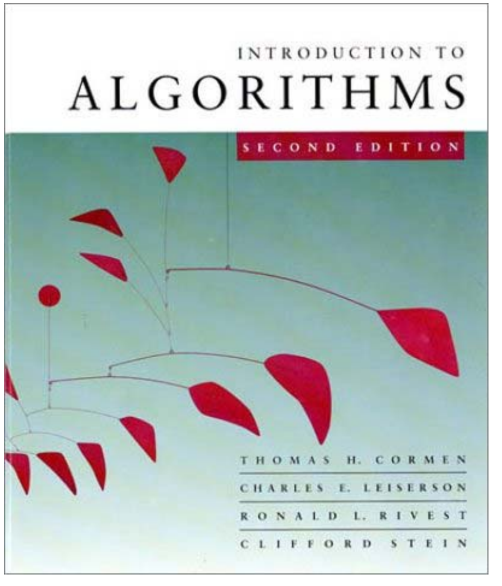
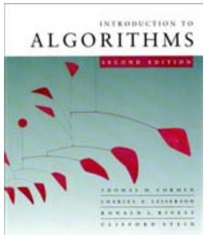
Introduction to Algorithms

6.046J/18.401J



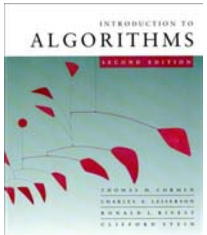
Lecture 4

Prof. Piotr Indyk



Today

- Randomized algorithms: algorithms that flip coins
  - Matrix product checker: is AB=C ?
  - Quicksort:
    - Example of divide and conquer
    - Fast and practical sorting algorithm
    - Other applications on Wednesday

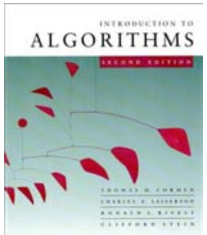


Randomized Algorithms

- Algorithms that make random decisions
- That is:
  - Can generate a random number x from some range {1…R}
  - Make decisions based on the value of x
- Why would it make sense ?

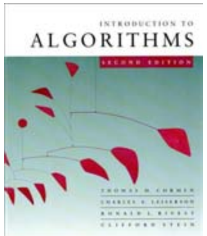
Two cups, one coin

- If you always choose a fixed cup, the adversary will put the coin in the other one, so the expected payoff = \$0
- If you choose a random cup, the expected payoff = \$0.5



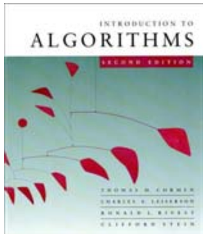
Randomized Algorithms

- Two basic types:
  - Typically fast (but sometimes slow): Las Vegas
  - Typically correct (but sometimes output garbage): Monte Carlo
- The probabilities are defined by the random numbers of the algorithm! (not by random choices of the problem instance)



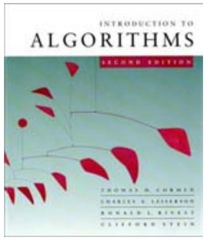
Matrix Product

- Compute $C=A\times B$
  - Simple algorithm: $O(n^3)$ time
  - Multiply two 2×2 matrices using 7 mult.
    $\rightarrow O(n^{2.81\ldots})$ time [Strassen'69]
  - Multiply two 70×70 matrices using 143640 multiplications $\rightarrow O(n^{2.795\ldots})$ time [Pan'78]
  - …
  - $O(n^{2.376\ldots})$ [Coppersmith-Winograd]



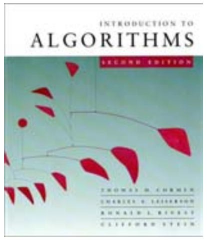
Matrix Product Checker

- Given: $n \times n$ matrices A, B, C
- Goal: is $A \times B = C$?
- We will see an $O(n^2)$ algorithm that:
  - If answer=YES, then $\Pr[\text{output}=\text{YES}]=1$
  - If answer=NO, then $\Pr[\text{output}=\text{YES}] \leq ½$



The algorithm

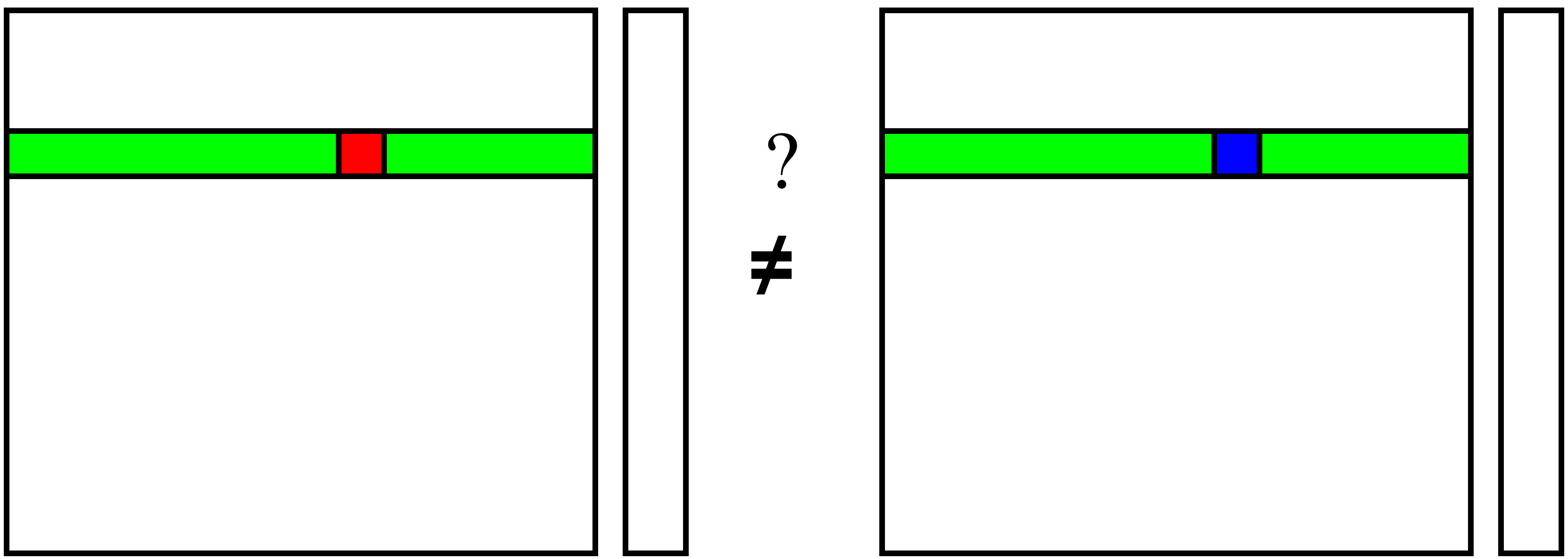
- Algorithm:
  - Choose a random binary vector $x[1\ldots n]$, such that $Pr[x_i=1]=½$, $i=1\ldots n$
  - Check if $ABx=Cx$
- Does it run in $O(n^2)$ time ?
  - YES, because $ABx = A(Bx)$

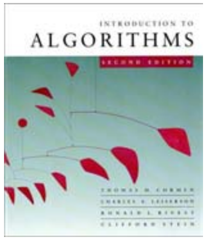


Correctness

- Let $D=AB$, need to check if $D=C$
- What if $D=C$?
  - Then $Dx=Cx$,so the output is YES
- What if $D\neq C$?
  - Presumably there exists x such that $Dx\neq Cx$
  - We need to show there are many such x

D≠C



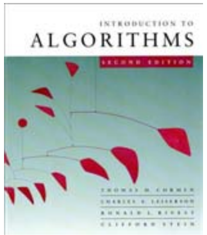


Vector product

- Consider vectors $d \neq c$ (say, $d_i \neq c_i$)
- Choose a random binary x
- We have $dx = cx$ iff $(d-c)x = 0$
- $\Pr[(d-c)x=0] =$?

(d-c):	d_1-c_1	d_2-c_2	…	d_i-c_i	…	d_n-c_n	
x:	x_1	x_2	…	x_i	…	x_n	

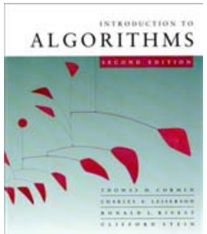
$$= \Sigma_{j \neq i}(d_j-c_j)x_j + (d_i-c_i)x_i$$



Analysis, ctd.

- If $x_i=0$, then $(c-d)x=S_1$
- If $x_i=1$, then $(c-d)x=S_2\neq S_1$
- So, ≥ 1 of the choices gives $(c-d)x\neq 0$

$\rightarrow$ $\Pr[cx=dx] \leq ½$



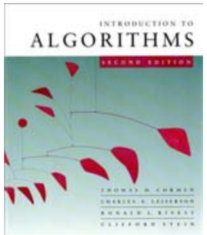
Matrix Product Checker

- Is A×B=C ?
- We have an algorithm that:
  - If answer=YES, then Pr[output=YES]=1
  - If answer=NO, then Pr[output=YES] ≤ ½
- What if we want to reduce ½ to ¼ ?
  - Run the algorithm twice, using independent random numbers
  - Output YES only if both runs say YES
- Analysis:
  - If answer=YES, then Pr[output_1=YES, output_2=YES]=1
  - If answer=NO, then

    Pr[output=YES] = Pr[output_1=YES, output_2=YES]

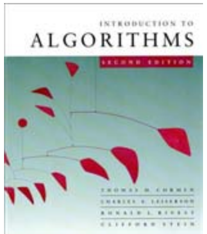
    = Pr[output_1=YES]*Pr[output_2=YES]

    ≤ ¼



Quicksort

- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.
- Sorts "in place" (like insertion sort, but not like merge sort).
- Very practical (with tuning).
- Can be viewed as a randomized Las Vegas algorithm



Divide and conquer

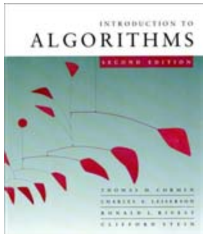
Quicksort an n-element array:

1. ***Divide:*** Partition the array into two subarrays around a ***pivot*** x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.

$\leq x$	x	$\geq x$

2. ***Conquer:*** Recursively sort the two subarrays.
3. ***Combine:*** Trivial.

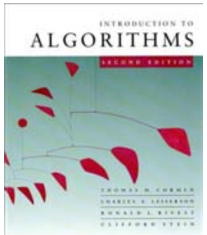
Key: *Linear-time partitioning subroutine.*



Pseudocode for quicksort

QUICKSORT(A, p, r)
    if $p < r$
        then $q \leftarrow$ PARTITION(A, p, r)
            QUICKSORT(A, p, q–1)
            QUICKSORT(A, q+1, r)

Initial call: QUICKSORT(A, 1, n)



Partitioning subroutine

PARTITION(A, p, r) $\triangleleft$ $A[p \,.\,.\, r]$

 $x \leftarrow A[p]$ $\triangleleft$ pivot $= A[p]$

 $i \leftarrow p$

 for $j \leftarrow p + 1$ **to** r

  do if $A[j] \leq x$

   then $i \leftarrow i + 1$

    exchange $A[i] \leftrightarrow A[j]$

 exchange $A[p] \leftrightarrow A[i]$

 return i

Invariant:

x	$\leq x$	$\geq x$	?
p	i	j	r

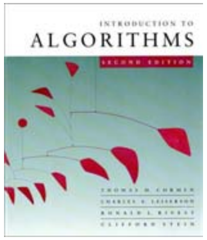
Example of partitioning

6	10	13	5	8	3	2	11
i	j						

Example of partitioning

6	10	13	5	8	3	2	11

i $\longrightarrow j$



Example of partitioning

6	10	13	5	8	3	2	11

i $\longrightarrow j$

Example of partitioning

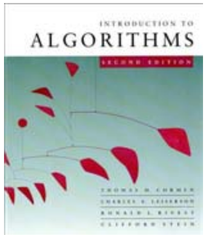
6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11

i j

Example of partitioning

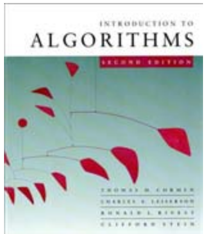
6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
	i			j			



Example of partitioning

6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11
	i				j		



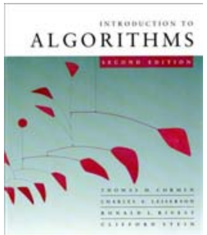
Example of partitioning

6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11

6	5	3	10	8	13	2	11

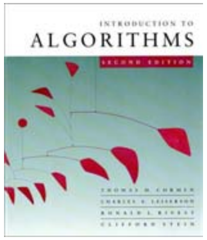
i j



Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11

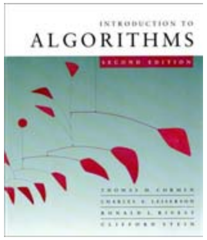
i j



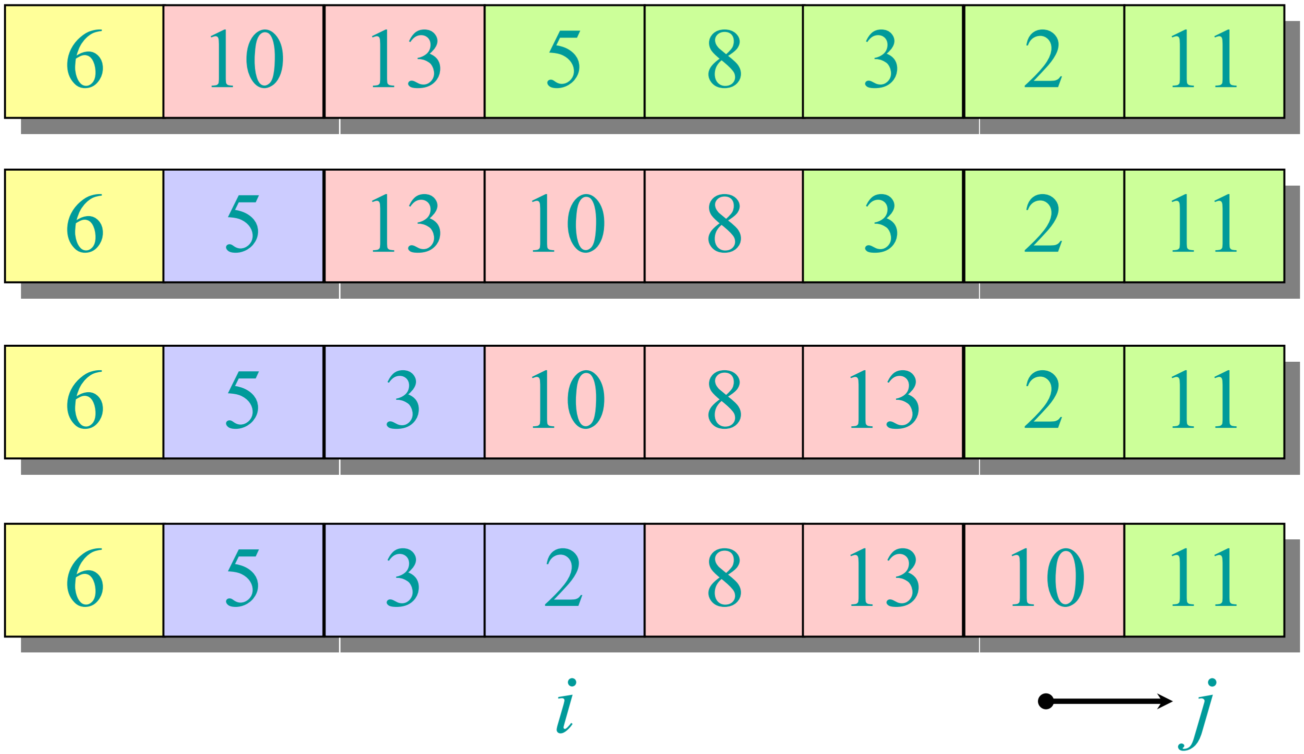
Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11

$\longrightarrow i$ $\qquad\qquad j$



Example of partitioning



Example of partitioning

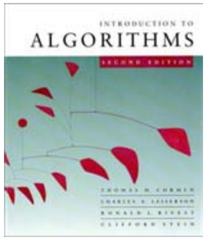
6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11

6	5	3	10	8	13	2	11

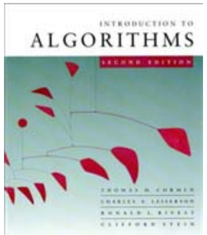
6	5	3	2	8	13	10	11

i $\longrightarrow j$



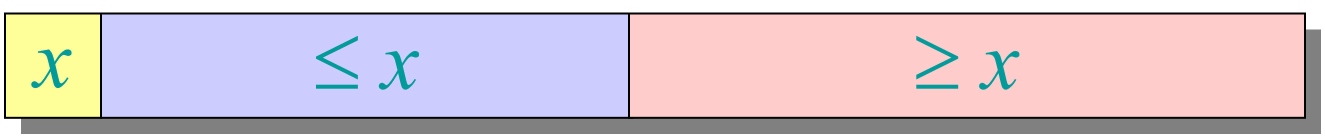
Example of partitioning

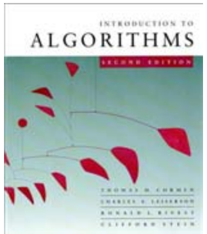
6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
2	5	3	6	8	13	10	11
			i				



Analysis of quicksort

- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- What is the worst case running time of Quicksort ?





Worst-case of quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$
\begin{aligned}
T(n) &= T(0) + T(n-1) + \Theta(n) \\
&= \Theta(1) + T(n-1) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \Theta(n^2) \qquad \textit{\textbf{(arithmetic series)}}
\end{aligned}
$$

Worst-case recursion tree

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

Worst-case recursion tree

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

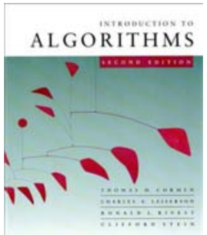
$$T(n)$$

Worst-case recursion tree

$$T(n) = T(0) + T(n–1) + cn$$

$$cn$$

$$T(0) \qquad T(n–1)$$



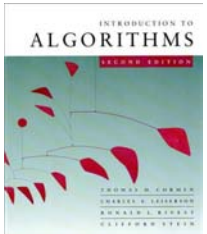
Worst-case recursion tree

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

$$cn$$

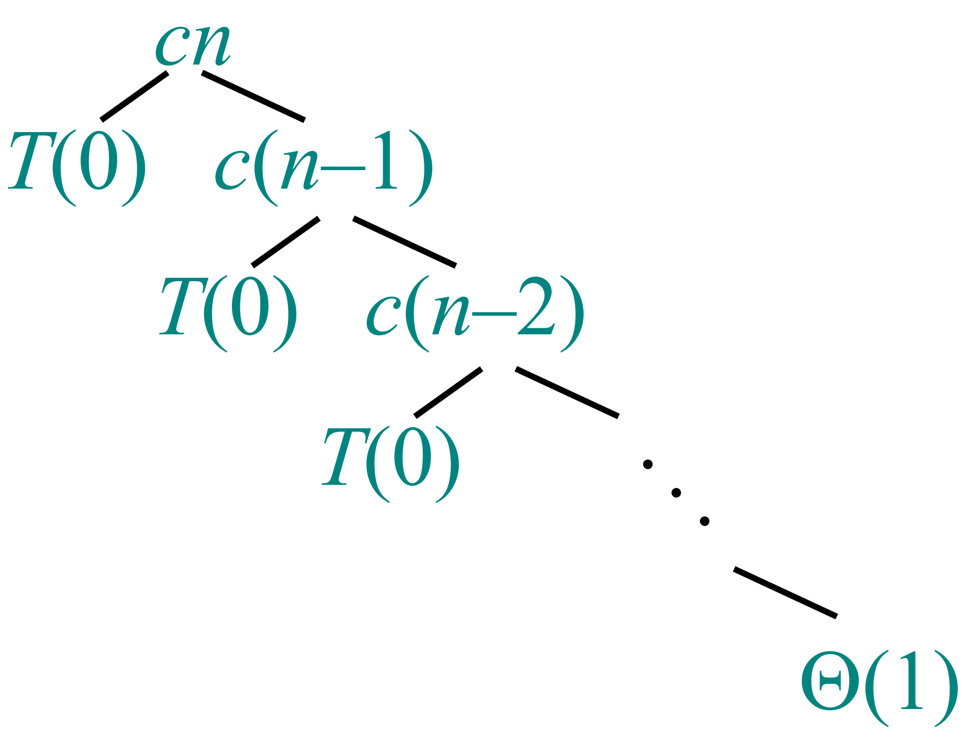
$$T(0) \qquad c(n-1)$$

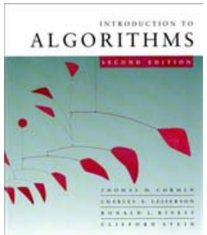
$$T(0) \qquad T(n-2)$$



Worst-case recursion tree

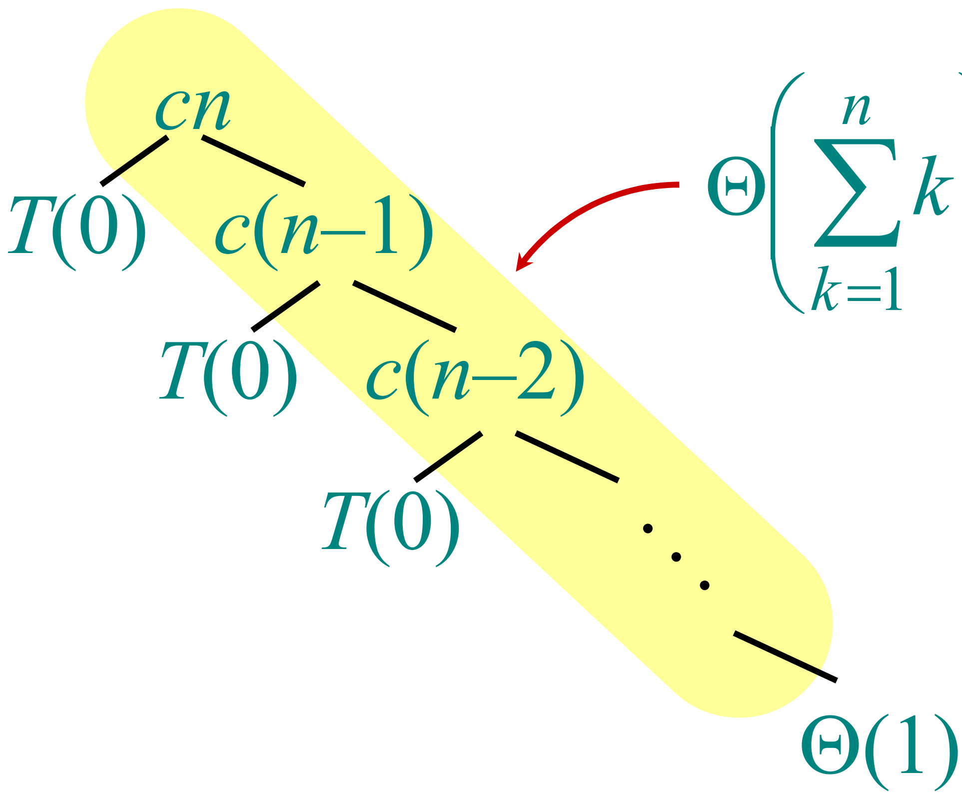
$$T(n) = T(0) + T(n-1) + cn$$



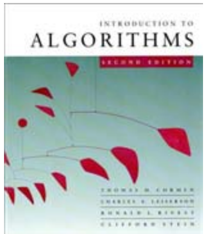


Worst-case recursion tree

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

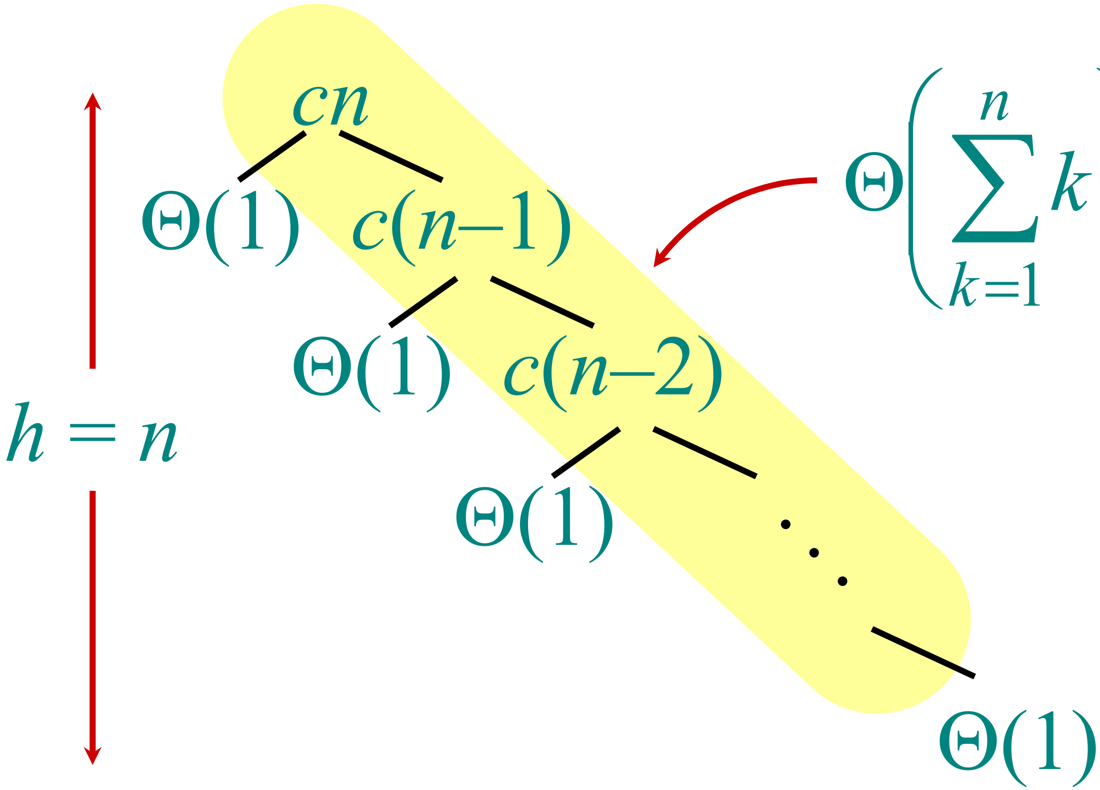


$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$

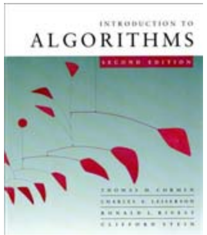


Worst-case recursion tree

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



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$



Nice-case analysis

If we're lucky, PARTITION splits the array evenly:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

(same as merge sort)

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$T(n) = T\left(\tfrac{1}{10}n\right) + T\left(\tfrac{9}{10}n\right) + \Theta(n)$$

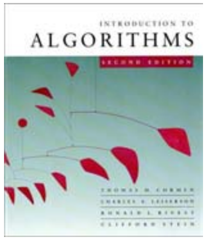
Analysis of nice case

$$T(n)$$

Analysis of nice case

$$cn$$

$$T\left(\tfrac{1}{10}n\right) \qquad\qquad T\left(\tfrac{9}{10}n\right)$$

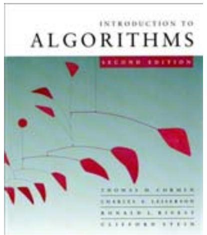


Analysis of nice case

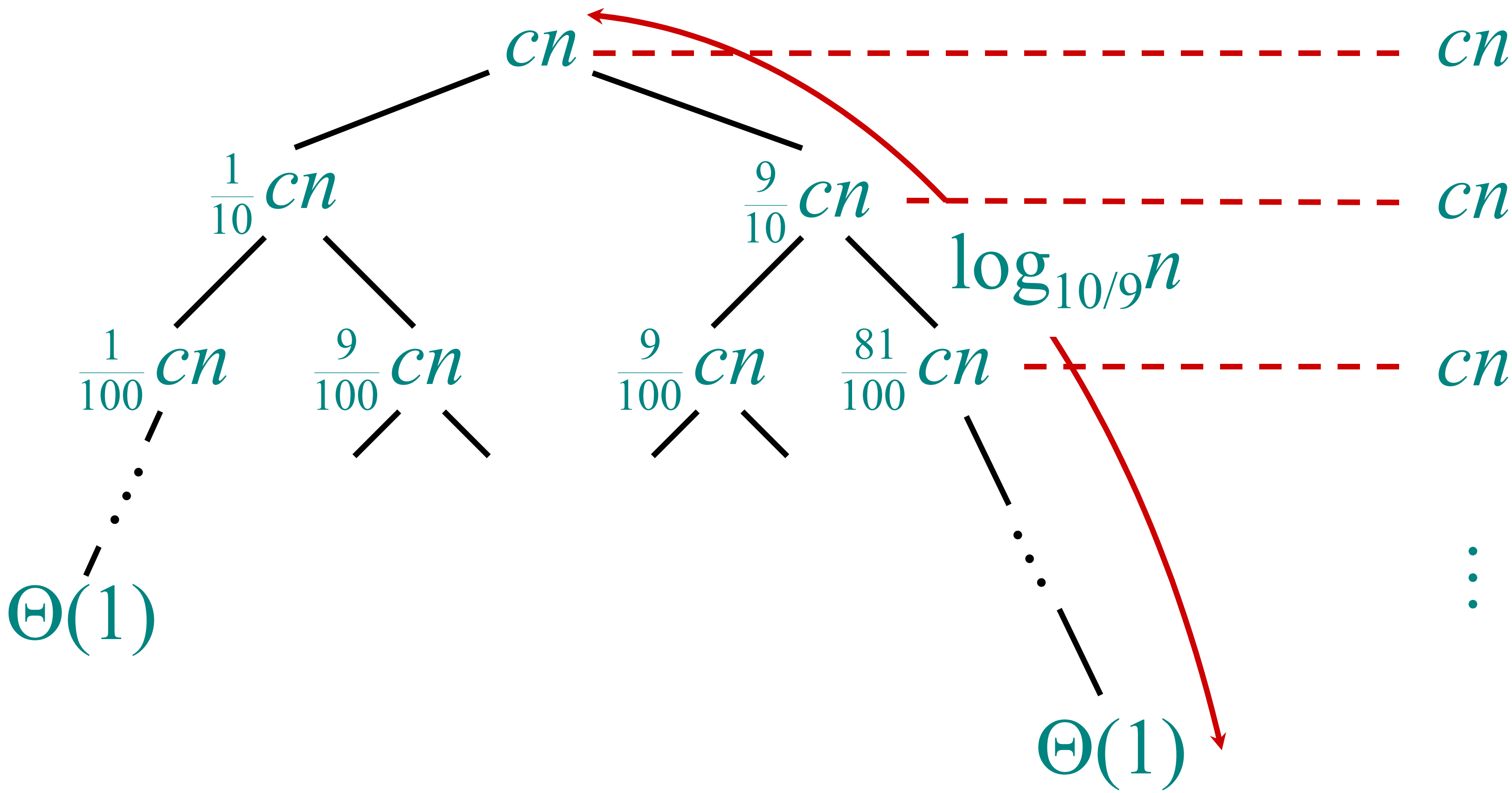
$$cn$$

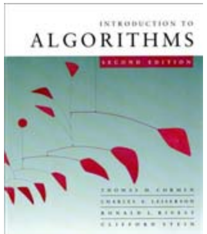
$$\frac{1}{10}cn \qquad\qquad \frac{9}{10}cn$$

$$T\left(\frac{1}{100}n\right)\ T\left(\frac{9}{100}n\right) \qquad T\left(\frac{9}{100}n\right)\ T\left(\frac{81}{100}n\right)$$

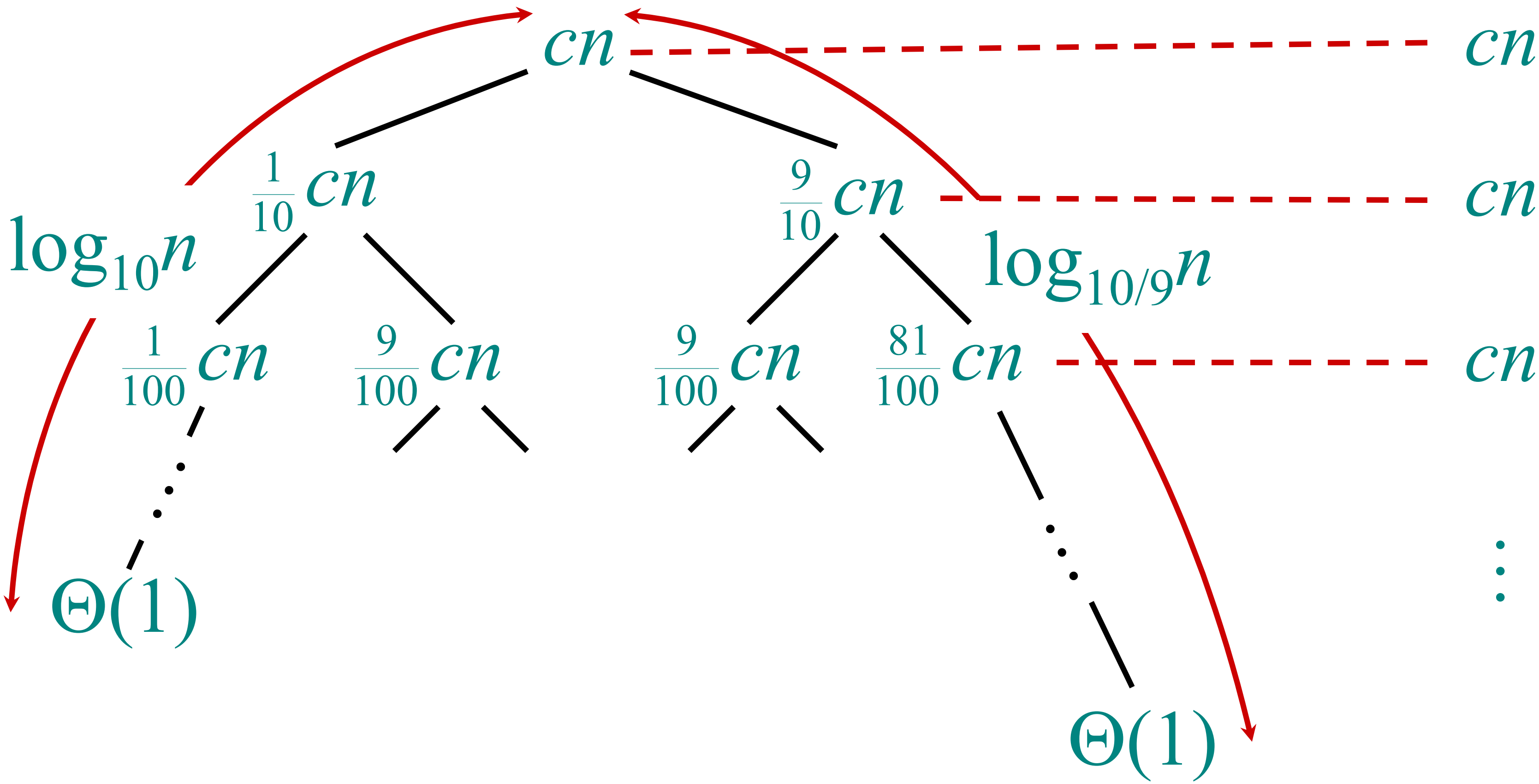


Analysis of nice case

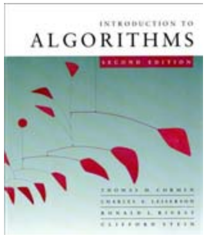




Analysis of nice case

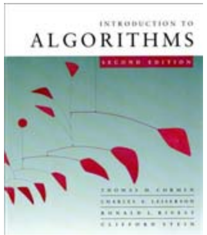


$$cn \log_{10} n \le T(n) \le cn \log_{10/9} n + O(n)$$



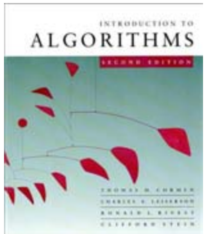
Randomized quicksort

- Partition around a ***random*** element. I.e., around A[t] , where t chosen uniformly at random from {p…r}
- We will show that the ***expected*** time is O(n log n)



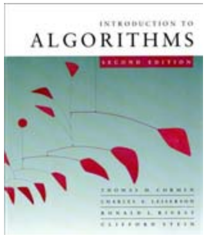
"Paranoid" quicksort

- Will modify the algorithm to make it easier to analyze:
  - Repeat:
    - Choose the pivot to be a random element of the array
    - Perform PARTITION
  - Until the resulting split is "lucky", i.e., not worse than 1/10: 9/10
  - Recurse on both sub-arrays



Analysis

- Let $T(n)$ be an upper bound on the ***expected*** running time on any array of n elements
- Consider any input of size n
- The time needed to sort the input is bounded from the above by a sum of
  - The time needed to sort the left subarray
  - The time needed to sort the right subarray
  - The number of iterations until we get a lucky split, times cn



Expectations

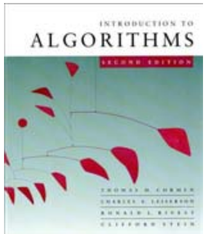
- By linearity of expectation:

$$T(n) \le \max T(i) + T(n-i) + E[\#\, partitions] \bullet cn$$

where maximum is taken over $i \in [n/10, 9n/10]$

- We will show that $E[\#partitions]$ is $\le 10/8$
- Therefore:

$$T(n) \le \max T(i) + T(n-i) + 2cn, i \in [n/10, 9n/10]$$



Final bound

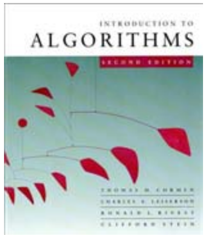
- Can use the recursion tree argument:
  - Tree depth is $\Theta(\log n)$
  - Total expected work at each level is at most $10/8\ cn$
  - The total expected time is $O(n \log n)$

Lucky partitions

- The probability that a random pivot induces lucky partition is at least 8/10

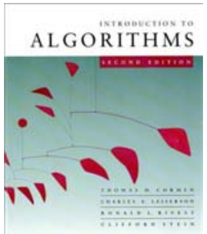
  (we are ***not*** lucky if the pivot happens to be among the smallest/largest $n/10$ elements)

- If we flip a coin, with heads prob. $p=8/10$, the expected waiting time for the first head is $1/p = 10/8$



Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from ***code tuning***.
- Quicksort behaves well even with caching and virtual memory.
- Quicksort is great!



More intuition

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ….

$$L(n) = 2U(n/2) + \Theta(n) \quad \textbf{\textit{lucky}}$$

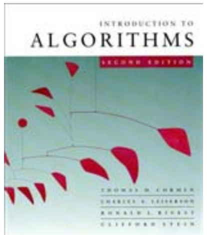
$$U(n) = L(n-1) + \Theta(n) \quad \textbf{\textit{unlucky}}$$

Solving:

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

Lucky!

How can we make sure we are usually lucky?



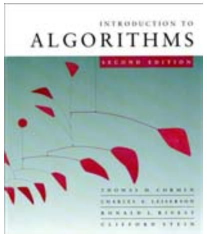
Randomized quicksort analysis

Let $T(n)$ = the random variable for the running time of randomized quicksort on an input of size n, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the ***indicator random variable***

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

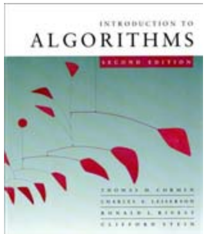
$E[X_k] = \Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.



Analysis (continued)

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} \end{cases}$$

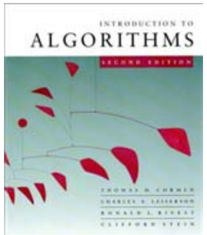
$$= \sum_{k=0}^{n-1} X_k \big(T(k) + T(n-k-1) + \Theta(n) \big) .$$



Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

Take expectations of both sides.

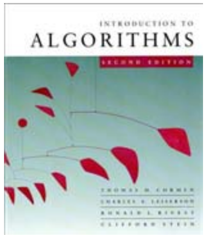


Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\bigl(T(k) + T(n-k-1) + \Theta(n)\bigr)\right]$$

$$= \sum_{k=0}^{n-1} E\bigl[X_k\bigl(T(k) + T(n-k-1) + \Theta(n)\bigr)\bigr]$$

Linearity of expectation.



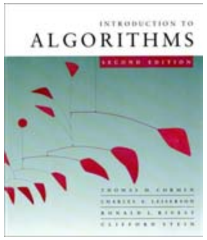
Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\bigl(T(k) + T(n-k-1) + \Theta(n)\bigr)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k\bigl(T(k) + T(n-k-1) + \Theta(n)\bigr)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

Independence of X_k from other random choices.



Calculating expectation

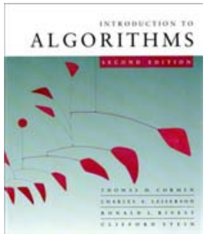
$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

$$= \sum_{k=0}^{n-1} E\big[X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\big]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

Linearity of expectation; $E[X_k] = 1/n$.



Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

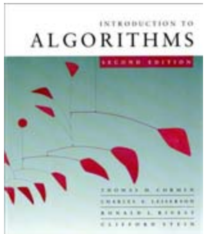
$$= \sum_{k=0}^{n-1} E\left[X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

$$= \frac{2}{n}\sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

Summations have identical terms.



Hairy recurrence

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

(The $k = 0, 1$ terms can be absorbed in the $\Theta(n)$.)

Prove: $E[T(n)] \le a\, n \lg n$ for constant $a > 0$.

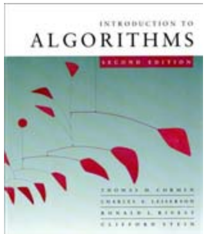
- Choose a large enough so that $a\, n \lg n$ dominates $E[T(n)]$ for sufficiently small $n \ge 2$.

Use fact: $\sum_{k=2}^{n-1} k \lg k \le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$ (exercise).

Substitution method

$$E[T(n)] \leq \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n)$$

Substitute inductive hypothesis.

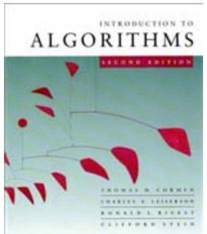


Substitution method

$$E[T(n)] \le \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n)$$

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n)$$

Use fact.



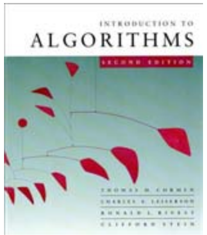
Substitution method

$$E[T(n)] \le \frac{2}{n} \sum_{k=2}^{n-1} ak \lg k + \Theta(n)$$

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \Theta(n)$$

$$= an \lg n - \left(\frac{an}{4} - \Theta(n)\right)$$

Express as ***desired – residual***.



Substitution method

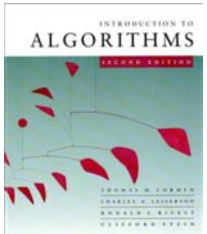
$$E[T(n)] \le \frac{2}{n} \sum_{k=2}^{n-1} ak \lg k + \Theta(n)$$

$$= \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \Theta(n)$$

$$= an \lg n - \left(\frac{an}{4} - \Theta(n)\right)$$

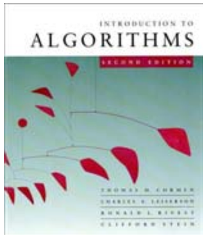
$$\le an \lg n,$$

if a is chosen large enough so that $an/4$ dominates the $\Theta(n)$.



- Assume

Running time $= O(n)$ for n elements.



Randomized Algorithms

- Algorithms that make decisions based on random coin flips.
- Can “fool” the adversary.
- The running time (or even correctness) is a random variable; we measure the ***expected*** running time.
- We assume all random choices are ***independent*** .
- This is ***not*** the average case !