Analyzing Recursive Algorithms (Ch. 4)

A recursive algorithm can often be described by a recurrence equation that describes the overall runtime on a problem of size \( n \) in terms of the runtime on smaller inputs.

For divide-and-conquer algorithms, we get recurrences like:

\[
T(n) = \begin{cases} 
  \theta(1) & \text{if } n = c \\
  aT(n/b) + D(n) + C(n) & \text{otherwise}
\end{cases}
\]

where

- \( a \) = number of subproblems we divide the problem into
- \( n/b \) = size of the subproblems (in terms of \( n \))
- \( D(n) \) = time to divide the size \( n \) problem into subproblems
- \( C(n) \) = time to combine the subproblem solutions to get the answer for the problem of size \( n \)

We will use the following methods to solve recurrences given in §4-3.

1. Backward Substitution: involves substitution and expansion until seeing a pattern, converting result to a summation.
2. The “Master Theorem”: If the recurrence has the form

\[
aT(n/b) + D(n) + C(n)
\]

then there are 2 formulae that can (often) be applied; one of these is

\[
\begin{cases} 
  \theta(n^l) & \text{if } a \leq c \log_b d \\
  \theta(n^k \log^k n) & \text{if } a = c \log_b d \\
  \theta(n^p) & \text{if } a > c \log_b d \text{ or } a = c \log_b d \text{ but } k < p
\end{cases}
\]

Recurrence trees can be used along with backward substitution to guess the running time of a recurrence relation. Most recurrences of the form shown above will be solved using the Master Theorem.

To make the solutions simpler, we will

- assume base cases are constant, i.e., \( T(n) = \theta(1) \) for \( n \) small enough.

Review of Logarithms

A logarithm is an inverse exponential function. Saying \( b^x = y \) is equivalent to saying \( \log_b y = x \).

- properties of logarithms:
  - \( \log_a(xy) = \log_a x + \log_a y \)
  - \( \log_a (x/y) = \log_a x - \log_a y \)
  - \( \log_a x^k = k \log_a x \)
  - \( \log_a a = \log_a b \) (reason log base doesn’t matter; asymp)
  - \( a = b^{\log_b a} \) (e.g., \( n = 2^{\log_2 n} \))
  - \( \log_2 n = (\log n)^2 \)
  - \( \log_{lg} n = \log_2 (\log n) \)

Solving Recurrences

We will use the following methods to solve recurrences:

1. Backward Substitution: involves substitution and expansion until seeing a pattern, converting result to a summation.
2. The “Master Theorem”: If the recurrence has the form

\[
T(n) = aT(n/b) + D(n) + C(n)
\]

then there are 2 formulae that can (often) be applied; one of these is given in §4-3.

Recurrence trees can be used along with backward substitution to guess the running time of a recurrence relation. Most recurrences of the form shown above will be solved using the Master Theorem.

To make the solutions simpler, we will

- assume base cases are constant, i.e., \( T(n) = \theta(1) \) for \( n \) small enough.

Analyzing Recursive Algorithms

For recursive algorithms such as computing the factorial of \( n \), we get an expression like the following:

\[
T(n) = \begin{cases} 
  \theta(1) & \text{if } n = 0 \\
  T(n-1) + D(n) + C(n) & \text{otherwise}
\end{cases}
\]

where

- \( n = \) size of the subproblems (in terms of \( n \))
- \( D(n) = \) time to divide the size \( n \) problem into subproblems
- \( C(n) = \) time to combine the subproblem solutions to get the answer for the problem of size \( n \)

Solving Recurrences: Backward Substitution

Example: \( T(n) = 2T(n/2) + n \)

\[
T(n) = 2T(n/2) + n
\]

\[
\begin{align*}
T(n) &= 2T(n/2) + n \\
&= 2[2T(n/4) + n/2] + n \\
&= 4T(n/4) + n + n \\
&= 8T(n/8) + n + n + n \\
&= 16T(n/8) + n + n + n + n \\
&\ldots \text{continue until } T(n/1) = T(1) \text{ is reached}
\end{align*}
\]

Therefore, this algorithm has linear running time.

We solved this recurrence (ie, found an expression of the running time \( T(n) \) that is not given in terms of itself) using a method known as backward substitution.
**Binary Search (recursive version)**

Algorithm Binary-Search-Rec(A[1…n], k, l, r)

- **Input:** a sorted array A of n comparable items, search key k, leftmost and rightmost index positions in A
- **Output:** Index of array's element that is equal to k or -1 if k not found

1. if (l > r) return -1
2. else
3. \( m = \lfloor (l + r) / 2 \rfloor \); m is midpoint
4. if \( k = A[m] \) return \( m \)
5. else if \( k < A[m] \) return Binary-Search-Rec(A, k, l, m-1)
6. else return Binary-Search-Rec(A, k, m+1, r)

**Solving Recurrences: Backward Substitution**

Example: \( T(n) = T(n/2) + 1 \)

- \( T(n) = T(n/2) + 1 \)
- \( T(n/2) + 2 \)
- \( T(n/4) + 3 \)
- \( T(n/8) + 4 \)

What is the running time of this algorithm for an input of size \( n \)?

Are there best and worst case input instances?
Solving Recurrences: Master Method (§4.3)

The master method provides a 'cookbook' method for solving recurrences of a certain form.

Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on nonnegative integers as:

$$T(n) = aT(n/b) + f(n)$$

Where $a$ is the number of subproblems, $n/b$ is the size of each subproblem, and $f(n)$ is the time to divide or combine data.

Then, $T(n)$ can be bounded asymptotically as follows:

1. $T(n) = \Theta(n^p)$ if $f(n) = \Theta(n^{p-1})$ for some constant $p > 0$ (case 1: $f(n)$ is "polynomially smaller than" $\Theta(n^p)$)
2. $T(n) = \Theta(n \log^k n)$ if $f(n) = \Theta(n \log^{k-1} n)$ for some constant $k > 0$ (case 2: $f(n)$ is "asymptotically equal to" $\Theta(n \log^k n)$)
3. $T(n) = \Theta(f(n))$ if $f(n) = \Theta(n \log^{k-1} n)$ for some constant $k > 0$ (case 3: $f(n)$ is "polynomially larger than" $\Theta(n \log^k n)$)

What is $\log_a b$? The number of times we divide a by $b$ to reach $O(1)$.

Solving Recurrences: Master Method (§4.3)

Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on nonnegative integers as:

$$T(n) = aT(n/b) + f(n)$$

Then, $T(n)$ can be bounded asymptotically as follows:

1. $T(n) = \Theta(n^p)$ if $f(n) = \Theta(n^{p-1})$ for some constant $p > 0$ (case 1: $f(n)$ is "polynomially smaller than" $\Theta(n^p)$)
2. $T(n) = \Theta(n \log^k n)$ if $f(n) = \Theta(n \log^{k-1} n)$ for some constant $k > 0$ (case 2: $f(n)$ is "asymptotically equal to" $\Theta(n \log^k n)$)
3. $T(n) = \Theta(f(n))$ if $f(n) = \Theta(n \log^{k-1} n)$ for some constant $k > 0$ (case 3: $f(n)$ is "polynomially larger than" $\Theta(n \log^k n)$)

These 3 cases do not cover all the possibilities for $f(n)$.

There is a gap between cases 1 and 2 when $f(n)$ is smaller than $n^p \log n$, but not polynomially smaller.

There is a gap between cases 2 and 3 when $f(n)$ is larger than $n^p \log n$, but not polynomially larger.

If the function falls into one of these 2 gaps, or if the regularity condition can’t be shown to hold, then the master method can’t be used to solve the recurrence.

Alternate Version of Master Method

Master Theorem: Let $a \geq 1$, $b > 1$, and $k \geq 0$, p a real number, and let $T(n)$ be defined on nonnegative integers as:

$$T(n) = aT(n/b) + f(n)$$

Then, $T(n)$ can be bounded asymptotically as follows:

1. If $a > b^k$, then $T(n) = \Theta(n^p)$
2. If $a = b^k$, then
   a) If $p > -1$, then $T(n) = \Theta(n^p \log^{-1} n)$
   b) If $p = -1$, then $T(n) = \Theta(n^p \log \log n)$
   c) If $p < -1$, then $T(n) = \Theta(n^p \log n)$
3. If $a < b^k$, then
   a) If $p \geq 0$, then $T(n) = \Theta(n^p \log^2 n)$
   b) If $p < 0$, then $T(n) = \Theta(n^p)$

Like the version of the master theorem in our book, this doesn’t hold for cases in which $a$, $b$, or $k$ are not in the correct range.
Solving Recurrences: Master Method

Example: $T(n) = 9T(n/3) + n$

- $a = 9$, $b = 3$, $f(n) = n$, $n \log_b a = n \log_3 9 = n^2$
- Compare $f(n) = n$ with $n^2$
  - $n = O(n^{2-\epsilon})$ (so $f(n)$ is polynomially smaller than $n \log_b a$)
  - Case 1 applies: $T(n) \in \theta(n^2)$

Example: $T(n) = T(n/2) + 1$

- $a = 1$, $b = 2$, $f(n) = 1$, $n \log_b a = n \log_2 1 = n^0 = 1$
- Compare $f(n) = 1$ with $1$
  - $n^2 = \Omega(n^{0+\epsilon})$ (so $f(n)$ is polynomially larger)
  - Since $f(n)$ is a polynomial in $n$, case 3 holds, $T(n) \in \theta(n^2)$

Example: $T(n) = 4T(n/2) + n$

- $a = 4$, $b = 2$, $f(n) = n^2$, $n \log_b a = n \log_2 4 = n^2$
- Compare $f(n) = n^2$ with $n^2$
  - $n^2 = \theta(n^2)$ (so $f(n)$ is polynomially equal)
  - Case 2 holds and $T(n) \in \theta(n^2 \log n)$

Example: $T(n) = 7T(n/3) + n$

- $a = 7$, $b = 3$, $f(n) = n^2$, $n \log_b a = n \log_3 7 = n^{1+\epsilon}$
- Compare $f(n) = n^2$ with $n^{1+\epsilon}$
  - $n^2 = \Omega(n^{1+\epsilon})$ (so $f(n)$ is polynomially larger)
  - Since $f(n)$ is a polynomial in $n$, case 3 holds and $T(n) \in \theta(n^2)$

Example: $T(n) = 7T(n/2) + n^2$

- $a = 7$, $b = 2$, $f(n) = n^2$, $n \log_b a = n \log_2 7 = n^{2+\epsilon}$
- Compare $f(n) = n^2$ with $n^{2+\epsilon}$
  - $n^2 = O(n^{2+\epsilon})$ (so $f(n)$ is polynomially smaller)
  - Case 1 holds and $T(n) \in \theta(n \log_2 7)$