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

Solving Recurrences

We will use the following methods to solve recurrences

1. Backward Substitution.
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 $T(n) = aT(n/b) + f(n)$ will be solved using the Master Theorem.

To make the solutions simpler, we will

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

Solving recurrence with backward substitution

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

Analyzing Recursive Algorithms

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

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

where

- a = number of subproblems (1)
- $n-1$ = size of the subproblems (in terms of n)
- $D(n)$ = time to divide the size n problem into subproblems = 1
- $C(n)$ = time to combine the subproblem solutions to get the answer for the problem of size n = 1

Solving Recurrences: Backward Substitution

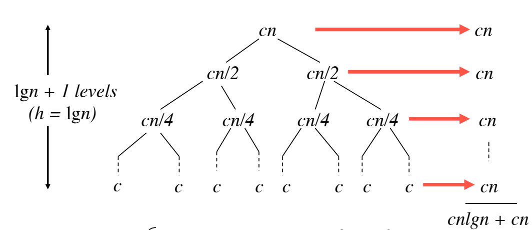
Example: T(n) = 2T(n/2) + n

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

=	$2^{lgn}T(n/2^{lgn})$ + . . . + n + n + n	/* after lgn iterations */
=	$2^{lgn}T(1)$ + . . . + n + n + n	/* $2^{lgn} = n^{lg2}$ = n */
=	$c2^{lgn}$ + nlgn	
=	cn + nlgn	
=	O(nlgn)	

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 (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
if (l > r) return -1              ; k not found
else
   m = ⌊(l + r)/2⌋                ; m is midpoint
   if k = A[m]
      return m                    ; found index of k at A[m]
   else if k < A[m]
        return Binary-Search-Rec(A, k, l, m-1)   ; look in lower half
   else
        return Binary-Search-Rec(A, k, m+1, r)   ; look in upper half
```

What is the running time of this algorithm for an input of size n? We need to figure out what T(n) is for this algorithm.

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

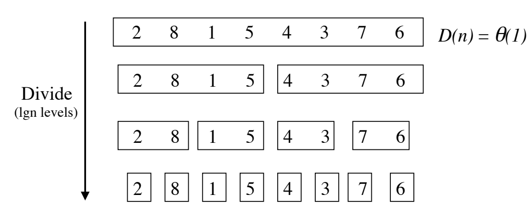
= O(lgn)

Analysis of Divide-and-Conquer Algorithms

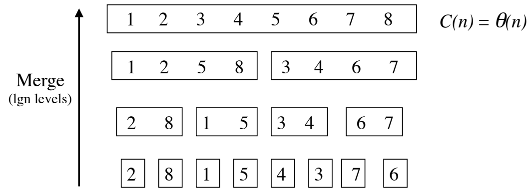
<u>**The divide-and-conquer paradigm (Ch.2)**</u>
- **divide the problem into a number of subproblems**
- **conquer the subproblems (solve them)**
- **combine the subproblem solutions to get the solution to the original problem**

<u>Example: Merge Sort</u>
- **divide the n-element sequence to be sorted into two n/2-element sequences.**
- **conquer the subproblems recursively using merge sort.**
- **combine the resulting two sorted n/2-element sequences by merging.**

Analyzing Merge-Sort



Analyzing Merge-Sort



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

Recurrence for worst-case running time for Merge-Sort

Merge-Sort(A,p,r)

```
if p < r then
    q ← ⌊(p+r)/2⌋
    Merge-Sort (A,p,q)
    Merge-Sort (A,q+1,r)
    Merge (A,p,q,r)
```

Initial call:

`Merge-sort(A,1,length(A))`

The Merge subroutine takes $\theta(n)$ time to merge n elements that are divided into two *sorted* arrays of $n/2$ elements each.

Merge(A,p,q,r)

```
n1← q-p+1; n2← r-q;
Create arrays
 L[1...n1+1] and
 R[1...n2+1]
for i← 1 to n1
    L[i]← A[p+i-1]
for i← 1 to n2
    R[i]← A[q+i]
L[n1+1] = R[n2+1] = ∞
i ← j ← 1
for k ← p to r
   if L[i] ≤ R[j]
      A[k]← L[i]
      i ← i+1
   else A[k]← R[j]
      j ← j+1
```

Analyzing Merge-Sort

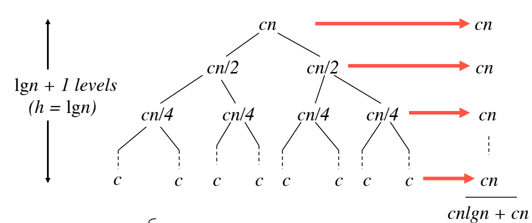
$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(n/2) + \theta(n) & \text{otherwise} \end{cases}$$

Recurrence for worst-case running time for Merge-Sort

$$aT(n/b) + D(n) + C(n)$$

- a = 2 (two subproblems)
- n/b = $n/2$ (each subproblem has size approx. $n/2$)
- $D(n) = \Theta(1)$ (just compute midpoint of array)
- $C(n) = \Theta(n)$ (merging can be done by scanning sorted subarrays)

Recursion Tree for Merge-Sort



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

Recurrence for worst-case running time of Merge-Sort

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).

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

$$T(n) = \begin{cases} \theta(n^{\log_b a}) & \text{if } d < \log_b a \quad (a > b^d) \\ \theta(n^d \lg n) & \text{if } d = \log_b a \quad (a = b^d) \\ \theta(n^d) & \text{if } d > \log_b a \quad (a < b^d) \end{cases}$$

Why? The proof uses a recursion tree argument (given in our textbook).

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

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 $1 = 1$
- 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 $b^k = 3^2, 7 < 9$
- Case 3) holds and $T(n) = \theta(n^2 \log^0 n) = \theta(n^2)$

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 $T(n) \in \theta(n^{\log_2 7})$