Divide-and-Conquer

- Merge Sort
- Divide-and-Conquer
- Master Theorem
- Quick Sort

Yanlin Zhang & Wei Wang | DSAA 2043 Spring 2025

Merge Sort

Merge Sort

```
MERGE-SORT $A[1 \mathinner{.\,.} n]$
    1. If $n = 1$, done
    2. Recursively sort $A[1 \mathinner{.\,.} \lceil n/2 \rceil]$
       and $A[\lceil n/2 \rceil + 1 \mathinner{.\,.} n]$
    3. "Merge" the $2$ sorted lists
```

Key subroutine: **MERGE**

Merging Two Sorted Arrays

20	12
13	11
7	9
2	1

Analyzing Merge Sort

$T(n)$	**MERGE-SORT** $A[1\,.\,.\,n]$
$\Theta(1)$	1. If $n = 1$, done
$2T(n/2)$	2. Recursively sort $A[\,1\,.\,.\lceil n/2\rceil\,]$ and $A[\,\lceil n/2\rceil+1\,.\,.\,n\,]$
$\Theta(n)$	***3. "Merge"*** the 2 sorted lists

Abuse

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 provides 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.

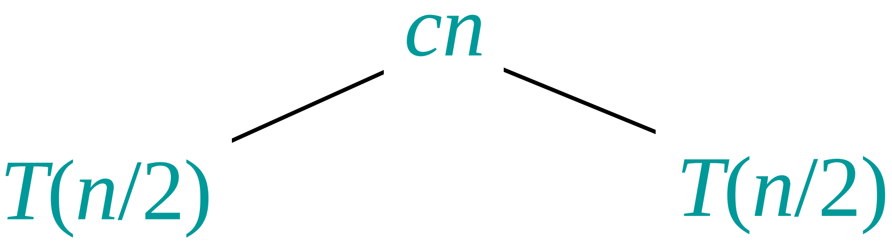
Recursion Tree

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.

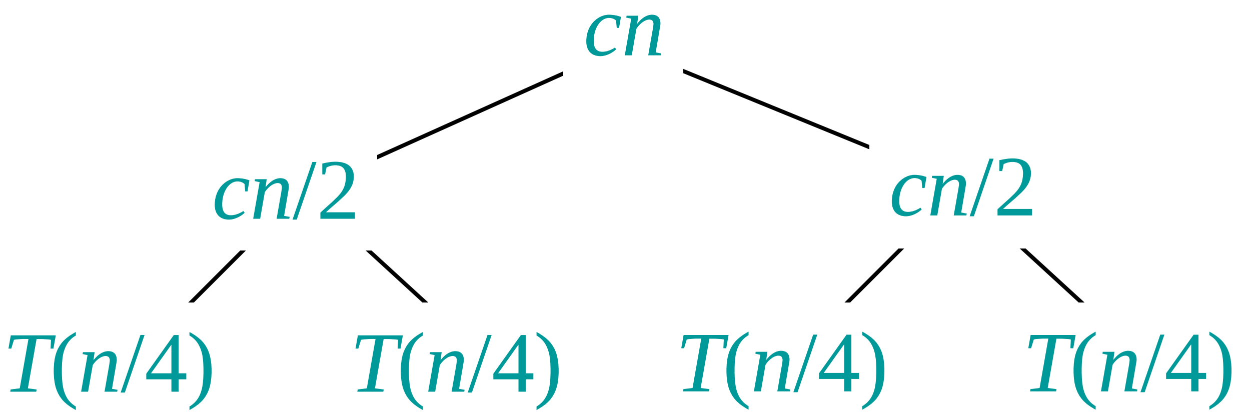


cn

$T(n/2)$ $T(n/2)$

Recursion Tree

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



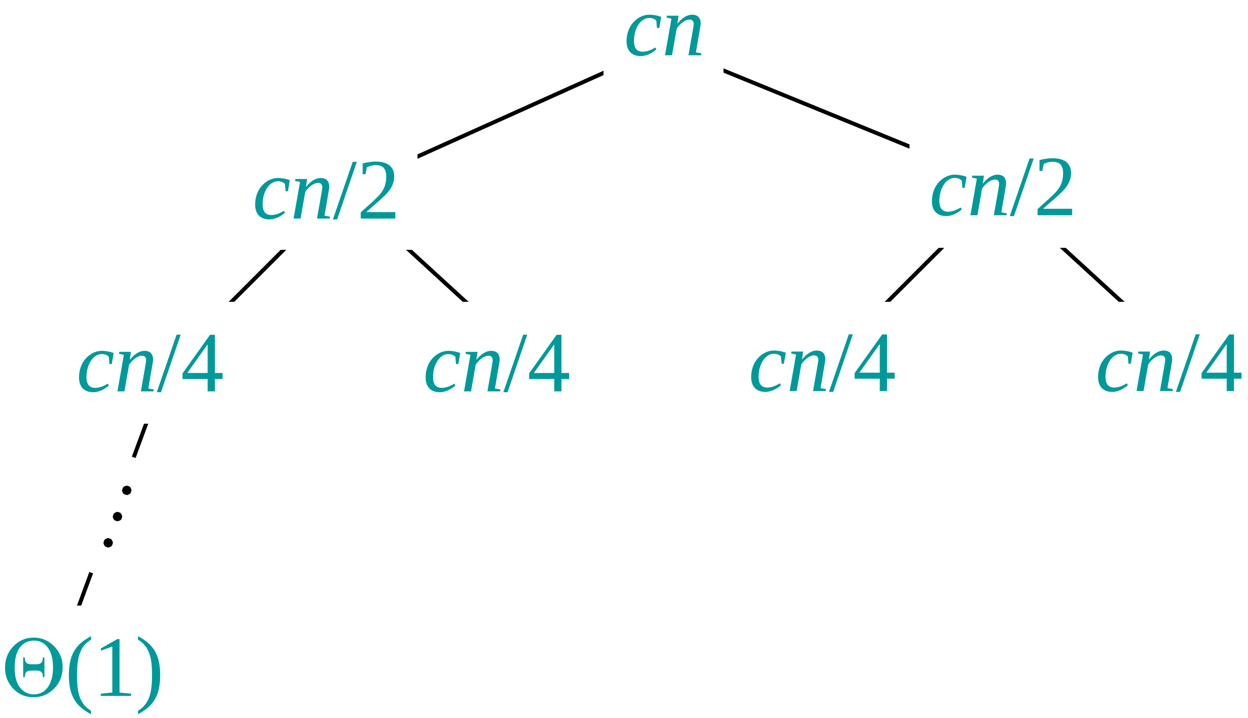
cn

$cn/2$ $cn/2$

$T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$

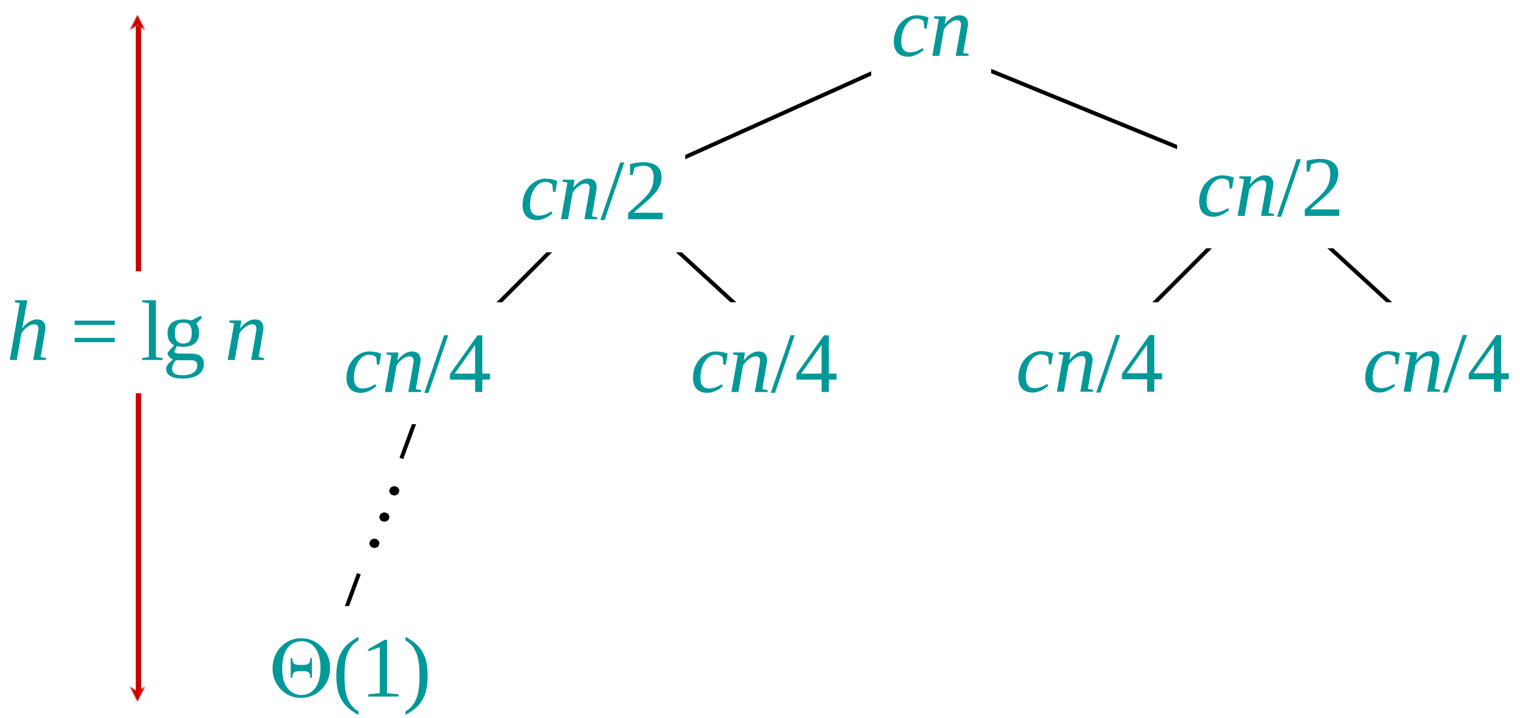
Recursion Tree

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



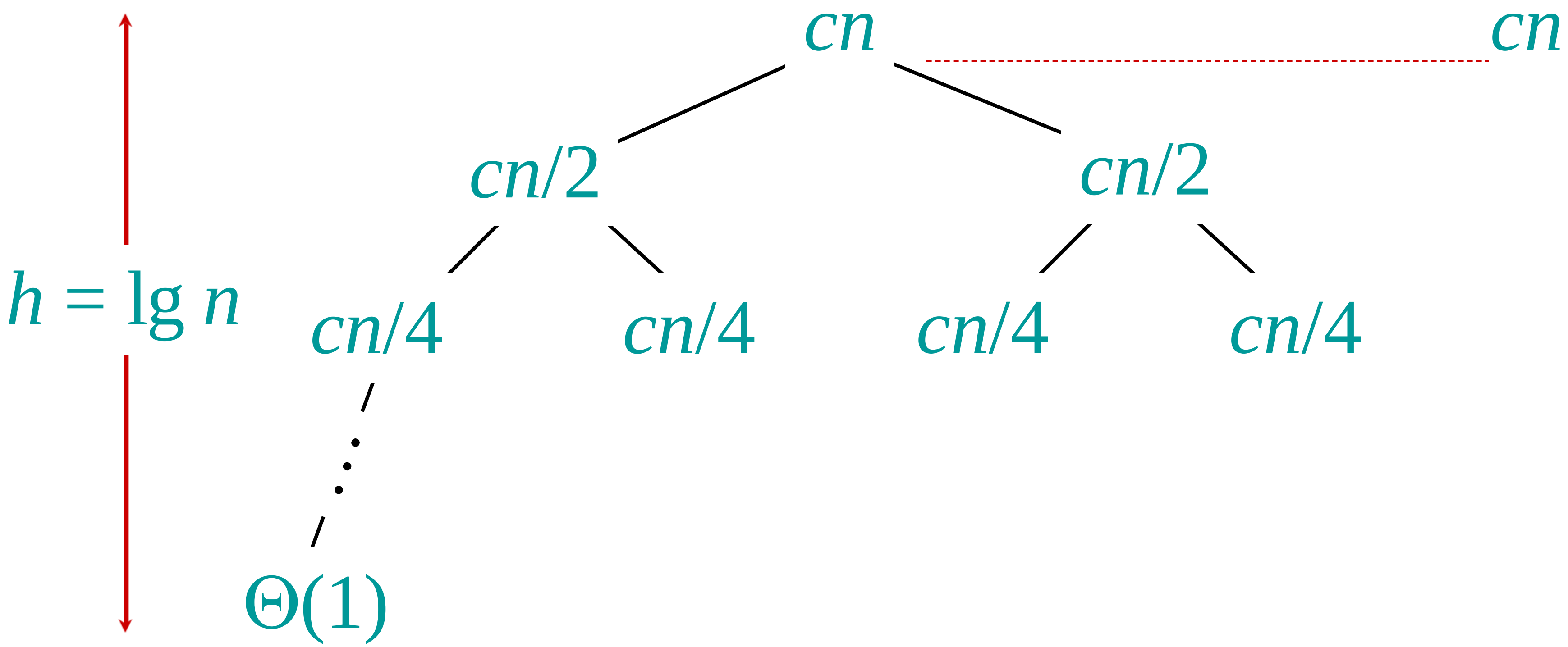
Recursion Tree

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



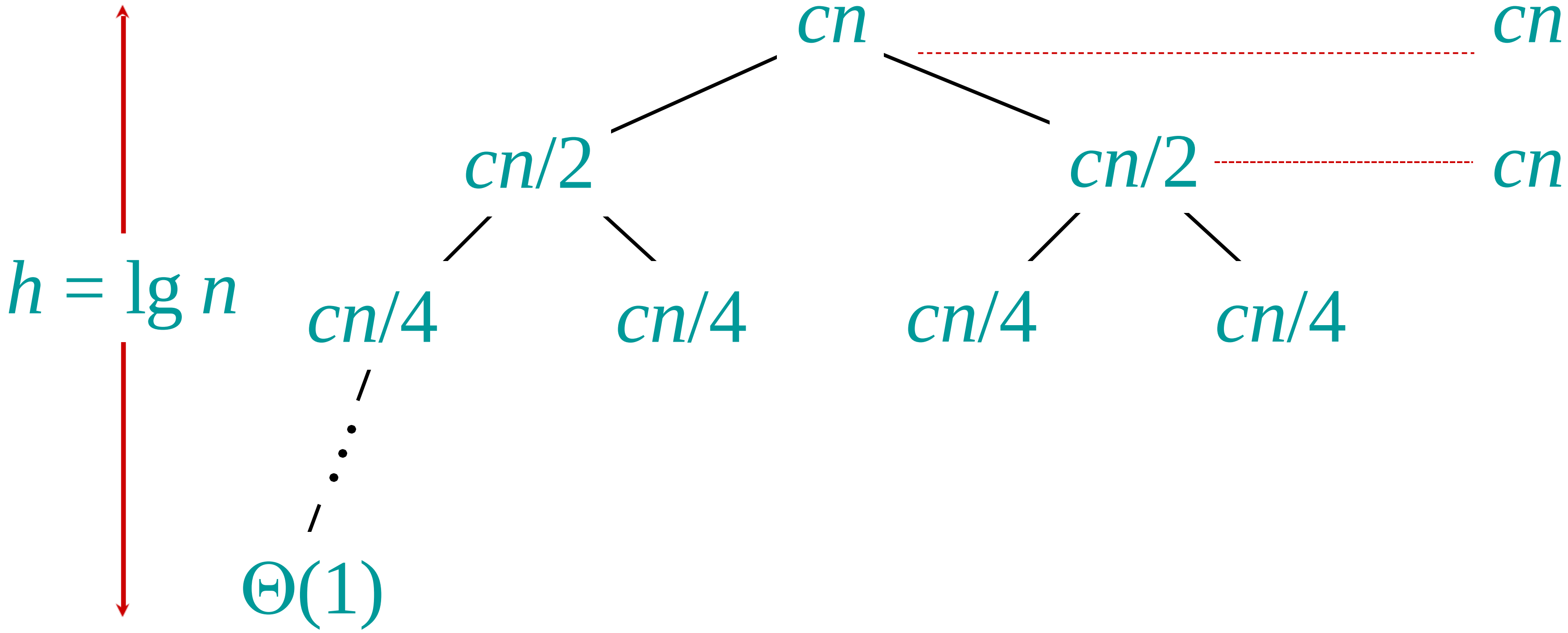
Recursion Tree

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



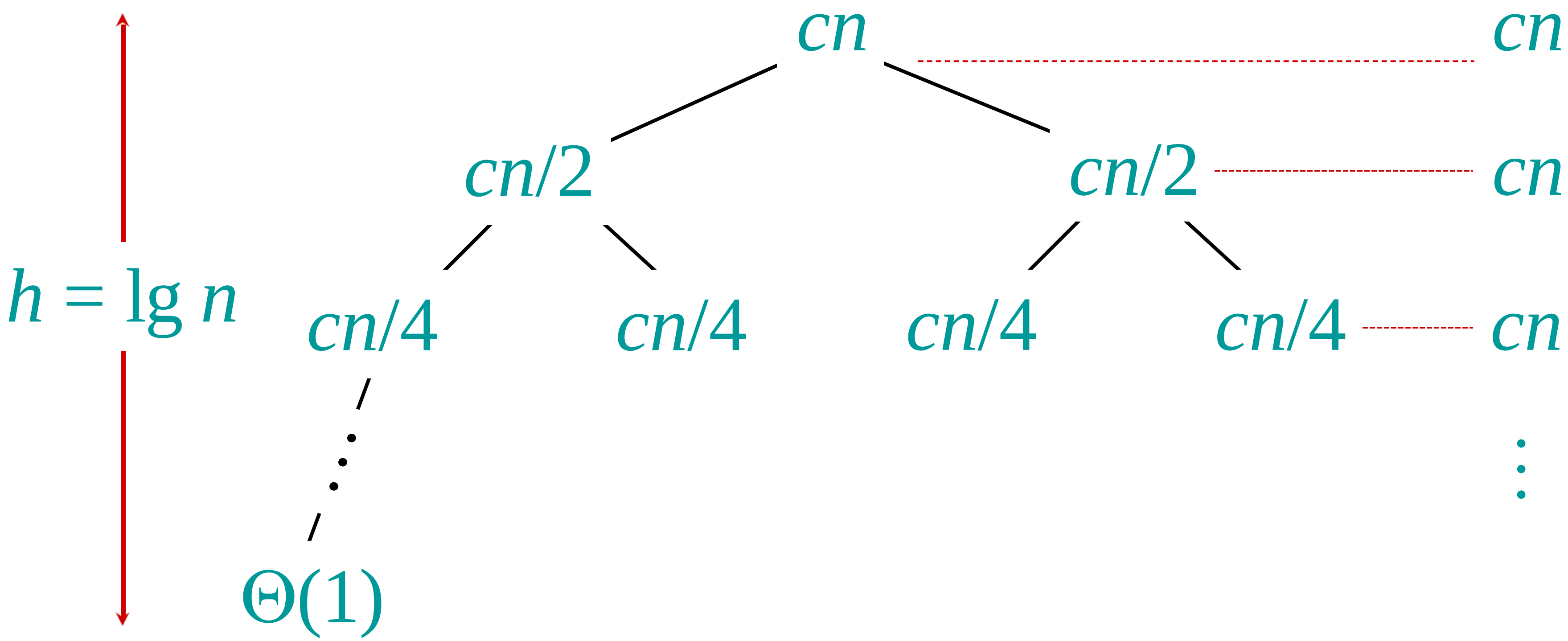
Recursion Tree

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



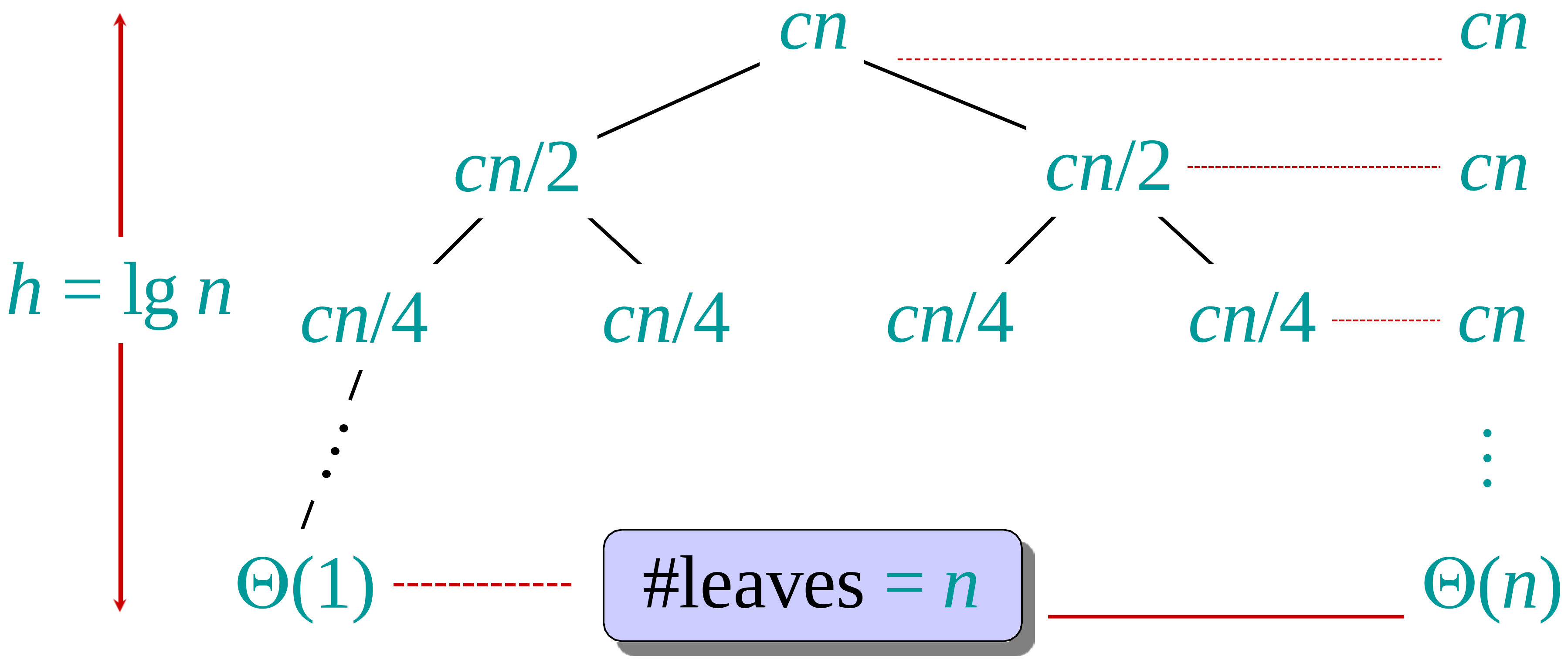
Recursion Tree

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



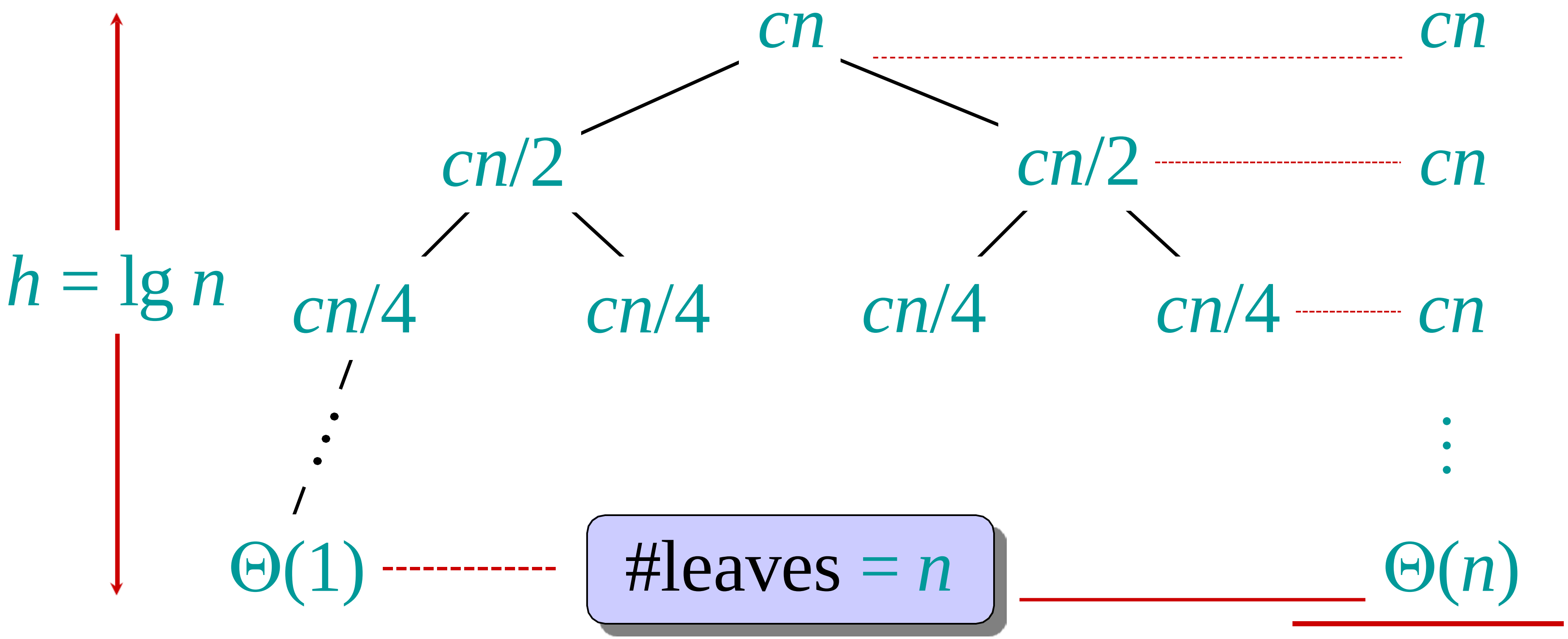
Recursion Tree

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.



Total $= \Theta(n \lg n)$

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.

$$T(n) = \underbrace{2}_{\#\ \text{subproblems}}\ T(\underbrace{n/2}_{\text{subproblem size}}) + \underbrace{\Theta(n)}_{\text{work dividing and combining}}$$

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

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

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$, constant $\varepsilon > 0$, and regularity condition $a\,f(n/b) \leq c\,f(n)$, constant $c < 1$ for all sufficiently large n

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

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

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

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$, constant $\varepsilon > 0$, and regularity condition $a\,f(n/b) \leq c\,f(n)$, constant $c < 1$ for all sufficiently large n

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

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

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

Master Theorem (Proof)

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

Try to solve case 2 in lab!

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

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

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

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

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

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

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

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

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

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

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

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

$$\vdots$$

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

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

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

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

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

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

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

3 5 7 8 9 12 15

Recurrence for Binary Search

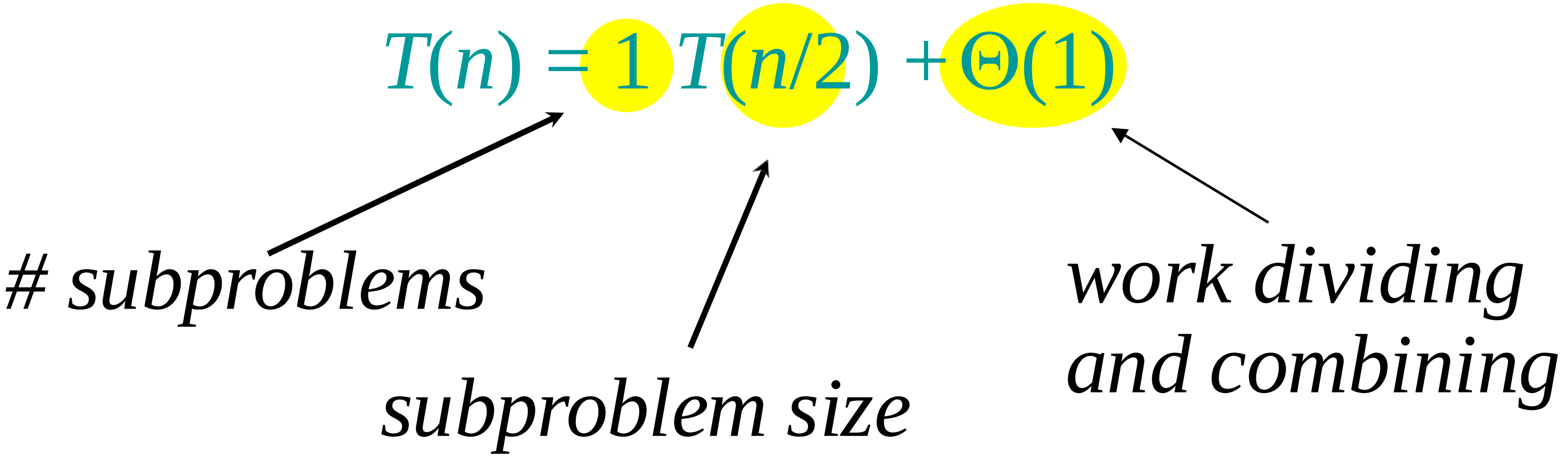
$$T(n) = 1\,T(n/2) + \Theta(1)$$

 # subproblems

subproblem size

work dividing and combining

Recurrence for Binary Search



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

subproblems

subproblem size

work dividing
and combining

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \Rightarrow \textsc{Case}\ 2\ (l = 0)$$

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

Powering a Number

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

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 $n \in \mathbf{N}$.

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

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:

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

$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \ldots$

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

Computing Fibonacci Numbers

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

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

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

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

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

$$\left.\begin{matrix} \\ \\ \end{matrix}\right\} \quad i, j = 1, 2, \ldots, n.$$

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

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

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

IDEA:

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

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

$$\underset{C}{} \quad = \quad \underset{A}{} \qquad\qquad \underset{B}{}$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned}\right\}$$

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

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

Divide-and-Conquer Algorithm

IDEA:

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

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

$$C \quad = \quad A \qquad\qquad B$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dh \\ u &= cf + dg \end{aligned}\right\}$$

recursive

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

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

Analysis of D&C Algorithm

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

submatrices

submatrix size

work adding submatrices

Analysis of D&C Algorithm

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

 # submatrices

submatrix size

work adding submatrices

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

Analysis of D&C Algorithm

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

 # submatrices

submatrix size

work adding submatrices

$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow$ 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.

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

Strassen’s Idea

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

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

$$\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
s &= P_1 + P_2 \\
t &= P_3 + P_4 \\
u &= P_5 + P_1 - P_3 - P_7
\end{aligned}$$

Strassen's Idea

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

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

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
s &= P_1 + P_2 \\
t &= P_3 + P_4 \\
u &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

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

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

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

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

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

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

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

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): $\Theta(n^{2.376\ldots})$.

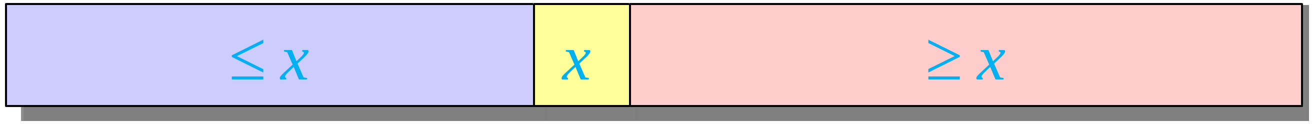
Conclusion

- 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, in $O(n \log n)$ time (for in-memory sorting)
- Basic scheme
  - Divide: partition an array into two subarrays around a pivot x such that elements in left subarray $\leq x \leq$ the elements



  - Conquer: recursively to quicksort each of these subarrays
  - Combine: trivial
- Some embellishments we can make
  - selection of the pivot
  - sorting of small partitions

Quick Sort (Pseudo-Code)

```
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

- 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

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 (two indexs)
  - 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

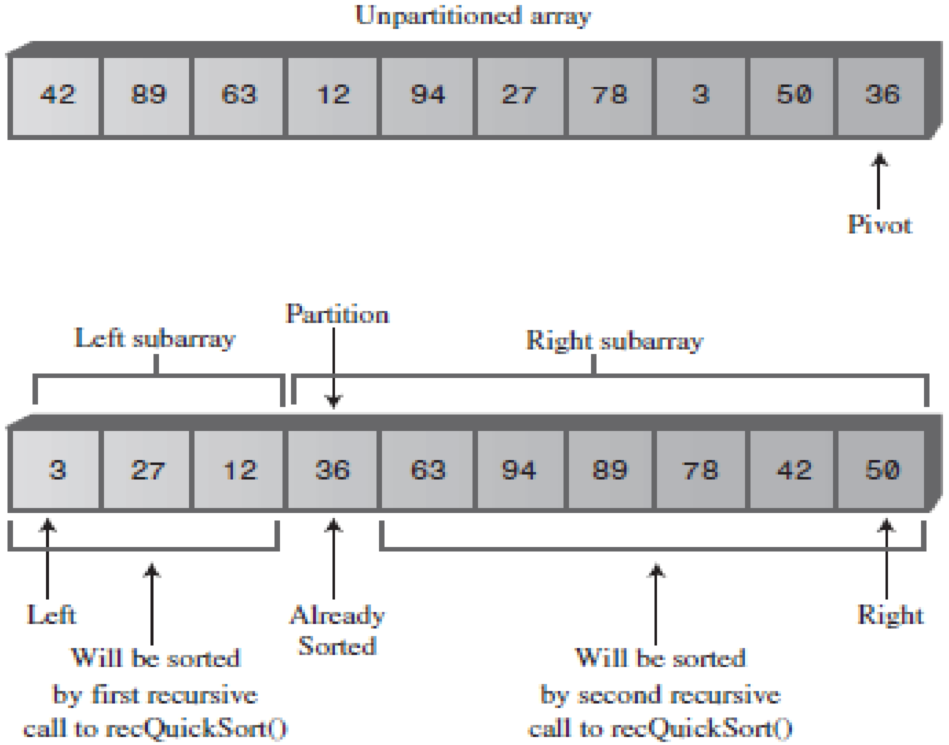
- In preparation for Quicksort:
  - Choose our pivot value to be the rightmost element
  - Partition the array around the pivot
  - Ensure the pivot is at the location of the partition
    - Meaning, the pivot should be the leftmost element of the right subarray
- Example: Unpartitioned **42 89 63 12 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**

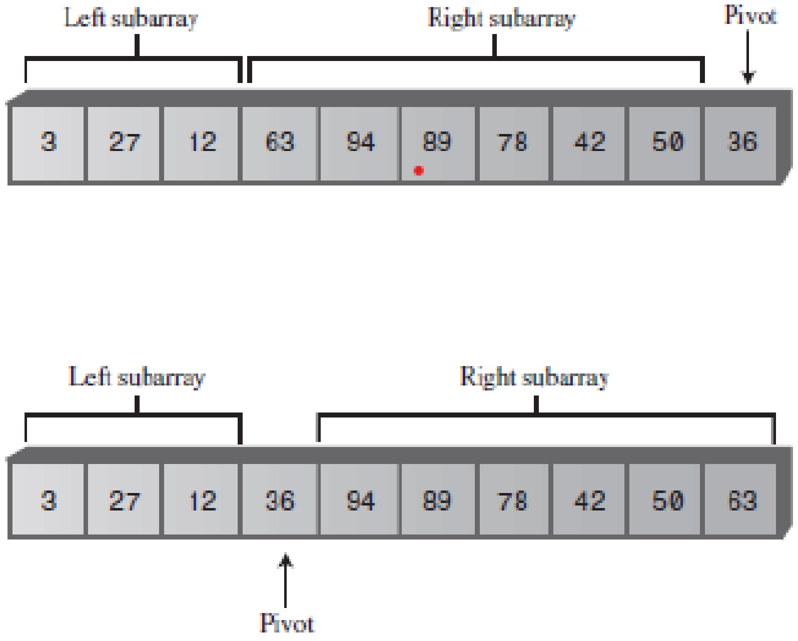
Shifting the PIVOT

- We have this:
  - 3 27 12 63 94 89 78 42 50 36
- Our goal is the position of 36
- Shift every element in the right suba
  - 3 27 12 36 63 94 89 78 42 50



Recursive calls sort subarrays.

Swapping the PIVOT



Swapping the pivot.

h the pivot! Better

63

e right subarray is not in any particular order

o our Python method

ap()

with A[left]

Shall We Try It On An ARRAY?

- 1 7 5 3 6 9 0 4 8 2
- Let’s go step-by-step via Quick Sort

Shall We Try It On An ARRAY?

- 1 7 5 3 6 9 0 4 8 2
- Let’s go step-by-step via Quick Sort
  - 1 0 2 3 6 9 7 4 8 5
  - 0 1 | 2 | 3 4 5 7 6 8 9
  - 0 | 1 | 2 | 3 4 | 5 | 7 6 8 9
  - 0 | 1 | 2 | 3 | 4 | 5 | 7 6 8 | 9
  - 0 | 1 | 2 | 3 | 4 | 5 | 7 6 | 8 | 9
  - 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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

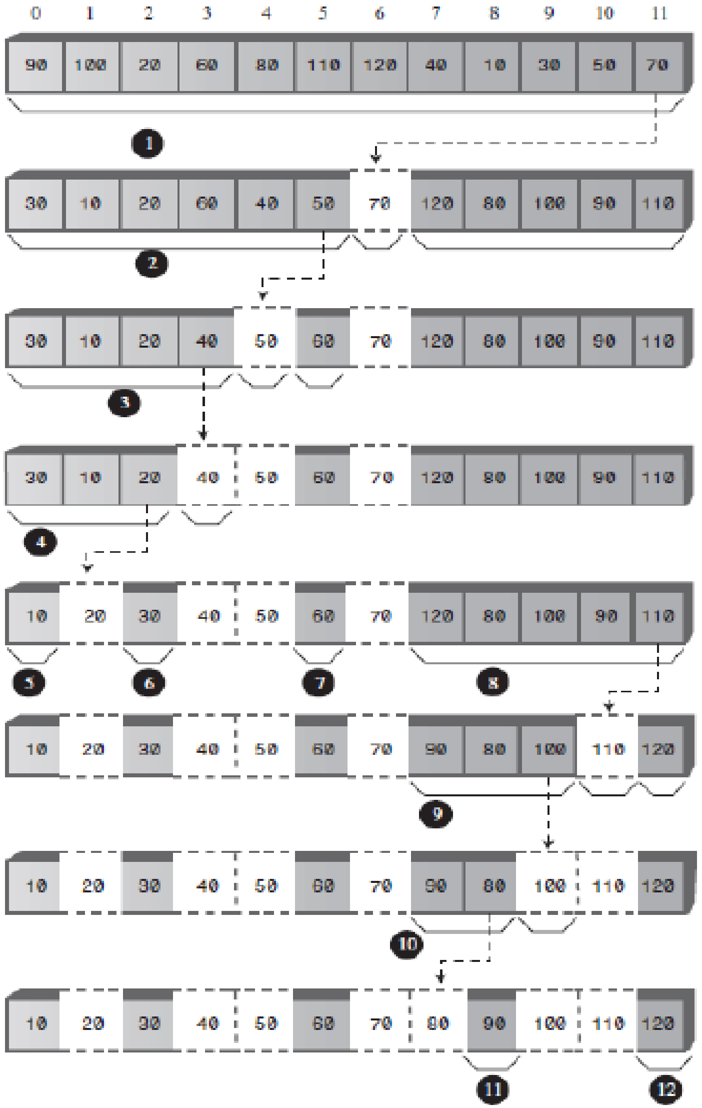
- Once you get to a very small subarray, you can just sort with insertion sort
- You can experiment a bit with ‘cutoff’ values
  - Knuth: n=9

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



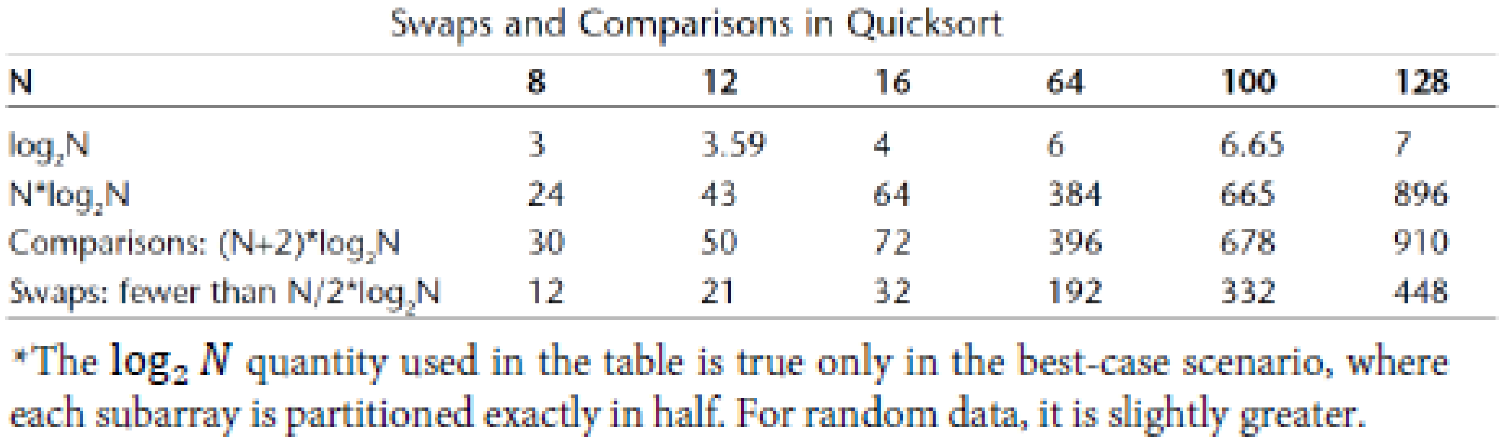
The quicksort process.

- $O(N*\log N)$ time (except when the simpler version is
ted data).

n a certain size (the cutoff) can be sorted by a method other

ommonly used to sort subarrays smaller than the cutoff.

also be applied to the entire array, after it has been sorted
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.