Computational Problem Solving

- Problems
- Designing solution strategies
- Developing algorithms (iterative and recursive)
  - Writing algorithms that implement the strategies
  - Understand existing algorithms and modify/reuse
- Understanding an algorithm by simulating its operation on an input
- Checking/proving correctness
- Analyzing and comparing performance: complexity or efficiency
  - Theoretically: Using a variety of mathematical tools
  - Empirically: Code, run and collect performance data
- Choosing the best

Complexity Analysis of Recursive Algorithms

Chapter 4: OMIT 4.2 & 4.6

Keep up with the reading assignments!

Why can't we use the complexity calculation technique that we have been discussing for non-recursive algorithms?

Because of recursion!

- Fibonacci numbers: $F_0 = 1$; $F_1 = 1$; $F_n = F_{n-1}+F_{n-2}$

Fib (n: non-negative integer)

1 if n=0 or 1 then return 1

2 else return Fib(n-1)+Fib(n-2)

So what do we do?

- Develop a pair of equations (called “recurrences” or “recurrence relations”) that characterize the behavior of a recursive algorithms and
- Solve those to obtain the algorithm’s complexity.

Recurrences

What are these?

A pair of equations giving T(n) of recursive algorithms in terms of the
cost of recursive calls
and the cost of other steps

Step 1: Develop Recurrence Relations

How?

1. Determine what the base case(cases) is (are).
2. Determine steps that will be executed when the input size matches the base case(s).
3. Calculate the complexity of those steps.
4. Write the first part of the RR as T(base case input sizes) = what you calculated

Step 1: Develop Recurrence Relations

How?

5. Determine steps that will be executed when the input size is n, different from the base case(s).
6. Determine how many recursive calls (and with what input size in relation to the original input size of n) will be made.
7. Calculate the complexity of all other steps, excluding the recursive calls.
8. Write the second part of the RR as T(n) = T(input size for first recursive call) + T(input size for next recursive call) +…+complexity of all other steps.

Recursive algorithm example

- $T(n) = 4 = \Theta(1)$ when $n < 2$
- $T(n) = T(n-1) + T(n-2) + 7 = T(n-1) + T(n-2) + \Theta(1)$ when $n \geq 2$

Fib (n: non-negative integer)

1 if n=0 or 1 then return 1

2 else return Fib(n-1)+Fib(n-2)

Divide & Conquer Algorithm

- ***A recursive algorithm that***
  - ***divides the input each time into non-overlapping parts & apply itself to these smaller inputs until the base case is reached, and then***
  - ***recursively combines the subproblem solutions to obtain a solution to the overall problem***
- ***Example: Merge Sort***

Divide & Conquer Algorithm

Find-Max-Recursive(A:array[i…j] of numbers)

if i=j then return A[i]

else

  mid=floor((i+j)/2)

  return max(Find-Max-Recursive(A[i…mid]), Find-Max-Recursive(A[mid+1…j]))

Thinking Assignments

- understand this algorithm
- is it correct? can you prove it? how?
- can you draw a recursion tree for A=[1,0,-5,7,23]?
- develop its recurrences

D & C Algorithms

- General form of divide-and-conquer algorithm recurrences

$$T(n) = \begin{cases} \Theta(1) & if\, n \le c \\ aT(n/b) + f(n) & otherwise \end{cases}$$

- Recursion tree method can be used to solve these kinds of recurrences, like we did for Merge Sort

Reading Assignment

- ***Section 4.1***
  - ***understand the maximum subarray problem***
  - ***understand the divide and conquer solution***
  - ***understand FIND-MAX-CROSSING-SUBARRAY***
    - ***simulate it on some specific arrays using paper and pen***
  - ***understand FIND-MAXIMUM-SUBARRAY***
    - ***draw the recursion tree for some specific arrays***
    - ***understand its recurrences (equations 4.7)***
    - ***Thinking Assignment: do exercise 4.1-1***
  - ***come to next class with questions/confusions***
  - ***other than answering your questions, this section will not be discussed in class***

Solving Recurrences

1. Recursion-tree method (4.4)
2. Substitution method (4.3: omit “changing variables”)
3. Master method (4.5)
4. Backward substitution method (not in text)
5. Forward substitution method (not in text)

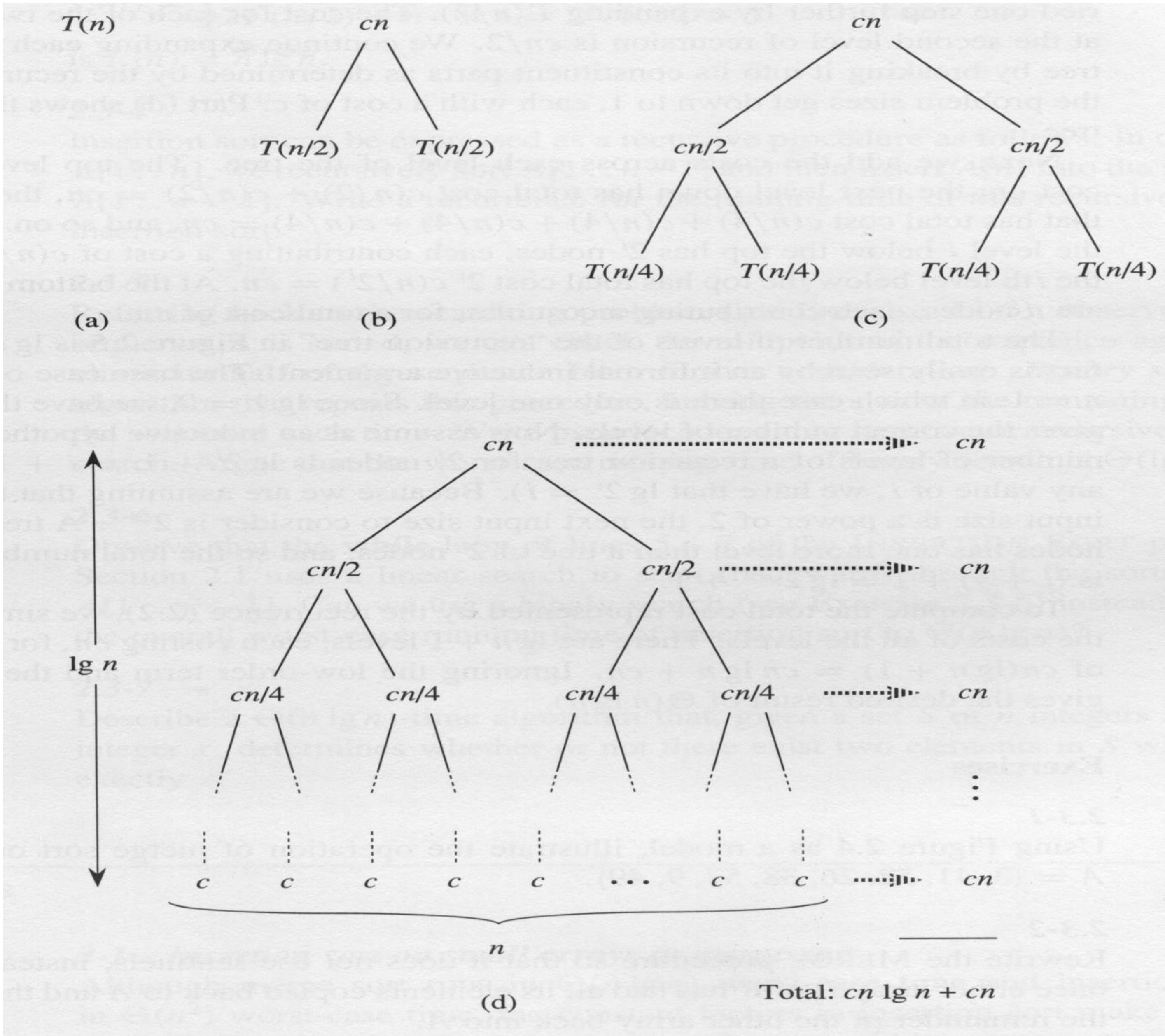
- Reading Assignments: 4.3, 4.4, 4.5

Recurrence relations of Merge Sort

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

$$\Downarrow \Uparrow$$

$$T(n) = \begin{cases} \Theta(1) & if \quad n = 1 \\ 2T(n/2) + \Theta(n) & if \quad n > 1 \end{cases}$$



T(n)
cn
T(n/2) T(n/2)
cn
cn/2 cn/2
T(n/4) T(n/4) T(n/4) T(n/4)
(a) (b) (c)
cn → cn
cn/2 cn/2 → cn
lg n
cn/4 cn/4 cn/4 cn/4 → cn
c c c c c ... c c → cn
n
Total: cn lg n + cn
(d)

Analysis of Merge Sort

$$T(n) = cn\log n + cn = \Theta(n\log n)$$

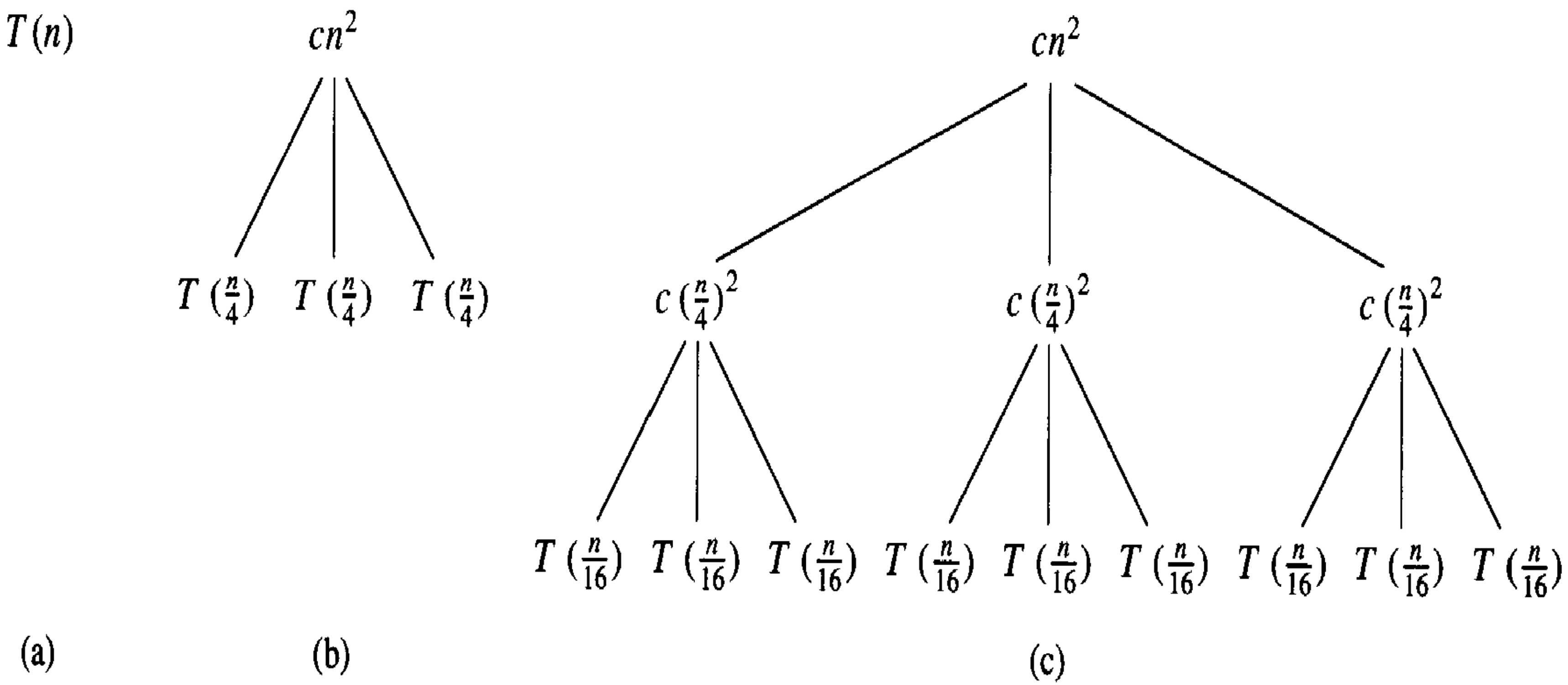
Another Example

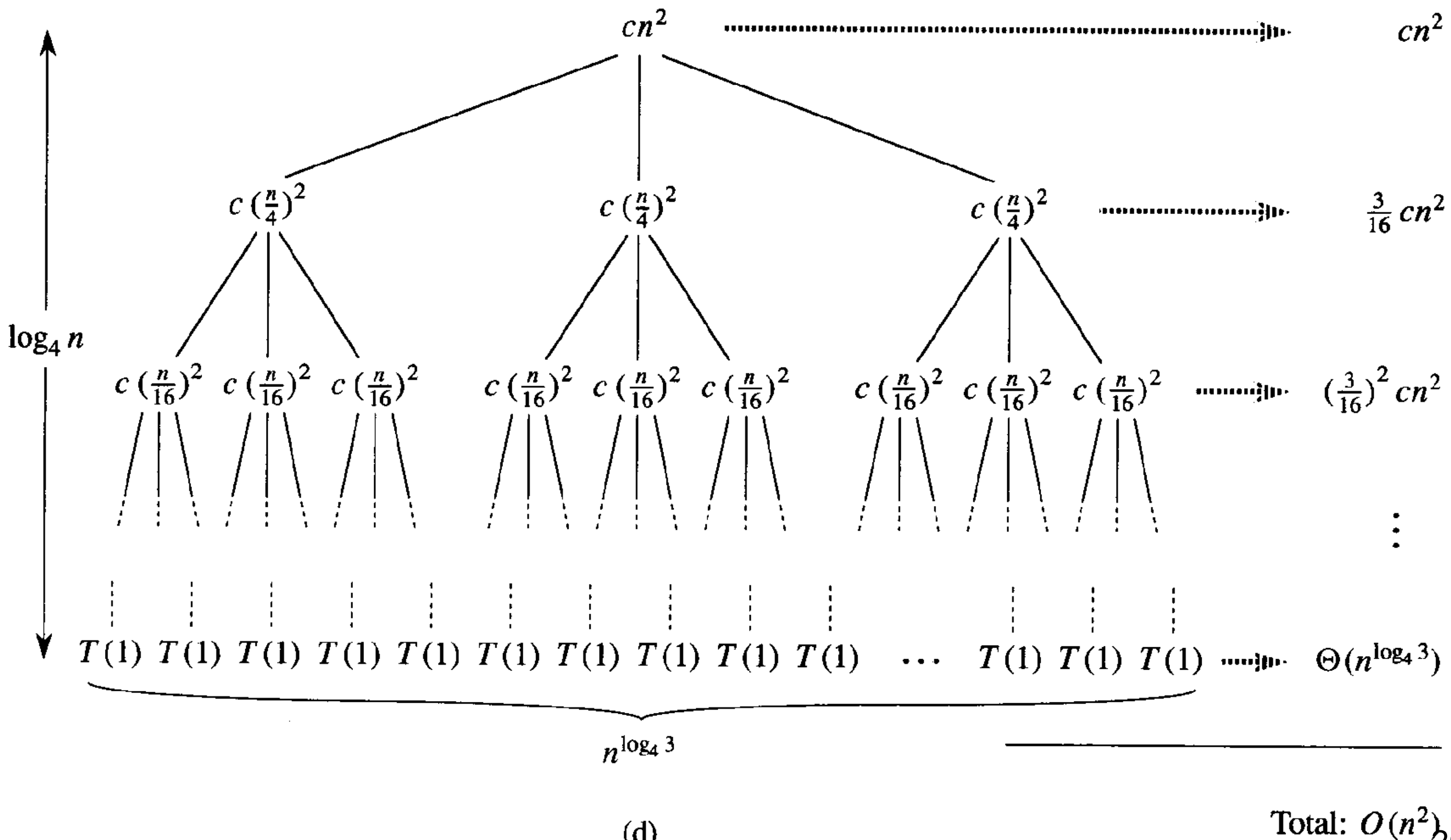
- Suppose a divide-and-conquer algorithm has these recurrences

$$T(n) = \begin{cases} \Theta(1) & if\, n = 1 \\ 3T(\lfloor n/4 \rfloor) + \Theta(n^2) & otherwise \end{cases}$$

Recursion-tree method

$$T(n) = \begin{cases} c & if\, n = 1 \\ 3T(\lfloor n/4 \rfloor) + cn^2 & otherwise \end{cases}$$





cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$\log_4 n$
$T(1)$
$n^{\log_4 3}$
cn^2
$\frac{3}{16}cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\Theta(n^{\log_4 3})$
Total: $O(n^2)$

(d)

The cost of the entire tree

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$= \frac{1}{1-(3/16)} cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$

$$I.e. T(n) < a\ quadratic\ polynomial, so\ T(n) = o(n^2)$$

Alternate solution using finite summation result

The substitution method

- The substitution method for solving recurrence entails two steps:
  1. Guess the form of the solution.
  2. Use mathematical induction to find the constants and show that the solution works.
  3. If the inductive step of the proof fails, your guess is probably wrong.

How do you guess?

(1) If a recurrence is similar to one you have seen, guess a similar solution.

(2) Draw the recursion tree to get an approximate idea of what the solution might be.

(3) Trial and error.

Example of (1)

$$\begin{cases} T(n) = 2T(\lfloor n/2 \rfloor) + n \\ T(1) = 1 \end{cases}$$

Guess $T(n) = O(n \log n)$

Now show using induction: $T(n) \le cn \log n$

$$\begin{cases} T(n) = 2T(\lfloor n/2 \rfloor) + n \\ T(1) = 1 \end{cases}$$

Base case

$$1 = T(1) \le c1\log 1 = 0?$$

However, $4 = T(2) \le c2\log 2 = 2c \ \ (\text{if } c \ge 2)$

$$\begin{cases} T(n) = 2T(\lfloor n/2 \rfloor) + n \\ T(1) = 1 \end{cases}$$

Now Assume

$$T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log\lfloor n/2 \rfloor$$

We have to show $T(n) \leq cn\log n$

$$T(n) \leq 2(c\lfloor n/2 \rfloor \log\lfloor n/2 \rfloor) + n \leq cn\log\frac{n}{2} + n$$

$$= cn\log n - cn\log 2 + n = cn\log n - n(c-1)$$

$$\leq cn\log n \quad (\text{if } c \geq 1.)$$

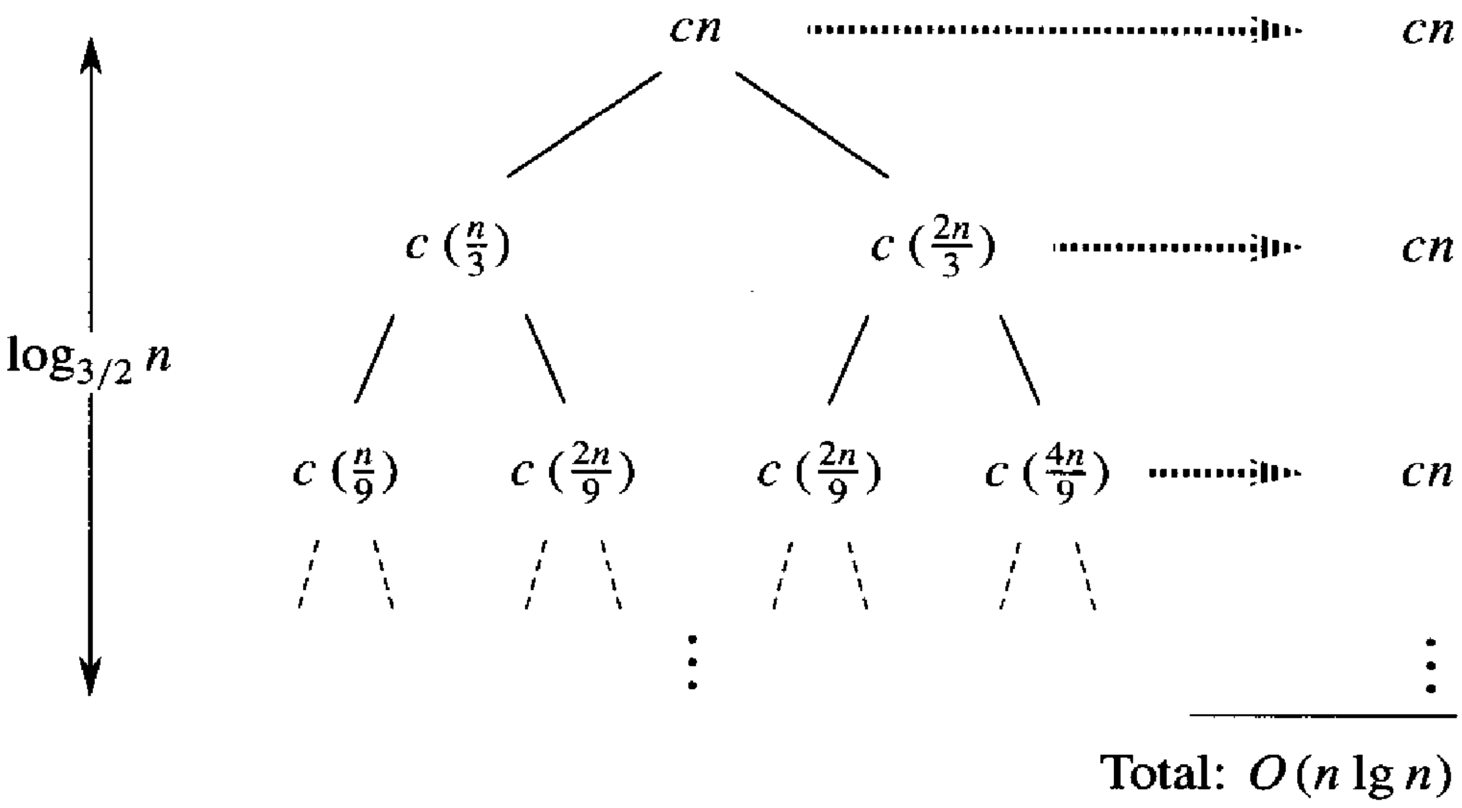
Example of (2)

$$T(n) = T(n/3) + T(2n/3) + cn$$

$$T(2) = 2$$

- Guess a solution using the recursion tree method approximately
- Then prove that it is correct

$$T(n) = T(n/3) + T(2n/3) + cn$$



$$T(n) = T(n/3) + T(2n/3) + cn$$

$$T(2) = 2$$

We need to prove by induction that

$$T(n) \le dn \log n$$

Base case

$$2 = T(2) \le d\,2 \log 2 = 2d \quad (\text{if } d \ge 1)$$

$$T(n) = T(n/3) + T(2n/3) + cn$$

$$T(2) = 2$$

Now Assume

$$T(n/3) \le d(n/3)\log(n/3)$$

$$T(2n/3) \le d(2n/3)\log(2n/3)$$

We have to show $T(n) \le dn \log n$

$$T(n) \le \frac{dn}{3}(\lg n - \lg 3) + \frac{2dn}{3}(\lg n - \lg(3/2)) + cn$$

$$= dn \lg n - dn(\frac{1}{3}\lg 3 + \frac{2}{3}\lg\frac{3}{2}) + cn$$

$$= dn \lg n - (dn(\lg 3 - \frac{2}{3}) - cn) \le dn \lg n \ \ when$$

$$(dn(\lg 3 - \frac{2}{3}) - cn) > 0 \ or \ d > c/(\lg 3 - \frac{2}{3})$$

Subtleties

Example of (3): Trial and error: correct guess; fixable problem with proof

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1;\ T(1) = 1$$

- Guess $T(n) = O(n)$
- Show $T(n) \le cn$
- Base Case: $T(1) = 1 \le c * 1\, when\, c\, is\, at\, least\, 1$
- Assume $T(n/2) \le cn/2$
- Inductive Step: proof fails

$$T(n) \le cn/2 + cn/2 + 1 \le cn + 1 \not\le cn$$

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

- Instead, show $T(n) = O(n) \Rightarrow T(n) \leq cn - 1$

- Base case:

$$T(1) = 1 \leq (c-1)\, holds\, when\, c\, is\, at\, least\, 2$$

- Assume $T(n/2) \leq cn/2 - 1$

- Then

$$T(n) \leq (cn/2 - 1) + (cn/2 - 1) + 1$$

$$T(n) \leq cn - 1$$

- **This technique can be used if the inductive step fails due to a constant additive or subtractive term**

Example of (3): Trial and error (wrong guess)

Avoiding pitfalls

$$\begin{cases} T(n) = 2T(\lfloor n/2 \rfloor) + n \\ T(1) = 1 \end{cases}$$

- Guess $T(n) = O(n)$
- Works for the base case:
  T(1)=1 ≤ c*1when c≥1
- Show $T(n) \le cn$
- Assume $T(n/2) \le cn/2$

$$T(n) \le 2(cn/2) + n \le cn + n = n(c+1) \le cn??$$

- You cannot find such a positive *constant*.
- This is an example of inductive proof failure suggesting that your guess is WRONG

Another example

(for you to read and understand)

$T(n)=3T(\lfloor n/4 \rfloor) + cn^2$, $T(1) = c$

Guess $T(n)=O(n^2)$

We want to show that $T(n) \leq dn^2$ for some constant $d > 0$. Works for the base case (why?) . So assume $T(n/4) \leq d(n/4)^2$

$$\begin{aligned} T(n) &= 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leq 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &= n^2(d\frac{3}{16} + c) \\ &\leq dn^2, when (d\frac{3}{16} + c) \leq d, or \frac{16}{13}c \leq d \end{aligned}$$

Changing variables method omit

$$T(n) = 2T(\sqrt{n}) + \lg n$$

Let $m = \lg n$.

$$T(2^m) = 2T(2^{m/2}) + m$$

Let $S(m) = T(2^m)$

$$T(2^{m/2}) = S(m/2)$$

To deal with recursive call input sizes that are square roots, cube roots and powers of n less than 1

Then $S(m) = 2S(m/2) + m$.

$$\Rightarrow S(m) = O(m \lg m)$$

$$\Rightarrow T(n) = T(2^m) = S(m) = O(m \lg m)$$

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

The master method

No need to memorize
Learn how to apply

The Master Theorem

If T(n)=aT(n/b) + f(n) (and T(base case)=some constant) and a and b are constants, then:

$$1: if\ \ f(n) = O(n^{(\log_b a) - \varepsilon})\ for\ \ \varepsilon > 0\ then\ T(n) = \Theta(n^{\log_b a})$$

$$2: if\ \ f(n) = \Theta(n^{\log_b a})\ \ then\ T(n) = \Theta(n^{\log_b a} \lg n)$$

$$3: if\ \ f(n) = \Omega(n^{(\log_b a) + \varepsilon})\ \ \ for\, \varepsilon > 0$$

$$and\, if\ \ af(n/b) \le cf(n)\ for\, some\, cons\tan t\, c < 1$$

$$then\ T(n) = \Theta(f(n))$$

- $T(n) = 9T(n/3) + n$

$a = 9, b = 3, \ f(n) = n$

$n^{\log_3 9} = n^2, \quad f(n) = O(n^{\log_3 9 - 1})$

$Case \quad 1 \Rightarrow T(n) = \Theta(n^2)$

- $T(n) = T(2n/3) + 1$

$a = 1, b = 3/2, \ f(n) = 1$

$n^{\log_{3/2} 1} = n^0 = 1 = f(n),$

$Case \quad 2 \Rightarrow T(n) = \Theta(\log n)$

- $T(n) = 3T(n/4) + n\log n$

$$a = 3, b = 4, f(n) = n\log n$$

$$n^{\log_4 3} = n^{0.793}, f(n) = \Omega(n^{\log_4 3+\varepsilon})$$

Case 3

Check

$$af(n/b) = 3(\frac{n}{4})\log(\frac{n}{4}) \leq \frac{3n}{4}\log n = cf(n)$$

for $c = \frac{3}{4}$, and sufficiently large n

$$\Rightarrow T(n) = \Theta(n\log n)$$

When master method doesn't apply omit

- The master method does not apply to the recurrence $T(n) = 2T(n/2) + n\lg n,$

  even though it has the proper form: *a* = 2, b=2, f(*n*)= *n* lg*n*, and $n^{\log_b a} = n.$
- It might seem that case 3 should apply, since f(*n*)= *n*lg*n* is asymptotically larger than $n^{\log_b a} = n.$
- The problem is that it is NOT polynomially larger, i.e., **not larger by a term n^{ε}**.

Method of backward substitutions

- Start with the recurrence relation for T(n), and
- repeatedly expand its right hand side by substituting for the T terms.
- After several such expansions, look for and find a pattern that allows you to express T(n) as a closed-form formula.
- If such a formula is evident, <u>check its validity</u> by direct substitution into the recurrence relations.

$T(n)=T(n-1)+n;\ T(0)=1$

$T(n-1)=T(n-2)+(n-1)$

So $T(n)=T(n-2)+n+(n-1)$

$T(n-2)=T(n-3)+(n-2)$

So $T(n)=T(n-3)+n+(n-1)+(n-2)$

$T(n-3)=T(n-4)+(n-3)$

So $T(n)=T(n-4)+n+(n-1)+(n-2)+(n-3)$

Eventually, $T(n)=T(n-n)+n+(n-1)+(n-2)+(n-3)+\ldots+(n-(n-1))=T(0)+n+(n-1)+(n-2)+(n-3)+\ldots+1= 1+ n(n+1)/2$

Check:

LHS of recurrence $T(n) = 1+ n(n+1)/2 = n^2/2+n/2+1$

RHS $= T(n-1) + n = 1+ (n-1)n/2 + n = n^2/2+n/2+1$

Method of forward substitutions

Start with the recurrence relation for T(base case), and repeatedly calculate non-base cases, e.g., T(1), T(2) etc. After several such calculations, look for and find a pattern that allows you to express T(n) as a closed-form formula. If such a formula is evident, check its validity by direct substitution into the recurrence relations.

$T(n)=T(n-1)+1;\ T(0)=1$

$T(1)=T(0)+1=1+1=2$

$T(2)=T(1)+1=2+1=3$

$T(3)=T(2)+1=3+1=4$

$T(4)=T(3)+1=4+1=5$

$T(5)=T(4)+1=5+1=6$

…

$T(n)=n+1$

Check:

LHS = $T(n) = n+1$

RHS = $T(n-1)+1 = n+1$

Complexity of Recursive Algorithms

- First develop the recurrences from the algorithm
- Then solve them using the most appropriate method

How do you know which method to apply?

Substitution method: generally applicable

Recursion tree method and Master method: for Divide and Conquer algorithms that reduce inputs by a fixed factor

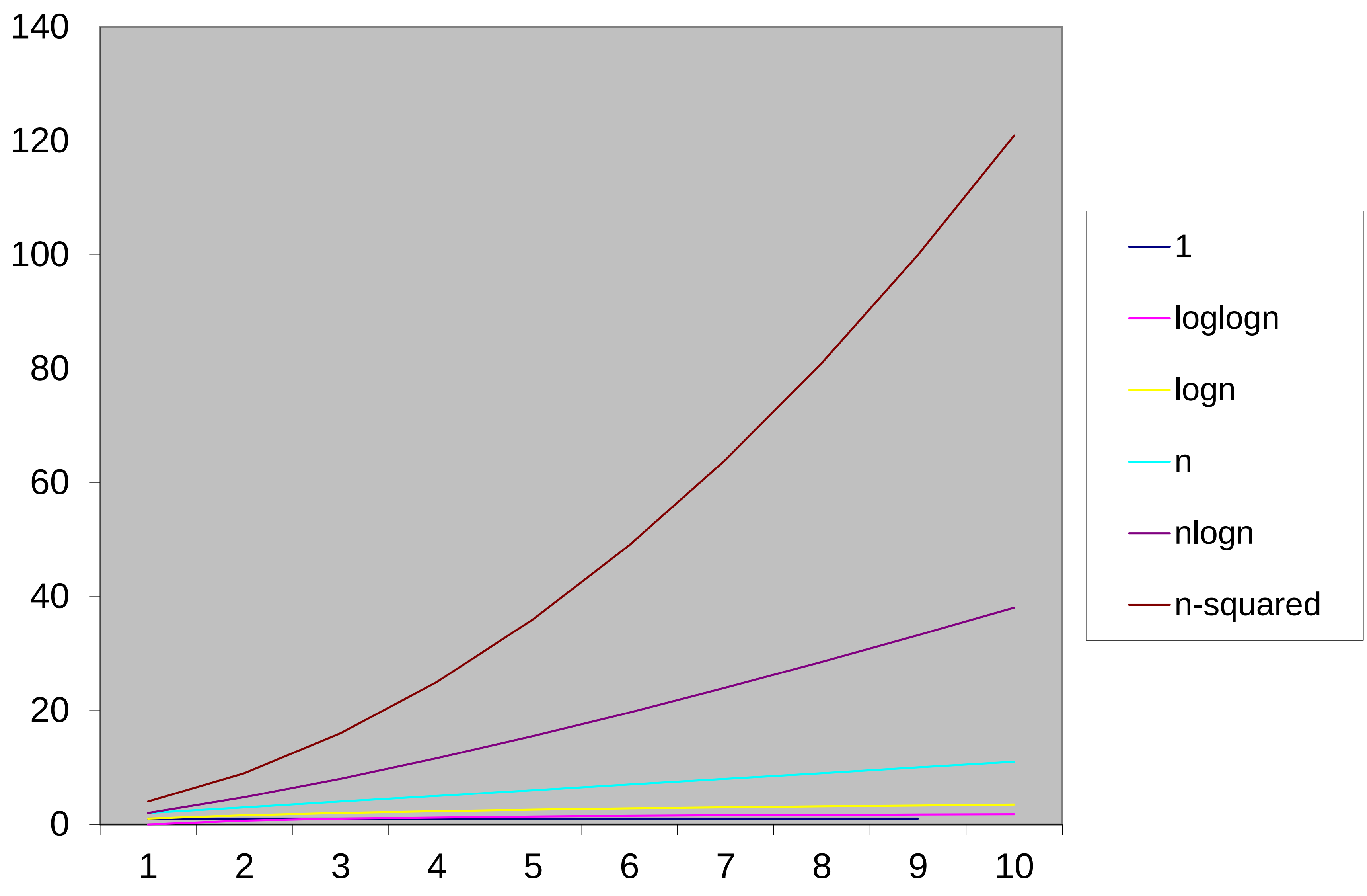
Backward/Forward substitution method: for algorithms that reduce input by a constant amount

Summary

- We have discussed several tools and techniques for mathematically determining the complexity of algorithms:
  - For non-recursive algorithms, calculate T(n) by adding up the (cost * # of executions) of each step
  - For recursive algorithms, develop the recurrence relations and solve them using a variety of techniques to obtain T(n)
  - Once you obtain an exact expression for T(n) [or through various approximations an upper or lower bound for T(n)] as a function of n, then you can determine the order of complexity of the algorithm.

Empirical Complexity

- Another approach is to determine T(n) by plotting it as a graph of actual time taken by the algorithm versus input size by:
  - Coding the algorithm in a programming language
  - Randomly generating inputs of different sizes: typically from small sizes up to 100K's or millions
  - Running the program on each of these inputs and measuring the time taken using the system clock
  - Plotting time against input size
  - Determining the appropriate g(n) that fits this graph or provides an upper or lower bound (see next slide for examples of g(n))
  - This function g will then give you the order of complexity of the algorithm


140
120
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10
1
loglogn
logn
n
nlogn
n-squared