Notes 10-23

Cameron Chalk

1 Divide and Conquer: Merge-Sort

Recall the algorithm Merge-Sort(A) discovered by John von Neumann in 1945:

- if $|A| = 1$ return A
- let L be the first $\lceil \frac{n}{2} \rceil$ elements of A
- let R be the remaining $\lfloor \frac{n}{2} \rfloor$ elements of A
- let LS be Merge-Sort(L) — if the algorithm takes worst-case $T(n)$ time,
- let RS be Merge-Sort(R) — these two lines each take $T(\frac{n}{2})$ time
- return merge(LS, RS) — takes $\mathcal{O}(n)$ time

The runtime of Merge-Sort(A) is given by the recurrence: $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n)$. We will see next that this implies $T(n) = \mathcal{O}(n \log n)$. Thus, it is an optimal comparison-based sorting algorithm—note that the comparisons occur in the call of the function merge(LS, RS).

We will write $T(n) = 2T(\frac{n}{2}) + c \cdot n$. We will assume $T(1) = d$ where d is a constant independent of n. Consider the recursion tree of the algorithm. There are $log_2 n$ layers. In layer i, each node does $c \cdot \frac{n}{2^i}$ work. The number of nodes in layer i is 2^i. Thus the total work in layer i is $c \cdot \frac{n}{2^i} \cdot 2^i = c \cdot n$. Since there are n nodes in the last layer, and $T(1) = d$, the total work in the last layer is $d \cdot n$. Then the total work is the work per layer, $c \cdot n$, times the number of layers, $\log_2 n$, plus the work done in the last layer, $d \cdot n$, for a total work of $c \cdot n \log_2 n + d \cdot n = \mathcal{O}(n \log n)$.

2 Master Theorem for solving recurrences

The above approach used for Merge-Sort(A) is a good general technique to solve recurrence equations to determine the runtime of divide and conquer algorithms. This technique has been generalized into a nice shortcut method for solving a large class of recurrences. This technique is outlined in the *Master Theorem* for solving recurrences.

First, we give some intuition for the Master Theorem. Consider our algorithm has a runtime given by the following recurrence equation:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

and $T(1) = d$. Now if we consider our recursion tree, layer i has a^i nodes. There are $\log_b n$ layers. The work per node in layer i is $f(\frac{n}{b^i})$. Then the total work in each layer is $a^i \cdot f(\frac{n}{b^i})$. The last layer has total work $d \cdot a^{\log_b n}$.

To use the Master Theorem, we classify recurrences into three cases. First, we see Case 2 for simplicity. The cases are split based on the work done in each recursive call, $f(n)$. Case 2 is when $f(n) = n^{\log_b a}$. Recall the work done in layer i:

$$\begin{aligned} a^i f\left(\frac{n}{b^i}\right) &= a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\ &= a^i \left(\frac{n^{\log_b a}}{b^{i \log_b a}}\right) \\ &= n^{\log_b a} \end{aligned}$$

and thus the work done is independent of which layer we consider—each layer does the same amount of work, $n^{\log_b a}$. For the last layer, we can rewrite the work done in the last layer as $d \cdot a^{\log_b n} = d \cdot n^{\log_b a}$. Therefore since the algorithm does $n^{\log_b a}$ work per layer, and there are $\log_b n$ layers, and the last layer does $d \cdot n^{\log_b a}$ work, the total work is $n^{\log_b a} \log_b n + d \cdot n^{\log_b a} = \mathcal{O}(n^{\log_b a} \log n)$.

The cases can be seen as the following:

- Case 1: Suppose $f(n)$ is "polynomially smaller" than $n^{\log_b a}$
- Case 2: Suppose $f(n) = n^{\log_b a}$
- Case 3: Suppose $f(n)$ is "polynomially larger" than $n^{\log_b a}$

Case 1 corresponds to "doing the most work in the last layer", therefore in this case $T(n) = \mathcal{O}(n^{\log_b a})$. Case 3 corresponds to "doing the most work in the top layer, therefore in this case $T(n) = \mathcal{O}(f(n))$. Now you can see the Master Theorem slide, which is posted on Canvas, and you should have some intuition for why it looks the way it does. If you are to use the Master Theorem on a quiz or exam, you will be given the Master Theorem in full to use, so you need not memorize the details.

Next we will see some example recurrences and use the Master Theorem to solve them. Recall the recurrence for Merge-Sort(A),

$$T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n)$$

noting that we have the general recurrence from the Master Theorem with $a = 2$, $b = 2$ and $f(n) = \mathcal{O}(n)$. Therefore $n^{\log_b a} = n$. We see we are in Case 2 of the Master Theorem, and thus the runtime is $\mathcal{O}(n \log n)$. Further, note that if we

instead have $a = b = k$ instead of $a = b = 2$, i.e., we split the list into more than two parts, we get the same runtime, $\mathcal{O}(n \log n)$. Thus by using the Master Theorem, we see that splitting the list into smaller parts than two halves does not improve the asymptotic runtime.

Next, consider the Matrix Multiplication problem, and see the "Divide and Conquer: First Attempt" slide on Canvas. The recurrence for this technique is:

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + \Theta(n^2)$$

. To write this in the form of the general Master Theorem recurrence, we have $a = 8$, $b = 2$, and $f(n) = \Theta(n^2)$. Then we have $n^{\log_b a} = n^3$ since $\log_b a = \log_2 8 = 3$. Then we are in Case 1, of the Master Theorem, and thus $T(n) = \Theta(n^3)$.

Now see Strassen's improved algorithm in the slide titled "Divide and Conquer: Strassen (1969)". This algorithm has a different recurrence:

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + \Theta(n^2)$$

.

We are still in Case 1 of the algorithm, but now the quantity $n^{\log_b a} = n^{2.81\cdots}$, and thus the runtime is $\Theta(n^{2.81\cdots})$. This was the first improvement to the asymptotic runtime of matrix multiplication. Since then, improvement have been made. It is conjectured that matrix multiplication can be done in time $\mathcal{O}(n^{2+\epsilon})$.