Chapter 3

Recursive Analysis - Recitation

3.1 Bounding recursive functions by hands.

Our primary tool to handle recursive relation is the Master Theorem, which was proved in the lecture. As we would like to have a more solid grasp, let's return on the calculation in the proof over a specific case. Assume that your algorithm analysis has brought the following recursive relation:

Example 3.1.1. $T(n) = \begin{cases} 4T\left(\frac{n}{2}\right) + c \cdot n & \textit{for } n > 1 \\ 1 & \textit{else} \end{cases}$. *Thus, the running time is given by*

$$T(n) = 4T\left(\frac{n}{2}\right) + c \cdot n = 4 \cdot 4T\left(\frac{n}{4}\right) + 4c \cdot \frac{n}{2} + c \cdot n = \dots =$$
$$\overbrace{4^h T(1)}^{\text{critical}} + c \cdot n \left(1 + \frac{4}{2} + \left(\frac{4}{2}\right)^2 \dots + \left(\frac{4}{2}\right)^{h-1}\right) = 4^h + c \cdot n \cdot \frac{2^h - 1}{2 - 1}$$

We will call the number of iteration till the stopping condition the recursion height, and we will denote it by h *. What should be the recursion height?* $2^h = n \Rightarrow h = \log(n)$*. So in total we get that the algorithm running time equals* $\Theta\left(n^2\right)$*.*

Question, Why is the term $4^h T(1)$ so critical? Consider the case $T(n) = 4T\left(\frac{n}{2}\right) + c$ .One popular mistake is to forget the final term, which yields a linear solution $\Theta(n)$ (instead of quadric $\Theta(n^2)$).

Example 3.1.2.

$$T(n) = \begin{cases} 3T\left(\frac{n}{2}\right) + c \cdot n & \textit{for } n > 1 \\ 1 & \textit{else} \end{cases}$$

and then the expanding yields:

$$T(n) = 3T\left(\frac{n}{2}\right) + c \cdot n = 3^2 T\left(\frac{n}{2^2}\right) + \frac{3}{2}cn + c \cdot n$$
$$= \overbrace{3^h T(1)}^{\text{critical}} + cn\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \ldots + \left(\frac{3}{2}\right)^{h-1}\right)$$
$$h = \log_2(n) \Rightarrow T(n) = 3^h T(1) + c \cdot n \cdot \left(\left(\frac{3}{2}\right)^{\log_2 n}\right) / \left(\frac{3}{2} - 1\right)$$
$$= \theta\left(3^{\log_2(n)}\right) = \theta\left(n^{\log 3}\right)$$

where $n^{\log 3} \sim n^{1.58} < n^2$.

3.2 Master Theorem, one Theorem to bound them all.

As you might already notice, the same pattern has been used to bound both algorithms. The master theorem is the result of the recursive expansion. it classifies recursive functions at the form of $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$, for positive function $f : \mathbb{N} \to \mathbb{R}^+$.

Master Theorem, simple version.

First, Consider the case that $f = n^c$. Let $a \geq 1, b > 1$ and $c \geq 0$. then:

1. if $\frac{a}{b^c} < 1$ then $T(n) = \Theta(n^c)$ (f **wins**).
2. if $\frac{a}{b^c} = 1$ then $T(n) = \Theta(n^c \log_b(n))$.
3. if $\frac{a}{b^c} > 1$ then $T(n) = \Theta\left(n^{\log_b(a)}\right)$ (f **loose**).

Example 3.2.1. $T(n) = 4T\left(\frac{n}{2}\right) + d \cdot n \Rightarrow T(n) = \Theta\left(n^2\right)$ *according to case (3). And* $T(n) = 3T\left(\frac{n}{2}\right) + d \cdot n \Rightarrow T(n) = \Theta\left(n^{\log_2(3)}\right)$ *also due to case (3).*

Master Theorem, strong version.

Now, let's generalize the simple version for arbitrary positive f and let $a \geq 1$, $b > 1$.

1. if $f(n) = O\left(n^{\log_b(a) - \varepsilon}\right)$ for some $\varepsilon > 0$ then $T(n) = \theta\left(n^{\log_b(a)}\right)$ (f **loose**).
2. if $f(n) = \Theta\left(n^{\log_b(a)}\right)$ then $T(n) = \Theta\left(n^{\log_b(a)} \log(n)\right)$
3. if there exist $\varepsilon > 0, c < 1$ and $n_0 \in \mathbb{N}$ such that $f(n) = \Omega\left(n^{\log_b(a) + \varepsilon}\right)$ and for every $n > n_0$ $a \cdot f\left(\frac{n}{b}\right) \leq cf(n)$ then $T(n) = \theta(f(n))$ (f **wins**).

Example 3.2.2. *1.* $T(n) = T\left(\frac{2n}{3}\right) + 1 \to f(n) = 1 = \Theta\left(n^{\log_{\frac{3}{2}}(1)}\right)$ *matches the second case. i.e* $T(n) = \Theta\left(n^{\log_{\frac{3}{2}}(1)} \log n\right)$.

2. $T(n) = 3T\left(\frac{n}{4}\right) + n\log n \to f(n) = \Omega\left(n^{\log_4(3)+\varepsilon}\right)$ *and notice that* $f\left(a\frac{n}{b}\right) = \frac{3n}{4}\log\left(\frac{3n}{4}\right)$. *Thus, it's matching to the third case.* $\Rightarrow T(n) = \Theta(n\log n)$.

3. $T(n) = 3T\left(n^{\frac{1}{3}}\right) + \log\log n$. *Let* $m = \log n \Rightarrow T(n) = T(2^m) = 3T\left(2^{\frac{m}{3}}\right) + \log m$. *Denote by* $S = S(m) = T(2^m) \to S(m) = 3T\left(2^{\frac{m}{3}}\right) + \log m = 3S\left(\frac{m}{3}\right) + \log m$. *And by the fact that* $\log m = O\left(m^{\log_3(3)-\varepsilon}\right) \to T(n) = T(2^m) = S(m) = \Theta(m) = \Theta(\log(n))$.

3.3 Recursive trees.

There are still cases which aren't treated by the *Master Theorem.* For example consider the function $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$. Note, that $f = \Omega\left(n^{\log_b(a)}\right) = \Omega(n)$. Yet for every $\varepsilon > 0 \Rightarrow f = n\log n = O\left(n^{1+\varepsilon}\right)$ therefore the third case doesn't hold. How can such cases still be analyzed?

Recursive trees Recipe

1. draw the computation tree, and calculate it's height. in our case, $h = \log n$.
2. calculate the work which done over node at the k-th level, and the number of nodes in each level. in our case, there are 2^k nodes and over each we perform $f(n) = \frac{n}{2^k}\log\left(\frac{n}{2^k}\right)$ operations.
3. sum up the work of the k-th level.
4. finally, the total time is the summation over all the $k \in [h]$ levels.

applying the above, yields

$$\begin{aligned} T(n) &= \sum_{k=1}^{\log n} n \cdot \log\left(\frac{n}{2^k}\right) = n\sum_{k=1}^{\log n}\left(\log n - \log 2^k\right) = n\sum_{k=1}^{\log n}(\log n - k) \\ &= \Theta\left(n\log^2(n)\right) \end{aligned}$$

Example 3.3.1. *Consider merge sort variation such that instead of splitting the array into two equals parts it's split them into different size arrays. The first one contains* $\frac{n}{10}$ *elements while second contains the others* $\frac{9n}{10}$ *elements.*

Note, that the master theorem achieves an upper bound,

$$\begin{aligned} T(n) = n + T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) \le n + 2T\left(\frac{9n}{10}\right) \\ \Rightarrow T(n) = O\left(n^{\log_{\frac{10}{9}}(2)}\right) \sim O\left(n^6\right) \end{aligned}$$

Yet, that bound is far from been tight. Let's try to count the operations for each node. Let's try another direction.

Result: returns the sorted permutation of $x_1 ... x_n \in \mathbb{R}^n$

if $n \leq 10$ **then**
  return bubble-sort $(x_1 ... x_n)$
end

else
  define $S_l \leftarrow x_1 ... x_{\frac{n}{10}-2}, x_{\frac{n}{10}-1}$
  define $S_r \leftarrow x_{\frac{n}{10}}, x_{\frac{n}{10}+1} ..., x_n$

  $R_l \leftarrow$ non-equal-merge(S_l)
  $R_r \leftarrow$ non-equal-merge(S_r)

  return Merge(R_l, R_r)
end

Algorithm 1: non-equal-merge alg.

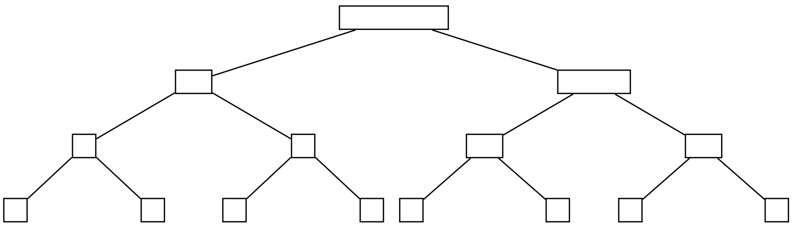

Figure 3.1: The tree matches the recursive calls made by Algorithm 1. Each node presents a rectangle with a length equal to the array given as input to the recursive call. The length of all the elements in a single level is equal to the original array length, thus we have that the linear work in each level sums up to $\Theta(n)$.

Claim 3.3.1. *Let n_i be the size of the subset which is processed at the i-th node. Then for every k:*

$$\sum_{i \in k \textit{ level}} n_i \leq n$$

Proof. Assuming otherwise implies that there exist index j such that x_j appear in at least two different nodes in the same level, denote them by u, v. As they both are in the same level, non of them can be ancestor of the other. denote by $m \in \mathbb{N}$ the input size of the sub array which is processed by the the lowest common ancestor of u and v, and by $j' \in [m]$ the position of x_j in that sub array. By the definition of the algorithm it steams that $j' < \frac{m}{10}$ and $j' \geq \frac{m}{10}$. contradiction. The height of the tree is bounded by $\log_{\frac{9}{10}}(n)$. Therefore the total work equals $\Theta(n \log n)$. Thus, the total running time equals to:

$$T(n) = \sum_{k \in \text{levels}} \sum_{i \in \text{k level}} f(n_i) = \sum_{k \in \text{levels}} \sum_{i \in \text{k level}} n_i \leq n \log n$$

$\square$