1.23 Master Theorem for Subtract and Conquer Recurrences

Let $T(n)$ be a function defined on positive n, and having the property

$$T(n) = \begin{cases} c, & \text{if } n \le 1 \\ aT(n-b) + f(n), & \text{if } n > 1 \end{cases}$$

for some constants $c, a > 0, b > 0, k \ge 0$, and function $f(n)$. If $f(n)$ is in $O(n^k)$, then

$$T(n) = \begin{cases} O(n^k), & \text{if a} < 1 \\ O(n^{k+1}), & \text{if a} = 1 \\ O\left(n^k a^{\frac{n}{b}}\right), & \text{if a} > 1 \end{cases}$$

1.24 Variant of Subtraction and Conquer Master Theorem

The solution to the equation $T(n) = T(\alpha\, n) + T((1-\alpha)n) + \beta n$, where $0 < \alpha < 1$ and $\beta > 0$ are constants, is $O(nlogn)$.

1.25 Method of Guessing and Confirming

Now, let us discuss a method which can be used to solve any recurrence. The basic idea behind this method is:

guess the answer; and then *prove* it correct by induction.

In other words, it addresses the question: What if the given recurrence doesn't seem to match with any of these (master theorem) methods? If we guess a solution and then try to verify our guess inductively, usually either the proof will succeed (in which case we are done), or the proof will fail (in which case the failure will help us refine our guess).

As an example, consider the recurrence $T(n) = \sqrt{n}\ T(\sqrt{n}) + n$. This doesn't fit into the form required by the Master Theorems. Carefully observing the recurrence gives us the impression that it is similar to the divide and conquer method (dividing the problem into $\sqrt{n}$ subproblems each with size $\sqrt{n}$). As we can see, the size of the subproblems at the first level of recursion is n. So, let us guess that $T(n) = O(nlogn)$, and then try to prove that our guess is correct.

Let's start by trying to prove an *upper* bound $T(n) \le cnlogn$:

$$\begin{aligned} T(n) &= \sqrt{n}\ T(\sqrt{n}) + n \\ &\le \sqrt{n}.\, c\sqrt{n}\ log\sqrt{n} + n \\ &= n.\ c\ log\sqrt{n} + n \\ &= n.c.\tfrac{1}{2}.\, logn + n \\ &\le cnlogn \end{aligned}$$

The last inequality assumes only that $1 \le c.\frac{1}{2}.logn$. This is correct if n is sufficiently large and for any constant c, no matter how small. From the above proof, we can see that our guess is correct for the upper bound. Now, let us prove the *lower* bound for this recurrence.

$$\begin{aligned} T(n) &= \sqrt{n}\ T(\sqrt{n}) + n \\ &\ge \sqrt{n}.\, k\ \sqrt{n}\ log\sqrt{n} + n \\ &= n.\ k\ log\sqrt{n} + n \\ &= n.k.\tfrac{1}{2}.\, logn + n \\ &\ge knlogn \end{aligned}$$

The last inequality assumes only that $1 \ge k.\frac{1}{2}.logn$. This is incorrect if n is sufficiently large and for any constant k. From the above proof, we can see that our guess is incorrect for the lower bound.

From the above discussion, we understood that $\Theta(nlogn)$ is too big. How about $\Theta(n)$? The lower bound is easy to prove directly:

$$T(n) = \sqrt{n}\ T(\sqrt{n}) + n \quad \ge n$$

Now, let us prove the upper bound for this $\Theta(n)$.

$$\begin{aligned} T(n) &= \sqrt{n}\ T(\sqrt{n}) + n \\ &\le \sqrt{n}.c.\sqrt{n} + n \\ &= n.\ c + n \\ &= n\ (c+1) \\ &\nleq cn \end{aligned}$$

From the above induction, we understood that $\Theta(n)$ is too small and $\Theta(nlogn)$ is too big. So, we need something bigger than n and smaller than $nlogn$. How about $n\sqrt{logn}$?

Proving the upper bound for $n\sqrt{logn}$:

$$\begin{aligned} T(n) &= \sqrt{n}\ T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n}\sqrt{log\sqrt{n}} + n \\ &= n.\ c.\tfrac{1}{\sqrt{2}}\ log\sqrt{n}+ n \\ &\leq cnlog\sqrt{n} \end{aligned}$$

Proving the lower bound for $n\sqrt{logn}$:

$$\begin{aligned} T(n) &= \sqrt{n}\ T(\sqrt{n}) + n \\ &\geq \sqrt{n}.k.\sqrt{n}\sqrt{log\sqrt{n}} + n \\ &= n.\ k.\tfrac{1}{\sqrt{2}}\ log\sqrt{n}+ n \\ &\ngeq knlog\sqrt{n} \end{aligned}$$

The last step doesn't work. So, $\Theta(n\sqrt{logn})$ doesn't work. What else is between n and $nlogn$? How about $nloglogn$?

Proving upper bound for $nloglogn$:

$$\begin{aligned} T(n) &= \sqrt{n}\ T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n}loglog\sqrt{n} + n \\ &= n.\ c.loglogn\text{-}c.n + n \\ &\leq cnloglogn,\ \text{if } c \geq 1 \end{aligned}$$

Proving lower bound for $nloglogn$:

$$\begin{aligned} T(n) &= \sqrt{n}\ T(\sqrt{n}) + n \\ &\geq \sqrt{n}.k.\sqrt{n}loglog\sqrt{n} + n \\ &= n.\ k.loglogn\text{-}k.n + n \\ &\geq knloglogn,\ \text{if } k \leq 1 \end{aligned}$$

From the above proofs, we can see that $T(n) \leq cnloglogn$, if $c \geq 1$ and $T(n) \geq knloglogn$, if $k \leq 1$. Technically, we're still missing the base cases in both proofs, but we can be fairly confident at this point that $T(n) = \Theta(nloglogn)$.

1.26 Amortized Analysis

Amortized analysis refers to determining the time-averaged running time for a sequence of operations. It is different from average case analysis, because amortized analysis does not make any assumption about the distribution of the data values, whereas average case analysis assumes the data are not "bad" (e.g., some sorting algorithms do well *on average* over all input orderings but very badly on certain input orderings). That is, amortized analysis is a worst-case analysis, but for a sequence of operations rather than for individual operations.

The motivation for amortized analysis is to better understand the running time of certain techniques, where standard worst case analysis provides an overly pessimistic bound. Amortized analysis generally applies to a method that consists of a sequence of operations, where the vast majority of the operations are cheap, but some of the operations are expensive. If we can show that the expensive operations are particularly rare we can *change them* to the cheap operations, and only bound the cheap operations.

The general approach is to assign an artificial cost to each operation in the sequence, such that the total of the artificial costs for the sequence of operations bounds the total of the real costs for the sequence. This artificial cost is called the amortized cost of an operation. To analyze the running time, the amortized cost thus is a correct way of understanding the overall running time — but note that particular operations can still take longer so it is not a way of bounding the running time of any individual operation in the sequence.

When one event in a sequence affects the cost of later events:

- One particular task may be expensive.
- But it may leave data structure in a state that the next few operations become easier.

Example: Let us consider an array of elements from which we want to find the k^{th} smallest element. We can solve this problem using sorting. After sorting the given array, we just need to return the k^{th} element from it. The cost of performing the sort (assuming comparison based sorting algorithm) is $O(nlogn)$. If we perform n such selections then the average cost of each selection is $O(nlogn/n) = O(logn)$. This clearly indicates that sorting once is reducing the complexity of subsequent operations.