Analyzing Recursive Algorithms (Ch. 4)

Analyzing Recursive Algorithms

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:

\[
T(n) = \begin{cases} 0 & \text{if } n = c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \\ \end{cases}
\]

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.

- Apply the “Master Theorem”: If the recurrence has the form
  \[
  T(n) = \begin{cases} \frac{1}{a} T(n/b) + D(n) + C(n) & \text{if } n/b = O(1) \\ T(n-1) + 1 & \text{otherwise} \\ \end{cases}
  \]

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

- Backward Substitution: involves substitution and expansion until seeing a pattern, converting result to a summation.

• properties of logarithms:
  - \( \log_a(b^x) = \log_a(b) \cdot x \)
  - \( \log_a(b^x) = x \cdot \log_a(b) \)
  - \( \log_a(b) = \log_b(a) / \log_b(c) \) (reason log base doesn’t matter, asymp)
  - \( a = b^\log_b(n) \) (e.g., \( n = 2^k \rightarrow \log_2(n) \))
  - \( \log(n) = (\log(n))^2 \)
  - \( \log(n) = \log(\log(n)) \)

• Review of Logarithms

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

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. Apply the “Master Theorem”: If the recurrence does not have the form

   \[
   T(n) = \begin{cases} \frac{1}{a} T(n/b) + D(n) + C(n) & \text{if } n/b = O(1) \\ T(n-1) + 1 & \text{otherwise} \\ \end{cases}
   \]

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

   - Analyzing Recursive Algorithms (Ch. 4)

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

\[
T(n) = \begin{cases} 0 & \text{if } n = c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \\ \end{cases}
\]

where

- \( a \) = 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: \( T(n) = 2T(n/2) + n \)

\[
T(n) = \begin{cases} 2T(n/2) + n & \text{if } n = 0 \\ \end{cases}
\]

\[
T(n) = \begin{cases} 2T(n/2) + n & \text{if } n = 0 \\ 2T(n/4) + n + n & \text{if } n = 0 \\ 2T(n/8) + n + n + n & \text{if } n = 0 \\ \ldots & \text{... continue until } T(n/0) = T(1) \text{ is reached} \\ \end{cases}
\]

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.
**Recursion Tree for** \( T(n) = 2T(n/2) + n \)

**Solving Recurrences: Backward Substitution**

**Example:** \( T(n) = 2T(n/2) + 4n \)

\[
T(n) = 2T(n/2) + 4n
\]

**Binary Search (iterative version)**

**Algorithm Binary-Search(A[1…n], k)**

**Input:** a sorted array \( A \) of \( n \) comparable items and search key \( k \)

**Output:** Index of array's element that is equal to \( k \) or -1 if \( k \) not found

1. **i = 1;** \( r = n \)
2. **while** \( l <= r \):
   3. \( m = \lceil (l + r)/2 \rceil \); \( m \) is midpoint
   4. \( \text{if } k = A[m] \) **return** \( m \); found \( k \), return index of \( k \)
   5. **else if** \( k < A[m] \) **return** Binary-Search-Rec(A, \( k \), \( l \), \( r \))
   6. **else return** Binary-Search-Rec(A, \( k \), \( m+1 \), \( r \))

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

**Are there best and worst case input instances?**

**Solving Recurrences:**

**Example:** \( T(n) = 2T(n/2) + 4n \)

\[
T(n) = 2T(n/2) + 4n
\]

**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. \( m = \lceil (l + r)/2 \rceil \); \( m \) 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, \( k \), \( m+1 \), \( r \))

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

**Binary Search (recursive version)**

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

\[
T(n) = T(n/2) + 1
\]
Solving Recurrences: Master Method (§4.3)

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

**Master Theorem:** Let a ≥ 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. If f(n) = \( \Theta(n^a) \) and if a > \( \log_b c \), then
   \[ T(n) = \Theta(n^a) \]
2. If f(n) = \( \Theta(n^a \log^k n) \) for some constant k ≥ 0, then
   \[ T(n) = \Theta(n^a \log^k n) \]
3. If f(n) is not \( \Theta(n^a \log^k n) \), then
   \[ T(n) = \Theta(f(n)) \]

Where \( c(f(n/b)) \) is a polynomial.

Case 3 requires us to also show \( a(f(n/b)) \leq c(f(n)) \) for some constant \( c < 1 \) and all sufficiently large n.

The regularity condition always holds whenever \( f(n) = n^a \) and \( f(n) = \Theta(n^a \log^k n) \), so we don’t need to check it when f(n) is a polynomial.

A more general version of Case 2 follows:

\[ T(n) = \Theta(n^a \log^{k+1} n) \quad \text{if} \quad f(n) = \Theta(n^a \log^k n) \text{ for } k \geq 0 \]

This case covers the gap between cases 2 and 3 in which \( f(n) \) is larger than \( n^a \log^k n \) by only a polylog factor. We’ll see an example of this type of recurrence in class.

**Alternate Version of Master Method**

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

\[ T(n) = aT(n/b) + \Theta(n^p \log^k n) \]

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

1. If a > b^p , then
   \[ T(n) = \Theta(n^a) \]
2. If a = b^p , then
   a) If p > -1, then
      \[ T(n) = \Theta(n^a \log^{p+1} n) \]
   b) If p = -1, then
      \[ T(n) = \Theta(n^a \log \log n) \]
   c) If p < -1, then
      \[ T(n) = \Theta(n^a \log^{k-1} n) \]
3. If a < b^p , then
   a) If p ≥ 0, then
      \[ T(n) = \Theta(n^a \log^\frac{a}{b^p} n) \]
   b) If p < 0, then
      \[ T(n) = \Theta(n^a) \]

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

Intuition: Compare \( f(n) \) with \( \Theta(n^a \log^k n) \).

- **case 1:** \( f(n) \) is "polynomially smaller than" \( \Theta(n^a \log^k n) \)
- **case 2:** \( f(n) \) is "asymptotically equal to" \( \Theta(n^a \log^k n) \)
- **case 3:** \( f(n) \) is "polynomially larger than" \( \Theta(n^a \log^k n) \)

What is \( \log_a a \)? The number of times we divide a by b to reach O(1).
Solving Recurrences: Master Method

Example: \( T(n) = 9T(n/3) + n \)
- \( a = 9, b = 3, f(n) = n, n^{\log_b a} = n^{\log_3 9} = n^2 \)
- \( \text{compare } f(n) = n \text{ with } n^2 \)
  - \( n = O(n^{\log_b a}) \) (so \( f(n) \) is polynomially smaller than \( n^{\log_b a} \))
- \( \text{case 1 applies: } T(n) \in \Theta(n^2) \)

Example: \( T(n) = T(n/2) + 1 \)
- \( a = 1, b = 2, f(n) = 1, n^{\log_b a} = n^0 = 1 \)
- \( \text{compare } f(n) = 1 \text{ with } 1 \)
  - \( 1 = \Theta(n^0) \) (so \( f(n) \) is polynomially equal to \( n^{\log_b a} \))
- \( \text{case 2 applies: } T(n) \in \Theta(n^0) \)

Solving Recurrences: Alt. Master Method

Example 1a: \( T(n) = 9T(n/3) + n \)
- \( a = 9, b = 3, k = 0, p = 0, \log_b a = \log_3 9 = 2 \)
- \( \text{compare } a = 9 \text{ with } b^k = 3^0 = 1 \)
- \( 9 > 3 \)
- \( \text{case 1 applies: } T(n) \in \Theta(n^{\log_b a}) \in \Theta(n^2) \)

Example 2a: \( T(n) = T(n/2) + 1 \)
- \( a = 1, b = 2, k = 0, p = 0, \text{ and } \log_b a = \log_2 1 = 0 \)
- \( \text{compare } a = 1 \text{ with } b^k = 2^0 = 1 \)
  - \( a = b^k \) because \( k = 1 \)
- Since \( p > -1 \), case 2(a) applies: \( T(n) \in \Theta(n^{\log_b a} \lg n) = (\lg^n n) \in \Theta(\lg n) \)

Solving Recurrences: Master Method

Example: \( T(n) = T(n/2) + n^2 \)
- \( a = 1, b = 2, f(n) = n^2, n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \)
- \( \text{compare } f(n) = n^2 \text{ with } 1 \)
  - \( n^2 = \Omega(n^{\log_b a}) \) (so \( f(n) \) is polynomially larger)
- \( \text{Since } f(n) \text{ is a polynomial in } n, \text{ case 3 holds, } T(n) \in \Theta(n^2) \)

Example: \( T(n) = 4T(n/2) + n^2 \)
- \( a = 4, b = 2, f(n) = n^2, n^{\log_b a} = n^{\log_2 4} = n^2 \)
- \( \text{compare } f(n) = n^2 \text{ with } n^2 \)
  - \( n^2 = \Theta(n^{\log_b a}) \) (so \( f(n) \) is polynomially equal)
  - \( \text{Case 2 holds and } T(n) \in \Theta(n^2) \)

Solving Recurrences: Alt. Master Method

Example: \( T(n) = T(n/2) + n^2 \)
- \( a = 4, b = 2, k = 2, p = 0, \text{ and } n^{\log_b a} = n^{\log_2 4} = n^2 \)
- \( \text{compare } a = 4 \text{ with } b^k = 2^2 = 4 \)
  - \( 4 = 4 \)
- \( \text{Since } p > -1, \text{ case 2a) applies and } T(n) \in \Theta(n^{\log_b a} \lg n) = (\lg^n n) \in \Theta(\lg n) \)

Solving Recurrences: Master Method

Example: \( T(n) = 7T(n/3) + n^2 \)
- \( a = 7, b = 3, f(n) = n^2, n^{\log_b a} = n^{\log_3 7} = n^{1.25} \)
- \( \text{compare } f(n) = n^2 \text{ with } n^{1.25} \)
  - \( n^2 = \Omega(n^{\log_b a}) \) (so \( f(n) \) is polynomially larger)
- \( \text{Since } f(n) \text{ is a polynomial in } n, \text{ case 3 holds and } T(n) \in \Theta(n^2) \)

Example: \( T(n) = 7T(n/2) + n^2 \)
- \( a = 7, b = 2, f(n) = n^2, n^{\log_b a} = n^{\log_2 7} = n^{1.8} \)
- \( \text{compare } f(n) = n^2 \text{ with } n^{1.8} \)
  - \( n^2 = O(n^{\log_b a}) \) (so \( f(n) \) is polynomially smaller)
- \( \text{Case 1 holds and } T(n) \in \Theta(n^{\log_b a}) \)
Checking an Upper Bound

Give an upper bound on the recurrence: \( T(n) = 2T(\lfloor n/2 \rfloor) + n \).
Show \( T(n) \leq cn \lg n \) for some \( c > 0 \).

Assume \( T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor \).

\[
T(n) \leq 2c \lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor + n \\
= cn \lg n - cn2 + n \\
= cn \lg n \quad \text{for } c \geq 1.
\]

Mathematical induction on a good guess

Suppose \( T(n) = 1 \) if \( n = 2 \), and \( T(n) = T(n/2) + \Theta(1) \) if \( n = 2^k \), for \( k > 1 \).
Show \( T(n) = \lg n \) by induction on the exponent \( k \).

**Basis:** When \( k = 1 \), \( n = 2 \). \( T(2) = \lg 2 = 1 \).

**IHOP:** Assume \( T(2^i) = \lg 2^i \) for some constant \( k > 1 \).

**Inductive step:** Show \( T(2^{i+1}) = \lg(2^{i+1}) = k+1 \).

\[
T(2^{i+1}) = T(2^{i+1}/2) + 1 \quad \text{by definition of } T(n) \quad \text{\( */\)} \\
= T(2^i) + 1 \\
= (\lg 2^i) + 1 \quad \text{by inductive hypothesis \( */\)} \\
= k + 1 \quad \text{\( \lg 2^i = k \quad \text{*/}\)}
\]

Checking an Upper Bound Using Induction

Suppose \( T(n) = 1 \) if \( n = 1 \), and \( T(n) = T(n-1) + \Theta(n) \) for \( n > 1 \).
Show \( T(n) = O(n^2) \) by induction.

**Basis:** When \( n = 1 \). \( T(1) = 1^2 = 1 \).

**IHOP:** Assume \( T(i) = i^2 \) for all \( i < k \).

**Inductive step:** Show \( T(k) = k^2 \).

\[
T(k) = T(k-1) + k \quad \text{\( */\)} \\
= (k-1)^2 + k \quad \text{by inductive hypothesis \( */\)} \\
= k^2 - k + 1 \\
\leq k^2 \quad \text{for } k > 1
\]