Recurrence Relations

Methods to Solve Recurrences, Solving Recurrence Relations

Rashed Hassan Siam
Department of CSE
Eastern University

Divide-and-Conquer (Recap)

- Paradigm: Solve a problem recursively by applying three steps at each level:

  1. Divide the problem into a number of subproblems that are smaller instances of the same problem.
  2. Conquer the subproblems by solving them recursively. If small enough (base case), solve directly.
  3. Combine the solutions to the subproblems into the solution for the original problem.

Divide-and-Conquer (Recap)

- Recursive Case: When subproblems are large enough to solve recursively.

- Base Case: When subproblems become small enough that we no longer recurse.

- Examples: Merge Sort , Maximum-subarray problem , Matrix multiplication (Strassen's algorithm).

Recurrences

- Definition: An equation or inequality that describes a function in terms of its value on smaller inputs.

- Application: Give a natural way to characterize the running times of divide-and-conquer algorithms.

- Example (Merge Sort): The worst-case running time T(n) is described by:

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

whose solution is claimed to be T(n) = Θ(n lg n).

Recurrences

- Other Forms:

    1. Unequal split: A recursive algorithm might divide subproblems into unequal sizes, such as a 2/3-to-1/3 split. If the divide and combine steps take linear time, such an algorithm would give rise to the recurrence $T(n) = T(2n/3) + T(n/3) + \Theta(n)$.

    2. Single subproblem: A recursive version of linear search would create just one subproblem containing only one element fewer than the original problem. Each recursive call would take constant time plus the time for the recursive calls it makes, yielding the recurrence $T(n) = T(n-1) + \Theta(1)$.

Methods for Solving Recurrences

- Goal: Obtain asymptotic "Θ" or "O" bounds on the solution.

1. Substitution Method: Guess a bound and use mathematical induction to prove the guess is correct. For example – solving the recurrence relation $T(n) = T(n-1) + \Theta(1)$.

2. Recursion-Tree Method: Converts the recurrence into a tree whose nodes represent costs at various levels. Use techniques for bounding summations. For example – solving the recurrence relation of Merge sort, $T(n) = 2T(n/2) + \Theta(n)$.

Methods for Solving Recurrences

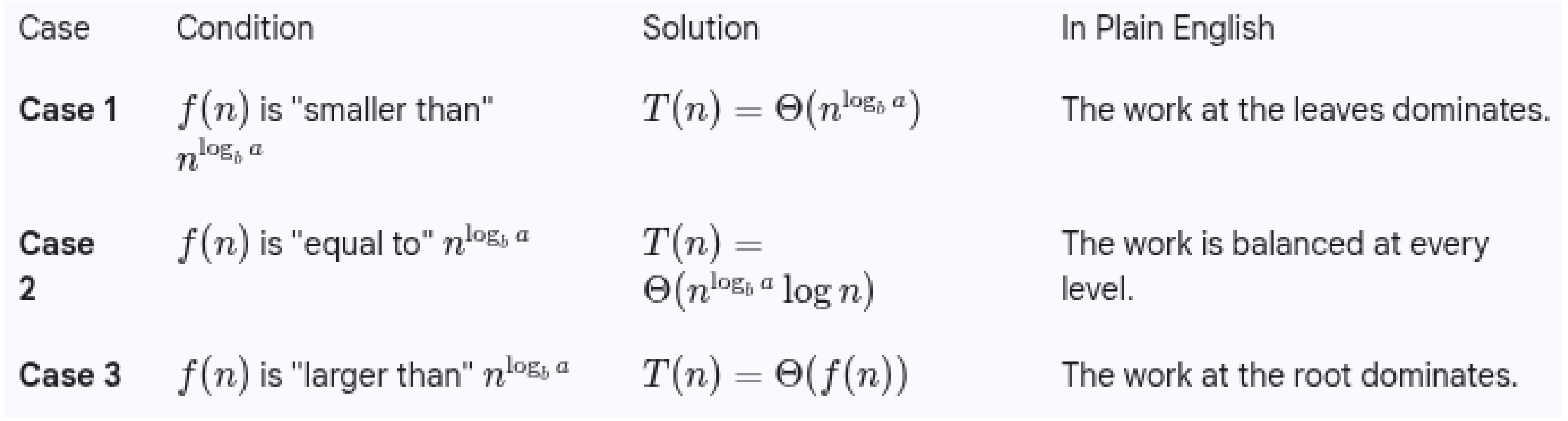
3. Master Method: Provides bounds for recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$, $b > 1$, and $f(n)$ is a given function. For example – solving the recurrence relation of Merge sort, $T(n) = 2T(n/2) + \Theta(n)$.

   - Characterizes algorithms creating **‘a’** subproblems, each **‘1/b’** the size of the original, with **f(n)** as the time for divide and combine steps.

   - You calculate a "critical value" $n^{\log_b a}$ and compare it to f(n). This tells you whether the work is dominated by the leaves of the tree (Case 1), balanced across all levels (Case 2), or dominated by the root (Case 3).

Methods for Solving Recurrences

- The 3 cases of Master method are -



Case	Condition	Solution	In Plain English
Case 1	$f(n)$ is "smaller than" $n^{\log_b a}$	$T(n) = \Theta(n^{\log_b a})$	The work at the leaves dominates.
Case 2	$f(n)$ is "equal to" $n^{\log_b a}$	$T(n) = \Theta(n^{\log_b a} \log n)$	The work is balanced at every level.
Case 3	$f(n)$ is "larger than" $n^{\log_b a}$	$T(n) = \Theta(f(n))$	The work at the root dominates.

Technicalities in Recurrences

- We often neglect technical details like floors, ceilings, and boundary conditions.
    1. Boundary Conditions: We generally omit them and assume T(n) is constant for small n. This typically doesn't change the asymptotic order of growth.
    2. Floors and Ceilings: For example, MERGE-SORT on n elements (odd n) creates subproblems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. We often omit these details as they usually do not affect asymptotic bounds.

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

is normally stated as T(n) = 2T(n/2) + Θ(n).

Which Method Should You Use?

1. Master Method: Always try this first. If your recurrence fits the $aT(n/b) + f(n)$ form, it's by far the fastest.

2. Recursion Tree: Use this when the Master Method doesn't apply or when you want to build intuition. It's fantastic for visualizing the problem and making a good guess.

3. Substitution Method: Use this when you need a formal proof, or when the recurrence is messy (e.g., $T(n) = T(n/2) + T(n/3) + n$) and the other methods fail.

Solving Recurrences: Merge Sort

- Recurrence: $T(n) = 2T(n/2) + O(n)$
- Why: It splits the array into 2 halves ($a=2$), reducing the problem size by 2 ($b=2$). The "merge" step takes linear time, $O(n)$, to combine the sorted halves ($f(n) = n$).
- Solving (Master Method):
  - $a = 2$, $b = 2$, $f(n) = n$
  - Calculate the critical value: $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Compare $f(n)$ to this value: $f(n) = n$, which is asymptotically equal to n.
  - This is Case 2 of the Master Method.
- Solution: $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$.

Solving Recurrences: Quick Sort

This one has two different cases.

A. Average or Best Case: This happens when the pivot chosen is good and splits the array evenly.

- Recurrence: $T(n) = 2T(n/2) + O(n)$
- Why: It splits the array into 2 halves ($a=2$) of size $n/2$ ($b=2$). The "partition" step takes linear time, $O(n)$, to rearrange elements around the pivot ($f(n) = n$).
- Solving (Master Method):
  - This is identical to Merge Sort.
  - $a = 2$, $b = 2$, $f(n) = n$, $n^{\log_b a} = n^{\log_2 2} = n$.
  - This is Case 2.
  - Solution: $T(n) = \Theta(n \log n)$.

Solving Recurrences: Quick Sort

B. Worst Case: This happens when the pivot is the smallest or largest element (e.g., in an already-sorted array).

- Recurrence: $T(n) = T(n-1) + O(n)$
- Why: The array is "split" into one empty part and one part with $(n-1)$ elements. It only makes 1 recursive call $(a=1)$ on a problem of size $(n-1)$. The partition still costs $O(n)$.

Solving Recurrences: Quick Sort

- Solving (Master Method doesn't apply):
  - This $T(n-1)$ form doesn't fit the $T(n/b)$ requirement for the Master Method.
  - We can solve it by expansion:
    - $T(n) = T(n-1) + n$
    - $T(n) = [T(n-2) + (n-1)] + n$
    - $T(n) = [T(n-3) + (n-2)] + (n-1) + n$
    - ...
    - $T(n) = n + (n-1) + (n-2) + \ldots + 1$
  - This is the sum of numbers from 1 to n, which is $n(n+1)/2$.
- Solution: $T(n) = O(n^2)$.

Solving Recurrences: Binary Search

- Recurrence: $T(n) = T(n/2) + O(1)$
- Why: It makes only 1 recursive call ($a=1$) on an array half the size ($b=2$). The work to find the middle and compare is constant time, $O(1)$ ($f(n) = 1$).
- Solving (Master Method):
  - $a = 1$, $b = 2$, $f(n) = 1$
  - Calculate the critical value: $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$.
  - Compare $f(n)$ to this value: $f(n) = 1$, which is asymptotically equal to 1.
  - This is Case 2 of the Master Method.
- Solution: $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(1 \cdot \log n) = \Theta(\log n)$.

Reference

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