On tap for today: more recurrences, more recursion trees, one new algorithm (binary search)

1 Recursion Trees: Two More Examples

Recursion trees are a general way to solve recurrences. I want to make sure you’re comfortable with how to construct and use them.

- Start with a recurrence for $T(n)$.
- Sketch structure of recursive calls.
- Account local work performed for each call.
- Sum up work over entire tree.

Here’s an example we haven’t seen before:

$$T(n) \geq \begin{cases} 
    c_0 & \text{if } n = 1 \\
    2T\left(\frac{n}{2}\right) + cn^2 & \text{if } n > 1 
\end{cases}$$

Assume $n$ is a power of 2. Let’s draw the tree:
Now we sum over all the levels of the tree.

\[ T(n) \geq \sum_{k=0}^{\log n - 1} \frac{cn^2}{2^k} + c_0 n \]

\[ = cn^2 \sum_{k=0}^{\log n - 1} \frac{1}{2^k} + c_0 n \]

\[ = cn^2 \left[ \sum_{k=0}^{\infty} \frac{1}{2^k} - \sum_{k=0}^{\infty} \frac{1}{2^k + \log n} \right] + c_0 n \]

\[ = cn^2 \left[ 2 - \frac{1}{2\log n} \sum_{k=0}^{\infty} \frac{1}{2^k} \right] + c_0 n \]

\[ = cn^2 \left[ 2 - \frac{1}{n} \right] + c_0 n \]

\[ = 2cn^2 + c'n. \]

**Asymptotic growth:** conclude that \( T(n) = \Omega(n^2) \). Why? Because recurrence itself is just a lower bound on \( T(n) \).

Here’s another example:

\[ T(n) = \begin{cases} 
  c_0 & \text{if } n = 1 \\
  3T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 
\end{cases} \]

Assume \( n \) is a power of 2. Let’s draw the tree:
Now we sum over all the levels of the tree.

\[
T(n) = \sum_{k=0}^{\log_2 n - 1} c n \left( \frac{3}{2} \right)^k + c_0 3^{\log_2 n}
\]

\[
= cn \sum_{k=0}^{\log_2 n - 1} \left( \frac{3}{2} \right)^k + c_0 n^{\log_2 3}
\]

\[
= cn \left[ \frac{(3/2)^{\log_2 n}}{3/2 - 1} - 1 \right] + c_0 n^{\log_2 3}
\]

\[
= 2cn \left[ n^{\log_2 (3/2)} - 1 \right] + c_0 n^{\log_2 3}
\]

\[
= (c_0 + 2c)n^{\log_2 3} - 2cn.
\]

Asymptotic growth: conclude that \( T(n) = \Theta(n^{\log_2 3}) \). Why? Because recurrence is exact – both upper and lower bound on \( T(n) \).

To review sneaky summation and log tricks, see Section 3.2 and Appendix A of your text!

2 A New Recursion: Binary Search

Binary search is a classic example of a divide-and-conquer algorithm, albeit with nothing to combine.

• Input:
  – a sorted array of numbers \( A[p . . r] \)
  – a number \( x \)

• Returns:
  – (an) index of \( x \) in \( A \) if it’s present
  – “notFound” otherwise
\textbf{Bsearch}(x, A, p, r) \\
if \( p = r \) \hspace{1cm} \triangleright \text{base case} \\
\quad \text{if } A[p] = x \\
\quad \quad \text{return } p \\
\quad \text{else} \\
\quad \quad \text{return } \text{notFound} \\

mid \leftarrow \lceil (p + r)/2 \rceil \\
\text{if } A[mid] > x \\
\quad \text{return } \text{BSEARCH}(x, A, p, mid - 1) \\
\text{else} \\
\quad \text{return } \text{BSEARCH}(x, A, mid, r) \\

\textbf{What is input size of binary search? } n = r - p + 1!

\section{Correctness of Binary Search}

Prove by induction on \( n \).

- \textbf{Base:} \( n = 1 \) – by inspection.
- \textbf{Inductive:} Consider \( A[p \ldots r] \), which is sorted. If \( x < A[mid] \) is present, it must be in the subarray \( A[p \ldots mid - 1] \). Otherwise, it must be in \( A[mid \ldots r] \). For each case, we recur on the correct subarray, which is shorter than \( A[p \ldots r] \). By inductive hyp., recursive call returns correct position of \( x \) if present, or \text{notFound} otherwise. QED.

\section{Running Time of Binary Search}

Let’s analyze the algorithm’s running time.

- \textbf{Base case} takes \( \Theta(1) \) time.
- \textbf{Inductive case} takes \( \Theta(1) \) time, plus one recursive call on an array of half the size.

For simplicity, assume again that \( n = r - p + 1 \) is power of two. Our recurrence is then

\[ T(n) = \begin{cases} 
  c_0 & \text{if } n = 1 \\
  T(n/2) + c & \text{if } n > 1 
\end{cases} \]

Recursion tree is a “stick.”
Now sum the cost over the tree...

\[ T(n) = \sum_{k=0}^{\log n - 1} c + c_0 \]
\[ = c \log n + c_0 \]
\[ = \Theta(\log n) \]