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:

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

where

- \( a \) = number of sub-problems
- \( n/b \) = size of the sub-problems
- \( D(n) \) = time to divide the size \( n \) problem into sub-problems
- \( C(n) \) = time to combine the sub-problem solutions

Solving Recurrences (Ch. 4)

We will use the following 3 methods to solve recurrences
1. Backward Substitution: find a pattern and convert it to a summation.
2. Make a guess and prove it is correct using induction.
3. Apply the "Master Theorem": If the recurrence has the form

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

then there is a formula that can (often) be applied, given in Section 4-3.

To make the solutions simpler, we will
- Assume \( T(n) = \theta(1) \) for \( n \) small enough.

The Factorial Problem

Algorithm F(n)

Input: a positive integer \( n \)
Output: \( n! \)
1. if \( n=0 \)
2. return 1
3. else
4. return \( F(n-1) \times n \)

\[ T(n) = T(n-1) + 1 \]
\[ T(0) = 0 \]

Solving recurrences for \( n! \)

Algorithm F(n)

Input: a positive integer \( n \)
Output: \( n! \)
1. if \( n=0 \)
2. return 1
3. else
4. return \( F(n-1) \times n \)

\[ T(n) = T(n-1) + 1 \]
\[ T(0) = 0 \]

We can solve this recurrence (ie, find an expression of the running time that is not given in terms of itself) using backward substitution.

Solving Recurrences: Backward Substitution

Example: \( T(n) = 2T(n/2) + n \) (look familiar?)

\[
T(n) = 2T(n/2) + n = 2[2T(n/4) + n/2] + n \quad \text{expand } T(n/2)
= 4T(n/4) + n + n \quad \text{simplify}
= 4[2T(n/8) + n/4] + n + n \quad \text{expand } T(n/4)
= 8T(n/8) + n + n + n \quad \text{simplify...notice a pattern?}
= \ldots
= 2^{\omega(n/2^\omega)} + \ldots + n + n + n \quad \text{after \( \omega \) iterations}
= c2^{\omega(n)} + \log n
= cn + \log n \quad 2^{\omega(n)} = n
= \theta(n\log n)
Solving Recurrences: Backward Substitution

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

\[
T(n) = 4T(n/2) + n \\
= 4[4T(n/4) + n/2] + n \\
= 16T(n/4) + 2n + n \\
= 16[4T(n/8) + n/4] + 2n + n \\
= 64T(n/8) + 4n + 2n + n \\
= \ldots \\
= 4^{\lceil \log_2 n \rceil}T(n(2^{\lceil \log_2 n \rceil})) + \ldots + 4n + 2n + n \\
as \lceil \log_2 n \rceil \text{ iterations}
\]

\[
= c4^{\lceil \log_2 n \rceil} + n(2^{\lceil \log_2 n \rceil} - 1) \\
= cn^2 + n(n - 1) \\
= \Theta(n^2)
\]

---

**Binary Search (non-recursive)**

**Algorithm** BinarySearch(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. \( l = 1; r = n \)
2. while \( l \leq r \)
3. \( m = \lfloor (l + r)/2 \rfloor \)
4. if \( k = A[m] \) return \( m \)
5. else if \( k < A[m] \) \( r = m - 1 \)
6. else \( l = m + 1 \)
7. return -1

What is the running time of this algorithm for an input of size n? Are there best- and worst-case input instances?

---

**Binary Search (recursive)**

**Algorithm** BinarySearchRec(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. \( m = \lfloor (l + r)/2 \rfloor \)
4. if \( k = A[m] \) return \( m \)
5. else if \( k < A[m] \) return BinarySearchRec(A, k, l, m-1)
6. else return BinarySearchRec(A, k, m+1, r)

---

**Substitution Method:**

1. Guess the solution.
2. Use induction to find the constants and show that the solution works.

---

**Substitution Method:**

If the recurrence has an exact function (with no asymptotic terms), the solution must be exact and a full inductive proof must be done (with basis, IHOP, and inductive step).

If the recurrence has an asymptotic term, we express the solution by asymptotic notation and we don’t worry about base cases in the substitution proof.
Substitution Method:

Example: $T(n) = 1$ if $n=1$ or $2T(n/2) + n$ if $n>1$.
1. Guess: $T(n) = n\log n + n$.
2. Induction:
   Basis: $n = 1$ $\Rightarrow$ $n\log n + n = 0 + 1 = 1 = T(n)$
   IHOP: $T(k) = k\log k + k$ for all $k < n$
   Now use this IHOP for $T(n/2)$:
   
   $T(n) = 2T(n/2) + n = 2(n\log n/2 + n/2) + n$
   $= n\log n + n + n = n\log n - n\log 2 + n + n = n\log n - n + n + n$
   $= n\log n + n$

Substitution Method:

Guessing an Upper Bound

Give an upper bound on the recurrence: $T(n) = 2T(n/2) + \theta n$.
Guess $T(n) = \Theta(n\log n)$ by showing that $T(n) \leq dn\log n$ for some $d > 0$.
For easier analysis, assume that $n = 2^k$.

IHOP: Assume $T(n/2) \leq d(n/2)\log(n/2)$.

$T(n) \leq 2(d(n/2)\log(n/2)) + cn = d\log n - d\log 2 + cn = d\log n - dn + cn \leq d\log n$ if $-dn + cn \leq 0, d \geq c$.

Substitution Method:

Guessing a Lower Bound

Give a lower bound on the recurrence: $T(n) = 2T(n/2) + \theta n$.
Guess $T(n) = \Omega(n\log n)$ by showing that $T(n) \geq d(n\log n)$ for some $d > 0$.
For easier analysis, assume that $n = 2^k$.

IHOP: Assume $T(n/2) \geq d(n/2)\log(n/2)$.

$T(n) \geq d(n\log n/2) + cn = d\log n - d\log 2 + cn = d\log n - dn + cn \geq d\log n$ if $-dn + cn \geq 0, d \leq c$.

Substitution Method:

Guessing a Lower Bound

Give a lower bound on the recurrence: $T(n) = T(n/2) + 1$.
Guess $T(n) = \Omega(\log n)$ by showing that $T(n) \geq d\log n$ for some $d > 0$.
For easier analysis, assume that $n = 2^k$.

IHOP: Assume $T(n/2) \geq d\log(n/2)$.

$T(n) \geq d\log(n/2) + c = d\log n - d\log 2 + c = d\log n - d + c \geq d\log n$ if $-d + c \geq 0, d \leq c$.

Using induction to prove a tight bound

Suppose $T(n) = 1$ if $n = 2$, and $T(n) = T(n/2) + 1$ if $n = 2^k$, for $k > 1$.

Show $T(n) = \log n$ by induction on the exponent $k$.

Basis: When $k = 1, n = 2$. $T(2) = \log 2 = 1 = T(n)$.

IHOP: Assume $T(2^k) = \log 2^k = k$ for some constant $k > 1$.

Inductive step: Show $T(2^{k+1}) = \log 2^{k+1} = k+1$.

$T(2^{k+1}) = T(2^k) + 1$ /* given */
$= (\log 2^k) + 1$ /* by IHOP */
$= k + 1$
Suppose \( T(n) = 1 \) if \( n = 1 \), and \( T(n) = T(n-1) + \frac{n!}{(n-1)!} \) for \( n > 1 \).
Show \( T(n) = O(n^2) \) by induction.

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

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

\[
T(k) = T(k-1) + k = (k-1)^2 + k \quad \text{/* given */}
= k^2 - k + 1
\]

Using induction to prove an upper bound

**How do you come up with a good guess?**

Example: \( T(n) = T(n/3) + T(2n/3) + \Theta(n) \)

The recursion tree shows us the cost at each level of recursion.

**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:

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

Recall, \( a \) is the number of sub-problems and \( n/b \) is the size of each sub-problem. \( f(n) \) is the cost to divide and recombine for a solution.

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

1. \( T(n) = \Theta(n^d) \) if \( f(n) = O(n^d \log^k n) \) for some constant \( \varepsilon > 0 \)
2. \( T(n) = \Theta(n^d \log n) \) if \( f(n) = \Omega(n^d \log^{\kappa} n) \) and \( d = \log_b a \)
3. \( T(n) = \Theta(n^d \log^k n) \) if \( f(n) = \Omega(n^d \log^{\kappa} n) \) for some constant \( \varepsilon > 0 \)

**Why?** The proof uses a recursion tree argument (§4.6)

**Master Method Restated**

**Master Theorem:** If \( T(n) = aT(n/b) + O(n^d) \) for some constants \( a \geq 1, b > 1, d \geq 0 \), then

\[
T(n) = \begin{cases} 
O(n^d \log^k n) & \text{if } d < \log_b a \quad (a > b^\varepsilon) \\
O(n^d \log n) & \text{if } d = \log_b a \quad (a = b^\varepsilon) \\
O(n^d) & \text{if } d > \log_b a \quad (a < b^\varepsilon)
\end{cases}
\]

Why? The proof uses a recursion tree argument (§4.6)

**Regularity Condition**

**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:

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

Case 3 requires us to also show \( a(f(n/b)) \leq c(f(n)) \) for some non-negative constant \( c < 1 \) and all sufficiently large \( n \), the "regularity condition".

The regularity condition always holds when \( f(n) = n^\varepsilon \) and

\[
f(n) = \Omega(n^{b^\varepsilon - \varepsilon'})
\]

for constant \( \varepsilon' > 0 \).
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 $n^{\log_b a}$, but not polynomially smaller.

There is a gap between cases 2 and 3 when $f(n)$ is larger than $n^{\log_b a}$, 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.

Example:

- $T(n) = T(n/2) + n^2$
  - $a = 4, b = 2, f(n) = n^2, \log_b a = \log_2 4 = 2$
  - $n^2 = \Theta(n^2)$ (so $f(n)$ is polynomially equal to $n^2$)
  - Case 2 holds and $T(n) = \Theta(n^2 \log n)$

Example:

- $T(n) = 4T(n/2) + n^2$
  - $a = 4, b = 2, f(n) = n^2, \log_b a = \log_2 4 = 2$
  - $n^2 = \Theta(n^2)$ (so $f(n)$ is polynomially equal)
  - Case 2 holds and $T(n) = \Theta(n^2 \log n)$

Example:

- $T(n) = 7T(n/3) + n^2$
  - $a = 7, b = 3, f(n) = n^2, \log_b a = \log_3 7 = n^{1.585}$
  - $n^2 = \Theta(n^{1.585})$ (so $f(n)$ is polynomially larger)
  - $7(n/3)^2 \leq cn^2$ for $c = 7/9$, so case 3 holds and $T(n) = \Theta(n^2)$
Recursion Tree for Merge-Sort

\[ T(n) = \begin{cases} \frac{1}{2}T(n/2) + cn & \text{if } n = 1 \\ T(n) + cn & \text{otherwise} \end{cases} \]

Solving Recurrences: Master Method (§4.3)

A more general version of Case 2 follows:

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

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

Practice

Use the master method to give tight bounds for the running time of \( T(n) \) in each of the following recurrences. Specify which case of the master theorem holds for each problem.

a) \( T(n) = 2T(n/2) + n^3 \)

b) \( T(n) = T(2n/3) + 1 \)

c) \( T(n) = 16T(n/4) + n^2 \)

d) \( T(n) = 5T(n/2) + n^2 \)

e) \( T(n) = 5T(n/2) + n^3 \)