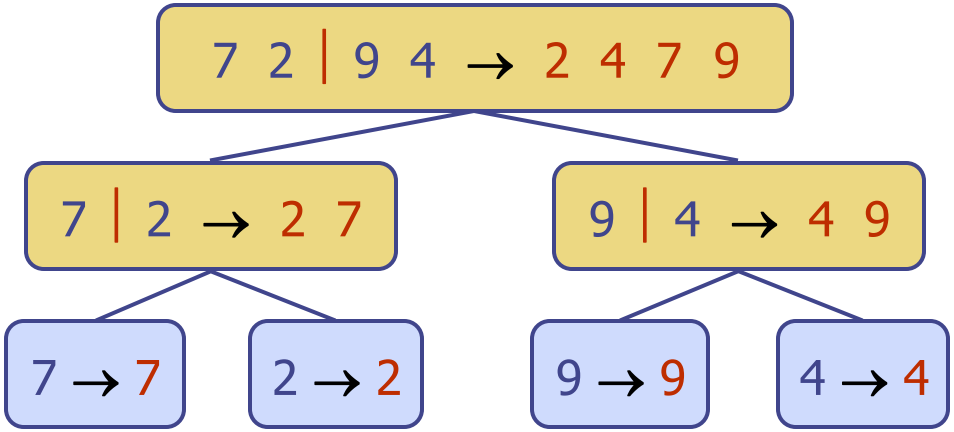
Divide-and-Conquer

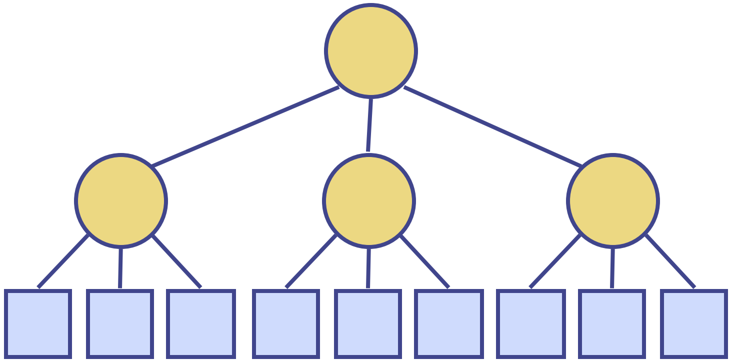


Outline and Reading

- Divide-and-conquer paradigm (§5.2)
- Review Merge-sort (§4.1.1)
- Recurrence Equations (§5.2.1)
  - Iterative substitution
  - Recursion trees
  - Guess-and-test
  - The master method
- Integer Multiplication (§5.2.2)

Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two or more disjoint subsets S_1, S_2, …
  - Recur: solve the subproblems recursively
  - Conquer: combine the solutions for S_1, S_2, …, into a solution for S
- The base case for the recursion are subproblems of constant size
- Analysis can be done using **recurrence equations**



Merge-Sort Review

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*(S, C)
    Input sequence S with n elements, comparator C
    Output sequence S sorted according to C
    if $S.size() > 1$
        $(S_1, S_2) \leftarrow partition(S, n/2)$
        $mergeSort(S_1, C)$
        $mergeSort(S_2, C)$
        $S \leftarrow merge(S_1, S_2)$

Recurrence Equation Analysis

- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes at most bn steps, for some constant b.
- Likewise, the basis case ($n < 2$) will take at b most steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- We can therefore analyze the running time of merge-sort by finding a **closed form solution** to the above equation.
  - That is, a solution that has $T(n)$ only on the left-hand side.

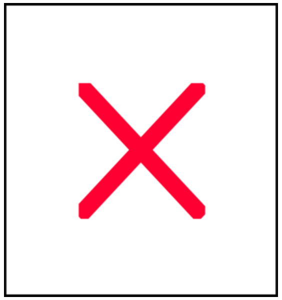
Iterative Substitution

- In the iterative substitution, or "plug-and-chug," technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$
\begin{aligned}
T(n) &= 2T(n/2) + bn \\
&= 2(2T(n/2^2)) + b(n/2)) + bn \\
&= 2^2 T(n/2^2) + 2bn \\
&= 2^3 T(n/2^3) + 3bn \\
&= 2^4 T(n/2^4) + 4bn \\
&= \ldots \\
&= 2^i T(n/2^i) + ibn
\end{aligned}
$$

- Note that base, T(1)=b, case occurs when 2^i=n. That is, i = log n.
- So, $$T(n) = bn + bn\log n$$
- Thus, T(n) is O(n log n).

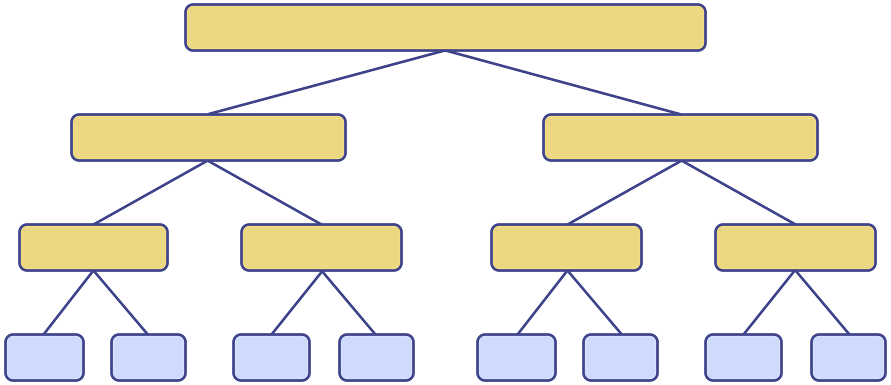
The Recursion Tree



- Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
…	…	…	…



Total time $= bn + bn \log n$

(last level plus all previous levels)

Guess-and-Test Method

$$T(n)=\begin{cases} b & \text{if } n<2 \\ 2T(n/2)+bn & \text{if } n\geq 2 \end{cases}$$

- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

$$T(n)=\begin{cases} b & \text{if } n<2 \\ 2T(n/2)+bn\log n & \text{if } n\geq 2 \end{cases}$$

- Guess: T(n) < cn log n.

$$\begin{aligned} T(n) &= 2T(n/2)+bn\log n \\ &= 2(c(n/2)\log(n/2))+bn\log n \\ &= cn(\log n-\log 2)+bn\log n \\ &= cn\log n-cn+bn\log n \end{aligned}$$

- Wrong: we cannot make this last line be less than cn log n

Guess-and-Test Method, Part 2

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \ge 2 \end{cases}$$

- Recall the recurrence equation:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \ge 2 \end{cases}$$

- Guess #2: T(n) < cn log² n.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &= 2(c(n/2)\log^2(n/2)) + bn\log n \\ &= cn(\log n - \log 2)^2 + bn\log n \\ &= cn\log^2 n - 2cn\log n + cn + bn\log n \\ &\le cn\log^2 n \end{aligned}$$

  - if c > b.
- So, T(n) is O(n log² n).
- In general, to use this method, you need to have a good guess and you need to be good at induction proofs.

Master Method

- Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

$$1. \text{ if } f(n) \text{ is } O(n^{\log_b a - \varepsilon}), \text{ then } T(n) \text{ is } \Theta(n^{\log_b a})$$

$$2. \text{ if } f(n) \text{ is } \Theta(n^{\log_b a} \log^k n), \text{ then } T(n) \text{ is } \Theta(n^{\log_b a} \log^{k+1} n)$$

$$3. \text{ if } f(n) \text{ is } \Omega(n^{\log_b a + \varepsilon}), \text{ then } T(n) \text{ is } \Theta(f(n)),$$

$$\text{provided } af(n/b) \leq \delta f(n) \text{ for some } \delta < 1.$$

Master Method, Example 1

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,

     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

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

Solution: $\log_b a = 2$, so case 1 says $T(n)$ is $\Theta(n^2)$.

Master Method, Example 2

- The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

$$1.\ \text{if } f(n) \text{ is } O(n^{\log_b a - \varepsilon}), \text{then } T(n) \text{ is } \Theta(n^{\log_b a})$$

$$2.\ \text{if } f(n) \text{ is } \Theta(n^{\log_b a} \log^k n), \text{then } T(n) \text{ is } \Theta(n^{\log_b a} \log^{k+1} n)$$

$$3.\ \text{if } f(n) \text{ is } \Omega(n^{\log_b a + \varepsilon}), \text{then } T(n) \text{ is } \Theta(f(n)),$$

$$\text{provided } af(n/b) \leq \delta f(n) \text{ for some } \delta < 1.$$

- Example:

$$T(n) = 2T(n/2) + n\log n$$

Solution: $\log_b a = 1$, so case 2 says T(n) is $\Theta(n \log^2 n)$.

Master Method, Example 3

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:

$$1.\ \text{if } f(n) \text{ is } O(n^{\log_b a-\varepsilon}), \text{ then } T(n) \text{ is } \Theta(n^{\log_b a})$$

$$2.\ \text{if } f(n) \text{ is } \Theta(n^{\log_b a} \log^k n), \text{ then } T(n) \text{ is } \Theta(n^{\log_b a} \log^{k+1} n)$$

$$3.\ \text{if } f(n) \text{ is } \Omega(n^{\log_b a+\varepsilon}), \text{ then } T(n) \text{ is } \Theta(f(n)),$$

$$\text{provided } af(n/b) \leq \delta f(n) \text{ for some } \delta < 1.$$

- Example:

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

Solution: $\log_b a=0$, so case 3 says T(n) is Θ(n log n).

Master Method, Example 4

- The form:
$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,

     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example:
$$T(n) = 8T(n/2) + n^2$$

Solution: $\log_b a = 3$, so case 1 says $T(n)$ is $\Theta(n^3)$.

Master Method, Example 5

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$

- The Master Theorem:

  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example:

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

Solution: $\log_b a = 2$, so case 3 says $T(n)$ is $\Theta(n^3)$.

Master Method, Example 6

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:

$$1.\ \text{if } f(n) \text{ is } O(n^{\log_b a - \varepsilon}), \text{ then } T(n) \text{ is } \Theta(n^{\log_b a})$$

$$2.\ \text{if } f(n) \text{ is } \Theta(n^{\log_b a} \log^k n), \text{ then } T(n) \text{ is } \Theta(n^{\log_b a} \log^{k+1} n)$$

$$3.\ \text{if } f(n) \text{ is } \Omega(n^{\log_b a + \varepsilon}), \text{ then } T(n) \text{ is } \Theta(f(n)),$$

$$\text{provided } af(n/b) \leq \delta f(n) \text{ for some } \delta < 1.$$

- Example:

$$T(n) = T(n/2) + 1 \quad \text{(binary search)}$$

Solution: $\log_b a = 0$, so case 2 says T(n) is $\Theta(\log n)$.

Master Method, Example 7

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:

$$1.\ \text{if } f(n) \text{ is } O(n^{\log_b a - \varepsilon}), \text{then } T(n) \text{ is } \Theta(n^{\log_b a})$$

$$2.\ \text{if } f(n) \text{ is } \Theta(n^{\log_b a} \log^k n), \text{then } T(n) \text{ is } \Theta(n^{\log_b a} \log^{k+1} n)$$

$$3.\ \text{if } f(n) \text{ is } \Omega(n^{\log_b a + \varepsilon}), \text{then } T(n) \text{ is } \Theta(f(n)),$$

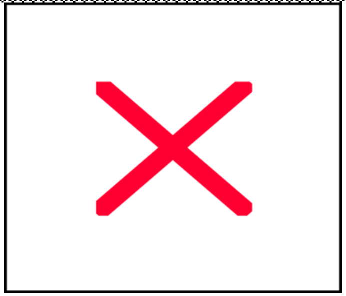
$$\text{provided } af(n/b) \leq \delta f(n) \text{ for some } \delta < 1.$$

- Example:

$$T(n) = 2T(n/2) + \log n \quad \text{(heap construction)}$$

Solution: $\log_b a = 1$, so case 1 says T(n) is Θ(n).

Iterative "Proof" of the Master Theorem



- Using iterative substitution, let us see if we can find a pattern:

$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a(aT(n/b^2)) + f(n/b)) + f(n) \\
&= a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2 f(n/b^2) + af(n/b) + f(n) \\
&\vdots \\
&= a^kT(n/b^k) + \sum_{i=0}^{k-1} a^i f(n/b^i) \\
&\vdots
\end{aligned}
$$

Iterative "Proof" of the Master Theorem

M

$$= a^{\log_b n} T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i)$$

$$= n^{\log_b a} T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i)$$

- We then distinguish the three cases as
  - The first term is dominant
  - Each part of the summation is equally dominant
  - The summation is a geometric series

Integer Multiplication

- Algorithm: Multiply two n-bit integers I and J.
  - Divide step: Split I and J into high-order and low-order bits

$$I = I_h 2^{n/2} + I_l$$

$$J = J_h 2^{n/2} + J_l$$

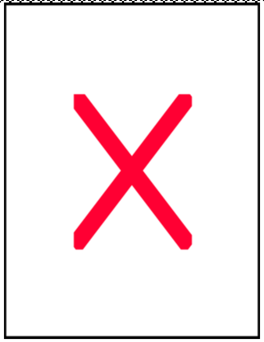
  - We can then define I*J by multiplying the parts and adding:

$$I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l)$$
$$= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$

  - So, $T(n) = 4T(n/2) + n$, which implies $T(n)$ is $\Theta(n^2)$.
  - But that is no better than the algorithm we learned in grade school.

where did this come from?

An Improved Integer Multiplication Algorithm



- Algorithm: Multiply two n-bit integers I and J.
  - Divide step: Split I and J into high-order and low-order bits

$$I = I_h 2^{n/2} + I_l$$

$$J = J_h 2^{n/2} + J_l$$

  - Observe that there is a different way to multiply parts:

$$\begin{aligned} I * J &= I_h J_h 2^n + [(I_h - I_l)(J_l - J_h) + I_h J_h + I_l J_l]2^{n/2} + I_l J_l \\ &= I_h J_h 2^n + [(I_h J_l - I_l J_l - I_h J_h + I_l J_h) + I_h J_h + I_l J_l]2^{n/2} + I_l J_l \\ &= I_h J_h 2^n + (I_h J_l + I_l J_h)2^{n/2} + I_l J_l \end{aligned}$$

requires only 3 mults!

  - So, T(n) = 3T(n/2) + n, which implies T(n) is $O(n^{\log_2 3})$, by the Master Theorem.
  - Thus, T(n) is $O(n^{1.585})$.

Matrix Multiplication

- Given $n \times n$ matrices X and Y, wish to compute the product $Z=XY$.
- Formula for doing this is

$$Z_{ij} = \sum_{k=0}^{n-1} X_{ik} Y_{kj}$$

- This runs in $O(n^3)$ time
  - In fact, multiplying an $n \times m$ by an $m \times q$ takes nmq operations

Matrix Multiplication

$$\begin{bmatrix} I & J \\ K & L \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$I = AE + BG$$
$$J = AF + BH$$
$$K = CE + DG$$
$$L = CF + DH$$

Matrix Multiplication

- Using the decomposition on previous slide, we can computer *Z* using 8 recursively computed (n/2)x(n/2) matrices plus four additions that can be done in $O(n^2)$ time
- Thus $T(n) = 8T(n/2) + bn^2$
- Still gives T(n) is $\Theta(n^3)$

Strassen's Algorithm

- If we define the matrices S_1 through S_7 as follows

$$S_1 = A(F - H)$$
$$S_2 = (A + B)H$$
$$S_3 = (C + D)E$$
$$S_4 = D(G - E)$$
$$S_5 = (A + D)(E + H)$$
$$S_6 = (B - D)(G + H)$$
$$S_7 = (A - C)(E + F)$$

Strassen's Algorithm

- Then we get the following:

$$I = S_5 + S_6 + S_4 - S_2$$
$$J = S_1 + S_2$$
$$K = S_3 + S_4$$
$$L = S_1 - S_7 - S_3 + S_5$$

- So now we can compute $Z=XY$ using only seven recursive multiplications

Strassen's Algorithm

- This gives the relation $T(n)=7T(n/2)+bn^2$ for some $b>0$.
- By the Master Theorem, we can thus multiply two $n \times n$ matrices in $\Theta(n^{\log 7})$ time, which is approximately $\Theta(n^{2.808})$.
  - May not seem like much, but if you're multiplying two *100 x 100* matrices:
    - n^3 is 1,000,000
    - $n^{2.808}$ is 413,048
- With added complexity, there are algorithms to multiply matrices in as little as $\Theta(n^{2.376})$ time
  - Reduces figures above to 56,494