 - P_2 + P_6$

$$
\begin{array}{lllllll}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & \\
 & & - A_{22} \cdot B_{11} & & + A_{22} \cdot B_{21} & & \\
 & - A_{11} \cdot B_{22} & & & & - A_{12} \cdot B_{22} & \\
 & & & - A_{22} \cdot B_{22} & - A_{22} \cdot B_{21} & + A_{12} \cdot B_{22} & + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} & & & & & & + A_{12} \cdot B_{21} \, ,
\end{array}
$$

- $C_{12} = C_{12} + P_1 + P_2$

$$
\begin{array}{lll}
A_{11} \cdot B_{12} & - A_{11} \cdot B_{22} & \\
 & + A_{11} \cdot B_{22} & + A_{12} \cdot B_{22} \\
\hline
A_{11} \cdot B_{12} & & + A_{12} \cdot B_{22} \, ,
\end{array}
$$

Strassen's Algorithm

- $C_{21} = C_{21} + P_3 + P_4$

$$
\begin{array}{llll}
A_{21} \cdot B_{11} & + A_{22} \cdot B_{11} & & \\
 & - A_{22} \cdot B_{11} & + A_{22} \cdot B_{21} & \\
\hline
A_{21} \cdot B_{11} & & + A_{22} \cdot B_{21} & ,
\end{array}
$$

- $C_{22} = C_{22} + P_5 + P_1 - P_3 - P_7$

$$
\begin{array}{llllllll}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & & \\
 & - A_{11} \cdot B_{22} & & & + A_{11} \cdot B_{12} & & & \\
 & & - A_{22} \cdot B_{11} & & & - A_{21} \cdot B_{11} & & \\
- A_{11} \cdot B_{11} & & & & - A_{11} \cdot B_{12} & + A_{21} \cdot B_{11} & + A_{21} \cdot B_{12} & \\
\hline
 & & & A_{22} \cdot B_{22} & & & + A_{21} \cdot B_{12} & ,
\end{array}
$$

- Altogether, we need 12 additions (or subtractions) in this step, which takes $\Theta(n^2)$ time.

Strassen's Algorithm: Time Complexity Analysis

- We calculate $T(n)$, the running time of Strassen's algorithm.

- When $n = 1$, the matrix multiplications takes one scalar multiplication plus one scalar addition, which takes $\Theta(1)$ time.

- When $n > 1$, we go through step 1-4. Step 1, 2, 4 takes $\Theta(n^2)$ time, and step 3 requires seven multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices.

- The recurrence of Strassen's algorithm is:
- $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$

- The solution to this recurrence relation is:
- $T(n) = \Theta(n^{\lg 7})$
- This is asymptotically better than $\Theta(n^3)$!

Exercise 4.2-1

- Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$

- Show your work.

4.3 The Substitution Method for Solving Recurrences

Substitution Method

- The most general method for solving recurrences.

- Steps
  - Guess the form of the solution using symbolic constants.
  - Use mathematical induction to show that the solution works, and find the constants.

- To apply the inductive hypothesis, you substitute the guessed solution for the function on smaller values → "substitution method"

Substitution Method

- We need to solve the following recurrence.
- $T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n)$

- We guess that the asymptotic upper bound is $T(n) = O(n \lg n)$.
- We will use the substitution method to prove it.

- The inductive hypothesis
  - $T(n) \le cn \lg n$ for all $n \ge n_0$
  - We will choose the specific constants $c > 0$ and $n_0 > 0$ later.

- Now we need to establish this inductive hypothesis.

Substitution Method

- The inductive hypothesis: if $T(n) \leq cn \lg n$ is true for all $n < k$, then it is also true for $n = k$.

- $$\begin{aligned} T(n) &= 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n) \\ &\leq 2c\frac{n}{2}\lg\frac{n}{2} + \Theta(n) \\ &= cn\lg\frac{n}{2} + \Theta(n) \\ &= cn\lg n - cn\lg 2 + \Theta(n) \\ &= cn\lg n - cn + \Theta(n) \\ &\leq cn\lg n \end{aligned}$$

- The last step holds if we constrain the constants c to be sufficiently large.

Substitution Method

- The base case: since no base case is given, we assume that the recurrence is algorithmic. If we pick $n_0 = 4$, it means that when $n < 4$, $T(n)$ is constant.

- Now we need to see if the inductive hypothesis hold for the base cases.

- $T(2) \leq c\ 2 \lg 2$
- $T(3) \leq c\ 3 \lg 3$

- If we pick c to be $\mathrm{c} = \max\{T(2), T(3)\}$, both inequalities hold.

- Thus, we have proven that $T(n) = O(n \lg n)$.

Substitution Method: Making a Good Guess

- There is no general way to correctly guess the tightest asymptotic solution to an arbitrary recurrence. We can gain intuition from experience.

- Suppose we have the following recurrence.
- $T(n) = 2T\left(\frac{n}{2} + 17\right) + \Theta(n)$
- This is similar to $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$, the recurrence we've already seen.
- The difference is 17, but when n is large, the difference between $\frac{n}{2}$ and $\frac{n}{2} + 17$ is not significantly different.
- So our guess is $T(n) = O(n \lg n)$.

Substitution Method: Making a Good Guess

- We can start from loose lower bound and loose upper bound.

- $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
- We can first establish that $T(n) = \mathrm{O}(\mathrm{n}^2)$, or $T(n) = \Omega(n)$.
- From there, we can tighten the bound by trying $O(n \lg n)$ and $\Omega(n \lg n)$.

A Trick of the Trade: Subtracting a Low-Order Term

- Consider the case: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1)$
- We guess the solution $T(n) = O(n)$, and try to show $T(n) \leq cn$ for $n \geq n_0$.

- Substituting our guess into the recurrence, we obtain
- $T(n) \leq 2\left(c\left(\frac{n}{2}\right)\right) + \Theta(1) = cn + \Theta(1)$
- This does not imply that $T(n) \leq cn$ for any choice of c. So we think we guessed wrong.

- In fact, the original guess $T(n) = O(n)$ is correct and tight.

A Trick of the Trade: Subtracting a Low-Order Term

- Intuitively, our guess is nearly right: we are off only by $\Theta(1)$, a lower-order term.
- Here comes the trick of subtracting a lower-order term from our previous guess.

- $T(n) \leq cn - d$, where $d \geq 0$ is a constant. We now have

$$\begin{aligned} T(n) &\leq 2(c(n/2) - d) + \Theta(1) \\ &= cn - 2d + \Theta(1) \\ &\leq cn - d - (d - \Theta(1)) \\ &\leq cn - d \end{aligned}$$

- as long as we choose d to be larger than the anonymous upper-bound constant hidden by the Θ-notation.
- We must choose the constant c large enough that $cn - d$ dominates the implicit base cases.

Avoiding Pitfalls

- Avoid using asymptotic notation in the inductive hypothesis for the substitution method because it is error prone.

- $$\begin{aligned} T(n) &\leq 2 \cdot O\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n) \\ &= 2 \cdot O(n) + \Theta(n) \\ &= O(n). \quad \leftarrow \text{wrong!} \end{aligned}$$

- The problem with this reasoning is that the constant hidden by the O-notation changes.

- $$\begin{aligned} T(n) &\leq 2\left(c \left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n) \\ &\leq cn + \Theta(n) \end{aligned}$$

- Since $cn + \Theta(n) \leq cn$ is not true, this reasoning is wrong.

Avoiding Pitfalls

- Another fallacious use of substitution method
- $T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n)$

- We guess $T(n) = O(n)$. We need to prove that $T(n) \leq cn$, when $T(k) \leq ck$ is true for all $k < n$.

- $T(n) \leq 2\left(c\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n)$
  $\leq cn + \Theta(n)$
  $= O(n)$ ← wrong!

- We need to explicitly prove $T(n) \leq cn$ in order to show that $T(n) = O(n)$.

4.4 The Recursion-Tree Method for Solving Recurrences

Recursion-Tree Method

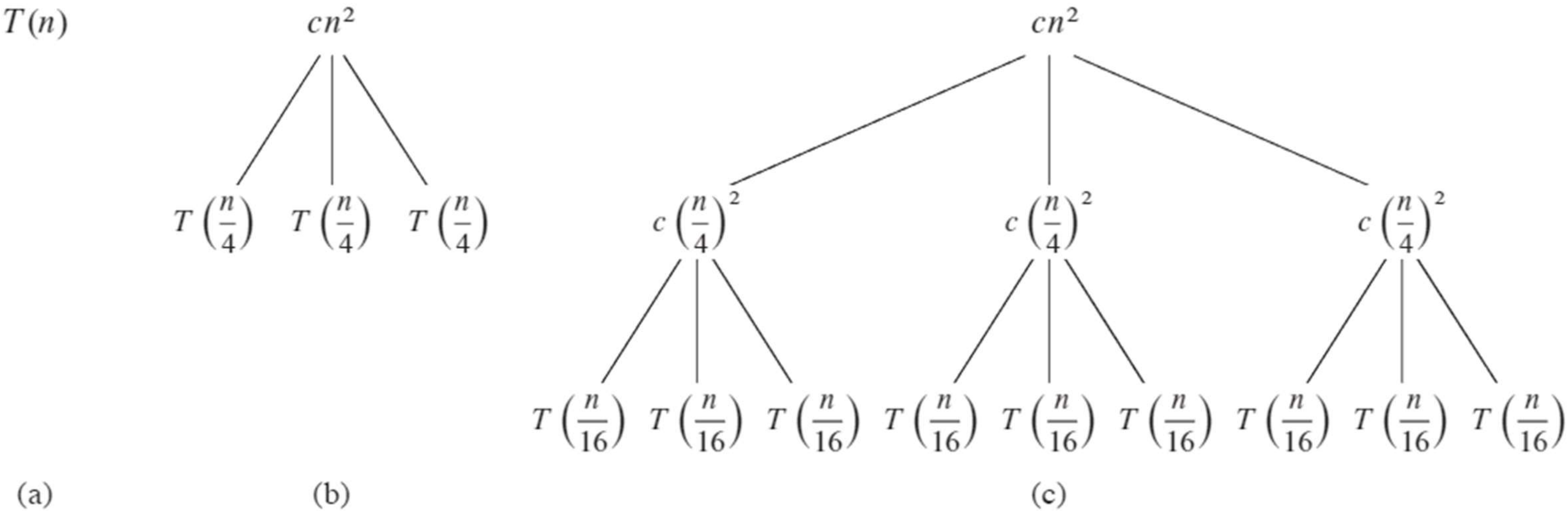
- Drawback of substitution method
  - Might have trouble coming up with a good guess

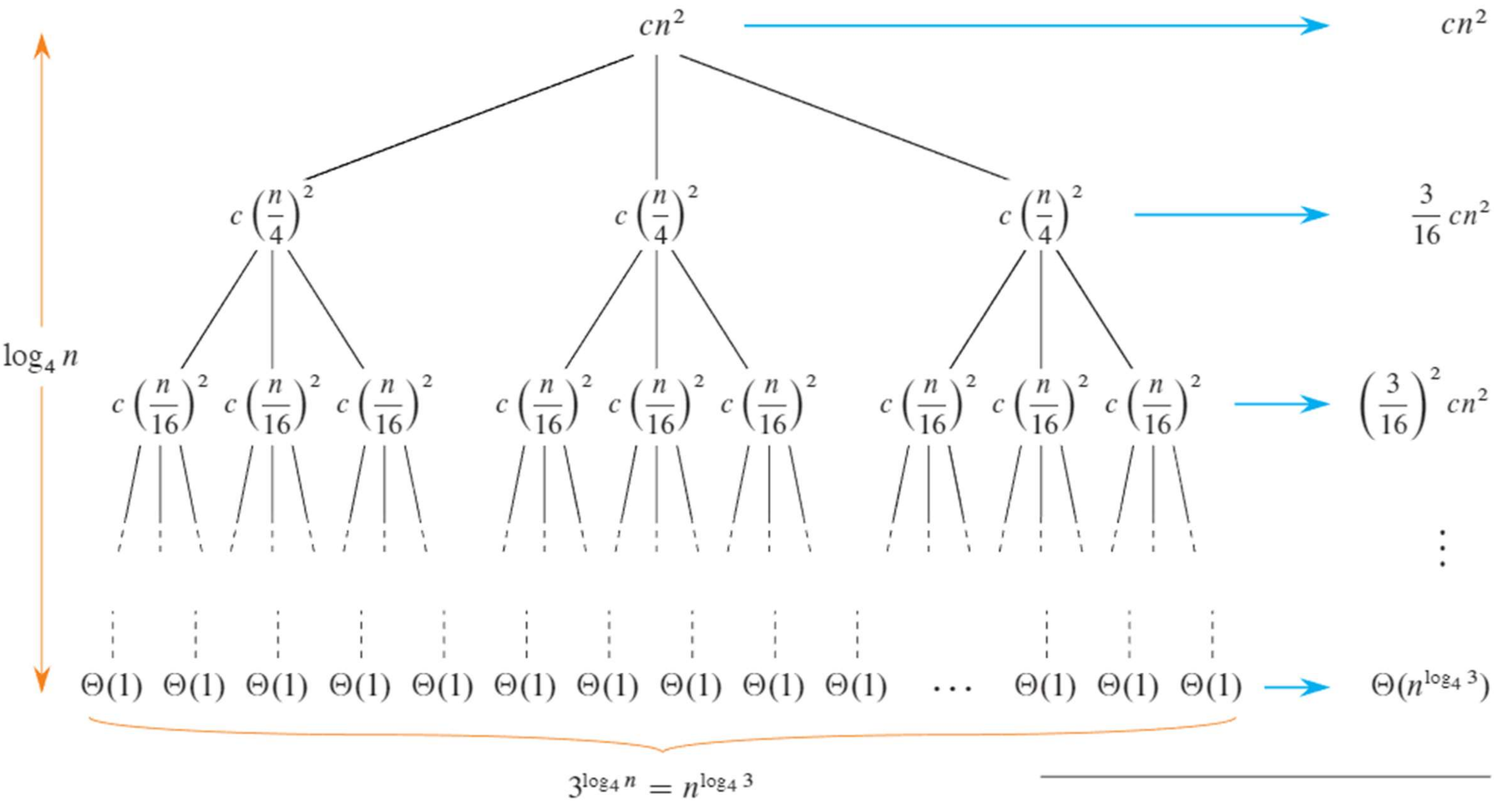
- Recursion tree for a good guess
  - Drawing a recursion tree can help generate intuition for a good guess, which can be verified using the substitution method

- Drawing a recursion tree
  - Each node represents the cost of a single subproblem
  - We sum the costs within each level of the tree to obtain per-level costs
  - Then we sum all the per-level costs to determine the total cost

Recursion Tree: An Illustrative Example

- We want to find out the upper-bound solution to the recurrence
  - $T(n) = 3T\left(\frac{n}{4}\right) + \Theta(n^2)$




cn^2
$c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$
$\frac{3}{16}cn^2$
$\log_4 n$
$c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\dots$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$
$\Theta(n^{\log_4 3})$
$3^{\log_4 n} = n^{\log_4 3}$
Total: $O(n^2)$

(d)

Recursion Tree: An Illustrative Example

- Characteristics of the recursion tree
  - The subproblem sizes decrease by a factor of 4 every time we go down one level.
  - Thus, the recursion must eventually bottom out in a base case where $n < n_0$.
  - By convention, this base case is $T(n) = \Theta(1)$ for $n < n_0$.
    - We can say $T(1) = \Theta(1)$.

- The height of the recursion tree
  - The subproblem size for a node at depth i is $\frac{n}{4^i}$.
  - As we descend from the root, the subproblem size hits $n = 1$ when $\frac{n}{4^i} = 1$.
    - $i = \log_4 n$
  - Thus, the tree has internal nodes at depths $0,1,2, \ldots, \log_4 n - 1$, and leaves at depth $\log_4 n$.

Recursion Tree: An Illustrative Example

- The cost at each level
  - Each level has three times as many nodes as the level above.
  - The number of nodes at depth i is 3^i.
  - Because the subproblem sizes reduce by a factor of 4 for each level further from the root, each internal node at depth $i = 0,1,2, \dots, \log_4 n - 1$ has a cost of $c\left(\frac{n}{4^i}\right)^2$.
  - Multiplying, we see that the total cost of all nodes at a given depth i is $3^i c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2$.
  - The bottom level, at depth $\log_4 n$, contains $3^{\log_4 n} = n^{\log_4 3}$ leaves.
  - Each leaf contributes $\Theta(1)$, leading to a total leaf cost of $\Theta(n^{\log_4 3})$.

Recursion Tree: An Illustrative Example

- The total cost
  - We add up the costs over all levels to determine the costs for the entire tree.

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{1}{1-(\frac{3}{16})} cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$= \frac{16}{13} cn^2 + \Theta\left(n^{\log_4 3}\right) = O(n^2)$$

Recursion Tree: A Tight Bound

- Now we know that the upper bound of $T(n) = 3T\left(\frac{n}{4}\right) + \Theta(n^2)$ is $O(n^2)$.
- We can easily see that $T(n) = \Omega(n^2)$, because the first recursion call contributes a cost of $\Theta(n^2)$.
- Thus, $T(n) = \Theta(n^2)$.

Verification using Substitution Method

- We may use the substitution method to verify whether $T(n) = O(n^2)$ is correct.
- We guess $T(n) = O(n^2)$, and we should show that $T(n) \le dn^2$ for some constant $d > 0$.

$$\begin{aligned} T(n) &\le 3T(n/4) + cn^2 \\ &\le 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\le dn^2 \,, \end{aligned}$$

- The last step holds if we choose $d \ge \left(\frac{16}{13}\right)c$.
- For the base case, let $n_0 > 0$ be a sufficiently large threshold constant that the recurrence is well defined when $T(n) = \Theta(1)$ for $n < n_0$.
- We can pick d large enough that d dominates the constant hidden by the Θ in which case $dn^2 \ge d \ge T(n)$ for $1 \le n < n_0$.

4.5 The Master Method for Solving Recurrences

Master Method

- The master method provides a "cookbook" method for solving algorithmic recurrences of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ where $a > 0$ and $b > 1$ are constants.

- We call $f(n)$ a driving function.

- Many divide-and-conquer algorithms can be analyzed using the master method.

The Master Theorem

- Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals.
- Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.
- Then, the asymptotic behavior of $T(n)$ can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.

2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \log^k n)$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the regularity condition $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Understanding The Master Theorem

- The function $n^{\log_b a}$ is called the watershed function.
- In each of the three cases, we compare the driving function $f(n)$ to the watershed function $n^{\log_b a}$.
- Case 1: the watershed function grows asymptotically faster than the driving function.
- Case 2: the two functions grow at nearly the same asymptotic rate.
- Case 3: the driving function grows asymptotically faster than the watershed function. (opposite of case 1)

Understanding The Master Theorem: Case 1

- In case 1, not only must the watershed function grow asymptotically faster than the driving function, it must grow polynomially faster.
- The watershed function $n^{\log_b a}$ must be asymptotically larger than the driving function $f(n)$ by at least a factor of $\Theta(n^{\epsilon})$ for some constant $\epsilon > 0$.
- In this case, the master theorem says that the solution is $T(n) = \Theta\left(n^{\log_b a}\right)$.

- If we look at the recursion tree, the cost per level grows at least geometrically from root to leaves, and <u>the total cost of leaves dominates the total cost of the internal nodes</u>.

Understanding The Master Theorem: Case 2

- In case 2, the watershed and driving functions grow at nearly the same asymptotic rate.
- More specifically, the driving function grows faster than the waterhed function by a factor of $\Theta(\log^k n)$, where $k \geq 0$.
- The master theorem says that we tack on an extra $\log n$ factor to $f(n)$, yielding the solution $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

- Each level of the recursion tree costs approximately the same - $\Theta\left(n^{\log_b a} \log^k n\right)$ - and there are $\Theta(\log n)$ levels.
- In practice, the most common situation for the case 2 occurs when $k = 0$, in which the case the watershed and driving functions have the same asymptotic growth, and the solution is $T(n) = \Theta(n^{\log_b a} \log n)$.

Understanding The Master Theorem: Case 3

- In case 3, not only must the driving function grow asymptotically faster than the watershed function, it must grow polynomially faster.
- The driving function $f(n)$ must be asymptotically larger than watershed function $n^{\log_b a}$ by at least a factor of $\Theta(n^{\epsilon})$ for some constant $\epsilon > 0$.
- Moreover, the driving function must satisfy the regularity condition, which is $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and sufficiently large n.
- This condition is satisfied by most of the polynomially bounded functions that we are likely to encounter when applying case 3.
  - The regularity condition might not be satisfied if the driving function grows slowly in local areas, yet relatively quickly overall.
- In this case, the master theorem says that the solution is $T(n) = \Theta(f(n))$.

- If we look at the recursion tree, the cost per level drops at least geometrically from root to leaves, and the root cost dominates the cost of all other nodes.

Using the Master Method

- $T(n) = 9T\left(\frac{n}{3}\right) + n$
- $a = 9$ and $b = 3$, and the watershed function $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$.
- Since the driving function $f(n) = n = O(n^{2-\epsilon})$ for any constant $\epsilon \leq 1$, this is case 1 in the master theorem.
- The solution is $T(n) = \Theta(n^2)$.

Using the Master Method

- $T(n) = T\left(\frac{2n}{3}\right) + 1$

- $a = 1$ and $b = \frac{3}{2}$, and the watershed function $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.

- Since $f(n) = 1 = \Theta\left(n^{\log_b a} \log^0 n\right) = \Theta(1)$, we apply case 2 of the master theorem.

- The solution is $T(n) = \Theta(\log n)$.

Using the Master Method

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

- $a = 3$ and $b = 4$, and the watershed function $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$.

- Since $f(n) = n \log n = \Omega(n^{\log_4 3+\epsilon})$, we apply case 3 of the master theorem.

- Regularity condition holds for $f(n)$, because for sufficiently large n, we have that $af\left(\frac{n}{b}\right) = 3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \le \left(\frac{3}{4}\right) n \log n = cf(n)$ for $c = 3/4$.

- The solution is $T(n) = \Theta(n \log n)$.

Using the Master Method

- $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$
- $a = 2$ and $b = 2$, and the watershed function $n^{\log_b a} = n^{\log_2 2} = n$.
- Since $f(n) = n \log n = \Theta\left(n^{\log_b a} \log^1 n\right)$, we apply case 2 of the master theorem.
- The solution is $T(n) = \Theta(n \log^2 n)$.

Using the Master Method: Exercise

- Solve the following recurrences using the master method.

- Merge Sort: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

- Matrix Multiplication - Divide and Conquer: $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(1)$

- Matrix Multiplication - Strassen's algorithm: $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$

When the Master Method Doesn't Apply

- The watershed function and the driving function cannot be asymptotically compared

- There is a gap between cases 1 and 2
  - when $f(n) = o(n^{\log_b a})$, yet the watershed function does not grow polynomially faster than the driving function.

- There is a gap between cases 2 and 3
  - when $f(n) = \omega(n^{\log_b a})$ and the driving function grows more than polylogarithmically faster than the watershed function, but it does not grow polynomially faster.

- Case 3 should be applied but the regularity condition does not hold.

When the Master Method Doesn't Apply: Example

- $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$
- Since $a = 2$ and $b = 2$, the watershed function is $n^{\log_b a} = n$.
- The driving function is $\frac{n}{\log n} = o(n)$, which means it grows asymptotically slowly than the watershed function n.
- However, $\frac{n}{\log n}$ grows only logarithmically slower than n, now polynomially slower.
- More precisely, $\log n = o(n^{\epsilon})$ for any constant $\epsilon > 0$, which means that $\frac{1}{\log n} = \omega(n^{-\epsilon}) = \omega(n^{\log_b a-\epsilon})$.
- Thus, no constant $\epsilon > 0$ exists such that $\frac{n}{\log n} = O(n^{\log_b a-\epsilon})$, which is required for case 1 to apply.
- Case 2 fails to apply as well, since $\frac{n}{\log n} = \Theta(n^{\log_b a}\log^k n)$, where $k = -1$, but k must be nonnegative for case 2 to apply.

When the Master Method Doesn't Apply

- When we cannot use the master method, we need to solve the recurrence using another method, such as the substitution method.

- Akra-Bazzi method is another method that can solve a more general form of recurrences.

- Although the master theorem does not handle some recurrences, it does handle the overwhelming majority of recurrences that tend to arise in practice.

End of Class

Questions?

Instructor office: AS-1013

Email: jso1@sogang.ac.kr