CSC236 Week 11: Recurrences…

Hisbaan Noorani

November 25 – November 1, 2021

Contents

1 Merge sort complexity, a sketch

1. Derive a recurrence to express worst-case run times in terms of $n = |A|$:

$$T(n) = \begin{cases} c' & \text{if } n = 1 \\ T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n & \text{if } n > 1 \end{cases}$$

2. Repeated substitution/unwinding in special case where $n = 2^k$ for some natural number k leads to:

$$T(2^k) = 2^k T(1) k 2^k = c'n + n \log_2(n)$$

We can neglect the $c'n$ term since it is of a lower order so we make the conjecture, $T(2^k) \in \Theta(n \log_2(n))$.

3. Prove T is non-decreasing (see Course Notes Lemma 3.6)
4. Prove $T \in \mathcal{O}(n \log_2(n))$ and $T \in \Omega(n \log_2(n))$

2 T is non-decreasing, see Course Notes Lemma 3.6

Exercise: prove the recurrence for binary search is non-decreasing...

Let $\hat{n} = 2^{\lceil \log_2(n) \rceil}$. We want to sandwich $T(n)$ between successive powers of 2.

$$\begin{aligned} \lceil \log_2(n) \rceil - 1 < \log_2(n) \leq \lceil \log_2(n) \rceil \\ 2^{\lceil \log_2(n) \rceil - 1} < n \leq 2^{\lceil \log_2(n) \rceil} \\ \frac{\hat{n}}{2} < n \leq \hat{n} \end{aligned}$$

As an aside, try working out for $n \in \{1, 2, 3, 4, 5, 6, 7, 8\}$.

3 Prove $T \in \mathcal{O}(n \log_2(n)$ for the general case

Let $d \in \mathbb{R}^+ = 2(2 + c)$. Let $B \in \mathbb{R}^+ = 2$. Let $n \in \mathbb{N}$ no smaller than B.

$$\begin{aligned}
T(n) &\leq T(\hat{n}) && \text{(since } T \text{ is non-decreasing and } n \leq \hat{n}) \\
&= \hat{n} \log_2(\hat{n}) + c\hat{n} && \text{by the unwinding} \\
&< 2n \log_2(2n) + c \cdot 2n && \text{(since } \hat{n} < 2n \text{ and } \log_2 \text{ is non-decreasing)} \\
&= 2n(\log_2(2) + \log_2(n)) + 2cn \\
&= 2n((1 + c) \log_2(n) + \log_2(n)) && (\log_2(n) \geq \log_2(2) = 1, \text{ since } n \geq B) \\
&= 2n(\log_2(n))(1 + 1 + c) \\
&= 2n(\log_2(n))(2 + c) \\
&= 2n \log_2(n) + 2n(1 + c) \\
&= dn \log_2(n) && \text{(since d = 2 (2 + c))}
\end{aligned}$$

This proves that T is bounded above by some constant times $n \log_2(n)$. $\blacksquare$.

Note: in proving Ω, you will want to use the fact that $\frac{n}{2} \leq \frac{\hat{n}}{2}$.

4 Divide-and-conquer general case

Divide-and-conquer algorithms: partition a problem into b roughly equal sub-problems, solve, and recombine.

$$T(n) = \begin{cases} k & n \leq B \\ a_1 T\left(\left\lceil \frac{n}{b} \right\rceil\right) + a_2 T\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n) & n > B \end{cases}$$

Where $b, k > 0, a_1, a_2 \geq 0,$ and $a = a_1 + a_2 > 0$. $f(n)$ is the cost of splitting and recombining.

b is the number of pieces we divide the problem into.

a is the number of recursive calls.

f is the cost of splitting and recombining, we hope $f \in \Theta(n^d)$.

4.1 Divide-and conquer Master Theorem

If f from the previous slide has $f \in \Theta(n^d)$

$$T(n) \in \begin{cases} \Theta(n^d) & a < b^d \\ \Theta(n^d \log_b n) & a = b^d \\ \Theta(n^{\log_b a}) & a > b^d \end{cases}$$

Note: the complexity is sensitive to a (the number of recursive calls) and d (the degree of polynomial for splitting and recombining).

There are three steps to the proof of the Master Theorem. They are exactly parallel to the Merge Sort proof.

1. Unwind the recurrence, and prove a result for $n = b^k$.

   Only valid for special values of n.

2. Prove that T is non-creasing

   See the course notes.

3. Extend to all n, similar to Merge Sort.

   Easiest step, remember that technique with $\hat{n}$.

4.2 Apply the master theorem

4.2.1 Merge sort

$a = b = 2,\ d = 1$. So the complexity is $\Theta(n^1 \log_2 n)$.

4.2.2 Binary search

$a = 1,\ b = 2,\ d = 1$. So the complexity is $\Theta(n^0 \log_2 n)$.

5 Multiply lots of bits

Machines are usually able to multiply able to process integers of machine size (64-bit, 32-bit, etc.) in constant time. But what if they don't fit into machine instruction? This process takes a longer time:

$$
\begin{array}{ccccccc}
 & & & 1 & 1 & 0 & 1 \\
 & & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & 1 & 1 & 0 & 1 \\
 & & 1 & 1 & 0 & 1 & \\
 & 0 & 0 & 0 & 0 & & \\
1 & 1 & 0 & 1 & & & \\
\hline
1 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}
$$

Let n be the number of bits of the numbers we are multiplying. We make n copies, and have n additions of 2 $n-$bit numbers. So the complexity of this "algorithm" is $\Theta(n^2)$.

5.1 Divide and recombine

Recursively, $2^n = n$ left-shifts, and addition/subtractions are $\Theta(n)$.

$$
\begin{array}{rr}
11 & 01 \\
\times\ 10 & 11
\end{array}
$$

Let x_0 be the top left of this table, x_1 be the top right, y_0 be the bottom left, y_1 be the bottom right.

$$
\begin{aligned}
xy &= (2^{\frac{n}{2}} x_1 + x_0)(2^{\frac{n}{2}} y_1 + y_0) \\
&= 2^n x_1 y_1 + 2^{\frac{n}{2}}(x_1 y_0 + x_0 y_1) + x_0 y_0
\end{aligned}
$$

The time complexity of this can be broken down as follows:

1. divide each factor (roughly) in half ($b = 2$).
2. Multiply the halves (recursively, if they're too big) ($a = 4$).
3. Combine he products with shifts and adds (linear in number of bits).

According to the master theorem, since we have $a = 4,\ b = 2,\ d = 1$, the time complexity of this algorithm is $\Theta(n^{log_2 4})) = \Theta(n^2)$.

5.2 Gauss's Trick

Gauss rewrote the multiplication of x and y as follows, to reduce the number of individual calculations by making some of the multiplications repeat themselves.

$$
xy = 2^n x_1 y_1 + 2^{\frac{n}{2}} x_1 y_1 + 2^{\frac{n}{2}}((x_1 - x_0)(y_0 - y_1) + x_0 y_0) + x_0 y_0
$$

Repeated products can be stored after the first calculation, so they don't need to be calculated multiple times. So, not we have just 3 multiplications, at the cost of 2 more subtractions and one more addition. Since addition and subtractions are linear in the number of bits, this greatly reduces the complexity.

So now, using the master theorem, since $a = 3$,$b = 2,\ d = 1$, the time complexity of this algorithm is $\Theta(n^{\log_2 3}) = \Theta(n^{1.5894})$. This is, in fact, better that $\Theta(n^2)$, even if not by much.