3 Growth of Functions

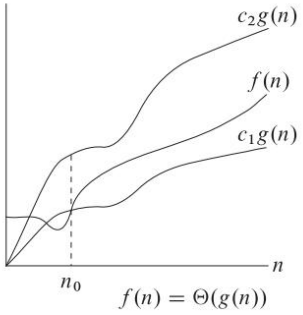
- Order of growth of the running time
  - gives a simple characterization of the algorithm's efficiency
  - allows us to compare the relative performance of alternative algorithms
- Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs

3.1 Asymptotic notation

- For a function $g(n)$ we denote by $\Theta(g(n))$ the set of functions

$\Theta(g(n)) = \{ f(n)$ there exist positive constants $c_1, c_2,$ and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$

- A function $f(n)$ belongs to $\Theta(g(n))$ if there exist c_1 and c_2 such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$ for sufficiently large n
- Because $\Theta(g(n))$ is a set, we could write "$f(n) \in \Theta(g(n))$"
- We usually write "$f(n) = \Theta(g(n))$" instead



- When $f(n) = \Theta(g(n))$ we say that $g(n)$ is an ***asymptotically tight bound*** for $f(n)$
- The definition requires that every member $f(n) \in \Theta(g(n))$ be asymptotically nonnegative
  - $f(n)$ be nonnegative whenever n is sufficiently large
- The function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty

- Let us use the formal definition to show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
  - We must determine positive c_1, c_2, and n_0 such that $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$ for all $n \geq n_0$
  - Dividing by n^2 yields $c_1 \leq \frac{1}{2} - 3/n \leq c_2$
  - We can make the right-hand inequality hold for any value of $n \geq 1$ by choosing $c_2 \geq \frac{1}{2}$
  - Likewise, we can make the left-hand inequality hold for any value of $n \geq 7$ by choosing $c_1 \leq 1/14$
  - Thus, by choosing $c_1 \leq 1/14$, $c_2 \geq \frac{1}{2}$, and $n \geq 7$ we can verify that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

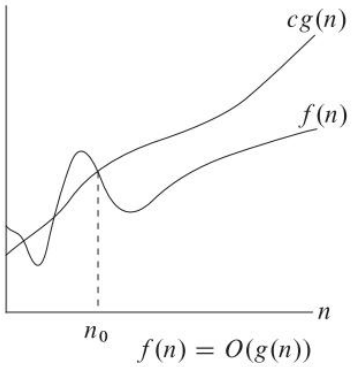
- We can also use the formal definition to verify that $6n^3 \neq \Theta(n^2)$
- Suppose for the purpose of contradiction that c_2 and n_0 exist such that $6n^3 \leq c_2 n^2$ for all $n \geq n_0$
- But then dividing by n^2 yields $n \leq c_2/6$, which cannot possibly hold for arbitrarily large n, since c_2 is constant

O-notation

- Θ-notation asymptotically bounds a function from above and below
- When we have only an asymptotic upper bound, we use O-notation
- For a function $g(n)$ we denote by $O(g(n))$ the set of functions

$O(g(n)) = \{ f(n) :$ there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$

- Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since Θ-notation is a stronger notion than O-notation
- When we use O-notation to bound the worst-case running time, we have a bound on the running time of the algorithm on every input



Ω-notation

- **Ω** –notation provides an asymptotic lower bound on a function
- For a function $g(n)$ we denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$$

Theorem 3.1

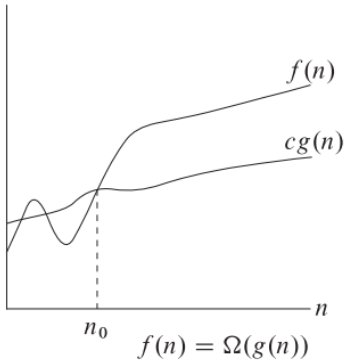
For any two functions $f(n)$ and $g(n)$, we have

$$f(n) = \Theta(g(n))$$

if and only if

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Proof Exercises. ■



- The running time of insertion sort is $\Omega(n)$
- The running time of insertion sort therefore belongs to both $\Omega(n)$ and $O(n^2)$
  - E.g., the running time of insertion sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time
- It is **not** contradictory, however, to say that the *worst-case* running time of insertion sort is $\Omega(n^2)$, since there exists an input that causes the algorithm to take $\Omega(n^2)$ time

o-notation

- The asymptotic bound provided by O-notation may or may not be asymptotically tight:
  - The bound $2n^2 = O(n^2)$ is asymptotically tight
  - The bound $2n = O(n^2)$ is not
- Formally define $o(g(n))$ as the set

  $o(g(n)) = \{ f(n) :$ for any positive constant $c > 0$ there exists $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$

- For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$
- The main difference between O-notation and o-notation is that in
  - $f(n) = O(g(n))$, the bound $0 \leq f(n) \leq cg(n)$ holds for some constant $c > 0$,
  - $f(n) = o(g(n))$, the bound $0 \leq f(n) < cg(n)$ holds for all constants $c > 0$
- Intuitively, in o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; i.e.,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

ω-notation

- We use ω-notation to denote a lower bound that is not asymptotically tight
- One way to define it is by $f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$
- Formally we, however, we $\omega(g(n))$ as the set

  $\omega(g(n)) = \{ f(n) :$ for any positive constant $c > 0$ there exists $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0 \}$
- E.g., $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$

Comparing functions

Transitivity:

- $f(n) = \Theta(g(n)), g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
- $f(n) = O(g(n)), g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- $f(n) = \Omega(g(n)), g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$
- $f(n) = o(g(n)), g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$
- $f(n) = \omega(g(n)), g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$

Reflexivity:

- $f(n) = \Theta(f(n))$
- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$

Symmetry:

- $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$

Transpose symmetry:

- $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
- $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

We can draw an analogy between the asymptotic comparison of functions and the comparison of real numbers

- $f(n) = O(g(n))$ is like $a \leq b$
- $f(n) = \Omega(g(n))$ is like $a \geq b$
- $f(n) = \Theta(g(n))$ is like $a = b$
- $f(n) = o(g(n))$ is like $a < b$
- $f(n) = \omega(g(n))$ is like $a > b$

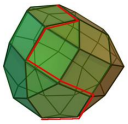
3.2 Standard notations and common functions

- For all real constants a and b s.t. $a > 1$,

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0$$

- from which we can conclude that $n^b = o(a^n)$
- Thus, any exponential function with a base strictly greater than 1 grows faster than any polynomial function

Simplex algorithm



- Dantzig's simplex algorithm is the classical method for solving linear programs
- The simplex method is remarkably efficient in practice and was a great improvement over earlier methods
- It has exponential worst-case complexity, which has led to the development of other measures of complexity
- The simplex has polynomial-time average-case complexity under various probability distributions

4 Divide-and-Conquer

- Three methods for obtaining asymptotic Θ or O bounds on the solution:
  - In ***substitution method***, we guess a bound and then use induction to prove it correct
  - ***recursion-tree method*** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence

- The ***master method*** provides bounds for recurrences of the form

$$T(n) = aT(n/b) + f(n),$$

where $a \geq 1$, $b > 1$, and $f(n)$ is a given function

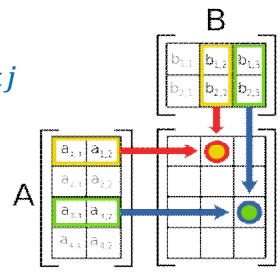
- This characterizes an algorithm that creates a subproblems, each of which is $1/b$ the size of the original problem, and in which the divide and combine steps together take $f(n)$ time

4.2 Strassen's algorithm for matrix multiplication

- Let $A = (a_{ij})$ and $B = (b_{ij})$ be square $n \times n$ matrices; in the product $C = A \cdot B$, we define the entry c_{ij}, for $i, j = 1, 2, \ldots, n$, by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

- We must compute n^2 matrix entries, and each is the sum of n values



SQUARE-MATRIX-MULTIPLY(A, B)

1. $n \leftarrow A.rows$
2. let C be a new $n \times n$ matrix
3. **for** $i \leftarrow 1$ **to** n
4. **for** $j \leftarrow 1$ **to** n
5. $c_{ij} \leftarrow 0$
6. **for** $k \leftarrow 1$ **to** n
7. $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$
8. **return** C

- Each of the nested **for** loops runs exactly n iterations, execution of line 7 takes constant time, the procedure takes $\Theta(n^3)$ time
- We have a way to multiply matrices in $o(n^3)$ time
- Strassen's remarkable recursive algorithm for multiplying $n \times n$ matrices runs in $\Theta(n^{\lg 7})$ time
- $\lg 7$ lies between 2.80 and 2.81, Strassen's algorithm runs in $O(n^{2.81})$

A simple divide-and-conquer algorithm

- Assume that n is an exact power of 2 in each of the $n \times n$ matrices
- Suppose that we partition each of A, B, and C into four $n/2 \times n/2$ matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

so that we rewrite $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

- This corresponds to the four equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

- Each of these four equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products
- Based on them we can create a straightforward, recursive, divide-and-conquer algorithm

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

1. $n \leftarrow A.rows$
2. let C be a new $n \times n$ matrix
3. **if** $n = 1$
4. $\quad c_{11} \leftarrow a_{11} \cdot b_{11}$
5. **else** partition A, B, and C
6. $\quad C_{11} \leftarrow$ RECURSE(A_{11}, B_{11}) + RECURSE(A_{12}, B_{21})
7. $\quad C_{12} \leftarrow$ RECURSE(A_{11}, B_{12}) + RECURSE(A_{12}, B_{22})
8. $\quad C_{21} \leftarrow$ RECURSE(A_{21}, B_{11}) + RECURSE(A_{22}, B_{21})
9. $\quad C_{22} \leftarrow$ RECURSE(A_{21}, B_{12}) + RECURSE(A_{22}, B_{22})
10. **return** C

- When $n = 1$, we perform just the one constant-time scalar multiplication in line 4
- When $n > 1$, partitioning the matrices in line 5 takes $\Theta(1)$ time, using index calculations
- Lines 6–9 recurse eight times, the time taken by all recursive calls is $8T(n/2)$
- We also must account for the four matrix additions in lines 6–9
- Each matrix contains $n^2/4$ entries, and so each of the four additions takes $\Theta(n^2)$ time

- The number of matrix additions is a constant, the total time spent adding matrices in lines 6–9 is $\Theta(n^2)$
- Total time for the recursive case is the sum of
  - the partitioning time,
  - the time for all the recursive calls, and
  - the time to add the matrices resulting from the recursive calls:

$$T(n) = \Theta(1) + 8T(n/2) + \Theta(n^2)$$
$$= 8T(n/2) + \Theta(n^2)$$

- The recurrence for the running time of SQUARE-MATRIX-MULTIPLY-RECURSIVE is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T(n/2) & \text{if } n > 1 \end{cases}$$

- We shall see that this recurrence has the solution $T(n) = \Theta(n^3)$
- The approach is no faster than the straightforward SQUARE-MATRIX-MULTIPLY procedure

- In accounting for the eight recursive calls we cannot just subsume the constant factor of 8
- We must say that together they take $8T(n/2)$ time, rather than just $T(n/2)$
- If we were to ignore the factor of 8, the recursion tree would just be linear, not "bushy," and each level would contribute only one term to the sum
- Although asymptotic notation subsumes constant multiplicative factors, recursive notation such as $T(n/2)$ does not

Strassen's method

- Makes the recursion tree slightly less bushy
  - Instead of performing eight recursive multiplications, it performs only seven
  - The cost is several new additions of $n/2 \times n/2$ matrices, but only a constant number of them
- Constant number of additions is subsumed by Θ-notation when we set up the recurrence to characterize the running time

1. Divide matrices A, B, and C into $n/2 \times n/2$ submatrices in $\Theta(1)$ time by index calculation
2. Create matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1

   We can create all 10 matrices in $\Theta(n^2)$ time
3. Using the matrices created, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$

   Each matrix P_i is $n/2 \times n/2$
4. Compute the submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ by adding and subtracting various combinations of the P_i matrices

   We can compute all submatrices in $\Theta(n^2)$ time

- We have enough information to set up a recurrence for the running time of the method
- Let's assume that once the matrix size n gets down to 1, we perform a simple multiplication
- When $n > 1$, steps 1, 2, and 4 take a total of $\Theta(n^2)$ time, and step 3 requires us to perform seven multiplications of $n/2 \times n/2$ matrices.
- Hence, we obtain the following recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T(n/2) & \text{if } n > 1 \end{cases}$$

- By the *master method* this recurrence has the solution $T(n) = \Theta(n^{\lg 7})$

Matrices of Step 2

- $S_1 = B_{12} - B_{22}$
- $S_2 = A_{11} + A_{12}$
- $S_3 = A_{21} + A_{22}$
- $S_4 = B_{21} - B_{11}$
- $S_5 = A_{11} + A_{22}$
- $S_6 = B_{11} + B_{22}$
- $S_7 = A_{12} - A_{22}$
- $S_8 = B_{21} + B_{22}$
- $S_9 = A_{11} - A_{21}$
- $S_{10} = B_{11} + B_{12}$

- we must add or subtract $n/2 \times n/2$ matrices 10 times, so this step does take $\Theta(n^2)$ time

- In **step 3**, we recursively multiply $n/2 \times n/2$ matrices seven times to compute the following $n/2 \times n/2$ matrices:
  - $P_1 = A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22}$
  - $P_2 = S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22}$
  - $P_3 = S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11}$
  - $P_4 = A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11}$
  - $P_5 = S_5 \cdot S_6 = \cdots$
  - $P_6 = S_7 \cdot S_8 = \cdots$
  - $P_7 = S_9 \cdot S_{10} = \cdots$

- **Step 4** adds and subtracts the P_i matrices created in step 3 to construct the four $n/2 \times n/2$ submatrices of the product C
  - $C_{11} = P_5 + P_4 - P_2 + P_6$
  - $C_{12} = P_1 + P_2$
  - $C_{21} = P_3 + P_4$
  - $C_{22} = P_5 + P_1 - P_3 - P_7$

E.g.,

$$\begin{array}{r} A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\ \hline A_{11} \cdot B_{12} \end{array} \qquad \begin{array}{r} A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\ + A_{12} \cdot B_{22} = C_{12} \end{array}$$

4.3 The substitution method

- We can use the method to establish either upper or lower bounds on a recurrence
- Let us, e.g., determine an upper bound on the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + n$
- We guess the solution $T(n) = O(n \lg n)$
- The substitution method requires us to prove that $T(n) \leq cn \lg n$ for an appropriate choice of the constant $c > 0$

- We start by assuming that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding
$$T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$$
- Substituting into the recurrence yields $T(n) \leq 2\left(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)\right) + n$
$$\leq cn \lg(n/2) + n$$
$$= cn \lg n - cn \lg 2 + n$$
$$= cn \lg n - cn + n$$
$$\leq cn \lg n \qquad \text{(as long as } c \geq 1\text{)}$$

- We now need to show that our solution holds for the boundary conditions
  - Show that we can choose the constant c large enough so that the bound $T(n) \leq cn \lg n$ works for the boundary conditions as well
- Let us assume that $T(1) = 1$ is the sole boundary condition of the recurrence
- For $n = 1$, we get $T(1) \leq c1 \lg 1 = 0$, which is at odds with $T(1) = 1$
- Asymptotic notation requires us only to prove the bound for $n \geq n_0$, where n_0 is a constant that we get to choose

- We keep the condition $T(1) = 1$, but remove it from consideration in the inductive proof
- First observe that for $n > 3$, the recurrence does not depend directly on $T(1)$
- Thus, we replace $T(1)$ by $T(2)$ and $T(3)$ as the base of inductive proof, letting $n_0 = 2$
- The recurrence yields $T(2) = 2T(1) + 2 = 4$ and $T(3) = 2T(1) + 3 = 5$
- To complete the proof, we choose c so that $T(2) \leq c2 \lg 2$ and $T(3) \leq c3 \lg 3$
- Any choice of $c \geq 2$ suffices for the base cases of $n = 2$ and $n = 3$ to hold

Making a good guess

- There is no general way to guess the correct solutions to recurrences
- If a recurrence is similar to one you have seen before, then guessing a similar solution is reasonable
- Another way to make a good guess is to prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainty

Subtleties

- Consider the recurrence
$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$
- We guess $T(n) = O(n)$, and try to show that $T(n) \leq cn$ for an appropriate choice of c
- Substituting our guess, we obtain
$$T(n) \leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1$$
$$= cn + 1,$$
which does not imply $T(n) \leq cn$ for any choice of c

- We could try a larger guess, e.g., $T(n) = O(n^2)$
- The original guess of $T(n) = O(n)$ is correct
  - We must make a stronger inductive hypothesis
- We are off only by the lower-order term 1
- To overcome the difficulty subtract a lower-order term from our previous guess
- Our new guess is $T(n) \leq cn - d$, where $d \geq 0$ is a constant

- We now have
$$T(n) = (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1$$
$$= cn - 2d + 1$$
$$\leq cn - d$$
as long as $d \geq 1$
- As before, we must choose c large enough to handle the boundary conditions
- The idea of subtracting a lower-order term may seem counterintuitive
- Proving an upper bound by induction, it may be more difficult to show a weaker bound

- To prove the weaker bound, we must use the same weaker bound inductively in the proof
- There is more than one recursive term, and we get to subtract out the lower-order term once per recursive term
- We subtracted out the constant d twice, once for $T(\lfloor n/2 \rfloor)$ and once for $T(\lceil n/2 \rceil)$
- We ended up with the inequality $T(n) \leq cn - 2d + 1$, and it was easy to find values of d to make $cn - 2d + 1$ be less than or equal to $cn - d$

Avoiding pitfalls

- It is easy to err in asymptotic notation
- E.g., in an earlier recurrence we can falsely guess $T(n) \leq cn$ and argue $T(n) \leq 2(c\lfloor n/2 \rfloor) + n \leq cn + n = O(n)$;
- wrong!!
- We have not proved the exact form of the inductive hypothesis, that is, that $T(n) \leq cn$
- We therefore will explicitly prove that $T(n) \leq cn$ when we want to show that $T(n) = O(n)$

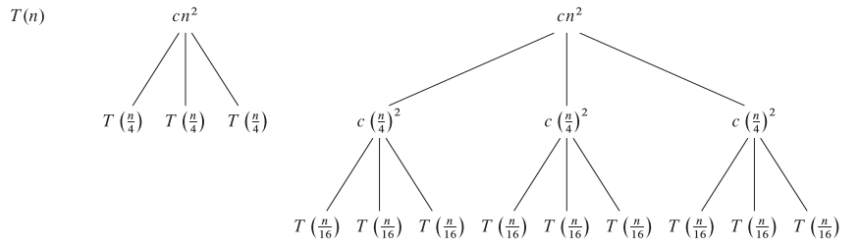
Changing variables

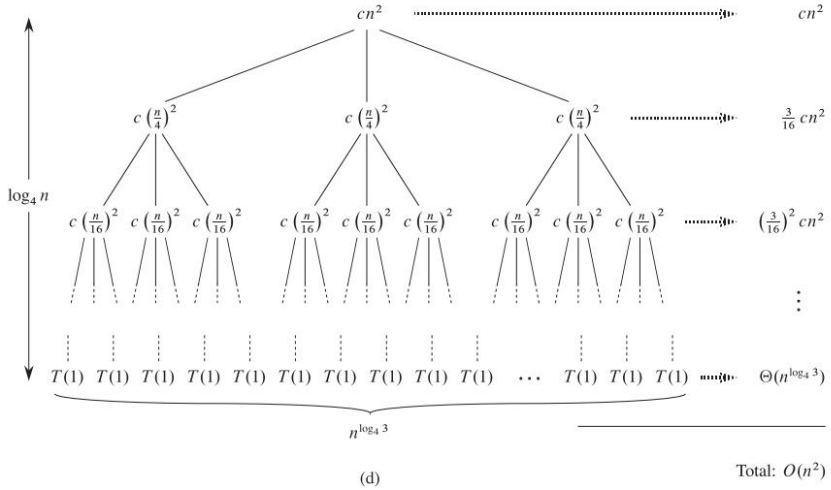
- A little algebraic manipulation can make an unknown recurrence more familiar
- The recurrence $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$ looks difficult
- We can simplify it with a change of variables
- For convenience, we shall not worry about rounding off values, such as $\sqrt{n}$, to be integers

- Renaming $m = \lg n$ yields
$$T(2^m) = 2T\left(2^{m/2}\right) + m$$
- We can now rename $S(m) = T(2^m)$ to produce the new recurrence
$$S(m) = 2S(m/2) + m$$
which looks more familiar
- Indeed, this new recurrence has the solution $S(m) = O(m \lg m)$
- Changing back from $S(m)$ to $T(n)$, we obtain
$$T(n) = T(2^m) = S(m) = O(m \lg m)$$
$$= O(\lg n \lg \lg n)$$

4.4 The recursion-tree method

- Let us see how a recursion tree would provide a good guess for the recurrence
$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$
- Start by finding an upper bound
- Floors and ceilings usually do not matter when solving recurrences
- Create a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$ having written out the implied constant coefficient $c > 0$


$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$
cn^2
$c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$


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

- Subproblem sizes decrease by a factor of 4 each time we go down one level =>
  - we eventually must reach a boundary condition
- Subproblem size for a node at depth i is $n/4^i$
  - It hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$
- Thus, the tree has $\log_4 n + 1$ levels (at depths $0, 1, 2, \ldots, \log_4 n$)

- Each level has 3 × the nodes of level above
  - the number of nodes at depth i is 3^i
- Subproblem sizes reduce by a factor of 4 for each level we go down, each node at depth i, $i = 0,1,2, \ldots, \log_4 n - 1$, has cost of $c(n/4^i)^2$
- Multiplying, we see that the total cost over all nodes at depth i is $3^i c(n/4^i)^2 = (3/16)^i cn^2$
- The bottom level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each contributing cost $T(1)$, for a total cost of $n^{\log_4 3} T(1)$, which is $\Theta(n^{\log_4 3})$, since $T(1)$ is a constant

- Add up the costs over all levels to determine the cost for the entire tree:

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

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

By the value of a ***geometric*** (or exponential) ***series***

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

- We can use an infinite decreasing geometric series as an upper bound

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

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

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

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

- We have derived a guess of $T(n) = O(n^2)$ for our recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$
- Root of the tree contributes cn^2 to the total cost – a constant fraction of the total cost
  - The cost of the root dominates the total cost
- If $O(n^2)$ is indeed an upper bound for the recurrence, then it must be a tight bound
- First recursive call contributes cost of $\Theta(n^2)$, $\Omega(n^2)$ must be a lower bound for the recurrence

- Use the substitution method to verify that the guess was correct, i.e., $T(n) = O(n^2)$ is an upper bound for the recurrence
- We want to show that $T(n) \leq dn^2$ for some constant $d > 0$

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

as long as $d \geq (16 / 13)c$