Solving Recurrences

CS313E - Elements of Software Design

Kia Teymourian

05/11/2022

Agenda

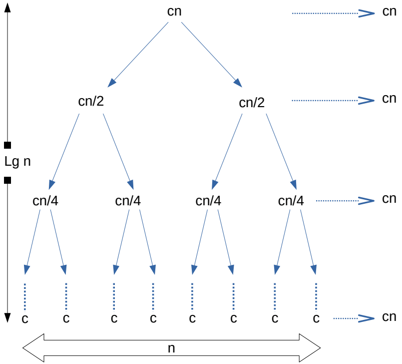
Recurrences - Running time of divide-and-conquer algorithms

- Recurrences provide us a good way to characterize the running time of divide-and-conquer algorithms.
- A recurrence is an equation or inequality to provide a function in terms of its value on smaller input sizes.

$$T(n) = C_1 + 2T(n/2) + C_2 n$$

- C_1 : cost of divide
- $2T(n/2)$ cost of recursion
- $C_2 n$ merge

Recurrence Tree Visualization - Merge Sort



- We stat with cn because it dominates the costs
- Number of leaves: n
- Number of levels: $1 + lg(n)$

$$\boldsymbol{T(n) = (1 + lg(n)) \times cn = \Theta(n\ lg(n))}$$

Solving Recurrences

We have 3 main techniques for solving recurrences which means converting them to their closed forms:

1. Visualization - Recurrences Trees
2. Substitution Method (aka Induction)
3. The Master Method (aka Master Theorem)

Substitution Method

- The substitution method or induction is the most general of the three methods.
- It requires to make an educational guess what the solution might be.
- We then can verify by doing a set of mathematical induction steps.
- The guess can be obtained using recursion trees or experience.

Example

Substitution Method

Solve the recurrence: $T(n) = 2T(\lfloor n/2 \rfloor) + n$

Our solution guess is: $T(n) = O(nlg(n))$

Example

Substitution Method

Solve the recurrence: $T(n) = 2T(\lfloor n/2 \rfloor) + n$

Our solution guess is: $T(n) = O(nlg(n))$

- Based on the substitution method we need to prove $T(n) \leq cnlg(n)$ for a constance $c > 0$
- We assume this bond should hold for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor lg(\lfloor n/2 \rfloor)$

We can then do:

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor lg(\lfloor n/2 \rfloor) + n) \\ &\leq cn\ lg(n/2) + n \\ &= cn\ lg(n) - cn\ lg(2) + n \\ &= cn\ lg(n) - cn + n \\ &\leq cn\ lg(n) \text{ holds as long as } c \geq 1 \end{aligned}$$

Substitution - Example (Continued)

$\boldsymbol{T(n) \leq cn\ lg(n)}$ where the last step holds as long as $\boldsymbol{c \geq 1}$.
Mathematical induction now requires us to show that our solution holds for the boundary conditions.

- We show that the boundary conditions are suitable as base cases for the inductive proof.
- We must show that we can choose the constant c large enough so that the bound $\boldsymbol{T(n) \leq cnlg(n)}$ works for the boundary conditions as well.

This leads sometimes to problems.

Substitution - Example (Continued)

$\boldsymbol{T(n) \leq cn\ lg(n)}$ where the last step holds as long as $\boldsymbol{c \geq 1}$.
Mathematical induction now requires us to show that our solution holds for the boundary conditions.

- We show that the boundary conditions are suitable as base cases for the inductive proof.
- We must show that we can choose the constant c large enough so that the bound $\boldsymbol{T(n) \leq cnlg(n)}$ works for the boundary conditions as well.

This leads sometimes to problems.

- Assume $\boldsymbol{T(1) = 1}$ is the sole boundary condition of the recurrence.
- For $\boldsymbol{n = 1, T(n) \leq cn\ lg(n)}$, yields, $\boldsymbol{T(1) \leq c1lg(1) = 0}$, which is at odds with $T(1) = 1$
  The base case of our inductive proof fails to hold.
  We can solve this obstacle in proving an inductive hypothesis for a specific boundary condition.

Substitution - Example (Continued)

We can solve this obstacle in proving an inductive hypothesis for a specific boundary condition.

- Use asymptotic notation that requiring us only to prove $\boldsymbol{T(n) \leq cnlgn}$ **, where** $\boldsymbol{n \geq 0}$ is a constant that we get to choose.
- We can remove $\boldsymbol{T(1) = 1}$ from our inductive proof.

Substitution - Example (Continued)

We can solve this obstacle in proving an inductive hypothesis for a specific boundary condition.

- Use asymptotic notation that requiring us only to prove $\boldsymbol{T(n) \leq cnlgn}$ **, where** $\boldsymbol{n \geq 0}$ is a constant that we get to choose.
- We can remove $\boldsymbol{T(1) = 1}$ from our inductive proof.

Distinct between the base case of the recurrence ($n = 1$) and the base cases of the inductive proof ($n = 2$ and $n = 3$).

Substitution - Example (Continued)

We can solve this obstacle in proving an inductive hypothesis for a specific boundary condition.

- Use asymptotic notation that requiring us only to prove $\boldsymbol{T(n) \le cnlgn}$ **, where** $\boldsymbol{n \ge 0}$ is a constant that we get to choose.
- We can remove $\boldsymbol{T(1) = 1}$ from our inductive proof.

Distinct between the base case of the recurrence ($n = 1$) and the base cases of the inductive proof ($n = 2$ and $n = 3$).

- With $T(1) = 1$, we can drive from the recurrence $\boldsymbol{T(2) = 4}$ **and** $\boldsymbol{T(3) = 5}$. (We had $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$)
- With the above, we can complete the inductive proof that $T(n) \le cnlgn$ for some constant $\boldsymbol{c \ge 1}$ by choosing c large enough so that $\boldsymbol{T(2) \le c2lg2}$ **and** $\boldsymbol{T(3) \le c3lg3}$.

Any choice of $c \ge 2$ would be sufficient for the base cases of $n = 2$ and $n = 3$ to hold.

Avoiding pitfalls

- Sometimes your guess is correct but somehow the math fails in the induction.
- **The inductive assumption is not strong enough to prove the detailed bound.**

Solution: Revise the guess by subtracting a lower-order term, then the math often goes through.

Example

For the recurrence
$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$

- For guess $O(n)$, we have to show $T(n) \leq cn$. In induction, we get $T(n) = cn + 1$ which does not imply $T(n) \leq cn$ for any choice of c.
- We overcome our difficulty by subtracting a lower-order term from our previous guess.
  Our new guess is $O(n) - d$, or $T(n) \leq cn - d$ where constant $d \geq 0$, can lead to

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 \\ &= cn - 2d + 1 \\ &\leq cn - d \text{ where } d \geq 0 \end{aligned}$$

Making a good guess

- Unfortunately, there is no general way to guess the correct solutions to recurrences. (A good guess takes experience and creativity)
- You can use some heuristics to help you become a good guesser.
- You can also use recursion trees.

The Master Method

The Master Method

The master method provides a **"Cookbook"** method for solving recurrences of the form

$$\boldsymbol{T(n) = aT(\frac{n}{b}) + O(n^d)}$$

Then the solution is:

$$T(n) = \begin{cases} O(n^d log(n)) & \text{if } a = b^d \text{ — Case 1} \\ O(n^d) & \text{if } a < b^d \text{ — Case 2} \\ O(n^{log_b a}) & \text{if } a > b^d \text{ — Case 3} \end{cases}$$

Parameters are:

- ▷ $\boldsymbol{a}$ is the number of recursive calls, or the number of sub-problems that we solve in our recurse algorithm
- ▷ $\boldsymbol{b}$ is the factor of the sub-problems or the factor that we divide the n size of the main problem into smaller problems
- ▷ $\boldsymbol{d}$ is the exponent of the running time outside of the recursive calls.

Merge Sort Example

Example

Merge Sort

Using the master method, we need to identify the 3 parameters a, b, d.

- a is the number of recursive calls, in merge sort we do 2 recursive calls so $a = 2$
- b is the factor size of the subproblem, in merge sort, the sub-problems have size of $n/2$ so $b = 2$
- In merge sort, we do merge of the results, so that d the exponent of the running time outside of the recursive calls is $d = 1$

So we plug in the above factors into the master method.
and we have $a = 2 = 2^1$, so that $T(n) = O(n^d log(n))$
as we knew it before.
$T(n) = O(n^d log(n)) = O(nlog(n))$

Strassen's Method

Example

Strassen's Method

Using the master method, we need to identify the 3 parameters a, b, d.

- a is the number of recursive calls, in Strassen's Method we do 7 recursive calls so $a = 7$, we have $(P_1, \ldots, P_7)$
- b is the factor size of the subproblem, in Strassen's Method, the sub-problems have size of $n/2$ so $b = 2$ with respect to the dimensions of matrices
- We do merge of the results, d the exponent of the running time outside of the recursive calls is $d = 2$

So we plug in the above factors into the master method.

$a = 7$

$b^d = 2^2 = 4 < a$ — Case 3

$T(n) = O(n^{log_b a})$

$T(n) = O(n^{log_2 7}) = O(n^{2.81})$

And we know that the naive algorithm was $O(n^3)$, so the Strassen's algorithm is faster.

A Case 2 Example

Example

Recurrence examples

$T(n) = 2T(n/2) + n^2$

So we plug in the above factors into the master method.

$a = 2$

$b = 2$

$d = 2$

$b^d = 2^2 = 4 > 2$

So, we have

$T(n) = O(n^d) = O(n^2)$

Additional Example - 1

$$\boldsymbol{T(n) = aT(\frac{n}{b}) + O(n^d)}$$

$$T(n) = \begin{cases} O(n^d log(n)) & \text{if } a = b^d \text{ — Case 1} \\ O(n^d) & \text{if } a < b^d \text{ — Case 2} \\ O(n^{log_b a}) & \text{if } a > b^d \text{ — Case 3} \end{cases}$$

Example

Recurrence examples

$T(n) = 7T(n/2) + n^2$

$a =?$
$b =?$
$d =?$
Which Case?

Additional Example -2

$$\boldsymbol{T(n) = aT(\frac{n}{b}) + O(n^d)}$$

$$T(n) = \begin{cases} O(n^d log(n)) & \text{if } a = b^d \text{ — Case 1} \\ O(n^d) & \text{if } a < b^d \text{ — Case 2} \\ O(n^{log_b a}) & \text{if } a > b^d \text{ — Case 3} \end{cases}$$

Example

Recurrence examples

$T(n) = 7T(n/3) + n^2$

a =?
b =?
d =?
Which Case?

Additional Example - 3

$$T(n) = aT(\frac{n}{b}) + O(n^d)$$

$$T(n) = \begin{cases} O(n^d log(n)) & \text{if } a = b^d \text{ — Case 1} \\ O(n^d) & \text{if } a < b^d \text{ — Case 2} \\ O(n^{log_b a}) & \text{if } a > b^d \text{ — Case 3} \end{cases}$$

Example

Recurrence examples
$T(n) = 2T(n/4) + \sqrt{n}$

$a =?$
$b =?$
$d =?$
Which Case?

Readings from CLRS Book (Introduction to Algorithms, 3rd Edition)

- Section 4.3 The substitution method for solving recurrences
- Section 4.4 The recursion-tree method for solving recurrences
- Section 4.5 The master method for solving recurrences