Recurrence

Heshan Du
University of Nottingham Ningbo China

March 2025

Aims and Learning Objectives

- To be able to understand and explain the definition of a recurrence.
- To be able to understand and use the substitution method, recursion-tree method and master method to solve recurrences.

Reading

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, Fourth Edition, 2022.
    - Chapter 4.

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, Third Edition, 2009.
    - Chapter 4.

Divide-and-conquer

In divide-and-conquer, we solve a problem recursively, applying three steps:

- **Divide** the problem into one or more subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- **Combine** the subproblem solutions to form a solution to the original problem.

e.g., merge sort and quick sort

Recursive Case and Base Case

- When the subproblems are large enough to solve recursively, we call that the **recursive case**.
- When the subproblems become small enough to solve directly without further recursing, we call that the **base case**.

Recurrence

Recurrences give us a natural way to characterize the running times of recursive or divide-and-conquer algorithms mathematically.

Definition

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

Recurrence of Merge Sort

Let n denote the size of input, $T(n)$ denote the worst-case running time of merge sort. Then the recurrence is:

- if $n = 1$, then $T(n) = \Theta(1)$.
- if $n > 1$, then $T(n) = 2T(n/2) + \Theta(n)$.

The solution to the recurrence is $T(n) = \Theta(n \log n)$

Different Forms of Recurrences

Recurrences can take many forms, e.g.

- $T(n) = T(2n/3) + T(n/3) + \Theta(n)$
- $T(n) = T(n-1) + \Theta(1)$
- $T(n) \leq 2T(n/2) + \Theta(n)$ (states only an upper bound on $T(n)$)
- $T(n) \geq 2T(n/2) + \Theta(n)$ (gives only a lower bound on $T(n)$)

Methods for Solving Recurrences

Three methods for solving recurrences to obain asymptotic O or Θ bounds on the solution:

- *substitution method*: guess a bound and then use mathematical induction to prove the guess correct.
- *recursion-tree method*: converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
- *master method*: provides bounds for recurrences of the form

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

where $a \geq 1$, $b > 1$, and $f(n)$ is a given function.

A Recurrence of the form $T(n) = aT(n/b) + f(n)$

- A recurrence of the form $T(n) = aT(n/b) + f(n)$ characterizes a divide-and-conquer algorithm that creates a subproblems, each of which is $1/b$ times the size of the original problem, using $f(n)$ time for the divide and combine steps.
- Such recurrences arise frequently.

Technicalities in recurrences 1

- In practice, we neglect certain technical details when we state and solve recurrences.
- e.g., call merge sort on n elements when n is odd
- Technically, the recurrence describing the worst-case running time of merge sort is really
  - if $n = 1$, then $T(n) = \Theta(1)$
  - if $n > 1$, then $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$

Technicalities in recurrences 2

- Boundary conditions are typically ignored.
- For convenience, we shall generally omit statements of the boundary conditions of recurrences and assume that $T(n)$ is constant for small n.
- e.g., we normally state the recurrence for merge sort as $T(n) = 2T(n/2) + \Theta(n)$, without explicitly giving values for small n.

Technicalities in recurrences 3

- When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions.
- We forge ahead without these details and later determine whether or not they matter.
- They usually do not, but you should know when they do.
- Experience helps, and so do some theorems (e.g., the master theorem) stating that these details do not affect the asymptotic bounds of many recurrences characterizing divide-and-conquer algorithms.

The Substitution Method for Solving Recurrences

The substitution method for solving recurrences comprises two steps:

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

The Substitution Method: Step 1

We want to determine an upper bound on the recurrence

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

- It is similar to $T(n) = 2T(n/2) + \Theta(n)$.
- We guess that the solution is $T(n) = O(n \log n)$.
- The substitution method requires us to prove that $T(n) \leq cn \log n$ for an appropriate choice of the constant $c > 0$.

The Substitution Method: Step 2

The substitution method requires us to prove that $T(n) \leq cn \log n$ for an appropriate choice of the constant $c > 0$. A skech proof:

- Assume this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$.
- Hence $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$.
- Then $T(n) = 2T(\lfloor n/2 \rfloor) + n \leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n$ (substitution)
- After some calculations, it can be proved that $T(n) \leq cn \log n$.

The Substitution Method: more details of Step 2

$$
\begin{aligned}
T(n) = \ & 2T(\lfloor n/2 \rfloor) + n \\
\leq \ & 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\
\leq \ & cn \log(n/2) + n \\
= \ & cn \log n - cn \log 2 + n \\
= \ & cn \log n - cn + n \\
\leq \ & cn \log n \text{ for every } c \geq 1
\end{aligned}
$$

Base Case

Mathematical induction now requires us to show that our solution holds for the boundary conditions. Typically, we do so by showing that the boundary conditions are suitable as **base cases** for the inductive proof.

- Assume that $T(1) = 1$ is the sole boundary condition of the recurrence. However $T(1) \not\leq c1 \log 1 = 0$.
- Note that asymptotic notation requires us only to prove $T(n) \leq cn \log n$ for $n \geq n_0$, where n_0 is a constant that we get to choose.
- We can replace $T(1)$ by $T(2)$ and $T(3)$ as the base cases in the inductive proof, letting $n_0 = 2$.

Base Case: more details

- With $T(1) = 1$, we derive from the recurrence that $T(2) = 4$ and $T(3) = 5$.
- Now we can complete the inductive proof that $T(n) \leq cn \log n$ for some constant $c \geq 1$ by choosing c large enough so that $T(2) \leq c * 2 \log 2$ and $T(3) \leq c * 3 \log 3$.
- Any choice of $c \geq 2$ suffices for the base cases of $n = 2$ and $n = 3$ to hold.

Making a good guess

- There is no general way to guess the correct solutions to recurrences.
- Guessing a solution takes experience and, occasionally, creativity.
- If a recurrence is similar to one you have seen before, then guessing a similar solution is reasonable. e.g.,
  $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$
- Another way to make a good guess is to prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainty.
- You can use some heuristics, recursion trees, etc.

Exercise

- Solve the recurrence $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$ by using the substitution method.
- Solve the recurrence $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$ by using the substitution method.

For answers, see "Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, Third Edition, 2009. Chapter 4. "

The Recursion-tree Method for Solving Recurrences

- In a recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.
- We sum the costs within each level of the tree to obtain the per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.
- A recursion tree is best used to generate a good guess, which you can then verify by the substitution method.
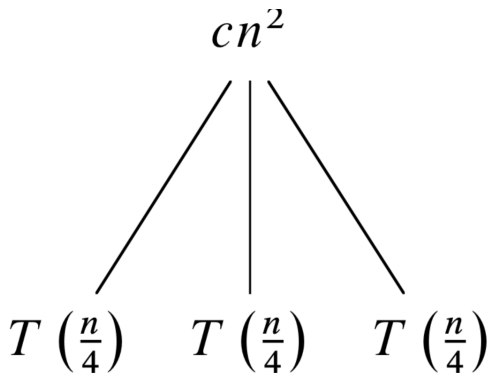
An Example

Let us see how a recursion tree would provide a good guess for the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

- We start by focusing on finding an upper bound for the solution.
- Floors and ceilings usually do not matter when solving recurrences.
- So we create a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$, having written out the implied constant coefficient $c > 0$.
- For convenience, we assume that n is an exact power of 4 (an example of tolerable sloppiness) so that all subproblem sizes are integers.
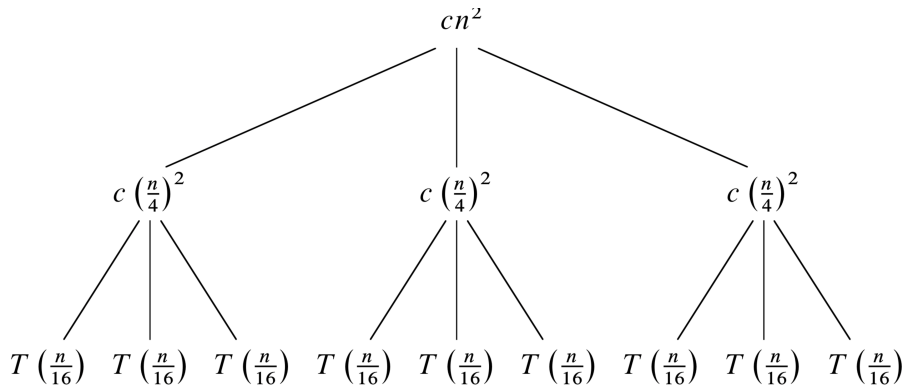
$T(n)$

cn^2

$$T\left(\frac{n}{4}\right) \quad T\left(\frac{n}{4}\right) \quad T\left(\frac{n}{4}\right)$$

$$cn^2$$

$$c\left(\frac{n}{4}\right)^2 \quad c\left(\frac{n}{4}\right)^2 \quad c\left(\frac{n}{4}\right)^2$$

$$T\left(\frac{n}{16}\right) \quad T\left(\frac{n}{16}\right) \quad T\left(\frac{n}{16}\right) \quad T\left(\frac{n}{16}\right) \quad T\left(\frac{n}{16}\right) \quad T\left(\frac{n}{16}\right) \quad T\left(\frac{n}{16}\right) \quad T\left(\frac{n}{16}\right) \quad T\left(\frac{n}{16}\right)$$

cn^2 .... cn^2

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

$\log_4 n$

$c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ .... $\left(\frac{3}{16}\right)^2 cn^2$

$\vdots$

$T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $\cdots$ $T(1)$ $T(1)$ $T(1)$ .... $\Theta(n^{\log_4 3})$

$n^{\log_4 3}$
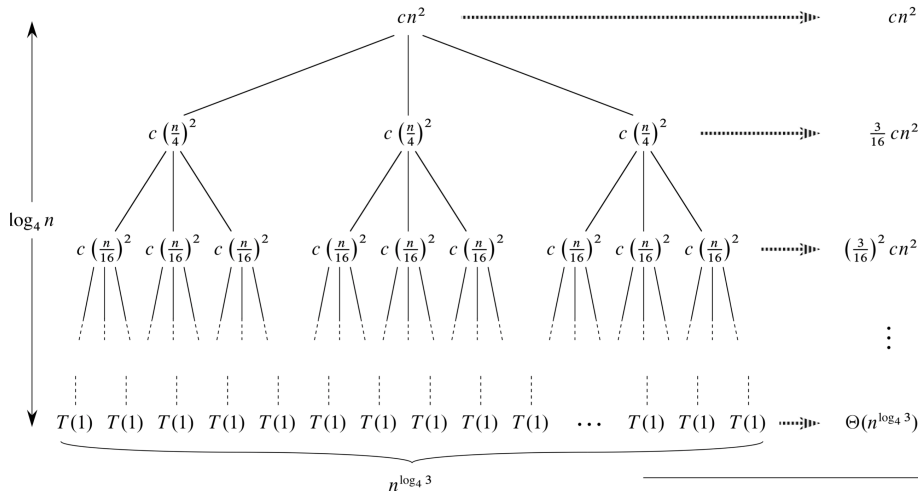
(d)

Total: $O(n^2)$

Cost Analysis 1

- Because subproblem sizes decrease by a factor of 4 each time we go down one level, we eventually must reach a boundary condition.
- The subproblem size for a node at depth i is $n/4^i$.
- The subproblem size is 1 when $n/4^i = 1$, i.e., $i = \log_4 n$.
- Thus, the tree has $\log_4 n + 1$ levels (at depths $0, 1, 2, \ldots, \log_4 n$).

Cost Analysis 2

Now we determine the cost at each level of the tree.

- Each level has three times more nodes than the level above.
- So the number of nodes at depth i is 3^i.
- Each node at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, has a cost of $c(n/4^i)^2$.
- The total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, is $3^i c(n/4^i)^2 = (3/16)^i cn^2$.
- The bottom level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes (to prove they are equal, take $\log_4$ on both sides), each contributing cost $T(1)$, for a total cost of $n^{\log_4 3} T(1)$, which is $\Theta(n^{\log_4 3})$.

Cost Analysis 3

Now we add up the costs over all levels to determine the cost for the entire tree:

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

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

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

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

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

$$= O(n^2)$$

Thus, we have derived a guess of $T(n) = O(n^2)$ for our original recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$.

Cost Analysis 4

- In fact, if $O(n^2)$ is indeed an upper bound for the recurrence (it can be verified using the substitution method), then it must be a tight bound.
- The first recursive call contributes a cost of $\Theta(n^2)$, and so $\Omega(n^2)$ must be a lower bound for the recurrence.

Exercise

- Use the substitution method to verify that $T(n) = O(n^2)$ is an upper bound for the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$.
- Use the recursion tree method to guess a solution of $T(n) = T(n/3) + T(2n/3) + O(n)$, then use the substitution method to verify your guess.

The Master Method for Solving Recurrences

The master method provides a 'cookbook' method for solving recurrences of the form

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

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

- The recurrence describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b, where a and b are positive constants.
- The a subproblems are solved recursively, each in time $T(n/b)$.
- The function $f(n)$ encompasses the cost of dividing the problem and combining the results of the subproblems.

Some Notes for the Master Method

- As a matter of technical correctness, the recurrence is not actually well defined, because n/b might not be an integer.
- However, replacing each of the a terms $T(n/b)$ with either $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ will not affect the asymptotic behavior of the recurrence.

The Master Theorem

Theorem (Master Theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.*

2. *If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.*

3. *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

Intuition of Master Theorem

In each of the three cases, we compare the function $f(n)$ with the function $n^{\log_b a}$ (**the watershed function**).

1. If the watershed function grows asymptotically faster, then Case 1 applies.
2. Case 2 applies if the two functions grow at nearly the same asymptotic rate.
3. If the function $f(n)$ grows asymptotically faster, then Case 3 applies.

More Explanations 1

- In Case 1, $f(n)$ should be polynomially smaller (i.e., asymptotically smaller than $n^{\log_b a}$ by a factor of n^ϵ for some constant $\epsilon > 0$).
- In Case 2, $f(n)$ grows faster than $n^{\log_b a}$ by a factor of $\Theta(\log^k n)$, where $k \geq 0$.
- In Case 2, each level of the recursion tree costs approximately the same - $\Theta(n^{\log_b a} \log^k n)$ -and there are $\Theta(\log n)$ levels.
- In practice, the most common situation for Case 2 occurs when $k = 0$.

- In Case 3, $f(n)$ should be polynomially larger than $n^{\log_b a}$ and also satisfy the 'regularity' condition that $af(n/b) \leq cf(n)$. This condition is satisfied by most of the polynomially bounded functions that we shall encounter.
- For Case 3, if we look at the recursion tree, the cost per level drops at least geometrically from the root to the leaves, and the root cost deominates the cost of all other nodes.

- There is a gap between cases 1 and 2 when $f(n)$ is in $o(n^{\log_b a})$, yet the watershed function does not grow polynomially faster than $f(n)$.
- There is a gap between cases 2 and 3 when $f(n)$ is in $\omega(n^{\log_b a})$ (i.e., $n^{\log_b a}$ is in $o(f(n))$) and $f(n)$ grows more than polylogarithmically faster than the watershed function, but it does not grow polynomially faster.

If the function $f(n)$ falls into one of these gaps, or if the regularity condition in case 3 fails to hold, then you need to use something other than the master method to solve the recurrence.

Example 1

Solve the recurrence $T(n) = 9T(n/3) + n$.

Example 2

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

Example 3

Solve the recurrence $T(n) = 3T(n/4) + n \log n$.

Example 4

Solve the recurrence $T(n) = 2T(n/2) + n \log n$.

Example 5

Solve the recurrence $T(n) = 2T(n/2) + n/\log n$.

Use the master theorem to solve the following recurrences:

1. $T(n) = 2T(n/2) + \Theta(n)$
2. $T(n) = 8T(n/2) + \Theta(n^2)$
3. $T(n) = 7T(n/2) + \Theta(n^2)$