香港科技大学(广州)
THE HONG KONG
UNIVERSITY OF SCIENCE AND
TECHNOLOGY (GUANGZHOU)

Design and Analysis of Algorithms

Jing Tang | DSAA 2043 Fall 2024

Merge Sort

Merge Sort

MERGE-SORT $A[1 \ldots n]$

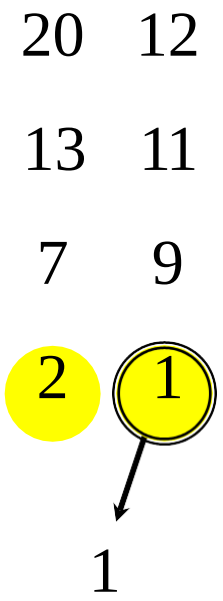
1. If $n = 1$, done
2. Recursively sort $A[1 \ldots \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \ldots n]$
3. "***Merge***" the 2 sorted lists

Key subroutine: **MERGE**

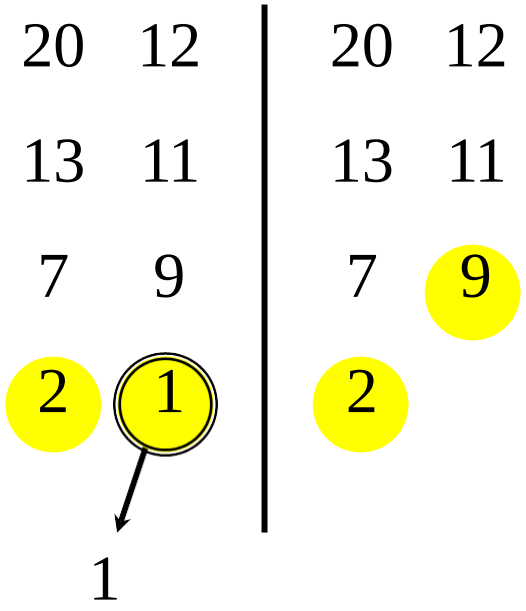
Merging Two Sorted Arrays

20	12
13	11
7	9
2	1

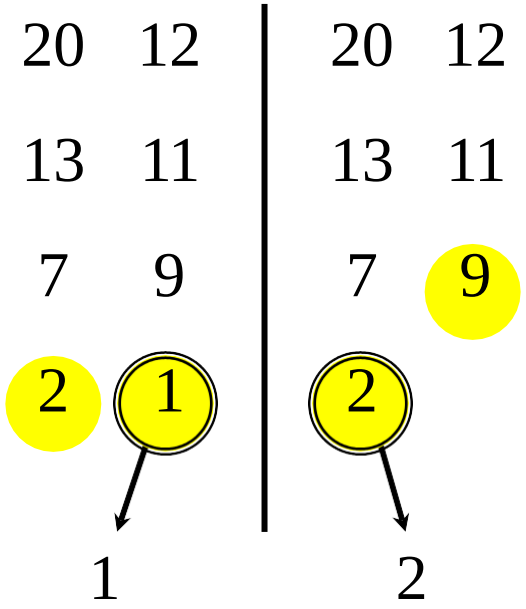
Merging Two Sorted Arrays



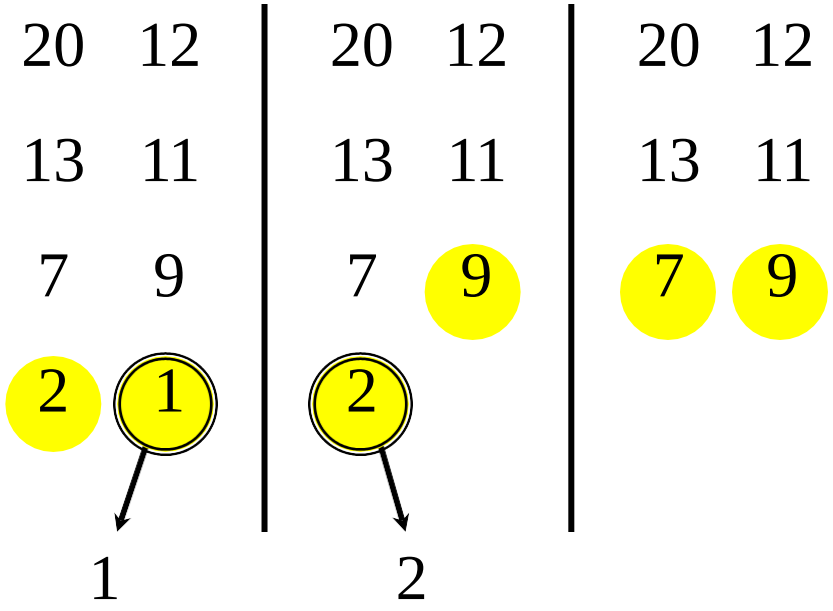
Merging Two Sorted Arrays



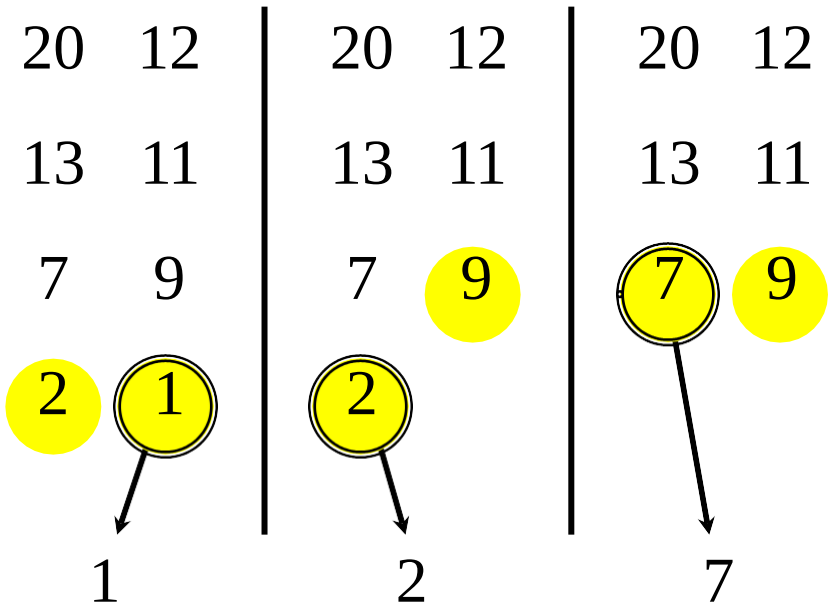
Merging Two Sorted Arrays



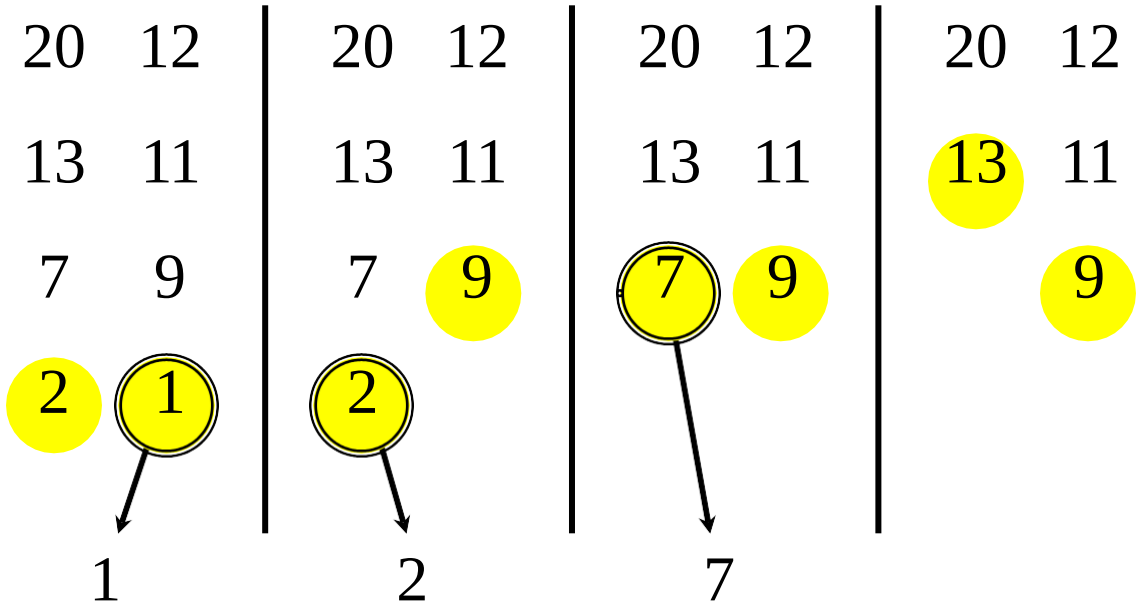
Merging Two Sorted Arrays



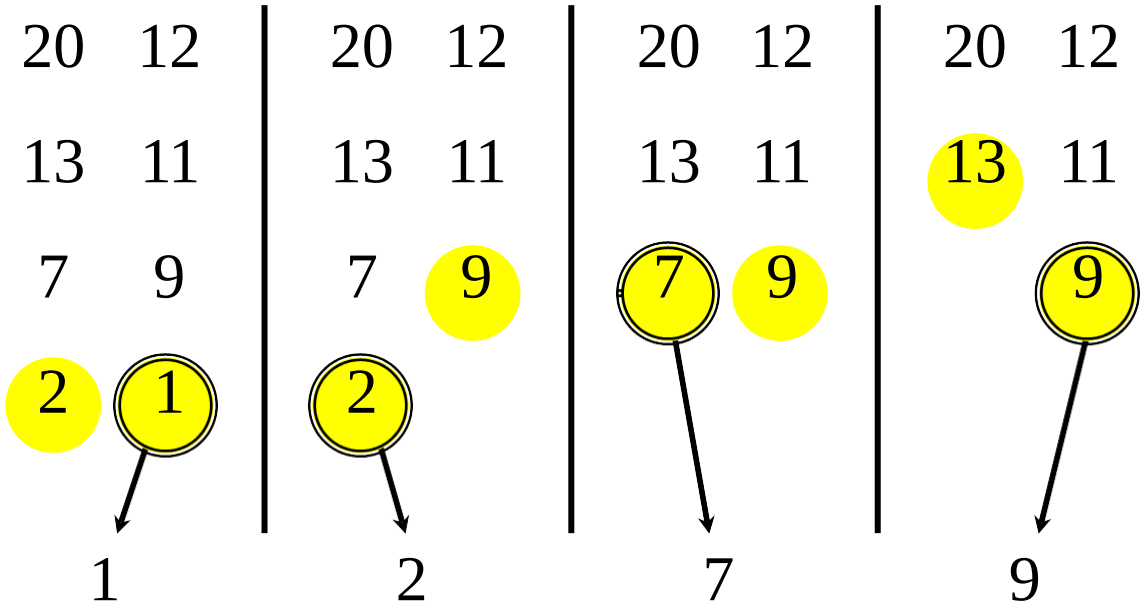
Merging Two Sorted Arrays



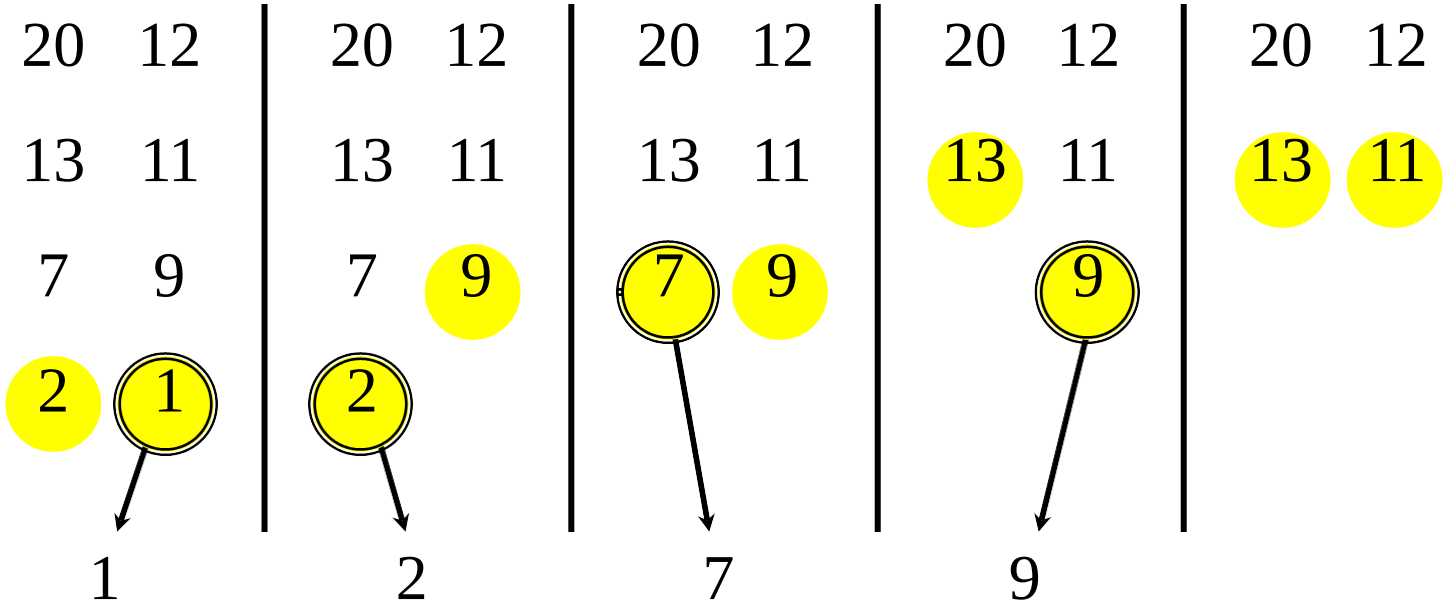
Merging Two Sorted Arrays



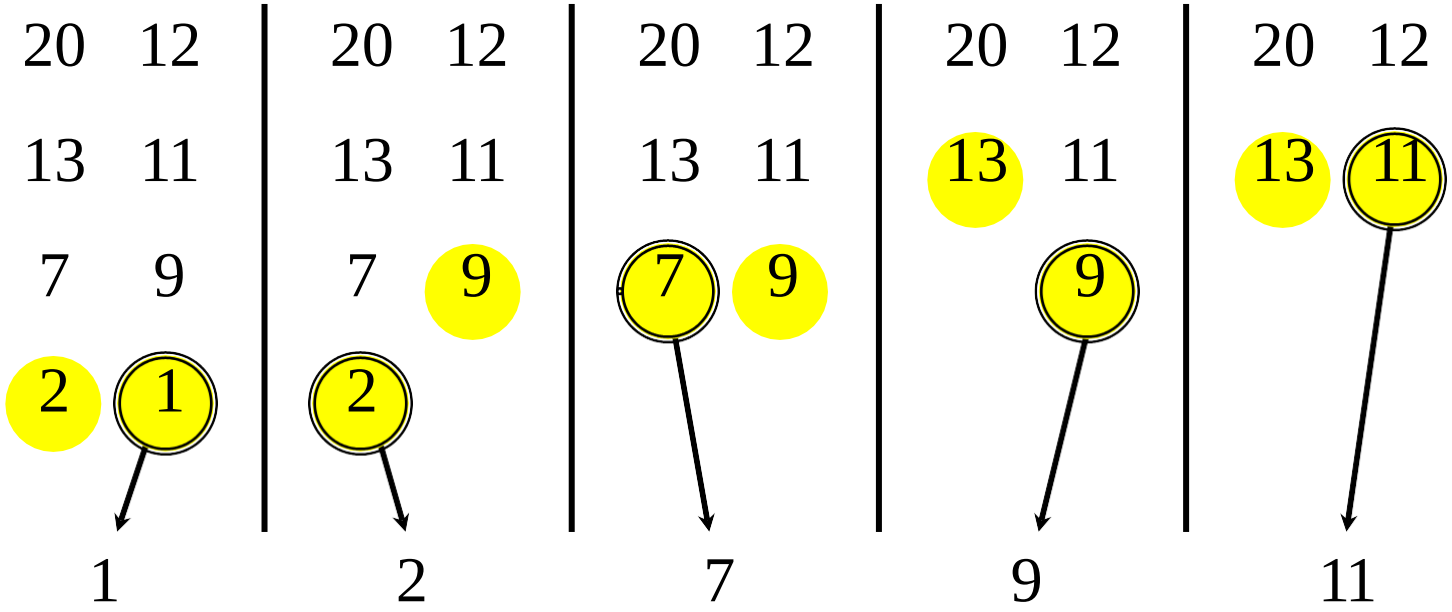
Merging Two Sorted Arrays



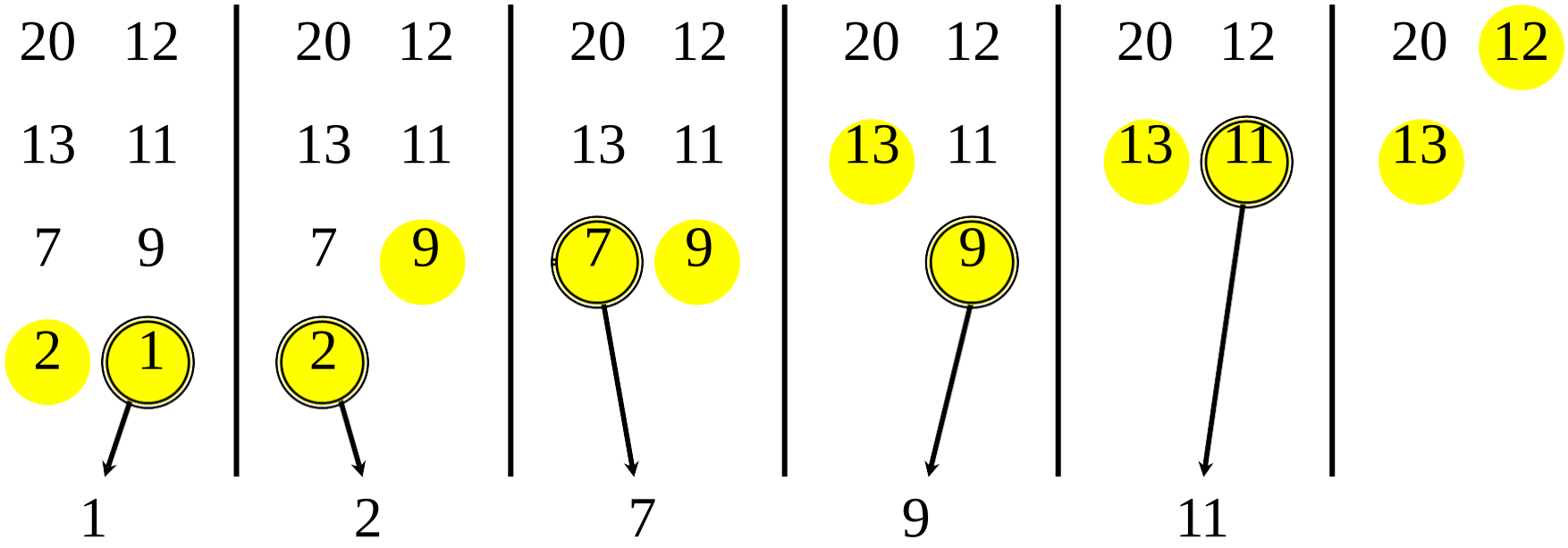
Merging Two Sorted Arrays



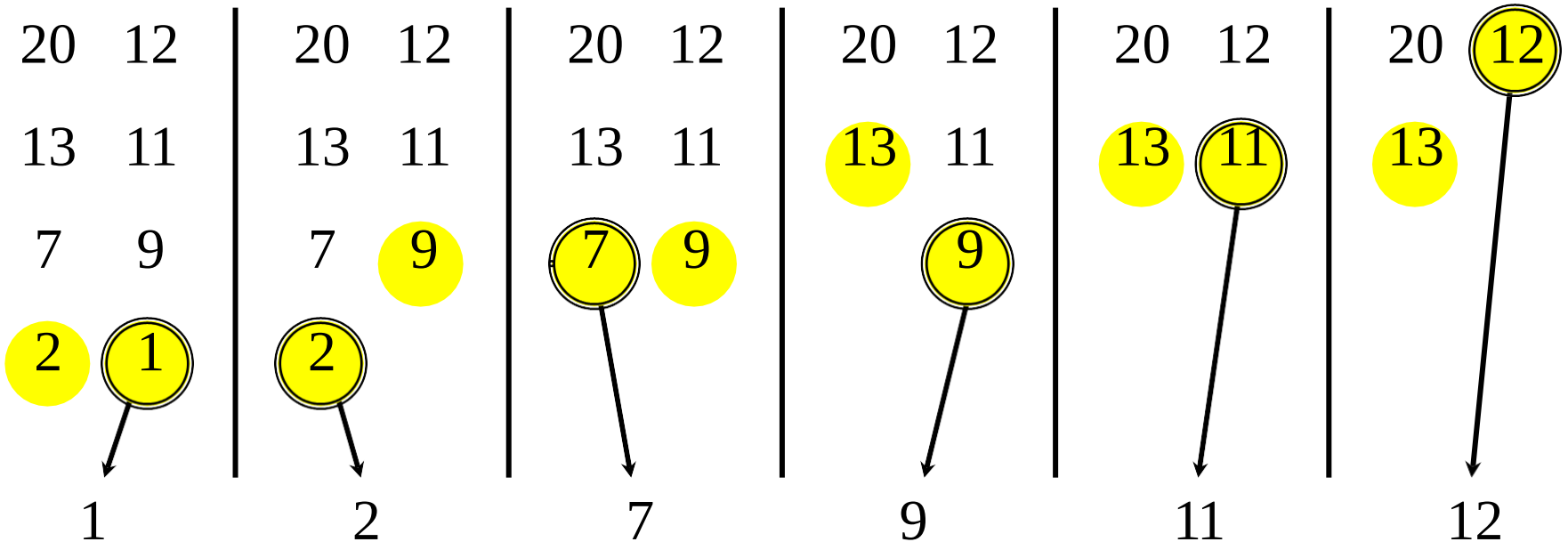
Merging Two Sorted Arrays



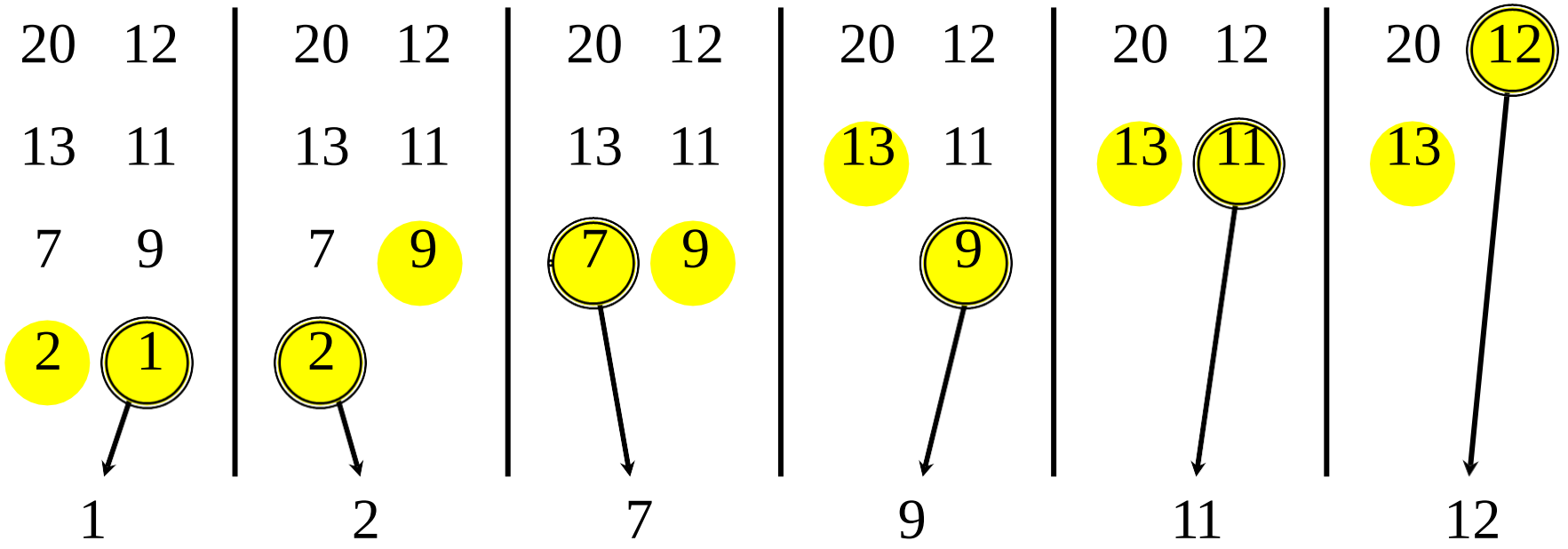
Merging Two Sorted Arrays



Merging Two Sorted Arrays

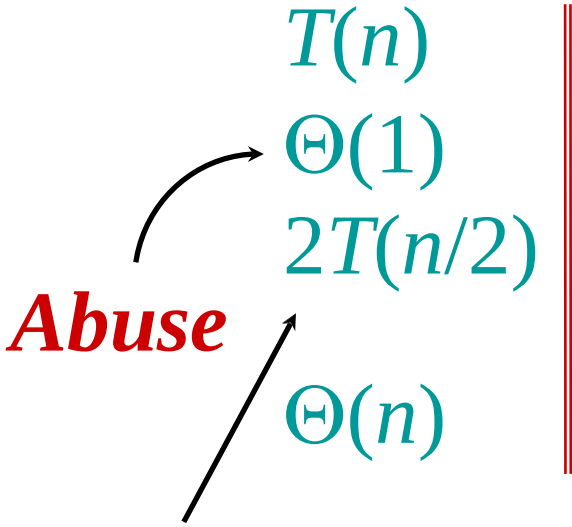


Merging Two Sorted Arrays



Time = $\Theta(n)$ to merge a total of n elements (linear time)

Analyzing Merge Sort



$T(n)$

$\Theta(1)$

$2T(n/2)$

$\Theta(n)$

MERGE-SORT $A[1 \ldots n]$

1. If $n = 1$, done
2. Recursively sort $A[1 \ldots \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \ldots n]$
3. ***"Merge"*** the 2 sorted lists

Sloppiness: Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically.

Recurrence for Merge Sort

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.
- CLRS and Lecture 2 provide several ways to find a good upper bound on $T(n)$.

Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

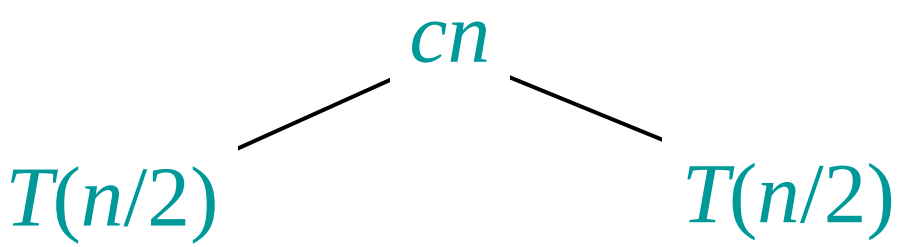
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$T(n)$

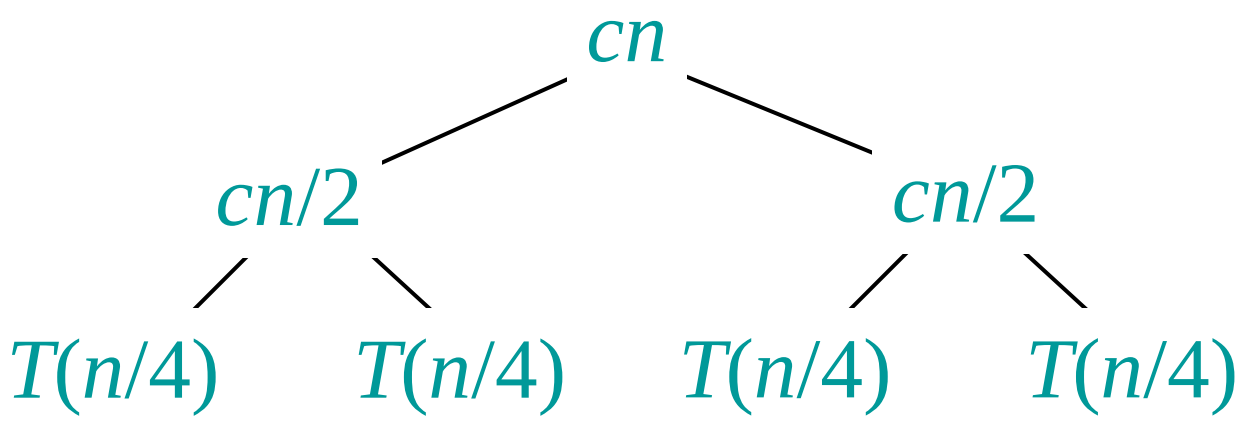
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



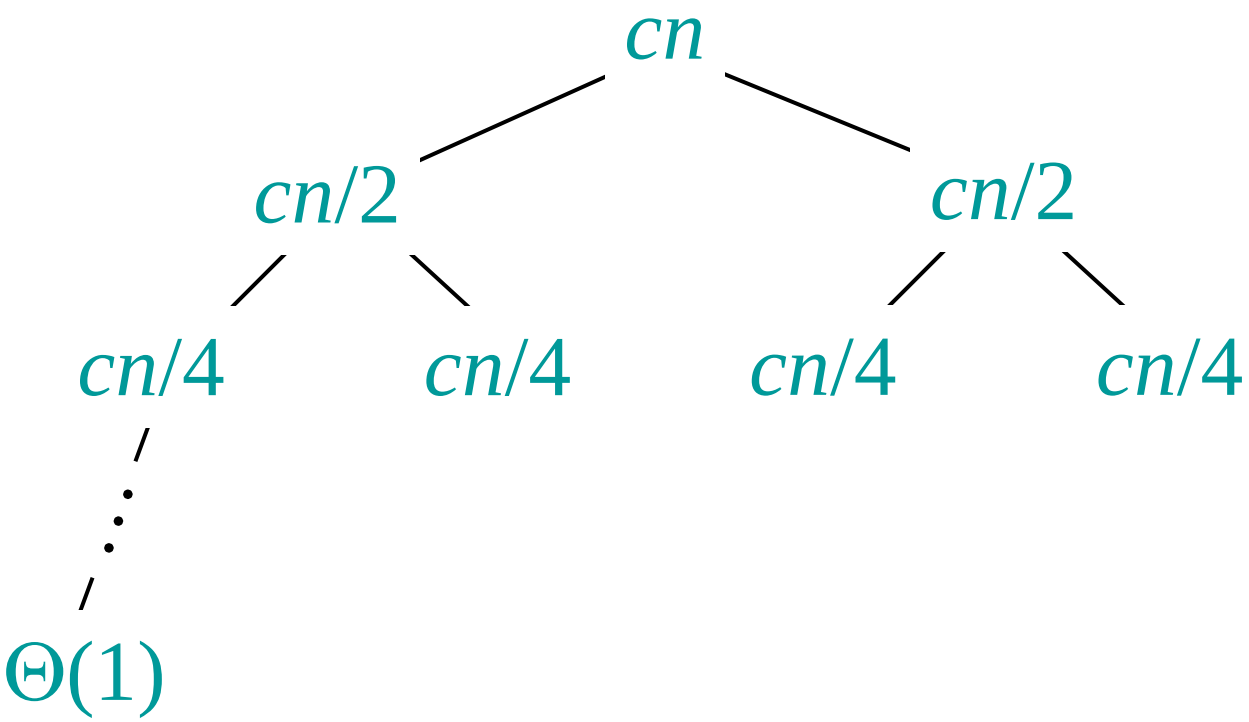
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



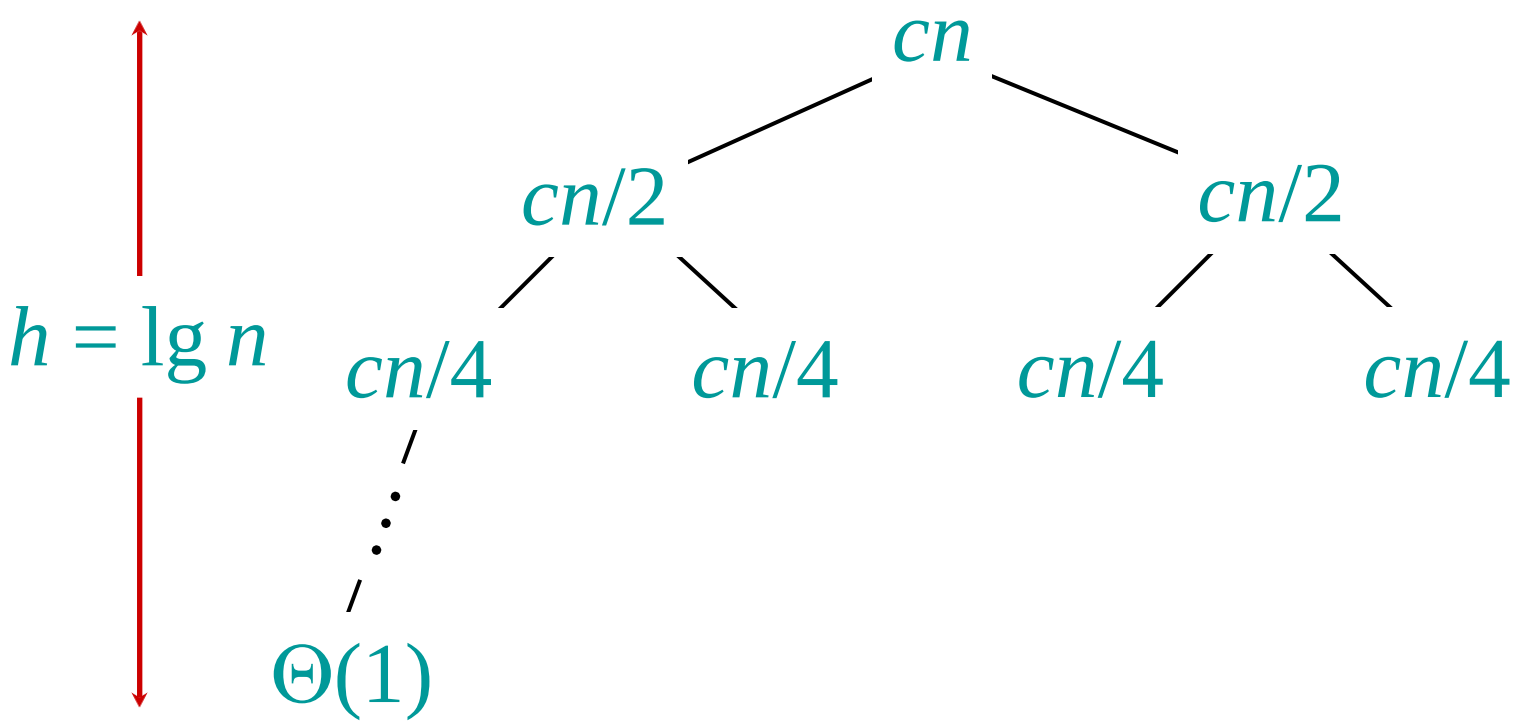
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



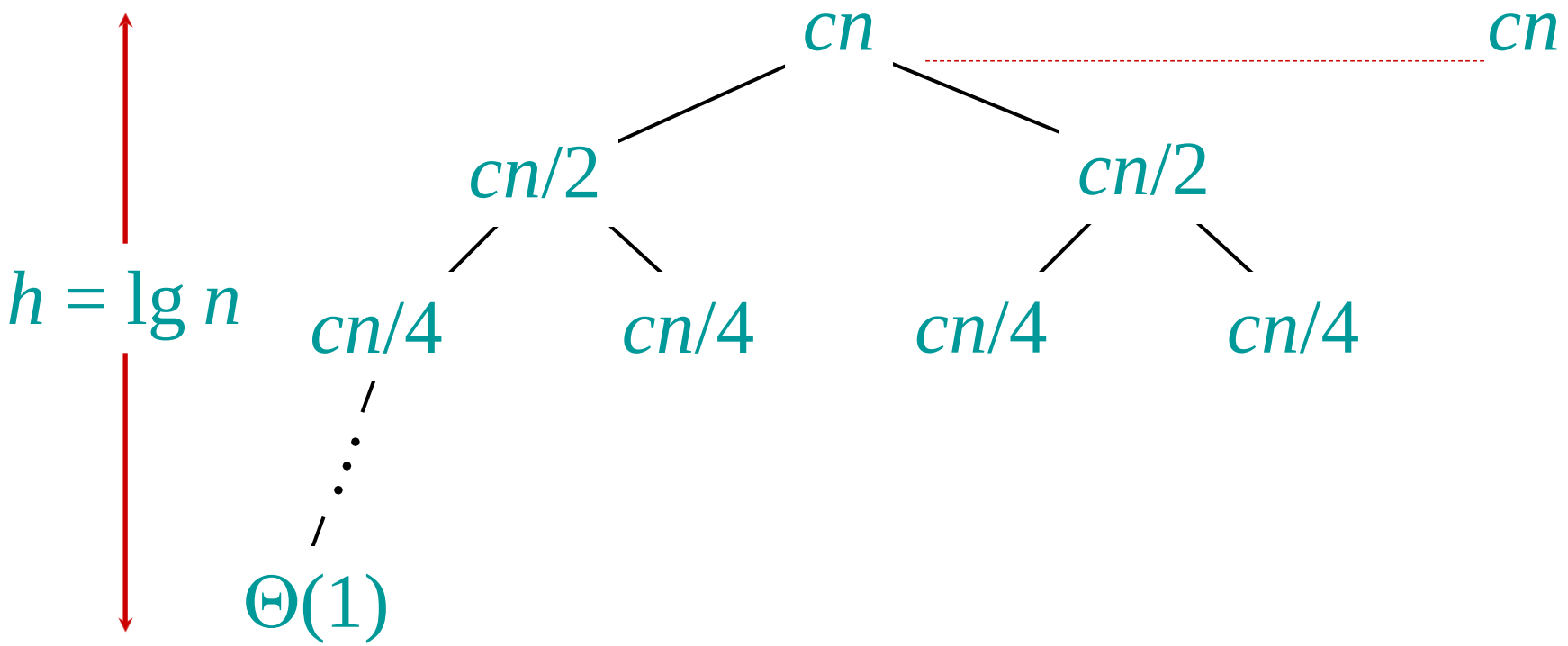
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



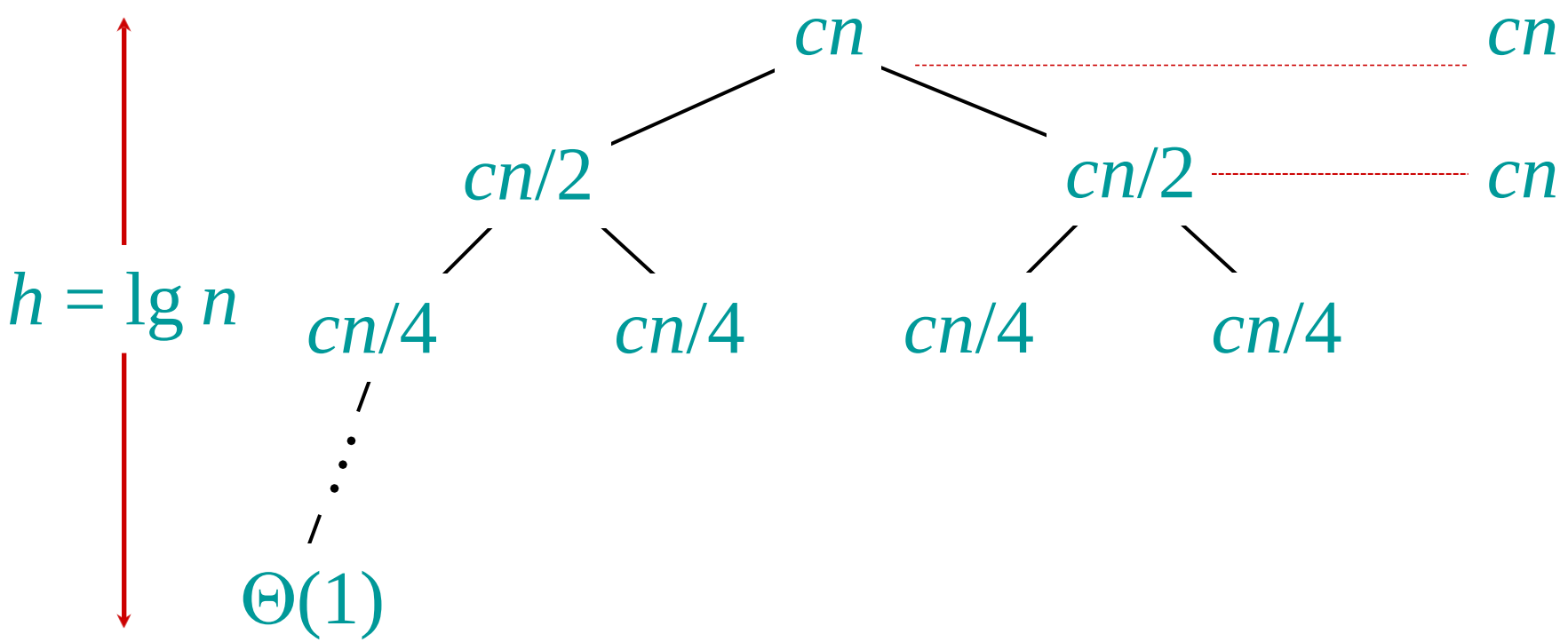
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



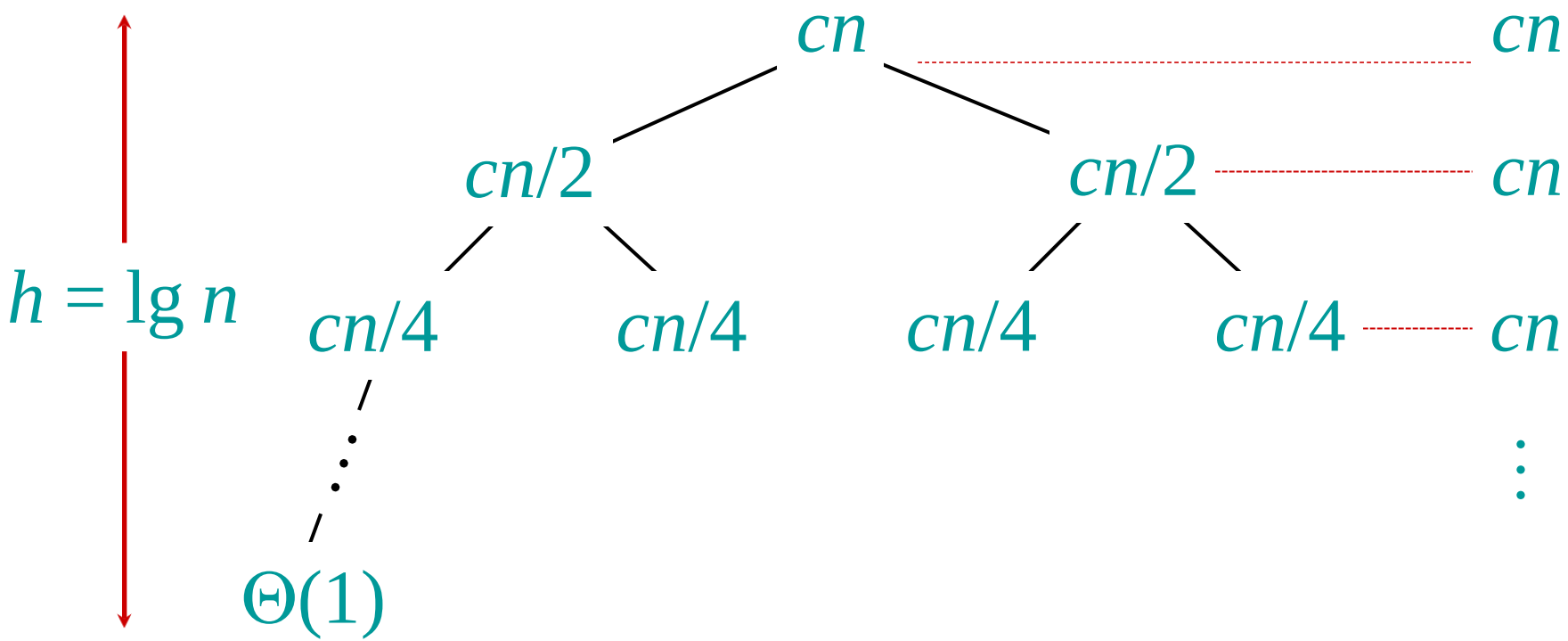
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



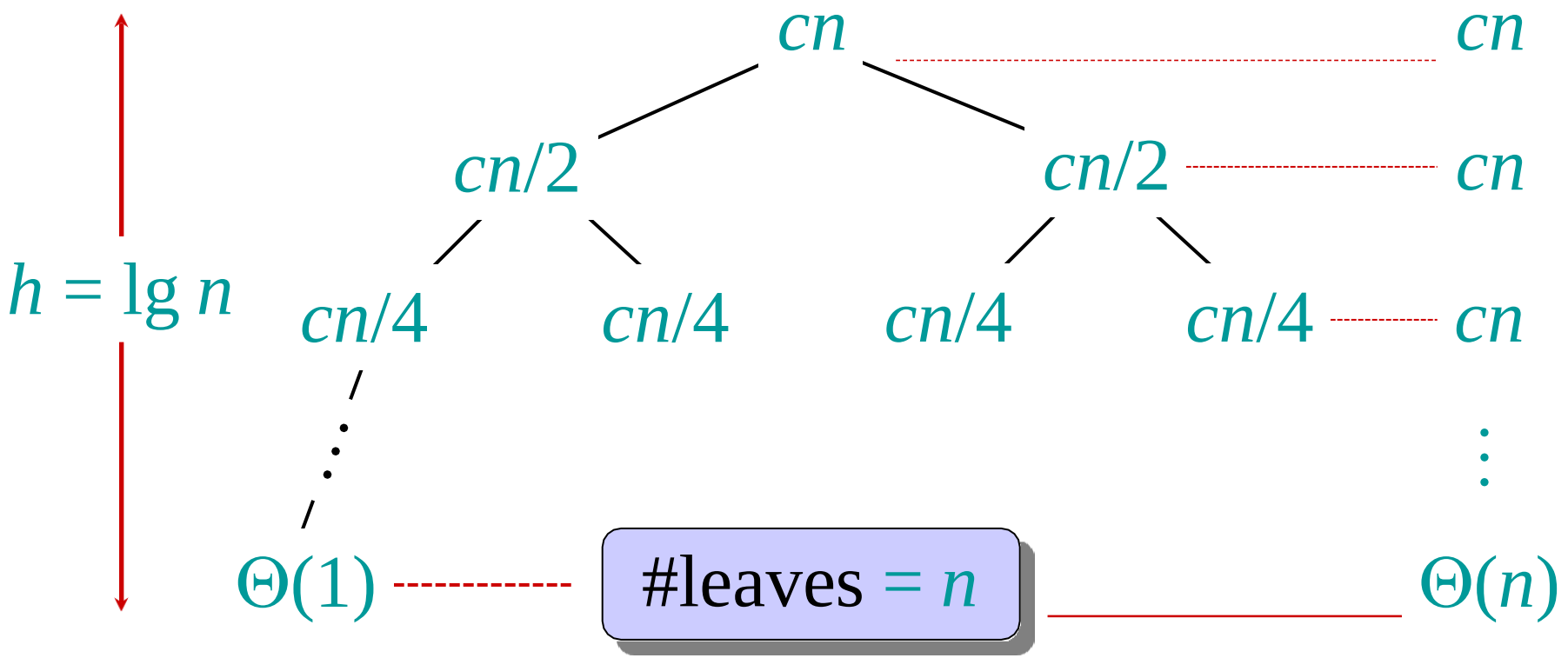
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



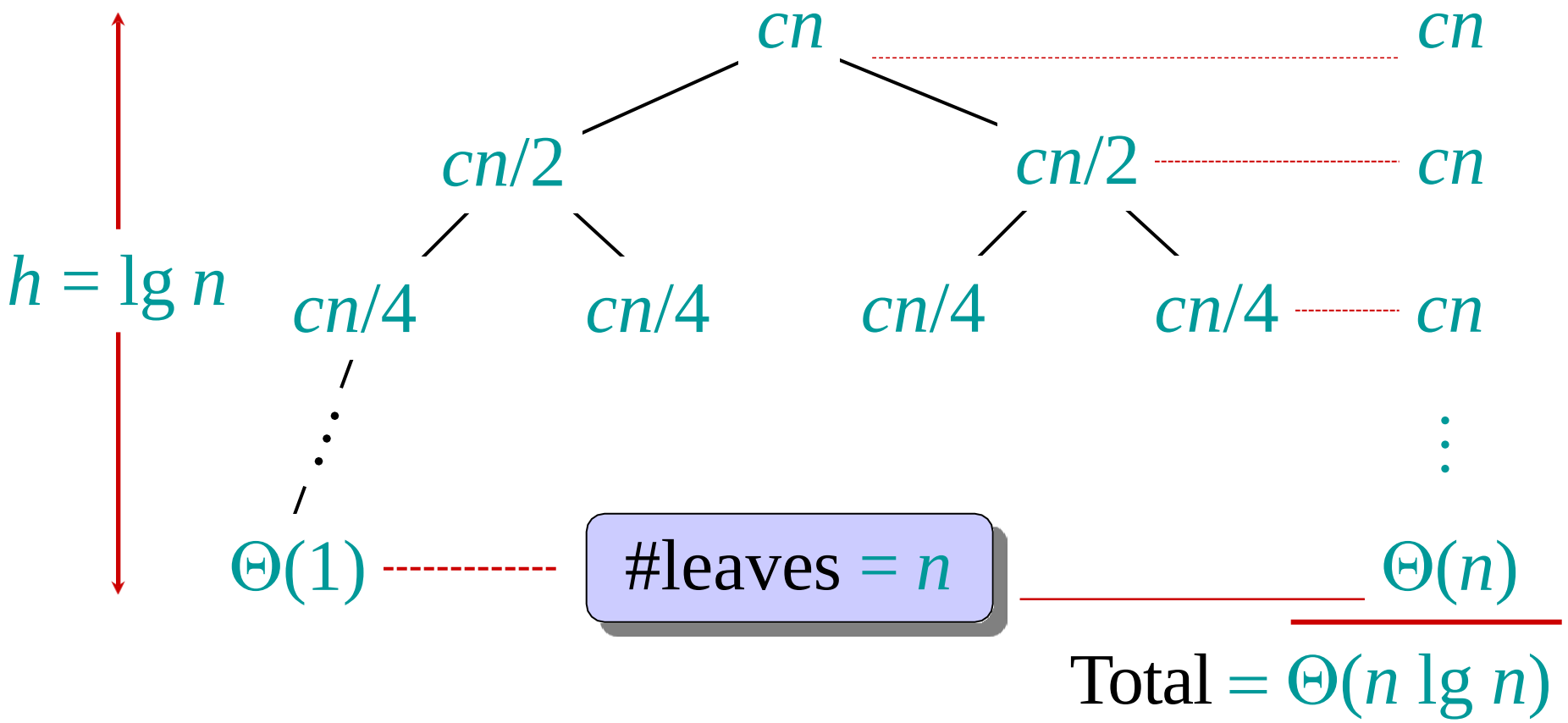
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Conclusions

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.
- Go test it out for yourself!

Divide and Conquer

The Divide-and-Conquer Design Paradigm

1. ***Divide*** the problem (instance) into subproblems.
2. ***Conquer*** the subproblems by solving them recursively.
3. ***Combine*** subproblem solutions.

Merge Sort

1. ***Divide:*** Trivial.
2. ***Conquer:*** Recursively sort 2 subarrays.
3. ***Combine:*** Linear-time merge.

Merge Sort

1. ***Divide:*** Trivial.
2. ***Conquer:*** Recursively sort 2 subarrays.
3. ***Combine:*** Linear-time merge.

$$T(n) = 2\,T(n/2) + \Theta(n)$$

 # subproblems

subproblem size

work dividing and combining

Master Theorem (Reprise)

$$T(n) = a\,T(n/b) + f(n)$$

CASE 1: $f(n) = O(n^{\log_b a - \varepsilon})$, constant $\varepsilon > 0$

$\Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$

$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$, constant $\varepsilon > 0$,
and regularity condition

$\Rightarrow T(n) = \Theta(f(n))$.

Master Theorem (Reprise)

$$T(n) = a\,T(n/b) + f(n)$$

CASE 1: $f(n) = O(n^{\log_b a - \varepsilon})$, constant $\varepsilon > 0$

$\Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$

$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$, constant $\varepsilon > 0$,
and regularity condition

$\Rightarrow T(n) = \Theta(f(n))$.

Merge sort: $a = 2, b = 2 \Rightarrow n^{\log_b a} = n^{\log_2 2} = n$

$\Rightarrow$ CASE 2 ($k = 0$) $\Rightarrow T(n) = \Theta(n \lg n)$.

Master Theorem (Proof)

$$T(n) = a\,T(n/b) + f(n)$$

$$\boldsymbol{T(n) = \Theta\left(n^{\log_b a}\right) + \sum_{i=0}^{k-1} a^i f\left(\frac{n}{b^i}\right),}$$

$$\boldsymbol{n = b^k, k = \log_b n\,, a^k = a^{\log_b n} = n^{\log_b a}}$$

For case1: $\boldsymbol{f(n) = O(n^{(\log_b a)-\varepsilon}),\ \varepsilon > 0}$

We have: $\boldsymbol{g(n) = O\left(\sum_{i=0}^{k-1} a^i \left(\frac{n}{b^i}\right)^{(\log_b a)-\varepsilon}\right)}$

$$= O\left(n^{(\log_b a)-\varepsilon} \sum_{i=0}^{k-1} \left(\frac{ab^\varepsilon}{b^{\log_b a}}\right)^i\right)$$

$$= O\left(n^{(\log_b a)-\varepsilon} \sum_{i=0}^{k-1} (b^\varepsilon)^i\right)$$

$$= O\left(n^{(\log_b a)-\varepsilon} \sum_{i=0}^{k-1} (b^\varepsilon)^i\right)$$

$$= O\left(n^{(\log_b a)-\varepsilon} \frac{n^\varepsilon - 1}{b^\varepsilon - 1}\right)$$

$$= O\left(n^{\log_b a}\right)$$

Try to solve case 2 in lab!

$$\longrightarrow \quad \boldsymbol{g(n) = \sum_{i=0}^{k-1} a^i f\left(\frac{n}{b^i}\right)}$$

For case3: $\boldsymbol{f(n) = \Omega(n^{(\log_b a)+\varepsilon}),\ \varepsilon > 0}$

$$\boldsymbol{af\left(\frac{n}{b}\right) \le cf(n),\ c < 1}$$

We have: $\boldsymbol{af\left(\frac{n}{b^2}\right) \le cf\left(\frac{n}{b}\right)}$

$$\vdots$$

$$\boldsymbol{af\left(\frac{n}{b^i}\right) \le cf\left(\frac{n}{b^{i-1}}\right)}$$

Multiply both sides: $\boldsymbol{a^i f\left(\frac{n}{b^i}\right) \le c^i f(n)}$

$$\boldsymbol{g(n) = \sum_{i=0}^{k-1} a^i f\left(\frac{n}{b^i}\right) \le \sum_{i=0}^{k-1} c^i f(n) = f(n) \sum_{i=0}^{k-1} c^i}$$

$$\boldsymbol{\le f(n) \frac{1}{1-c} = \Theta(f(n))}$$

Binary Search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Binary Search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary Search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

Binary Search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

Binary Search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

Binary Search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

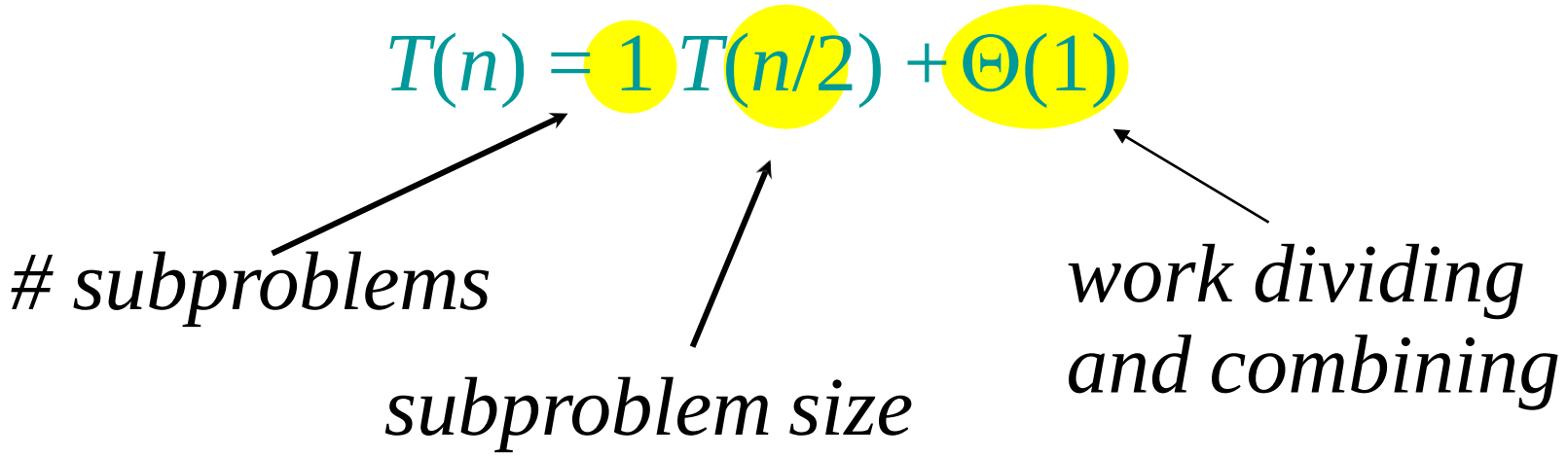
Binary Search

Find an element in a sorted array:

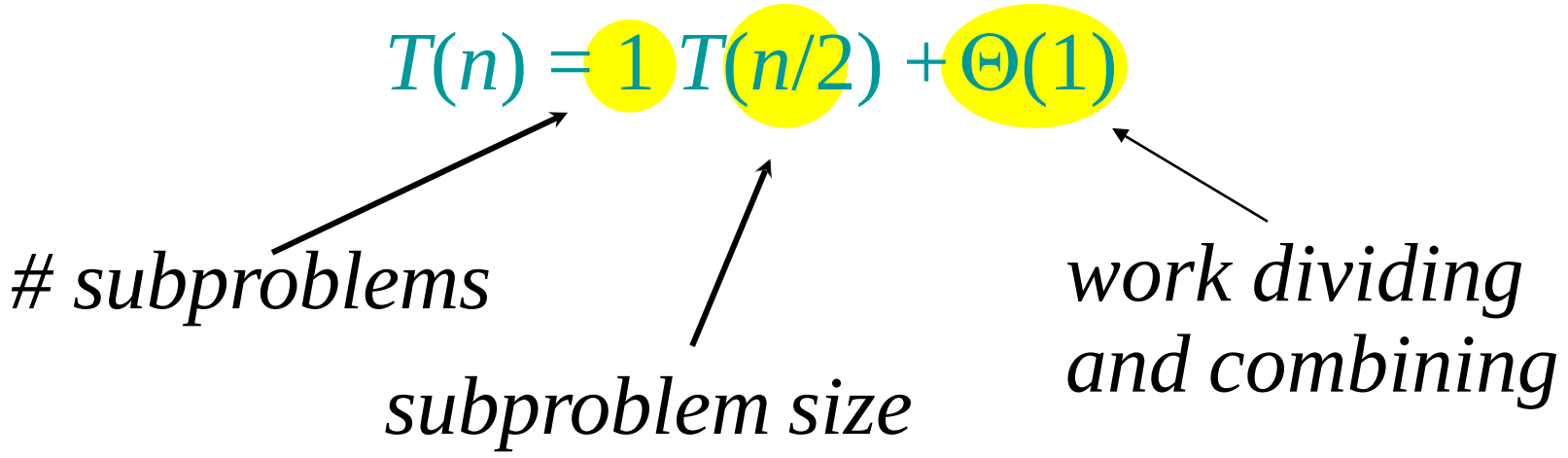
1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

Recurrence for Binary Search



Recurrence for Binary Search



$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \Rightarrow \text{CASE 2 } (k = 0)$$

$$\Rightarrow T(n) = \Theta(\lg n).$$

Powering a Number

Problem: Compute a^n, where $n \in \boldsymbol{N}$.

Naive algorithm: $\Theta(n)$.

Powering a Number

Problem: Compute a^n, where $n \in \boldsymbol{N}$.

Naive algorithm: $\Theta(n)$.

Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

Powering a Number

Problem: Compute a^n, where $n \in \boldsymbol{N}$.

Naive algorithm: $\Theta(n)$.

Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(\lg n).$$

Fibonacci Numbers

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Recursive definition:

$$F_n = \begin{cases} 1 & \text{if } n = 0; \\ 2 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

0 1 1 2 3 5 8 13 21 34 ...

Fibonacci Numbers

Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

0 1 1 2 3 5 8 13 21 34 ...

Naive recursive algorithm: $\Omega(\phi^n)$ (exponential time), where $\phi = (1+\sqrt{5})/2$ is the ***golden ratio***.

Computing Fibonacci Numbers

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Bottom-up:

- Compute $F_0, F_1, F_2, \ldots, F_n$ in order, forming each number by summing the two previous.
- Running time: $\Theta(n)$.

Computing Fibonacci Numbers

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Bottom-up:

- Compute $F_0, F_1, F_2, \ldots, F_n$ in order, forming each number by summing the two previous.
- Running time: $\Theta(n)$.

Naive recursive squaring:

$F_n = \phi^n/5$ rounded to the nearest integer.

- Recursive squaring: $\Theta(\lg n)$ time.
- This method is unreliable, since floating-point arithmetic is prone to round-off errors.

Recursive Squaring

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Theorem: $$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

Recursive Squaring

Theorem: $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$

Algorithm: Recursive squaring.

Time = $\Theta(\lg n)$.

Recursive Squaring

Theorem: $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$

Algorithm: Recursive squaring.

Time = $\Theta(\lg n)$.

Proof of theorem. (Induction on n.)

Base ($n = 1$): $\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1$

Recursive Squaring

Inductive step ($n \geq 2$):

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n}$$

Or Equivalently

$$\begin{bmatrix} F_{n+1} & F_n \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Matrix Multiplication

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Input: $A = [a_{ij}], B = [b_{ij}].$
Output: $C = [c_{ij}] = A \cdot B.$

$\Big\}\ i, j = 1, 2, \ldots, n.$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

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

Standard Algorithm

for $i \leftarrow 1$ **to** n
  do for $j \leftarrow 1$ **to** n
    do $c_{ij} \leftarrow 0$
      for $k \leftarrow 1$ **to** n
        do $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$

Standard Algorithm

for $i \leftarrow 1$ **to** n
  do for $j \leftarrow 1$ **to** n
    do $c_{ij} \leftarrow 0$
      for $k \leftarrow 1$ **to** n
        do $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$

Running time = $\Theta(n^3)$

Divide-and-Conquer Algorithm

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

IDEA:

$n \times n$ matrix = 2×2 matrix of $(n/2) \times (n/2)$ submatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \quad B$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned}\right\} \begin{aligned} &8 \text{ mults of } (n/2) \times (n/2) \text{ submatrices} \\ &4 \text{ adds of } (n/2) \times (n/2) \text{ submatrices} \end{aligned}$$

Divide-and-Conquer Algorithm

IDEA:

$n \times n$ matrix = 2×2 matrix of $(n/2) \times (n/2)$ submatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \quad B$$

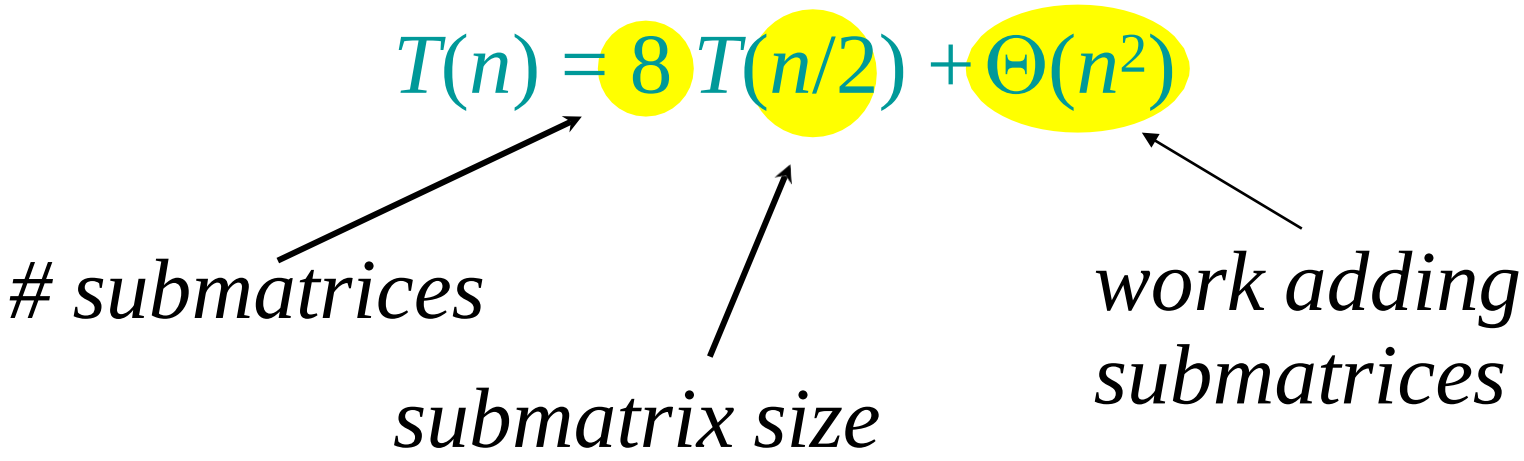
$r = ae + bg$
$s = af + bh$
$t = ce + dh$
$u = cf + dg$

recursive

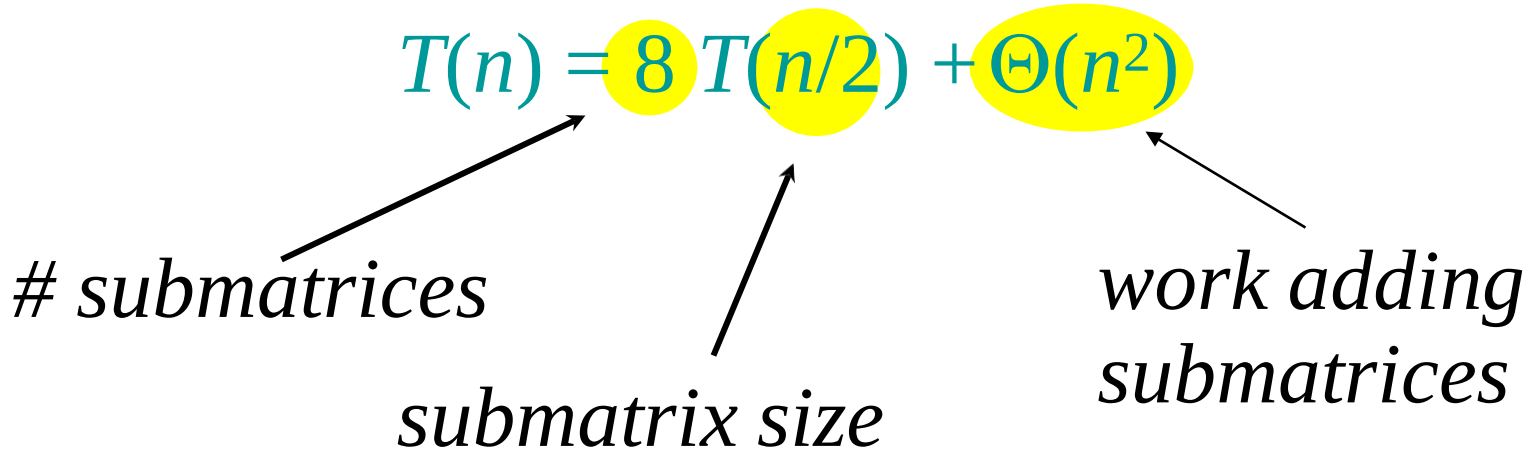
8 mults of $(n/2) \times (n/2)$ submatrices

4 adds of $(n/2) \times (n/2)$ submatrices

Analysis of D&C Algorithm



Analysis of D&C Algorithm



$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow$ CASE 1 $\Rightarrow T(n) = \Theta(n^3)$.

Analysis of D&C Algorithm

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

 # submatrices

submatrix size

work adding submatrices

$$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^3).$$

No better than the ordinary algorithm.

Strassen's Idea

- Multiply 2×2 matrices with only 7 recursive mults.

Strassen's Idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$P_1 = a \cdot (f - h)$$
$$P_2 = (a + b) \cdot h$$
$$P_3 = (c + d) \cdot e$$
$$P_4 = d \cdot (g - e)$$
$$P_5 = (a + d) \cdot (e + h)$$
$$P_6 = (b - d) \cdot (g + h)$$
$$P_7 = (a - c) \cdot (e + f)$$

Strassen's Idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$P_1 = a \cdot (f - h)$$
$$P_2 = (a + b) \cdot h$$
$$P_3 = (c + d) \cdot e$$
$$P_4 = d \cdot (g - e)$$
$$P_5 = (a + d) \cdot (e + h)$$
$$P_6 = (b - d) \cdot (g + h)$$
$$P_7 = (a - c) \cdot (e + f)$$

$$r = P_5 + P_4 - P_2 + P_6$$
$$s = P_1 + P_2$$
$$t = P_3 + P_4$$
$$u = P_5 + P_1 - P_3 - P_7$$

Strassen's Idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$P_1 = a \cdot (f - h)$$
$$P_2 = (a + b) \cdot h$$
$$P_3 = (c + d) \cdot e$$
$$P_4 = d \cdot (g - e)$$
$$P_5 = (a + d) \cdot (e + h)$$
$$P_6 = (b - d) \cdot (g + h)$$
$$P_7 = (a - c) \cdot (e + f)$$

$$r = P_5 + P_4 - P_2 + P_6$$
$$s = P_1 + P_2$$
$$t = P_3 + P_4$$
$$u = P_5 + P_1 - P_3 - P_7$$

7 mults, 18 adds/subs.
Note: No reliance on commutativity of mult!

Strassen's Idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$
\begin{aligned}
P_1 &= a \cdot (f - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (g - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (g + h) \\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
$$

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
&= (a + d)(e + h) \\
&\quad + d(g - e) - (a + b)h \\
&\quad + (b - d)(g + h) \\
&= ae + ah + de + dh \\
&\quad + dg - de - ah - bh \\
&\quad + bg + bh - dg - dh \\
&= ae + bg
\end{aligned}
$$

Strassen's Algorithm

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

1. ***Divide:*** Partition A and B into $(n/2)\times(n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.
2. ***Conquer:*** Perform 7 multiplications of $(n/2)\times(n/2)$ submatrices recursively.
3. ***Combine:*** Form C using $+$ and $-$ on $(n/2)\times(n/2)$ submatrices.

Strassen's Algorithm

1. ***Divide:*** Partition A and B into $(n/2)\times(n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.
2. ***Conquer:*** Perform 7 multiplications of $(n/2)\times(n/2)$ submatrices recursively.
3. ***Combine:*** Form C using $+$ and $-$ on $(n/2)\times(n/2)$ submatrices.

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

Analysis of Strassen

$$T(n) = 7\ T(n/2) + \Theta(n^2)$$

Analysis of Strassen

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^{\lg 7}).$$

Analysis of Strassen

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \Rightarrow$ CASE 1 $\Rightarrow T(n) = \Theta(n^{\lg 7})$.

The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 32$ or so.

Analysis of Strassen

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \Rightarrow$ CASE 1 $\Rightarrow T(n) = \Theta(n^{\lg 7})$.

The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 32$ or so.

Best to date (of theoretical interest only): $\Theta(n^{2.376\ldots})$.

Conclusion

- Divide and conquer is just one of several powerful techniques for algorithm design.
- Divide-and-conquer algorithms can be analyzed using recurrences and the master method (so practice this math).
- The divide-and-conquer strategy often leads to efficient algorithms.

Quick Sort

Quick Sort

- A popular sorting algorithm discovered by C.A.R. Hoare in 1962
  - In many situations, it's the fastest, operating in $O(N*\log N)$ time (for in-memory sorting)
- Basic scheme
  - The algorithm operates by partitioning an array into two subarrays
  - The algorithm then calls itself recursively to quicksort each of these subarrays
- Some embellishments we can make
  - selection of the pivot
  - sorting of small partitions

Partitioning

- Idea: Divide data into two groups, such that:
  - All items with a key value higher than a specified amount (the pivot) are in one group
  - All items with a lower key value are in another
- Applications:
  - Divide employees who live within 15 miles of the office with those who live farther away
  - Divide households by income for taxation purposes
  - Divide computers by processor speed
- Let's see an example with an array

Partitioning

- Say I have 12 values:
  - **175 192 95 45 115 105 20 60 185 5 90 180**
- I pick a pivot=104, and partition (NOT sorting yet):
  - **95 45 20 60 5 90 | 175 192 115 105 185 180**
  - Note: In the future the pivot will be an actual element
  - Also: Partitioning need not maintain order of elements and usually won't, although I did in this example
- The partition is the leftmost item in the right array:
  - **95 45 20 60 5 90 | 175 192 115 105 185 180**
- Which we return to designate where the division is located

Partitioning

- The partition process
  - Start with two pointers: *leftIndex* initialized to one position to the left of the first cell; *rightIndex* to one position to the right of the last cell
  - *leftIndex* moves to the right; *rightIndex* moves to the left

- Stopping and Swapping
  - When *leftIndex* encounters an item smaller than the pivot, it keeps going; when it finds a larger item, it stops
  - When *rightIndex* encounters an item larger than the pivot, it keeps going; when it finds a smaller item, it stops
  - When the two *indexs* eventually meet, the process is complete
  - When the two *indexs* stop, swap the two elements

Efficiency: Partitioning

- O(n) time
  - left starts at 0 and moves one-by-one to the right
  - right starts at n-1 and moves one-by-one to the left
  - When left and right cross, we stop.
    - So we'll hit each element just once
- Number of comparisons is n+1
- Number of swaps is worst case n/2
  - Worst case, we swap every single time
  - Each swap involves two elements
  - Usually, it will be less than this
    - Since in the random case, some elements will be on the correct side of the pivot

Modified Partitioning

- In preparation for Quicksort:
  - Choose our pivot value to be the rightmost element
  - Partition the array around the pivot
  - Ensure the pivot is at the location of the partition
    - Meaning, the pivot should be the leftmost element of the right subarray
- Example: Unpartitioned **42 89 63 12 94 27 78 3 50 36**
- Partitioned around Pivot: **3 27 12 36 63 94 89 78 42 50**
- What does this imply about the pivot element after the partition?

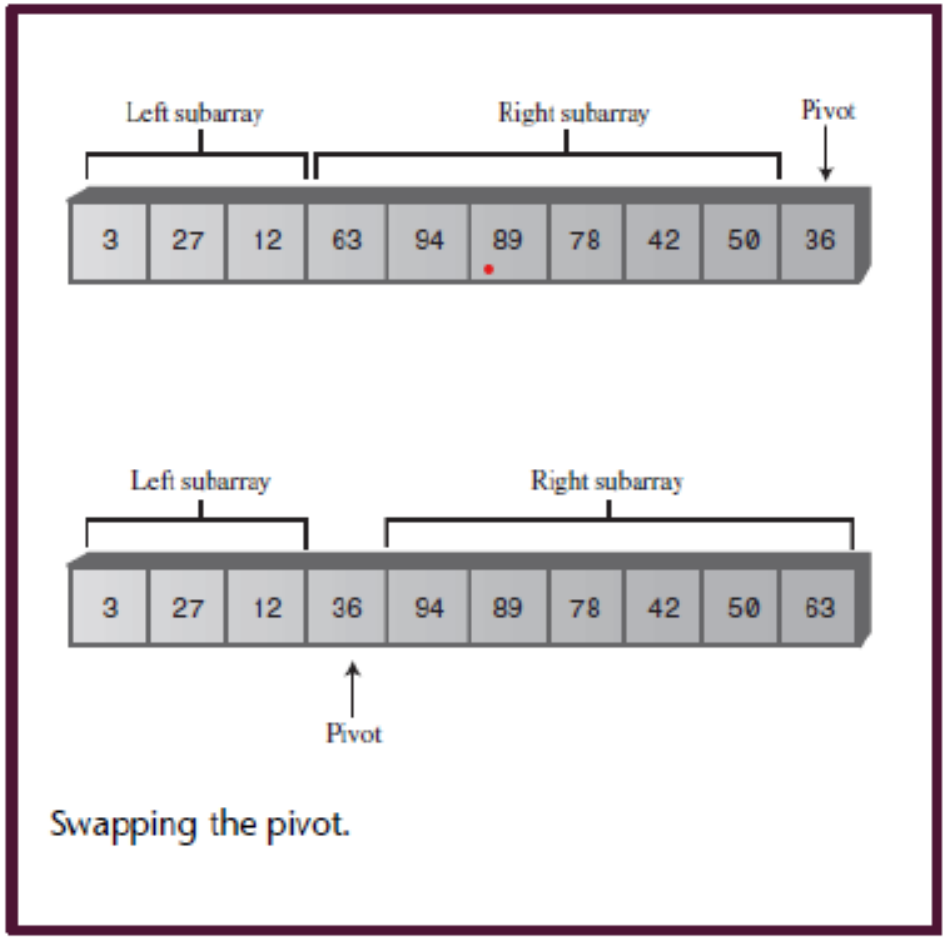
Placing the PIVOT

- Goal: Pivot must be in the leftmost position in the right subarray
  - **3 27 12 36 63 94 89 78 42 50**
- Our algorithm does not do this currently
- It currently will not touch the pivot
  - left increments till it finds an element > pivot
  - right decrements till it finds an element < pivot
  - So the pivot itself won't be touched, and will stay on the right:
  - **3 27 12 63 94 89 78 42 50 36**

OPTIONS

- We have this:
  - **3 27 12 63 94 89 78 42 50 36**
- Our goal is the position of 36:
  - **3 27 12 36 63 94 89 78 42 50**
- We could either:
  - Shift every element in the right subarray up (inefficient)
  - Just swap the leftmost with the pivot! Better
  - We can do this because the right subarray is not in any particular order
- **3 27 12 36 94 89 78 42 50 63**

Swapping the PIVOT

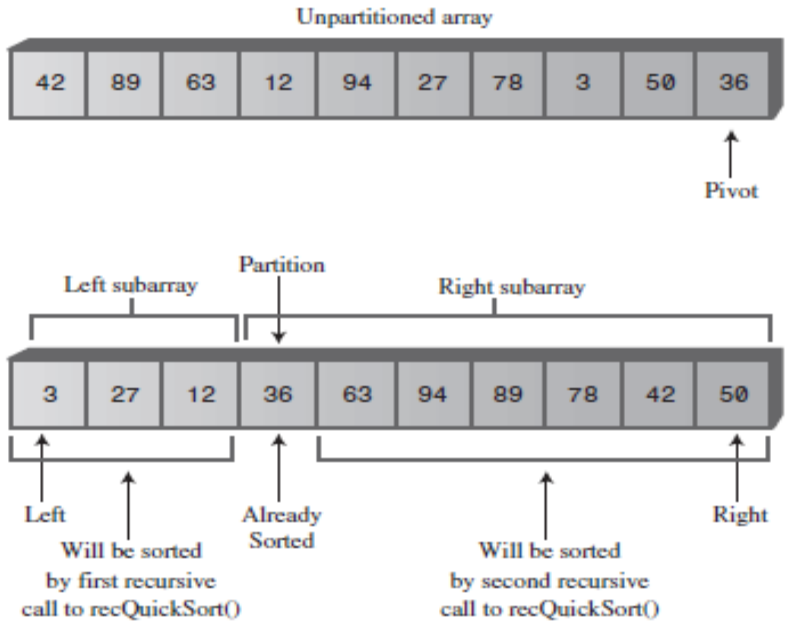


Swapping the pivot.

- Just takes one more line to our Python method
- Basically, a single call to swap()
  - Swaps A[end-1] (the pivot) with A[left] (the partition index)

Quick Sort

```
public void recQuickSort(int left, int right)
      {
      if(right-left <= 0)        // if size <= 1,
          return;                //    already sorted
      else                       // size is 2 or larger
         {
         long pivot = theArray[right];      // rightmost item
                                            // partition range
         int partition = partitionIt(left, right, pivot);
         recQuickSort(left, partition-1);   // sort left side
         recQuickSort(partition+1, right);  // sort right side
         }
      }  // end recQuickSort()
```



Recursive calls sort subarrays.

- Partition array/subarray into left (smaller keys) and right (larger keys) groups.
- Call ourselves to sort the left group.
- Call ourselves to sort the right group.

Shall We Try It On An ARRAY?

- 10 70 50 30 60 90 0 40 80 20
- Let’s go step-by-step on the board

BEST Case…

- We partition the array each time into two equal subarrays
- Say we start with array of size n = 2^i
- We recurse until the base case, 1 element

- Draw the tree
  - First call -> Partition n elements, n operations
  - Second calls -> Each partition n/2 elements, 2(n/2)=n operations
  - Third calls -> Each partition n/4, 4(n/4) = n operations
  - …
  - (i+1)th calls -> Each partition n/2^i = 1, 2^i(1) = n(1) = n ops
- Total: (i+1)*n = (log n + 1)*n -> O(n log n)

The Very BAD Case….

- If the array is sorted
- Let’s see the problem:
  - **0 10 20 30 40 50 60 70 80 90**
- What happens after the partition? This:
  - **0 10 20 30 40 50 60 70 80 90**
- This is sorted, but the algorithm doesn’t know it.
- It will then call itself on an array of zero size (the left subarray) and an array of n-1 size (the right subarray).
- Producing:
  - **0 10 20 30 40 50 60 70 80 90**

The Very BAD Case….

- In the worst case, we partition every time into an array of n-1 elements and an array of 0 elements
- This yields O(n^2) time:
  - First call: Partition n elements, n operations
  - Second calls: Partition n-1 and 0 elements, n-1 operations
  - Third calls: Partition n-2 and 0 elements, n-2 operations
  - Draw the tree

- Yielding: Operations = n + n-1 + n-2 + … + 1 = n(n+1)/2 -> O(n^2)

Summary

- What caused the problem was “blindly” choosing the pivot from the right end.
- In the case of a reverse sorted array, this is not a good choice at all

- Can we improve our choice of the pivot? Let’s choose the middle of three values

Median-Of-Three Partitioning

- Every time you partition, choose the median value of the left, center and right element as the pivot
- Example:
  - **44 11 55 33 77 22 00 99 101 66 88**
- Pivot: Take the median of the leftmost, middle and rightmost
  - **44 11 55 33 77 22 00 99 101 66 88 - Median: 44**
- Then partition around this pivot:
  - **11 00 33 22 44 77 55 99 101 66 88**
- Increases the likelihood of an equal partition
  - Also, it cannot possibly be the worst case

How This Fixes The WORST Case?

- Here's our array:
  - **0 10 20 30 40 50 60 70 80 90**

- Let's see on the board how this fixes things
- In fact in a perfectly sorted array, we choose the middle element as the pivot!
  - Which is optimal
  - We get $O(N\log N)$

- Vast majority of the time, if you use QuickSort with a Median-Of-Three partition, you get $O(N\log N)$ behavior

One Final Optimization…

- After a certain point, just doing insertion sort is faster than partitioning small arrays and making recursive calls

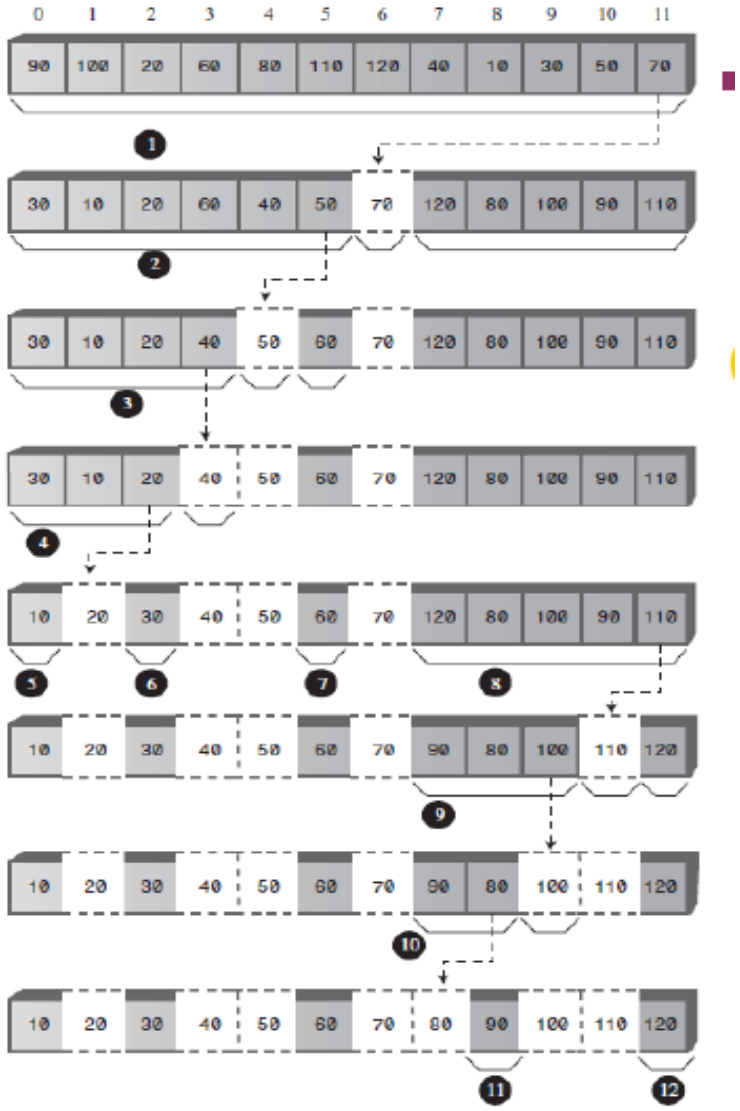
- Once you get to a very small subarray, you can just sort with insertion sort
- You can experiment a bit with ‘cutoff’ values
  - Knuth: n=9

Operation Count Estimates

- For QuickSort
- n=8: 30 comparisons, 12 swaps
- n=12: 50 comparisons, 21 swaps
- n=16: 72 comparisons, 32 swaps
- n=64: 396 comparisons, 192 swaps
- n=100: 678 comparisons, 332 swaps
- n=128: 910 comparisons, 448 swaps

- The only competitive algorithm is MergeSort
  - But, takes much more memory like we said

Summary of Quicksort



The quicksort process.

- Quick sort operates in $O(N*\log N)$ time (except when the simpler version is applied to already-sorted data).
- Subarrays smaller than a certain size (the cutoff) can be sorted by a method other than quicksort.
- The insertion sort is commonly used to sort subarrays smaller than the cutoff.
- The insertion sort can also be applied to the entire array, after it has been sorted down to a cutoff point by quicksort.

Swaps and Comparisons in Quicksort

N	8	12	16	64	100	128
$\log_2 N$	3	3.59	4	6	6.65	7
$N*\log_2 N$	24	43	64	384	665	896
Comparisons: $(N+2)*\log_2 N$	30	50	72	396	678	910
Swaps: fewer than $N/2*\log_2 N$	12	21	32	192	332	448

*The $\log_2 N$ quantity used in the table is true only in the best-case scenario, where each subarray is partitioned exactly in half. For random data, it is slightly greater.

The End

Median and Selection

Solving Recurrence Relations

- A **recurrence relation** expresses $T(n)$ in terms of $T(\text{less than } n)$
- For example, $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 11 \cdot n$
- Two methods of solution:
  1. Master theorem (aka, generalized “tree method”)
  2. Substitution method (aka, guess and check)

The Master Theorem

- Suppose $a \geq 1, b > 1,$ and d are constants (that don't depend on n).
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

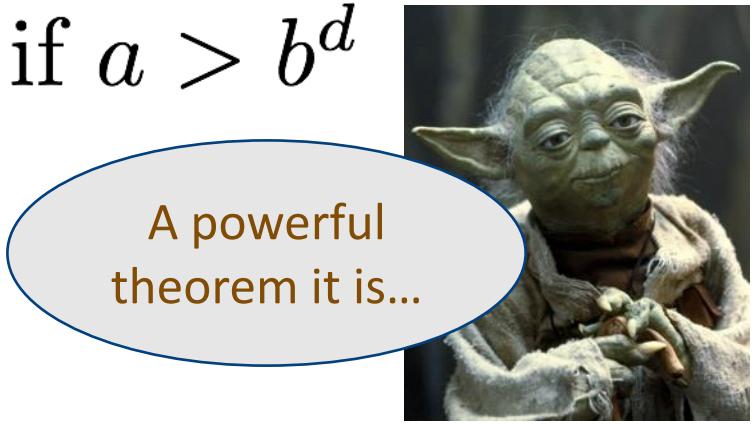
$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Three parameters:

a : number of subproblems

b : factor by which input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions.



The Substitution Method

- Step 1: Guess what the answer is.
- Step 2: Prove by induction that your guess is correct.
- Step 3: Profit.

The Plan for This Section

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
4. Return of the Substitution Method.

A Fun Recurrence Relation

- $T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$ for $n > 10$.
- Base case: $T(n) = 1$ when $1 \leq n \leq 10$

The Substitution Method

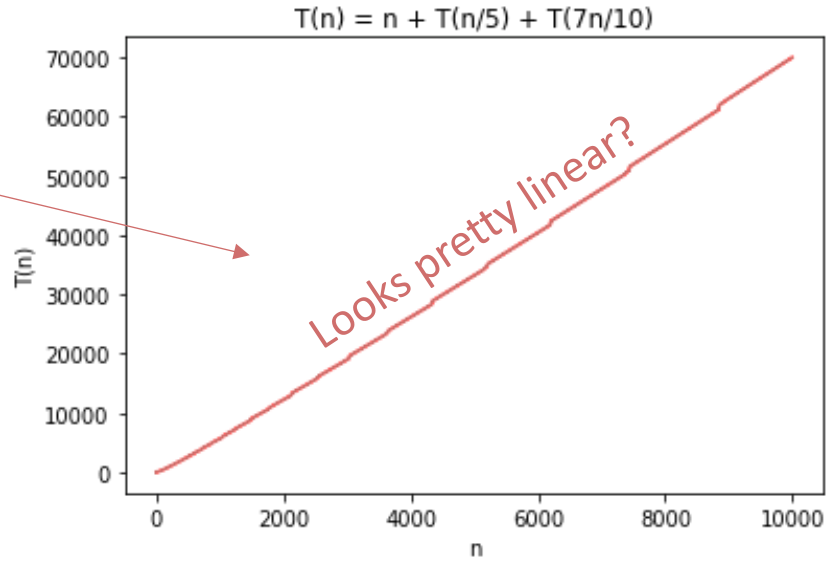
- Step 1: Guess what the answer is.
- Step 2: Prove by induction that your guess is correct.
- Step 3: Profit.

Step 1: Guess the Answer

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n \text{ for } n > 10.$$

Base case: $T(n) = 1$ when $1 \leq n \leq 10$

- Trying to work backwards gets gross fast…
- We can also just try it out.
  - (see Python notebook)
- Let's guess $O(n)$ and try to prove it.



Aside: Warning!

- It may be tempting to try to prove this with the inductive hypothesis "$T(n) = O(n)$"
- But that doesn't make sense!
- Formally, that's the same as saying:
  - Inductive Hypothesis for n:
  - There is some $n_0 > 0$ and some $C > 0$ so that, for all $n \geq n_0$, $T(n) \leq C \cdot n$.

The IH is supposed to hold for a *specific* n.

But now we are letting n be anything big enough!

- Instead, we should pick C first…

Step 2: Prove Our Guess Is Right

$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$ for $n > 10$.

Base case: $T(n) = 1$ when $1 \leq n \leq 10$

- Inductive Hypothesis: $T(n) \leq \boldsymbol{C}n$
- Base case: $1 = T(n) \leq \boldsymbol{C}n$ for all $1 \leq n \leq 10$
- Inductive step:
  - Let k > 10. Assume that the IH holds for all n so that $1 \leq n < k$.
  - $T(k) \leq k + T\left(\frac{k}{5}\right) + T\left(\frac{7k}{10}\right)$

    $\leq k + \boldsymbol{C} \cdot \left(\frac{k}{5}\right) + \boldsymbol{C} \cdot \left(\frac{7k}{10}\right)$

    $= k + \frac{\boldsymbol{C}}{5}k + \frac{7\boldsymbol{C}}{10}k$

    $\leq \boldsymbol{C}k$??
  - (aka, want to show that IH holds for n=k).
- Conclusion:
  - There is some $\boldsymbol{C}$ so that for all $n \geq 1$, $T(n) \leq \boldsymbol{C}n$
  - By the definition of big-Oh, T(n) = O(n).

We don't know what C should be yet! Let's go through the proof leaving it as "C" and then figure out what works…

Whatever we choose C to be, it should have C≥1

Let's solve for C and make this true! C = 10 works. *(write out)*

Step 3: Profit

$T(n) \leq n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$ for $n > 10$.

Base case: $T(n) = 1$ when $1 \leq n \leq 10$

Theorem: $T(n) = O(n)$

Proof:

- Inductive Hypothesis: $T(n) \leq \mathbf{10}n$.
- Base case: $1 = T(n) \leq \mathbf{10}n$ for all $1 \leq \mathrm{n} \leq 10$
- Inductive step:
  - Let k > 10. Assume that the IH holds for all n so that $1 \leq n < k$.
  - $T(k) \leq k + T\left(\frac{k}{5}\right) + T\left(\frac{7k}{10}\right)$
    $\leq k + \mathbf{10} \cdot \left(\frac{k}{5}\right) + \mathbf{10} \cdot \left(\frac{7k}{10}\right)$
    $= k + 2k + 7k = \mathbf{10}k$
  - Thus, IH holds for n=k.
- Conclusion:
  - For all $n \geq 1, T(n) \leq \mathbf{10}n$
  - Then, T(n) = O(n), using the definition of big-Oh with $n_0 = 1, c = 10$.

What Have We Learned?

- The substitution method can work when the master theorem doesn't.
  - For example, with different-sized sub-problems.

- Step 1: generate a guess
  - Throw the kitchen sink at it.
- Step 2: try to prove that your guess is correct
  - You may have to leave some constants unspecified till the end – then see what they need to be for the proof to work!!
- Step 3: profit
  - Pretend you didn't do Steps 1 and 2 and write down a nice proof.

The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
4. Return of the Substitution Method.

The k-SELECT Problem

A is an array of size n, k is in {1,…,n}

- SELECT(A, k):
  - Return the k-th smallest element of A.

For today, assume all arrays have distinct elements.

7	4	3	8	1	5	9	14

- SELECT(A, 1) = 1
- SELECT(A, 2) = 3
- SELECT(A, 3) = 4
- SELECT(A, 8) = 14

- SELECT(A, 1) = MIN(A)
- SELECT(A, n/2) = MEDIAN(A)
- SELECT(A, n) = MAX(A)

Being sloppy about floors and ceilings!

Note that the definition of Select is 1-indexed…

An O(nlog(n))-Time Algorithm

- SELECT(A, k):
  - A = MergeSort(A)
  - **return** A[k-1]

It's k-1 and not k since my pseudocode is 0-indexed and the problem is 1-indexed…

- Running time is O(n log(n)).
- So that's the benchmark….

Can we do better?

We're hoping to get O(n)

Show that you can't do better than O(n).

Goal: An O(n)-Time Algorithm

- On your pre-lecture exercise: SELECT(A, 1).
  - (aka, MIN(A))
- MIN(A):
  - ret = ∞
  - **For** i=0, ..., n-1:
    - If A[i] < ret:
      - ret = A[i]

    This stuff is O(1)

    This loop runs O(n) times
  - **Return** ret
- Time O(n). Yay!

How About SELECT(A,2)?

- SELECT2(A):
  - ret = ∞
  - minSoFar = ∞
  - **For** i=0, .., n-1:
    - **If** A[i] < ret and A[i] < minSoFar:
      - ret = minSoFar
      - minSoFar = A[i]
    - **Else** if A[i] < ret and A[i] >= minSoFar:
      - ret = A[i]
  - **Return** ret

(The actual algorithm here is not very important because this won't end up being a very good idea…)

Still O(n)

SO FAR SO GOOD.

SELECT(A, n/2) aka MEDIAN(A)?

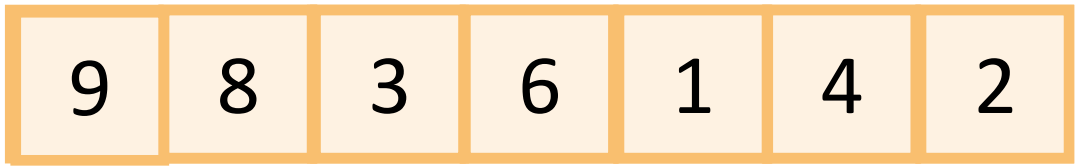
- MEDIAN(A):
  - ret = ∞
  - minSoFar = ∞
  - secondMinSoFar = ∞
  - thirdMinSoFar = ∞
  - fourthMinSoFar = ∞
  - ….
- This is not a good idea for large k (like n/2 or n).
- Basically, this is just going to turn into something like INSERTIONSORT…and that was $O(n^2)$.

The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
4. Return of the Substitution Method.

Idea: Divide and Conquer!

Say we want to find SELECT(A, k)



How about this pivot?

First, pick a "pivot." We'll see how to do this later.

Next, partition the array into "bigger than 6" or "less than 6"

This PARTITION step takes time O(n). (Notice that we don't sort each half).

L = array with things smaller than A[pivot]

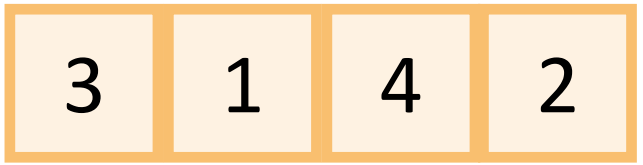
R = array with things larger than A[pivot]

Idea: Divide and Conquer!

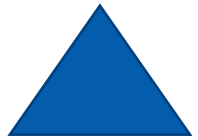
Say we want to find SELECT(A, k)

First, pick a "pivot." We'll see how to do this later.

Next, partition the array into "bigger than 6" or "less than 6"

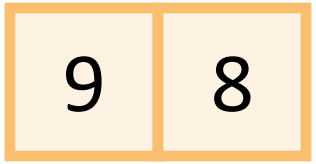


L = array with things smaller than A[pivot]



How about this pivot?

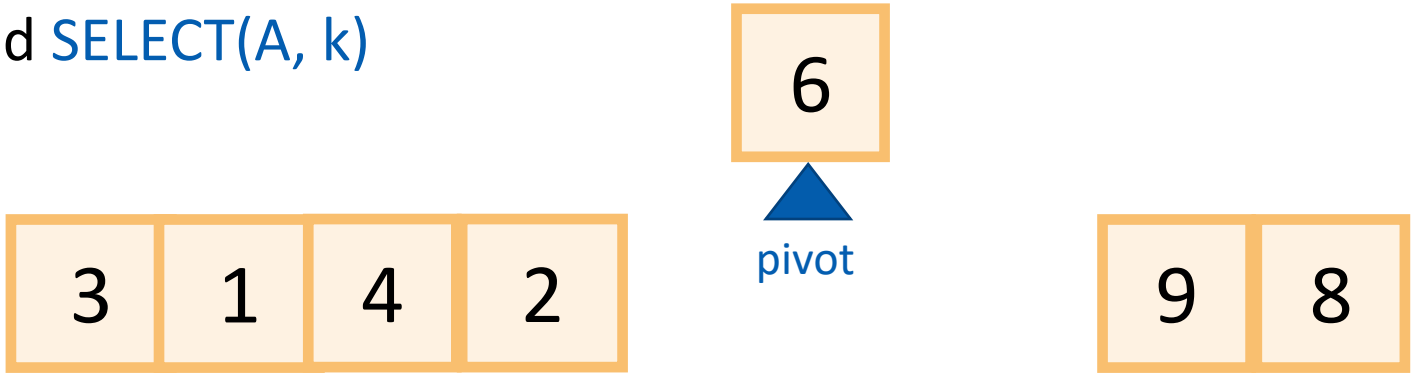
This PARTITION step takes time O(n). (Notice that we don't sort each half).



R = array with things larger than A[pivot]

Idea Continued…

Say we want to find SELECT(A, k)



L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

- If k = 5 = len(L) + 1:
  - We should return A[pivot]
- If k < 5:
  - We should return SELECT(L, k)
- If k > 5:
  - We should return SELECT(R, k – 5)

This suggests a recursive algorithm

(still need to figure out how to pick the pivot…)

Pseudocode

- **Select**(A,k):
  - **If** len(A) <= 50:
    - A = **MergeSort**(A)
    - **Return** A[k-1]
  - p = **getPivot**(A)
  - L, pivotVal, R = **Partition**(A,p)
  - **if** len(L) == k-1:
    - return pivotVal
  - **Else if** len(L) > k-1:
    - return **Select**(L, k)
  - **Else if** len(L) < k-1:
    - return **Select**(R, k – len(L) – 1)

- **getPivot**(A) returns some pivot for us.
  - How?? We'll see later…
- **Partition**(A,p) splits up A into L, A[p], R.
  - See Lecture 4 Python notebook for code

Base Case: If len(A) = O(1), then any sorting algorithm runs in time O(1).

Case 1: We got lucky and found exactly the k'th smallest value!

Case 2: The k'th smallest value is in the first part of the list

Case 3: The k'th smallest value is in the second part of the list

Does It Work?

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- Check out the Python notebook for Lecture 4, which implements this with a bunch of different pivot-selection methods.
  - Seems to work!

- Check out the lecture notes for a rigorous proof based on induction that this works, with any pivot-choosing mechanism.
  - It provably works!
  - Also, this is a good example of proving that a recursive algorithm is correct.

What Is the Running Time?

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

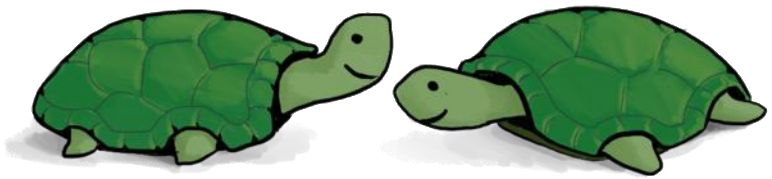
Assuming we pick the pivot in time O(n)…

- $$T(n) = \begin{cases} T(\text{len(L)}) + O(n) & \text{len(L)} > k - 1 \\ T(\text{len(R)}) + O(n) & \text{len(L)} < k - 1 \\ O(n) & \text{len(L)} = k - 1 \end{cases}$$

- What are len(L) and len(R)?
- That depends on how we pick the pivot…

Think: one minute
Share: (wait) one minute

What would be a “good” pivot?
What would be a “bad” pivot?



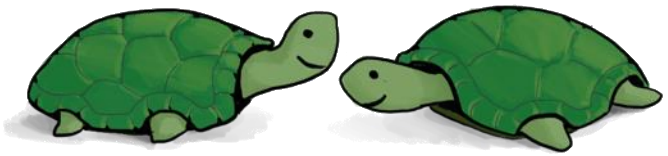
Think-Share Terrapins

The best way would be to always pick the pivot so that len(L) = k-1. But say we don’t have control over k, just over how we pick the pivot.

The Ideal Pivot

- We split the input exactly in half:
  - len(L) = len(R) = (n-1)/2

What happens in that case?



Think: one minute
Share: (wait) one minute

In case it's helpful…

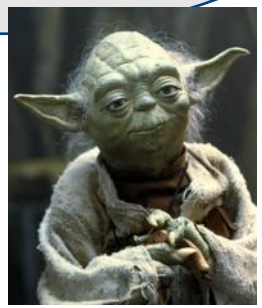
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

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

The Ideal Pivot

- We split the input exactly in half:
  - len(L) = len(R) = (n-1)/2

Apply here, the Master Theorem does NOT. Making unsubstantiated assumptions about problem sizes, we are.



Jedi master Yoda

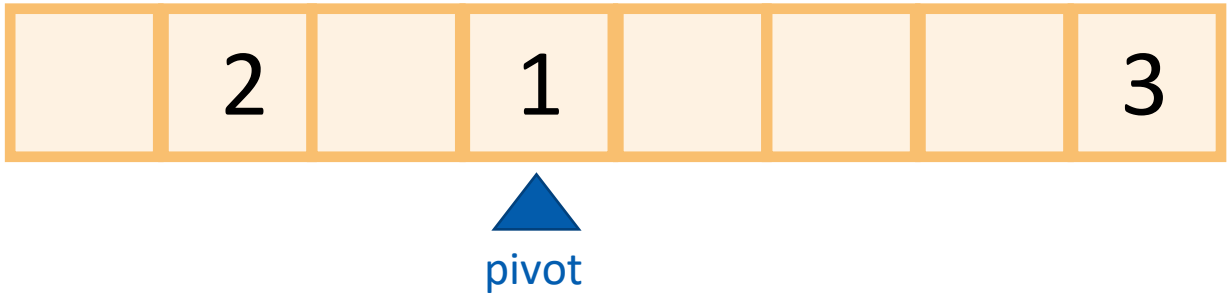
- Let's pretend that's the case and use the **Master Theorem**!
  - $T(n) \leq T\left(\frac{n}{2}\right) + O(n)$
  - So a = 1, b = 2, d = 1
  - $T(n) \leq O(n^d) = O(n)$

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

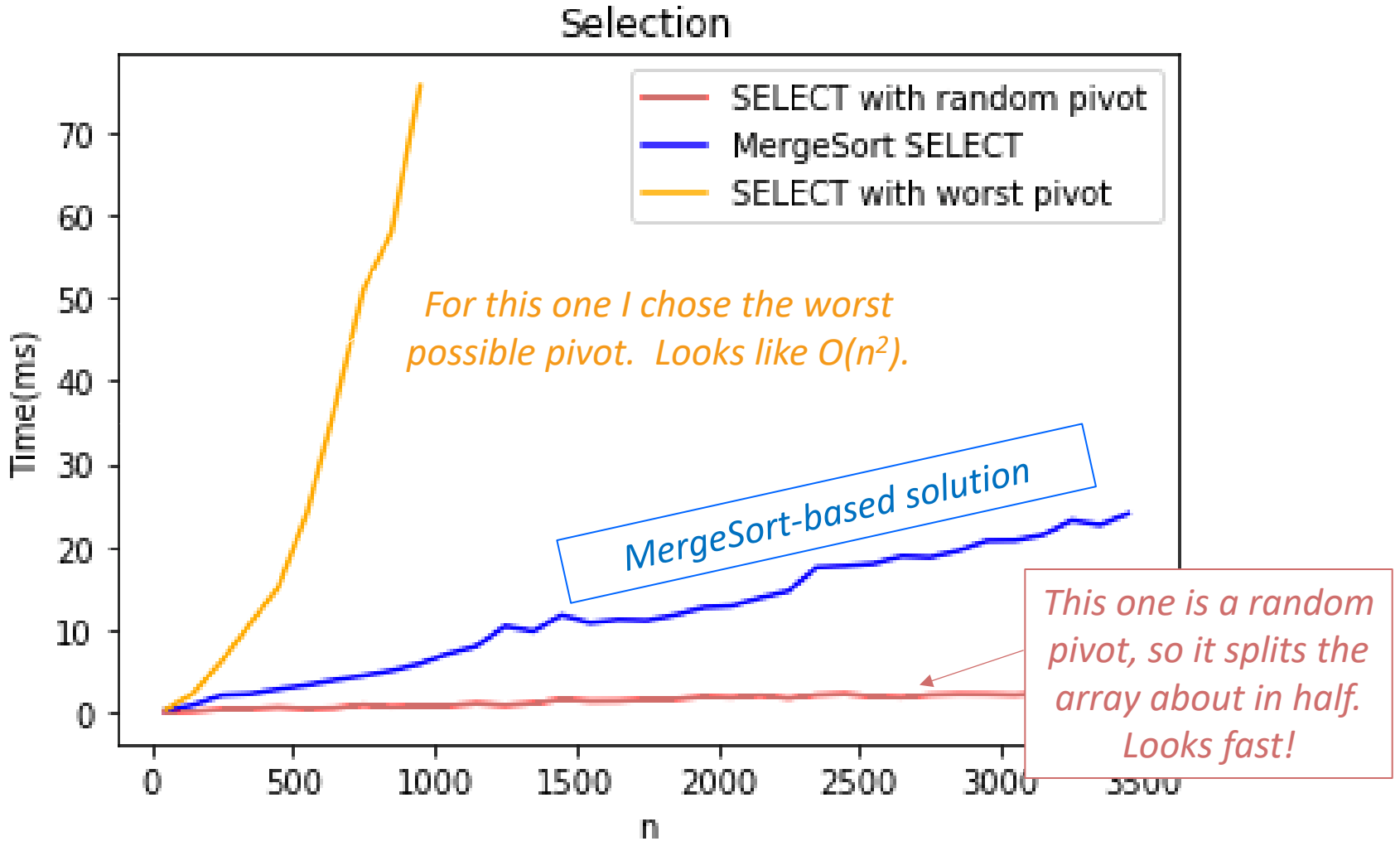
That would be great!

The Worst Pivot

- Say our choice of pivot doesn't depend on A.
- A bad guy who **knows what pivots we will choose** gets to come up with A.



The Distinction Matters!



How Do We Pick a Good Pivot?

- Randomly?
  - That works well if there's no bad guy.
  - But if there is a bad guy who gets to see our pivot choices, that's just as bad as the worst-case pivot.

Aside:

- In practice, there is often no bad guy. In that case, just pick a random pivot and it works really well!
- (More on this next lecture)

How Do We Pick a Good Pivot?

- For today, let's assume there's this bad guy.
- Reasons:
  - This gives us a very strong guarantee
  - We'll get to see a really clever algorithm.
    - Necessarily it will look at A to pick the pivot.
  - We'll get to use the substitution method.

The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
   a) The outline of the algorithm.
   b) How to pick the pivot.
4. Return of the Substitution Method.

Approach

- First, we'll figure out what the ideal pivot would be.
  - But we won't be able to get it.
- Then, we'll figure out what a **pretty good** pivot would be.
  - But we still won't know how to get it.
- Finally, we will see how to get our pretty good pivot!
  - And then we will celebrate.

How Do We Pick Our Ideal Pivot?

- We'd like to live in the ideal world.

- Pick the pivot to divide the input in half.
- Aka, pick the median!
- Aka, pick SELECT(A, n/2)!

How About a Good Enough Pivot?

- We'd like to approximate the ideal world.

- Pick the pivot to divide the input about in half!
- Maybe this is easier!

A Good Enough Pivot

- We split the input not quite in half:
  - 3n/10 < len(L) < 7n/10
  - 3n/10 < len(R) < 7n/10

Lucky the lackadaisical lemur

- If we could do that (let's say, in time O(n)), the **Master Theorem** would say:

- $T(n) \leq T\left(\frac{7n}{10}\right) + O(n)$
- So a = 1, b = 10/7, d = 1
- $T(n) \leq O(n^d) = O(n)$

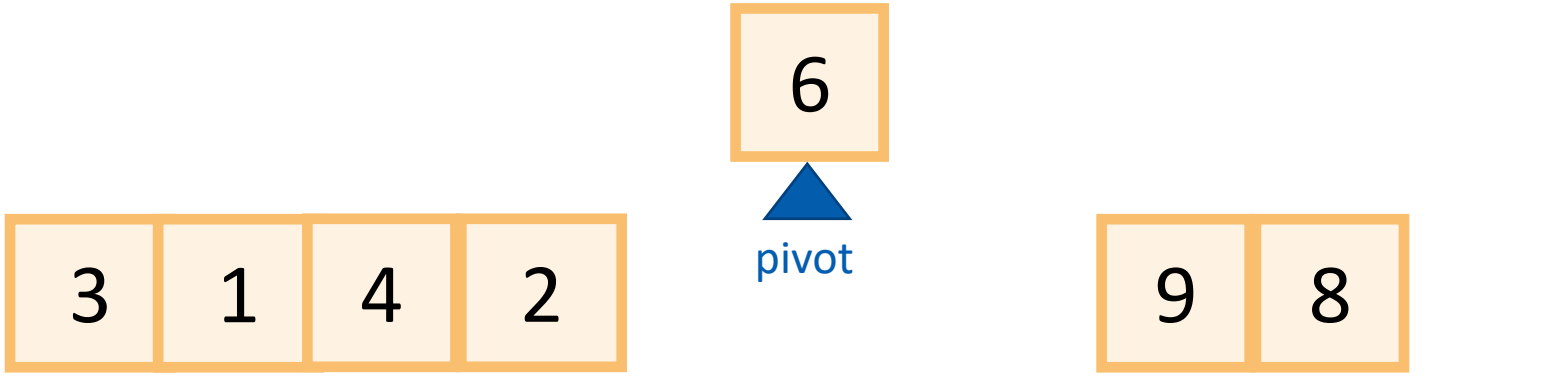
STILL GOOD!

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

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

Goal

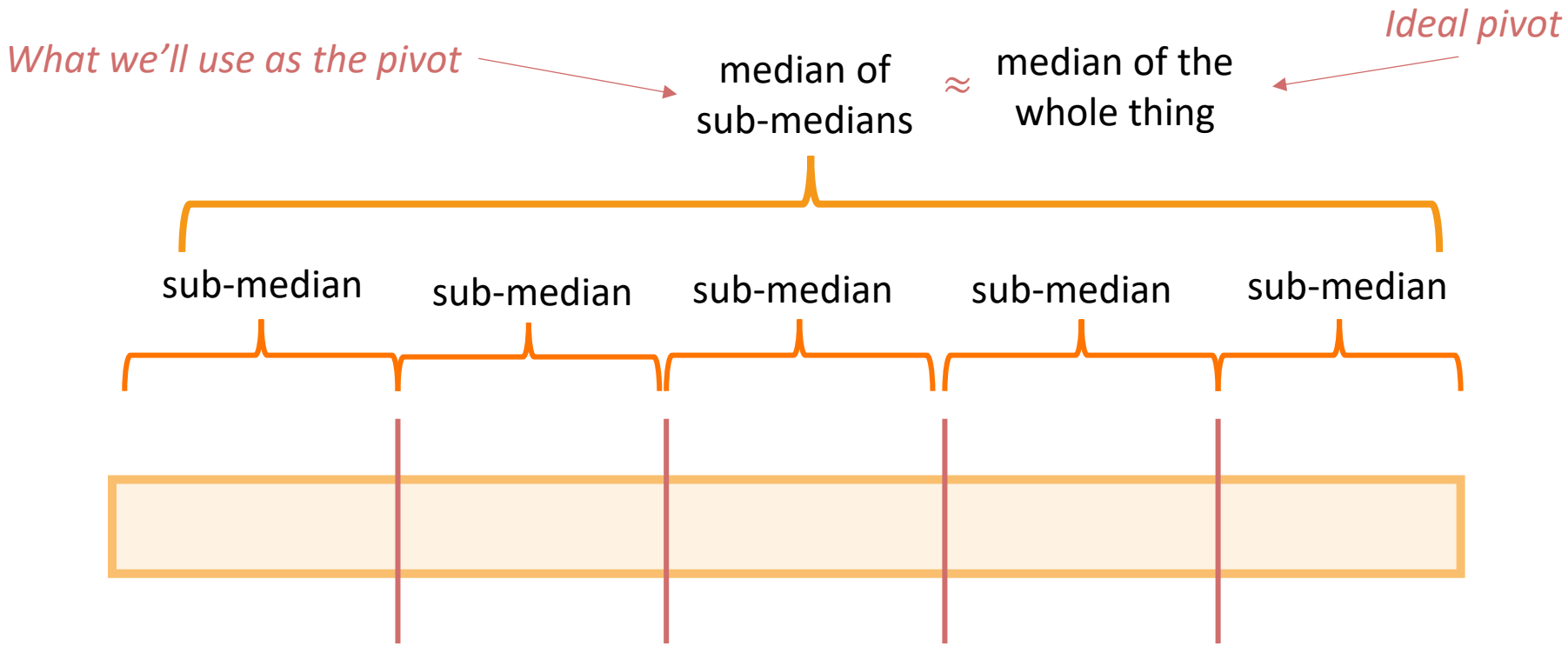
- Efficiently pick the pivot so that



$$\frac{3n}{10} < \text{len}(L) < \frac{7n}{10} \qquad \frac{3n}{10} < \text{len}(R) < \frac{7n}{10}$$

Another Divide-and-Conquer Algorithm!

- We can't solve SELECT(A,n/2) (yet)
- But we can divide and conquer and solve SELECT(B,m/2) for smaller values of m (where len(B) = m).
- **Lemma*:** The median of sub-medians is close to the median.

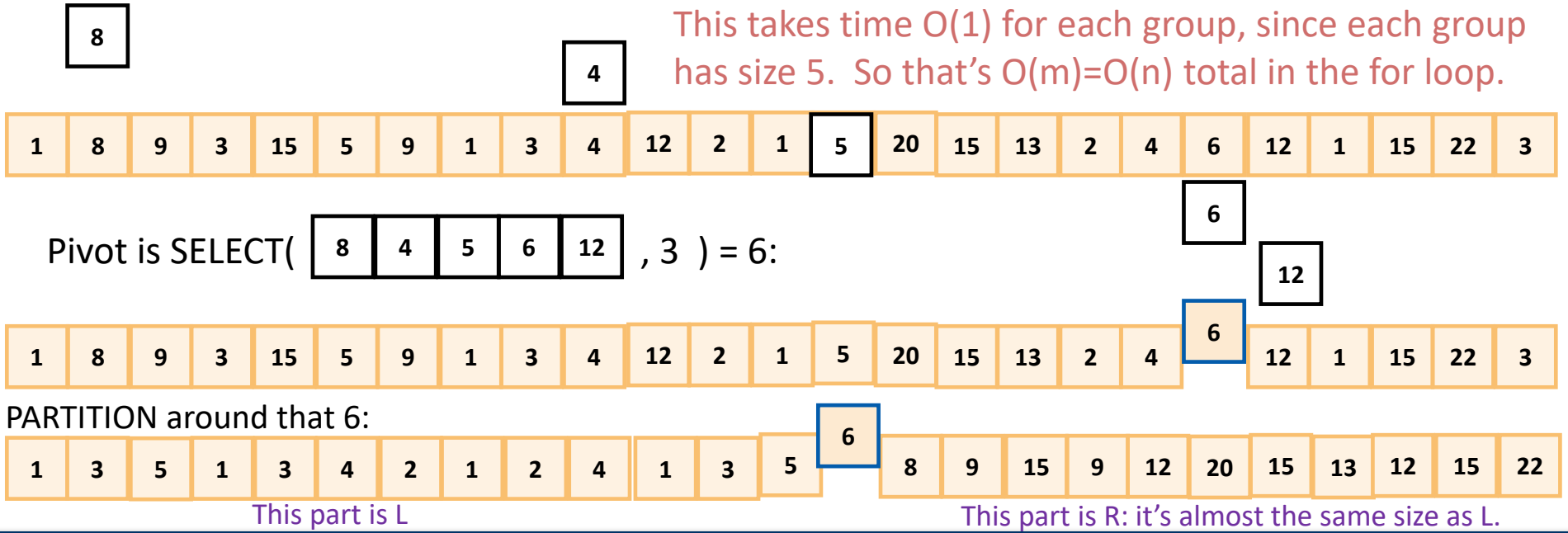


*we will make this a bit more precise.

How to Pick the Pivot

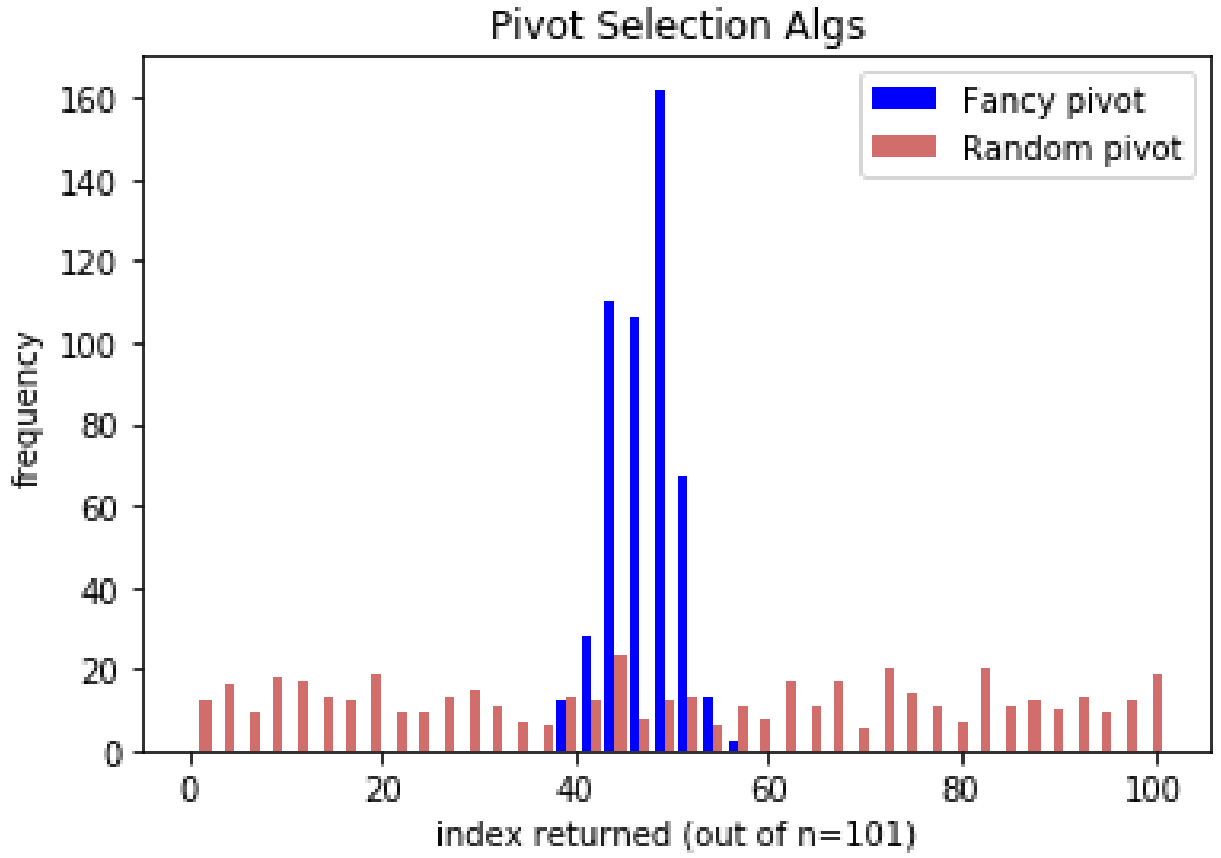
- CHOOSEPIVOT(A):
  - Split A into m = $\lceil \frac{n}{5} \rceil$ groups, of size <=5 each.
  - **For** i=1, .., m:
    - Find the median within the i'th group, call it p_i
  - p = SELECT([$p_1, p_2, p_3, \ldots, p_m$] , m/2)
  - **return** the index of p in A

This takes time O(1) for each group, since each group has size 5. So that's O(m)=O(n) total in the for loop.



CLAIM: This Works

Divides the Array *Approximately* in Half



CLAIM: This Works

Divides the Array *Approximately* in Half

- Formally, we will prove (later):

Lemma: If we choose the pivots like this, then

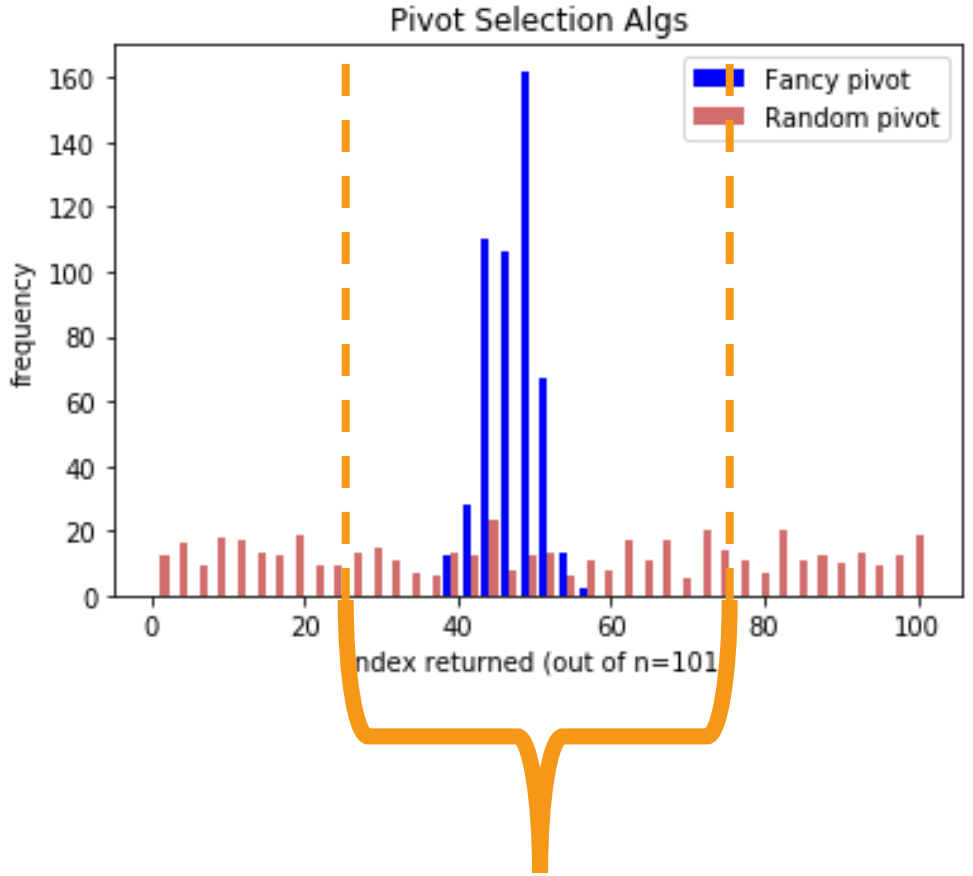
$$|L| \leq \frac{7n}{10} + 5$$

and

$$|R| \leq \frac{7n}{10} + 5$$

Sanity Check

$$|L| \leq \frac{7n}{10} + 5 \text{ and } |R| \leq \frac{7n}{10} + 5$$



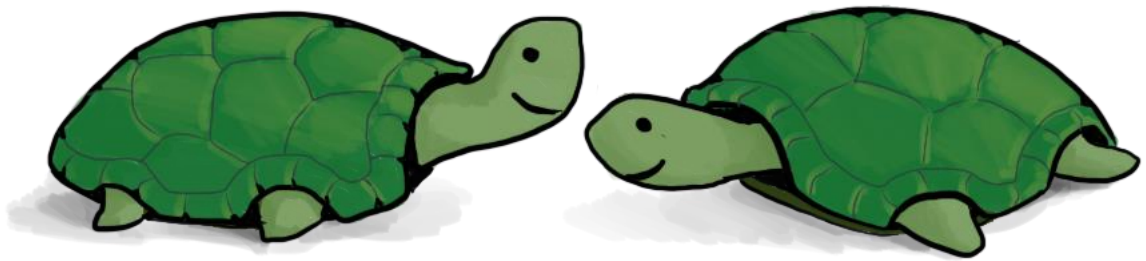
That's this window

How About the Running Time?

- Suppose the Lemma is true. (It is).
  - $|L| \leq \frac{7n}{10} + 5$ and $|R| \leq \frac{7n}{10} + 5$

- Recurrence relation:

$$T(n) \leq ?$$



Pseudocode

- **Select**(A,k):
  - **If** len(A) <= 50:
    - A = **MergeSort**(A)
    - **Return** A[k-1]
  - p = **choosePivot**(A)
  - L, pivotVal, R = **Partition**(A,p)
  - **if** len(L) == k-1:
    - return pivotVal
  - **Else if** len(L) > k-1:
    - return **Select**(L, k)
  - **Else if** len(L) < k-1:
    - return **Select**(R, k – len(L) – 1)

- Lemma says that $|L| \leq \frac{7n}{10} + 5$ and $|R| \leq \frac{7n}{10} + 5$
- Suppose **Partition** runs in time O(n)
- Come up with a recurrence relation for T(n), the running time of **Select**, using the **choosePivot** algorithm we just described.

Base Case: If len(A) = O(1), then any sorting algorithm runs in time O(1).

Case 1: We got lucky and found exactly the k'th smallest value!

Case 2: The k'th smallest value is in the first part of the list

Case 3: The k'th smallest value is in the second part of the list

How About the Running Time?

- Suppose the Lemma is true. (It is).
  - $|L| \leq \frac{7n}{10} + 5$ and $|R| \leq \frac{7n}{10} + 5$

- Recurrence relation:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

The call to CHOOSEPIVOT makes one further recursive call to SELECT on an array of size n/5.

Outside of CHOOSEPIVOT, there's at most one recursive call to SELECT on array of size 7n/10 + 5.

We're going to drop the "+5" for convenience, but it does not change the final answer. Why?
Hint: Define T'(n) := T(n+1000) and write recurrence for T'

Siggi the Studious Stork

The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
   a) The outline of the algorithm.
   b) How to pick the pivot.
4. Return of the Substitution Method.

This Sounds Like a Job For…

Step 1: generate a guess
Step 2: try to prove that your guess is correct
Step 3: profit

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

That's convenient! We did this at the beginning of lecture!

Conclusion: $T(n) = O(n)$

Technically we only did it for $T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$, not when the last term has a big-Oh…

Plucky the Pedantic Penguin

Recap of Approach

- First, we figured out what the ideal pivot would be.
  - Find the median
- Then, we figured out what a **pretty good** pivot would be.
  - An approximate median
- Finally, we saw how to get our pretty good pivot!
  - Median of medians and divide and conquer!
  - Hooray!

In Practice?

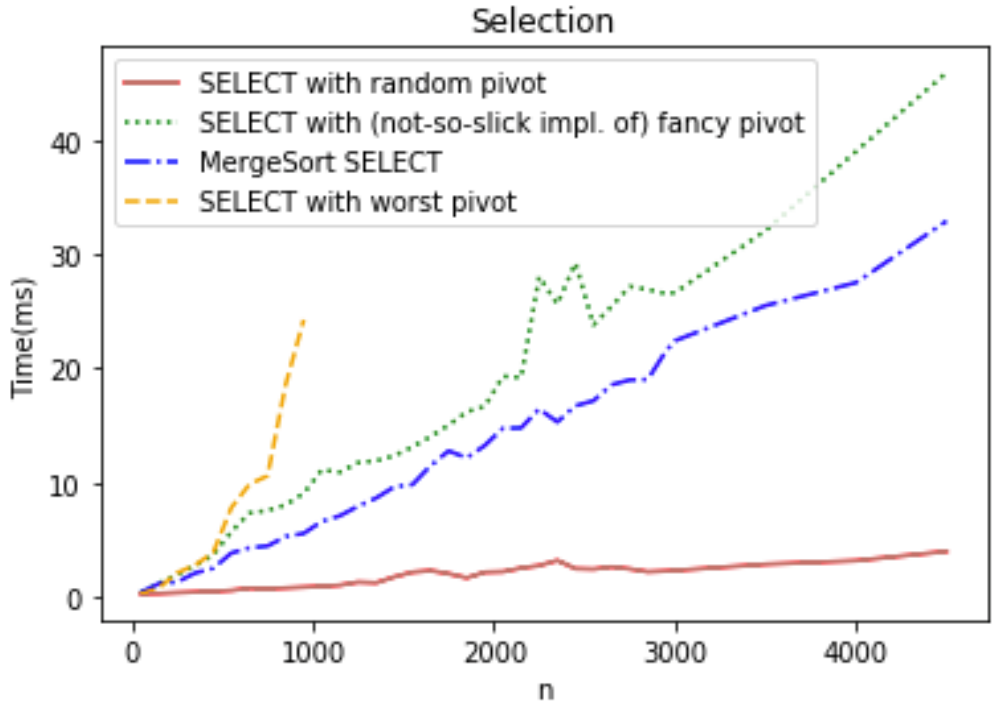
- With not-very-slick implementation, our fancy version of SELECT is worse than the MergeSort-based SELECT ☹
  - But O(n) is better than O(n log(n))! How can that be?
  - *What's the constant in front of the n in our proof? 20? 30?*

- On **non-adversarial** inputs, random pivot choice is much better.

Moral:
Just pick a random pivot if you don't expect nefarious arrays.

Optimize the implementation of SELECT (with the fancy pivot). Can you beat MergeSort?

Siggi the Studious Stork



What Have We Learned?

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Pending the Lemma

- It is possible to solve SELECT in time O(n).
  - Divide and conquer!

- If you want a deterministic algorithm or expect that a bad guy will be picking the list, **choose a pivot cleverly.**
  - More divide and conquer!

- If you don't expect that a bad guy will be picking the list, in practice it's better just to **pick a random pivot.**