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:

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

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*

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:**
  $\log_b(xy) = \log_b x + \log_b y$
  $\log_b (x/y) = \log_b x - \log_b y$
  $\log_b x^a = a\log_b x$
  $\log_b a = \log_x a / \log_x b$ (reason log base doesn't matter, asymp)
  $a = b^{\log_b a}$ (e.g., $n = 2^{\lg n} = n^{\lg 2}$)
  $\lg^k n = (\lg n)^k$
  $\lg\lg(n) = \lg(\lg n)$

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. Apply the "Master Theorem": If the recurrence has the form
   $T(n) = aT(n/b) + f(n)$
   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:

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

where

- *n-1* = 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 recurrence with backward substitution

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

T(n) = T(n-1) + 1
T(0) = 0

$T(n) = T(n-1) + 1$ subst $T(n-1) = T(n-2) + 1$
$= [T(n-2) + 1] + 1 = T(n-2) + 2$
subst $T(n-2) = T(n-3) + 1$
$= [T(n-3) + 1] + 2 = T(n-3) + 3$
…
$= T(n-i) + i =$
…
$= T(n-n) + n = T(0) + n = 0 + n = O(n)$

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.*

Solving Recurrences: Backward Substitution

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

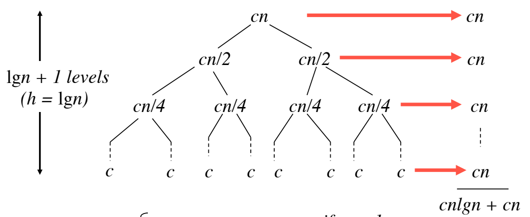
$T(n) = 2T(n/2) + n$
$= 2[2T(n/4) + n/2] + n$ /* expand T(n/2) */
$= 4T(n/4) + n + n$ /* simplify */
$= 4[2T(n/8) + n/4] + n + n$ /* expand T(n/4) */
$= 8T(n/8) + n + n + n$ /* simplify…see a pattern? */

. . . continue until $T(n/n) = T(1)$ is reached

$= 2^{\lg n}T(n/2^{\lg n}) + \ldots + n + n + n$ /* after lgn iterations */
$= 2^{\lg n}T(1) + \ldots + n + n + n$ /* $2^{\lg n} = n^{\lg 2} = n$ */
$= c2^{\lg n} + n\lg n$

$= cn + n\lg n$
$= O(n\lg n)$

Recursion Tree for T(n) = 2T(n/2) + n



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

Solving Recurrences: Backward Substitution

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

T(n)
= 2T(n/2) + 4n
= 2[2T(n/4) + 4(n/2)] + 4n /* expand T(n/2) */
= 4T(n/4) + 8n/2 + 4n /* simplify */
= 4T(n/4) + 4n + 4n
= 4[2T(n/8) + 4(n/4)] + 4n + 4n /* expand T(n/4) */
= 8T(n/8) + 16n/4 + 4n + 4n /* simplify…see a pattern? */
= 8T(n/8) + 4n + 4n + 4n
. . . continue until T(n/n) = T(1) is reached
= $2^{lgn}T(n/2^{lgn})$ + . . . + 4n + 4n + 4n /* after lgn iterations */
= nT(1) + lgn(4n) /* $2^{lgn} = n^{lg2} = n$ */
= cn + 4nlgn
= O(nlgn)

Solving Recurrences: Backward Substitution

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

T(n)
= 4T(n/2) + n
= 4[4T(n/4) + n/2] + n /* expand T(n/2) */
= 16T(n/4) + 4n/2 + n /* simplify */
= 16[4T(n/8) + n/4] + 2n + n /* expand T(n/4) */
= 64T(n/8) + 16n/4 + 2n + n /* simplify */
= 64T(n/8) + 4n + 2n + n
. . . continue until T(n/n) = T(1) is reached
= $4^{lgn}T(n/2^{lgn})$ + . . . + 4n + 2n + n /* after lgn iterations */
= $c4^{lgn} + n \sum_{k=0}^{lgn-1} 2^k = 2^0 + 2^1 + \ldots + 2^{lgn-1}$ /* convert to summation */ /* p. 1147 */
= $cn^{lg4} + n(2^{lgn} - 1)$ /* $4^{lgn} = n^{lg4} = n^2$ */
= $cn^2 + n(n - 1)$ /* $2^{lgn} = n^{lg2} = n$ */
= $O(n^2)$

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. l = 1; r =n
2. **while** l <= r
3. m = ⌊(l + r)/2⌋ ; m is midpoint
4. **if** k = A[m] **return** m ; found k, return index of k
5. **else if** k < A[m] r = m – 1 ; k is in lower half
6. **else** l = m + 1 ; k is in upper half
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 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 = ⌊(l + r)/2⌋ ; 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?

Solving Recurrences: Backward Substitution

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

T(n)
= T(n/2) + 1
= [T(n/4) + 1] + 1 /* expand T(n/2) */
= T(n/4) + 2 /* simplify */
= [T(n/8) + 1] + 2 /* expand T(n/4) */
= T(n/8) + 3 /* simplify */
= . . . . . .
= $T(n/2^{lgn})$ + lgn /* $2^{lgn} = n^{lg2} = n$ */
= T(1) + lgn
= c + lgn
= O(lgn)

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. $T(n) = \theta(n^{\log_b a})$ if $f(n) = O(n^{\log_b a-\varepsilon})$ for some constant $\varepsilon > 0$
2. $T(n) = \theta(n^{\log_b a}\lg n)$ if $f(n) = \theta(n^{\log_b a})$
3. $T(n) = \theta(f(n))$ if $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$

Solving Recurrences: Master Method

Intuition: Compare f(n) with $\theta(n^{\log_b a})$.

case 1: $f(n)$ is "polynomially smaller than" $\theta(n^{\log_b a})$
case 2: $f(n)$ is "asymptotically equal to" $\theta(n^{\log_b a})$
case 3: $f(n)$ is "polynomially larger than" $\theta(n^{\log_b a})$

What is $\log_b a$? 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. $T(n) = \theta(n^{\log_b a})$ if $f(n) = O(n^{\log_b a-\varepsilon})$ for some constant $\varepsilon > 0$
2. $T(n) = \theta(n^{\log_b a}\lg n)$ if $f(n) = \theta(n^{\log_b a})$
3. $T(n) = \theta f(n)$ if $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$ and if $a(f(n/b)) \leq c(f(n))$ for some positive constant $c < 1$ and all sufficiently large n.

Case 3 requires us to also show $a(f(n/b)) \leq c(f(n))$, the "regularity" condition.

The regularity condition *always* holds whenever $f(n) = n^k$ and $f(n) = \Omega(n^{\log_b a+\varepsilon})$, so we don't need to check it when f(n) is a polynomial.

Solving Recurrences: Master Method (§4.3)

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.

Solving Recurrences: Master Method (§4.3)

A more general version of Case 2 follows:

$T(n) = \theta(n^{\log_b a}\lg^{k+1} n)$ if $f(n) = \theta(n^{\log_b a}\lg^k n)$ 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.

Alternate Version of Master Method

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

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

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

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

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
  $n = O(n^{2-\varepsilon})$ (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
  $1 = \theta(n^0)$ (so f(n) is polynomially equal to $n^{\log_b a}$)
- case 2 applies: $T(n) \in \theta(n^0 \lg n) \in \theta(\lg n)$

Solving Recurrences: Alt. Master Method

Example 1a: $T(n) = 9T(n/3) + n$

- $a = 9, b = 3, k = 1, p = 0, \log_3 9 = 2$
- compare $a = 9$ with $b^k = 3^1 = 3$
  $9 > 3$
- case 1 applies: $T(n) \in \theta(n^{\log_3 9}) \in \theta(n^2)$

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

- $a = 1, b = 2, k = 0, p = 0$, and $\log_2 1 = 0$
- compare $a = 1$ with $b^k = 2^0 = 1$
  $a = b^0$ because $1 = 1$
- since $p > -1$, case 2(a) applies: $T(n) \in \theta(n^{\log_2 1} \lg n) = (n^0 \lg^{p+1} n)$
  $= (\lg^1 n) \in \theta(\lg n)$

Solving Recurrences: Master Method

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

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

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

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

Solving Recurrences: Alt. Master Method

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

- $a = 1, b = 2, k = 2, p = 0$, and $n^{\log_2 1} = n^0$
- compare $a = 1$ with $b = 2^k$, where $k = 2$
  $1 < 4$
- Since $p \geq 0$, case 3a) applies and $T(n) = \theta(n^2 \log^0 n) \in \theta(n^2)$

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

- $a = 4, b = 2, k = 2, p = 0$, and $n^{\log_2 4} = n^2$
- compare $a = 4$ with $b^k = 2^2 = 4$
  $4 = 4$
- Since $p > -1$, case 2a) applies and $T(n) = \theta(n^{\log_2 4} \log^1 n)$
  $\in \theta(n^2 \log n)$

Solving Recurrences: Master Method

Example: $T(n) = 7T(n/3) + n^2$

- $a = 7, b = 3, f(n) = n^2, n^{\log_b a} = n^{\log_3 7} = n^{1+\varepsilon}$
- compare $f(n) = n^2$ with $n^{1+\varepsilon}$
  $n^2 = \Omega(n^{1+\varepsilon})$ (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: $T(n) = 7T(n/2) + n^2$

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

Solving Recurrences: Alt. Master Method

Example: $T(n) = 7T(n/3) + n^2$

- $a = 7, b = 3, k=2, p = 0$, and $n^{\log_b a} = n^{\log_3 7} = n^{1+\varepsilon}$
- compare $a = 7$ with $b^k = 3^2$, $7 < 9$
- Since $p \geq 0$, case 3a) holds and $T(n) \in \theta(n^2 \log^0 n)$
  $\theta(n^2)$

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

- $a = 7, b = 2, k = 2, p = 0$, $n^{\log_b a} = n^{\log_2 7} = n^{1+\varepsilon}$
- compare $a = 7$ with $b^k = 2^2$, $a > b$ because $7 > 4$
- Case 1 holds and $T(n) \in \theta(n^{\log_2 7})$

Checking an Upper Bound

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

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

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\le cn\lg(n/2) + n \\ &= cn\lg n - cn\lg 2 + n \\ &= cn\lg n - cn + n \\ &\le cn\lg n \quad \text{for } c >= 1. \end{aligned}$$

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.

<u>*Basis:*</u> *When* $k = 1, n = 2$. $T(2) = \lg 2 = 1$.

<u>*IHOP:*</u> *Assume* $T(2^k) = \lg 2^k$ for *some constant* $k > 1$.

<u>*Inductive step:*</u> *Show* $T(2^{k+1}) = \lg(2^{k+1}) = k+1$.

$$\begin{aligned} T(2^{k+1}) &= T(2^{k+1}/2) + 1 && \text{/* by definition of T(n) */} \\ &= T(2^k) + 1 \\ &= (\lg 2^k) + 1 && \text{/* by inductive hypothesis */} \\ &= k + 1 && \text{/* } \lg 2^k = k \text{ */} \end{aligned}$$

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.

<u>*Basis:*</u> *When* $n = 1$. $T(1) = 1^2 = 1$.

<u>*IHOP:*</u> *Assume* $T(i) = i^2$ for *all* $i < k$.

<u>*Inductive step:*</u> *Show* $T(k) = k^2$.

$$\begin{aligned} T(k) &= T(k-1) + k && \text{/* given */} \\ &= (k-1)^2 + k && \text{/* by inductive hypothesis */} \\ &= k^2 - k + 1 \\ &\le k^2 \quad \text{for } k > 1 \end{aligned}$$