CMPUT 275

Topic 2: The Divide and Conquer Paradigm

Rob Hackman and Xiao-Bo Li

Winter 2024

Reference

These slides are based on material from previous CMPUT 275 courses, and also the textbook "Introduction to Algorithms, 3rd edition" or 4th edition, by Cormen, Leiserson, Rivest, and Stein.

Motivation

Many useful algorithms are recursive in structure. They recurse (call themselves) one or more times to handle a closely related subproblem.

Recursive algorithms typically follow the **divide-and-conquer** paradigm: break the problem into several subproblems, solve the subproblems recursively, combine these solutions.

Definition: Divide-and-Conquer

The divide and conquer paradigm solves a problem recursively.

- **Divide** the problem into subproblem(s).
- **Conquer**
  - Recursive case: solve the subproblem(s) recursively,
  - Base case: solve subproblem(s) directly.
- **Combine** the solutions to the subproblems into the solution for the original problem.

The recursion **bottoms out** when it reaches a base case.

Definition: Recurrences

Recurrence equations characterize the running times, denoted $T(n)$, of divide-and-conquer algorithms.

A **recurrence** is an equation that describes a function in terms of its value on other, typically smaller arguments (inputs).

For example:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

or

$$T(n) = T(2n/3) + T(n/3) + \Theta(n).$$

A recurrence is **well defined** if there is at least one function that satisfies it. Else it is **ill defined**.

Definition: 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 $1 \leq n < n_0$, $T(n) = \Theta(1)$. The number of base cases is finite, they all share the same constant in $\Theta(1)$, but the constant can be very big.
- For all $n \geq n_0$, every path of recursion terminates in a defined base case within a finite number of recursive invocations.

This definition is constrained to values of n for which $T(n)$ is defined.

Also, since $T(n)$ is time, it must be *strictly* bigger than 0, $T(n) > 0$.

Solving Recurrences

Solving a recurrence means to find a close-form asymptotic bound on $T(n)$.

We discuss three methods for solving recurrences:

- The **Substitution method**: guess and use mathematical induction to prove our guess is correct.
- The **Recursion tree method**: convert the recurrence into a tree.
- The **Master method**

Conventions for Algorithmic Recurrences

When a recurrence is stated without an explicit base case, assume it is algorithmic.

- This means the base case can use any sufficiently large threshold n_0.

The asymptotic solution of a recurrence likely does not depend on the choice of the threshold constant n_0.

Conventions for Algorithmic Recurrences (cont. 1)

Floors, ceilings, and boundary conditions usually do not impact the asymptotic bounds.

For example,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

and

$$T(n) = 2T(n/2) + \Theta(n).$$

has the same asymptotic run time. So often recurrences are stated without floors and ceilings.

Conventions for Algorithmic Recurrences (cont. 2)

- When a recurrence is an equality:

$$T(n) = 2T(n/2) + \Theta(n).$$

  its solution is a Θ-bound.
- When a recurrence is a less than inequality

$$T(n) \leq 2T(n/2) + \Theta(n).$$

  its solution is a big O-bound.
- When a recurrence is a greater than inequality:

$$T(n) \geq 2T(n/2) + \Theta(n).$$

  its solution is a big Ω-bound.

The Substitution Method

The substitution method has two steps.

1. Guess the form of the solution.
2. Use strong mathematical induction to show the solution is correct, and find the constants.

Use the substitution method to establish either an upper *or* lower bound, not both at the same time.

Strong Induction for Algorithmic Recurrences

A statement $P(n)$ involving the asymptotic running time is proved as follows:

- Base case: **verify** $P(k)$ for $k_0 \leq k < n_0$ are all bounded by a constant. The base case does not need to start at 1, so $k_0 \geq 1$.
- Inductive hypothesis: **assume** $\exists k,\ n_0 \leq k < n$, such that

$$P(k_0), \ldots, P(k-1), P(k)$$

  are all true. When $k = n_0$, the assumption uses just the base cases.
- Inductive conclusion: **Show** $P(n)$ is true for $n > k$.

Conclude: $P(n)$ is true for $\forall n,\ n_0 \leq n$.

Textbook Notation n_0 and n_0'

Note the textbook sometimes uses n_0 for k_0, n_0' is the threshold for base case. For example, 4th ed p91, when discussing the substitution method:

> *...the recurrence always bottoms out in a constant-sized base case between n_0 and n_0'.*

But also uses n_0 to mean the threshold for the base case. For example, in 4th ed p98, when using the recursion tree to generate a guess,

> *...the recurrence is well defined when $T(n) = \Theta(1)$ for $n < n_0$.*

Their notation is confusing. Our definition of k_0 and n_0 is what you need to know.

Strong Induction for Algorithmic Recurrences: Number of Base Cases

Here are some guidelines for establishing the base case(s).

- Make the inductive hypothesis (guess) first. Suppose $T(k) \leq cg(k)$ is the guess. Use this guess to check the base case input size(s).
- Since $T(k) > 0$, check that $\forall k,\ k_0 \leq k < n_0,\ 0 < g(k)$. If not, then do not use that k.
- If $T(k)$ cannot be expressed in terms of the running time of smaller input sizes, then it is likely one of the base cases. For example, if the recurrence is

$$T(k) = 2T(k/4) + \ldots$$

and if $k = 2$, then

$$T(2) = 2T(0.5) + \ldots$$

But $T(0.5)$ is not the running time of a subproblem. This likely means $k = 2$ is one of the base cases.

Constant for Inductive Hypothesis and Inductive Conclusion

The constant used in the asymptotic bound for both the inductive hypothesis and inductive conclusion must be the same. For example, to show $0 < f(n) \leq cg(n)$ for all $n \geq n_0$, the same c is used for the inductive hypothesis and inductive conclusion.

The Substitution Method: Example

Use the substitution method to establish an asymptotic **upper-bound** on the following recurrence:

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n).$$

Guess $T(n) = O(n \lg n)$. This means we need to prove:

- There exists constants $c > 0$ and $n_0 > 0$ such that $T(n) \leq c \cdot n \lg n$ for all $n \geq n_0$.

As part of the proof, a convenient and helpful choice of c and n_0 also needs to be stated.

The Substitution Method: Example Proof

Proof : Base Case.

Let $k = 1$, then since $\lg 1 = 0$, $T(1) = 0$. But a base case should have constant running time, not 0. Therefore, this case must be removed. If $2 \leq k < 4$, then

$$T(k) \leq c \cdot k \lg k$$

can be made true by picking c such that

$$\max\{T(2), T(3)\} \leq c.$$

This concludes verification of the base case with $n_0 = 4$.

$$T(k) \leq c \cdot k \lg k \text{ if } 2 \leq k < n_0.$$

The Substitution Method: Example Proof (cont. 1)

Proof : Inductive Hypothesis.

The inductive hypothesis is

$$\text{assume } \exists k \ n_0 \leq k < n \ T(k) = O(k \lg k) \text{ for .}$$

Let $n_0 \leq \lfloor n/2 \rfloor < n$, the inductive hypothesis holds for $\lfloor n/2 \rfloor$,

$$T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor.$$

The Substitution Method: Example Proof (cont. 2)

Proof : Inductive Conclusion.

Use this inductive hypothesis in the recurrence for n:

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

This inequality $T(n) \leq c \cdot n \lg n$ is true for all $n \geq n_0$ as long as c is large enough that cn dominates the anonymous function $\Theta(n)$.

The Substitution Method: Example Proof (cont. 3)

The proof for $T(n)$ used the hypothesis for $T(\lfloor n/2 \rfloor)$. This works because

- When $k = n_0 = 4$, $T(4)$ uses $T(2)$, which is the smallest base case.
- When $k = 5 > n_0$, $T(5)$ also uses the base case $T(2)$.
- When $k = 6 > n_0$, $T(6)$ uses the base case $T(3)$.
- When $k = 7$, $T(7)$ uses the base case $T(3)$.
- When $k = 8$, $T(8)$ uses the case $T(4)$, this is the first recursive case.
- Continue with this reasoning for other $k \geq n_0 = 4$.

There is enough base cases such that starting with $n_0 = 4$, each inductive hypothesis for $k \geq n_0$ can be shown to be true using a base case.

In other words, if $k = 3$ is not included as a base case, then we cannot show that $T(6)$ and $T(7)$ are true.

Making a Good Guess

Making a good guess requires experience and creativity. Here are some suggestions

- If a recurrence is similar to one you have seen before, guess a similar solution.
- Prove loose upper and lower bounds first, then reduce the range of uncertainty.

Subtracting a Lower Order Term

Consider the recurrence

$$T(n) = 2T(n/2) + \Theta(1)$$

Guess that $T(n) = O(n)$ and substituting $T(n) \leq cn$ gives:

$$T(n) \leq cn + \Theta(1)$$

This is *not* $T(n) \leq cn$. The guess must be strengthened to $T(n) \leq cn - d$ where $d \geq 0$ is a constant.

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

Constant for Inductive Hypothesis and Inductive Conclusion: Example

Consider the following example where the inductive hypothesis is $T(k) = O(k)$ for $n_0 \leq k < n$

$$T(n) = 2T(n/2) + \Theta(n) \leq O(n) + \Theta(n) = O(n)$$

This substitution cannot lead to the correct inductive conclusion $T(n) = O(n)$ because the inductive conclusion will have a *different constant* than the inductive hypothesis.

Similarly, if the inductive hypothesis is $T(k) \leq ck$, $n_0 \leq k < n$, then

$$T(n) = 2T(n/2) + \Theta(n) \leq cn + \Theta(n) = O(n)$$

is incorrect because the inductive hypothesis and the inductive conclusion are using different constants.

Changing Variables

Algebraic manipulation can make a recurrence simpler. For example:

$$T(n) = 2T(\sqrt{n}) + \lg n$$

can be simplified using the change of variable $m = \lg n$:

$$T(2^m) = 2T(2^{m/2}) + m.$$

This recurrence can be re-expressed as:

$$S(m) = 2S(m/2) + m.$$

The Recursion-Tree Method

In a **recursion-tree**, each node represents the cost of a subproblem. Sum the costs within each level of the tree to obtain a per-level cost, then sum all per-level costs to determine the total cost.

A recursion-tree can be used to generate a good guess for the substitution method or to directly prove the solution to a recurrence.

Recursion-Tree Example

Provide a good guess for an upper-bound solution to the recurrence:

$$T(n) = 3T(n/4) + \Theta(n^2)$$

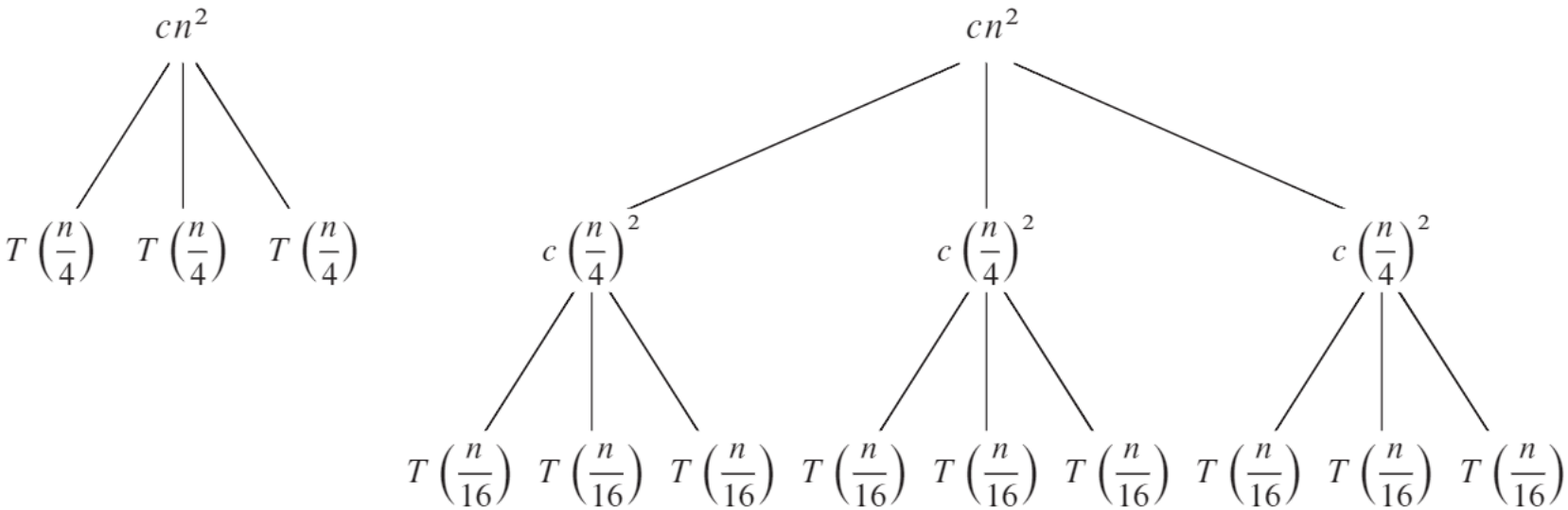
Assume n is an exact power of 4. The base case is

$$T(1) = \Theta(1).$$

Recursion-Tree Example: Branching Out

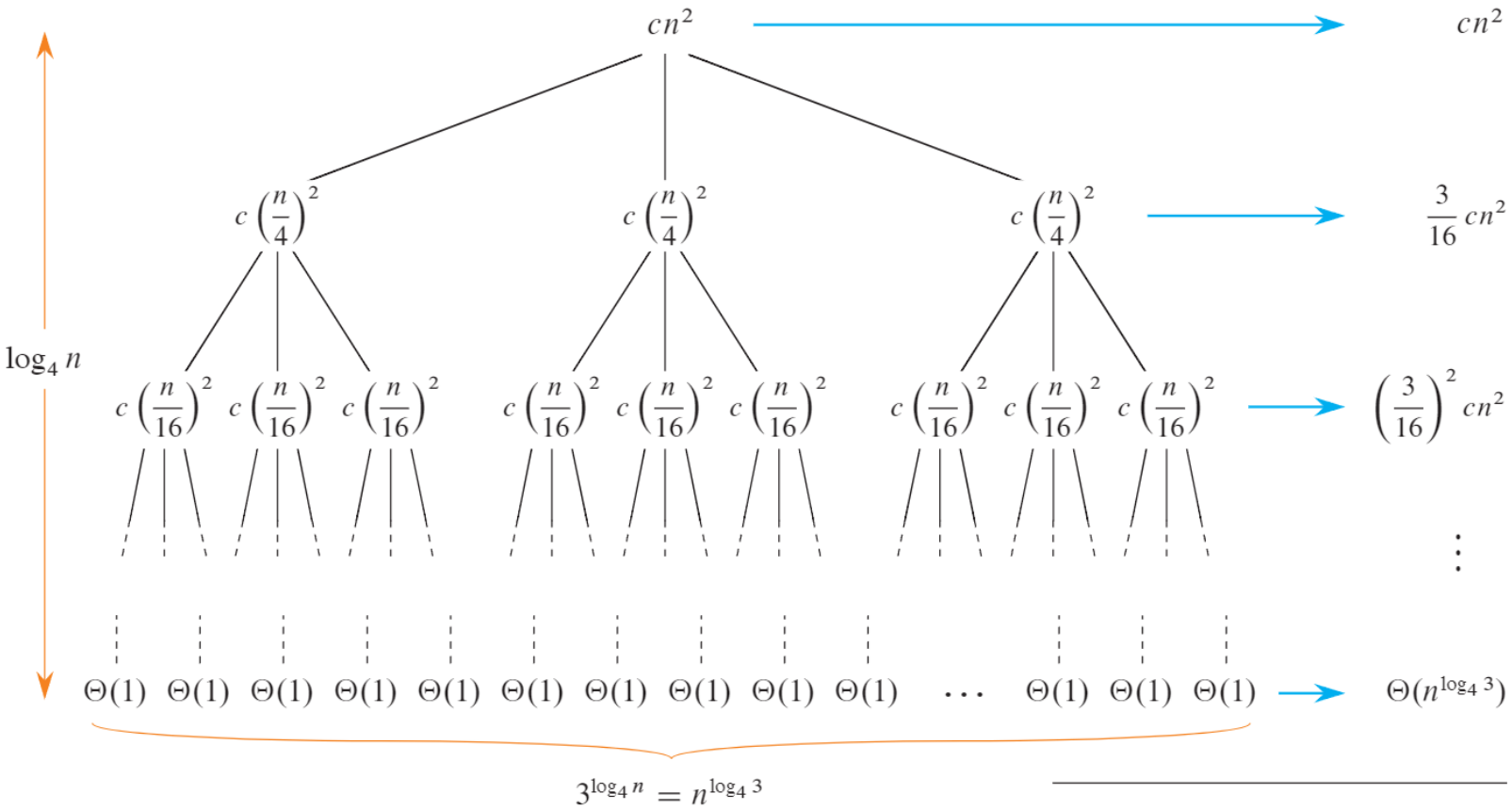
The root has cost cn^2. c represents the constant hidden by the Θ-notation, we don't need to pick a c.

In the first level of the recursion tree, there are three children. These each branch off to three children in the next level.



Recursion-Tree Example: Tree Height

The problem size at level i is $n/4^i$. When the problem size reaches $n/4^i = 1$, the height of this tree is $i = \lg_4 n$.



Recursion-Tree Example: Solve the Recurrence

Level 0 has 1 node, each level has 3 times more nodes, so level i has 3^i nodes. At the lowest level, $i = \lg_4 n$, the number of leaves is

$$3^{\lg_4 n} = n^{\lg_4 3},$$

where $\lg_4 3 < 0.8$.

Cost reduces by a factor of 4, so at level i, each node has cost $c(n/4^i)^2$. The total cost of all nodes at level i is:

$$3^i \cdot c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2$$

At level $\lg_4 n$, each leaf has a cost of $\Theta(1)$, there are $n^{\lg_4 3}$ leaves, so the total leaf cost is $\Theta(n^{\lg_4 3})$.

Recursion-Tree Example: Solve the Recurrence (cont.)

The total cost is the sum of all costs for all levels:

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\lg_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + O(n^2) \\
&= \left(\frac{1}{1-\frac{3}{16}}\right) c \cdot n^2 + O(n^2) \\
&= O(n^2)
\end{aligned}
$$

Verifying the Recursion-Tree

Use the substitution method to verify that $T(n) = O(n^2)$ is an upper-bound for the recurrence

$$T(n) = 3T(n/4) + \Theta(n^2)$$

Verifying the Recursion-Tree: Number of Base Cases

In the following recurrence:

$$T(n) = 3T(n/4) + \Theta(n^2)$$

$T(4)$ needs $T(1)$, so $k = 1$ is the first base case. $T(2)$ and $T(3)$ are also base cases to avoid fractional problem sizes. Therefore, $n_0 = 4$.

Verifying the Recursion-Tree: Proof

Proof : Base Case.

Let $n_0 = 4$ be the threshold constant such that $T(k) = \Theta(1)$ for $1 \leq k < n_0$.

Pick a cosntant δ large enough such that for $1 \leq k < n_0$

$$\delta k^2 \geq \delta \geq \max_{1 \leq k < n_0} T(k) > 0.$$

This verify that the base cases hold: $T(k) = \Theta(1)$ for $1 \leq k < n_0$.

Verifying the Recursion-Tree: Proof (cont. 1)

Proof : Inductive Hypothesis.

Assume that

$$\exists k \ T(k) < dk^2 \ n_0 \leq k < n.$$

Proof : Inductive Conclusion.

An upper-bound for the recurrence

$$T(n) = 3T(n/4) + \Theta(n^2)$$

is established by showing that $T(n) \leq dn^2$ for some n, $n > k$, for the same d used in the inductive hypothesis.

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

Verifying the Recursion-Tree: Proof (cont. 2)

Proof : Inductive Conclusion.

The constant d in the inductive hypothesis and inductive conclusion are the same d.

This inequality holds when

$$\begin{aligned} \frac{3}{16}d + c &\leq d \\ c &\leq \frac{13}{16}d \\ \frac{16}{13}c &\leq d \end{aligned}$$

The Master Method Overview

Suppose an algorithmic recurrence can be written in the form:

$$T(n) = aT(n/b) + f(n),$$

where $a > 0$ and $b > 1$ are constants. $f(n)$ is called the **driving function**. A recurrence of this general form is called a **master recurrence**.

The master recurrence describes the running-time of a divide and conquer algorithms that divides a problem of size n into a subproblems, each subproblem size is $n/b < n$. The driving function is the cost of dividing the problem before the recursion and the cost of combining the results of the recursive solutions.

Ignoring Floors and Ceilings

The master method allows the recurrence to be stated without any floors and ceilings.

We can define the running time $T(n)$ with the assumption that n is a real number rather than an integer.

The Master Theorem

Theorem

Let $a > 0$ and $b > 1$ be constants. Let $f(n)$ be a driving function that is defined, and nonnegative on sufficiently large reals. Let $n \in \mathbb{N}$. Suppose the recurrence relation for $T(n)$ is of the form:

$$T(n) = aT(n/b) + f(n).$$

The asymptotic behaviour of $T(n)$ can be characterized by the following three cases.

Case 1: If $\exists \epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then

$$T(n) = \Theta(n^{\log_b a}).$$

The Master Theorem (cont.)

Theorem (cont.)

Case 2: If $\exists k \geq 0$ *such that* $f(n) = \Theta(n^{\log_b a} \lg^k n)$, *then*

$$T(n) = \Theta(n^{\log_b a} \lg^{k+1} n).$$

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

$$T(n) = \Theta(f(n)).$$

Case 2 is different in the 4th edition of Intro to Algorithms.

Watershed Function

The function $n^{\log_b a}$ is called the **watershed function**. The master theorem compares the watershed function to the driving function.

- Case 1: The watershed function grows asymptotically faster, polynomially, than the driving function by a factor $T(n) = \Theta(n^\epsilon)$. The cost per recursion-tree level grows geometrically from root to leaves, and the leaves dominates the total cost.
- Case 2: The driving function grows faster than the watershed function by factor $\Theta(\lg^k n)$, where $k \geq 0$. The solution for $T(n)$ then adds an extra $\lg n$ factor to $f(n)$, where $\Theta(\lg n)$ is the number of levels. $k = 0$ is common.
- Case 3: The driving function grows asymptotically larger than the watershed function by factor $\Theta(n^\epsilon)$, $\epsilon > 0$. The driving function also must satisfy the regularity condition. Cost per level of the recursion-tree drops geometrically from root to leaves, and the root cost dominates cost of all other nodes.

Proof of the Master Theorem

We will not discuss the proof. You just need to know how to use the theorem.

Using Master Method

Use the following three steps solve a recurrence of the form

$$T(n) = aT(n/b) + f(n),$$

using the master method

1. Justify which case applies.
   - Determine a and b.
   - Calculate the watershed function.
   - Compare the watershed function and the driving function using the master theorem.
2. State which case of the Master Theorem applies.
3. State the solution to the recurrence.

Master Method Example 1

Solve the recurrence

$$T(n) = 9T(n/3) + n$$

Solution:

Since $a = 9$, $b = 3$, $n^{\log_3 9} = \Theta(n^2)$. $f(n) = n = O(n^{2-\epsilon})$ where $\epsilon \leq 1$.

Apply Case 1 of the master theorem to conclude

$$T(n) = \Theta(n^2)$$

Master Method Example 2

Solve the recurrence

$$T(n) = T(2n/3) + 1.$$

Solution:

$a = 1$ and $b = 3/2$ so the watershed function is $n^{\log_{3/2} 1} = n^0 = 1$.

Case 2 applies because $f(n) = \Theta(n^{\log_b a} \lg^0 n) = \Theta(1)$. The solution to the recurrence is:

$$T(n) = \Theta(\lg n).$$

Master Method Example 3

Solve the recurrence

$$T(n) = 3T(n/4) + n \lg n.$$

Solution:

$a = 3$, $b = 4$ so $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$.

Since $n \lg n \in \Omega(n)$, $f(n) = n \lg n = \Omega(n^{\log_4 3 + \epsilon})$ where $0 < \epsilon < 0.2$. Check the regularity condition to see if Case 3 applies.

$$af(n/b) = 3(n/4) \lg(n/4) \leq (3/4) n \lg n = cf(n).$$

Therefore, the regularity condition holds for $c = 3/4$. The solution to the recurrence is

$$T(n) = \Theta(n \lg n).$$

Master Method Example 4

Solve the recurrence

$$T(n) = 2T(n/2) + n \lg n.$$

Solution:

$a = 2,\ b = 2,\ n^{\log_b a} = n^{\log_2 2} = n.$

Since $f(n) = n \lg n$ and $n^{\log_2 2} = n$, $f(n) = \Theta(n^{\log_b a} \lg^1 n)$. Therefore, Case 2 holds with $k = 1 \geq 0$. The solution to the recurrence is:

$$T(n) = \Theta(n \lg^2 n).$$

Master Method Example 5

Solve the recurrence:

$$T(n) = 2T(n/2) + \Theta(n)$$

Solution:

$a = 2$, $b = 2$. $n^{\log_b a} = n^{\log_2 2} = n$. Since $f(n) = \Theta(n \lg^0 n)$, Case 2 applies with $k = 0$. The solution to the recurrence is:

$$T(n) = \Theta(n \lg^{0+1} n) = \Theta(n \lg n).$$

Master Method Example 6

Solve the recurrence:

$$T(n) = 8T(n/2) + \Theta(1).$$

Solution:

$a = 8,\ b = 2.\ n^{\log_b a} = n^{\log_2 8} = n^3.$

The watershed functon n^3 is polynomially larger than the driving function $f(1) = \Theta(1)$, that is $f(n) = O(n^{3-\epsilon})$ for any $0 < \epsilon < 3$. Case 1 applies. The solution to the recurrence is

$$T(n) = \Theta(n^3).$$

Master Method Example 7

Solve the recurrence:

$$T(n) = 7T(n/2) + \Theta(n^2).$$

Solution:

$a = 7$, $b = 2$. $n^{\log_b a} = n^{\log_2 7} = n^{\lg 7}$. Observe that $\lg 7 < 2.81$, so $f(n) = O(n^{\lg 7 - \epsilon})$ where $\epsilon = 0.8$ and $f(n) = \Theta(n^2)$. Case 1 applies.

The solution to the recurrence is:

$$T(n) = \Theta(n^{\lg 7})$$

When the Master Theorem Doesn't Apply

There are situations where you can't use the master theorem.

We will ignore this discussion.

Learning Objective

You are not expected to solve any difficult recurrences. The basic examples discussed here is what you need to know.