Analysis of Algorithms

Yan Gu

$\boldsymbol{O}, \boldsymbol{\Omega}, \boldsymbol{\Theta}, \mathbf{o}, \boldsymbol{\omega}$ notations

Functions	Real numbers analogy
$\boldsymbol{f(n) = O\big(g(n)\big)}$	$a \leq b$
$\boldsymbol{f(n)} = \Omega\big(\boldsymbol{g(n)}\big)$	$a \geq b$
$\boldsymbol{f(n)} = \Theta\big(\boldsymbol{g(n)}\big)$	$a = b$
$\boldsymbol{f(n)} = \mathrm{o}\big(\boldsymbol{g(n)}\big)$	$a < b$
$\boldsymbol{f(n)} = \omega\big(\boldsymbol{g(n)}\big)$	$a > b$

Popular Classes of Functions

		Example
• **Constant:**	$f(n) = \Theta(1)$	$1, 10, 10000000$
• **Logarithmic:**	$f(n) = \Theta(\log(n))$	$\log n, \log n + 3 \log\log n$
• **Poly-logarithmic:** (poly-logarithmically bounded)	$f(n) = O(\log^k n)$	$\log^2 n, \log^9 n + 8$
• **Sublinear:**	$f(n) = o(n)$	$\log n, \sqrt{n}, n^{1/5}$
• **Linear:**	$f(n) = \Theta(n)$	$n, 5n + \log n$
• **Super-linear:**	$f(n) = \omega(n)$	$n^3, n \log n$
• **Quadratic:**	$f(n) = \Theta(n^2)$	$n^2, 3n^2 + n$
• **Polynomial:** (polynomially bounded)	$f(n) = O(n^k)$	$n^3 + 2n^2 + 4, 4n^5$
• **Exponential:**	$f(n) = \Theta(k^n)$	2^n

($k \geq 0$ is a constant)

Some notes

- **Most of the content is already covered in CS14 and CS111, what's new?**
  - Two new concepts: o and ω
  - We'll really use these notations to describe / analyze running time bounds!
  - Understand the five notations and use them in different settings

- **More details about definitions and examples of $\boldsymbol{O}, \boldsymbol{\Omega}, \boldsymbol{\Theta}, \mathbf{o}, \boldsymbol{\omega}$ can be found in the CLRS book**
  - Read Section 1-3 carefully to understand them

Relationship between asymptotic notations and analyzing algorithms

- **Time complexity: count the number of operations (usually assuming the input size is n)**

- **Usually using asymptotic notation can make our life much easier**

- **We can omit lower order terms**
  - When n is small, the algorithm is fast anyway; when n is large, lower-order terms do not dominate

- **We can omit constant factors**
  - Mainly for simplicity reasons

What is the exact number of operations?

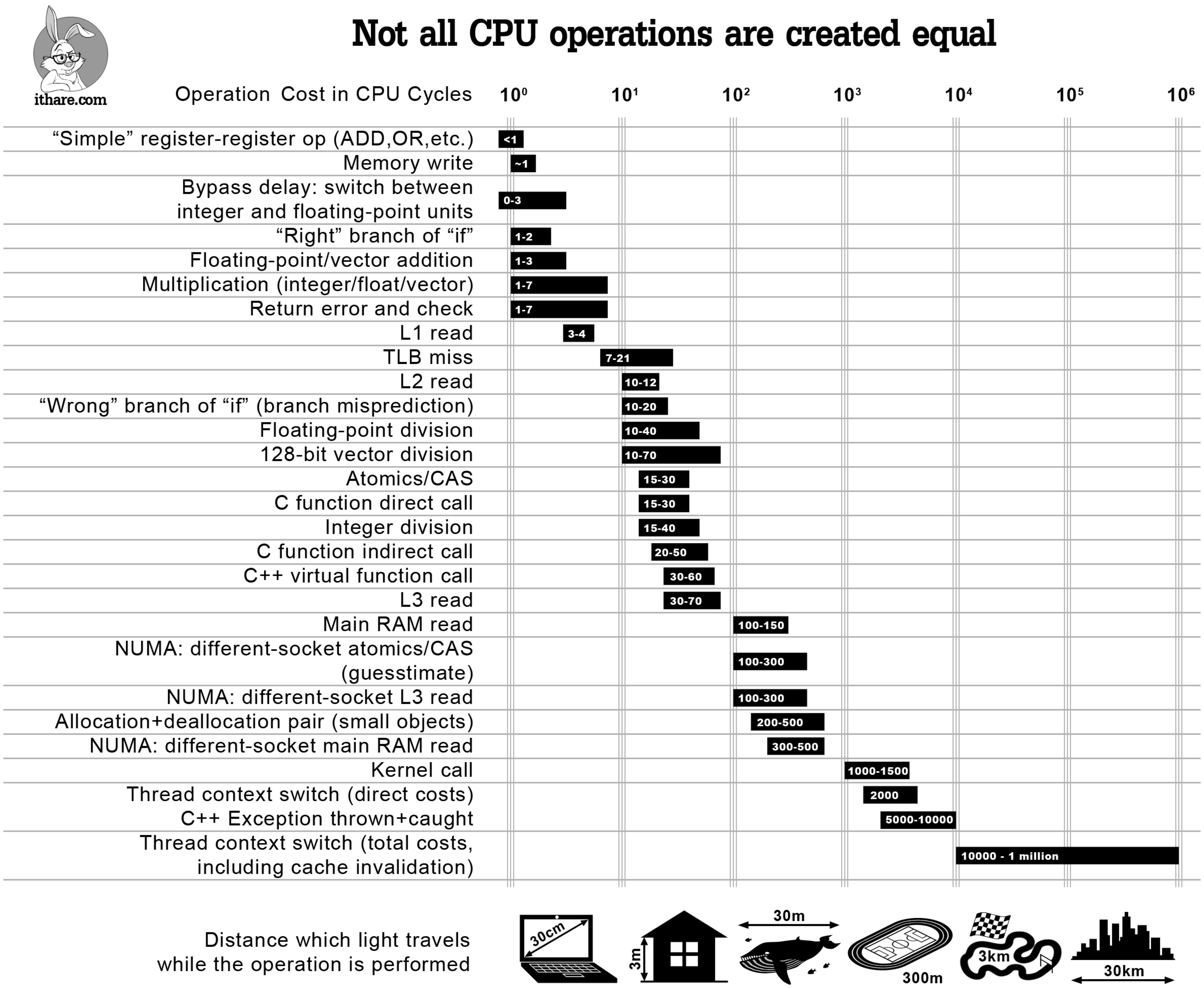
```
sum = 0;
for (int i = 1; i <=n; i++)
  sum += i;
```

```
sum = (1+n)*n/2;
```

3n+2 operations?

3 operations?

Do these operations cost the same?



- **You can take CS 142: Algorithm Engineering in Winter 2022 to study how to accurately estimate the running time of an algorithm**
- **But here we just omit all the details and ignore all the constant factors here**

Image from ithare.com: http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

Relationship between asymptotic analysis and analyzing algorithms

- **Time complexity: count the number of operations (usually assuming the input size is n)**

- **Usually using asymptotic notation can make our life much easier**

- **We can omit lower order terms**
  - When n is small, the algorithm is fast anyway; when n is large, lower-order terms do not dominate

- **We can omit constant factors**
  - Mainly for simplicity reasons

CS141: Intermediate Data Structures and Algorithms

Divide-and-conquer Algorithms and the Analysis

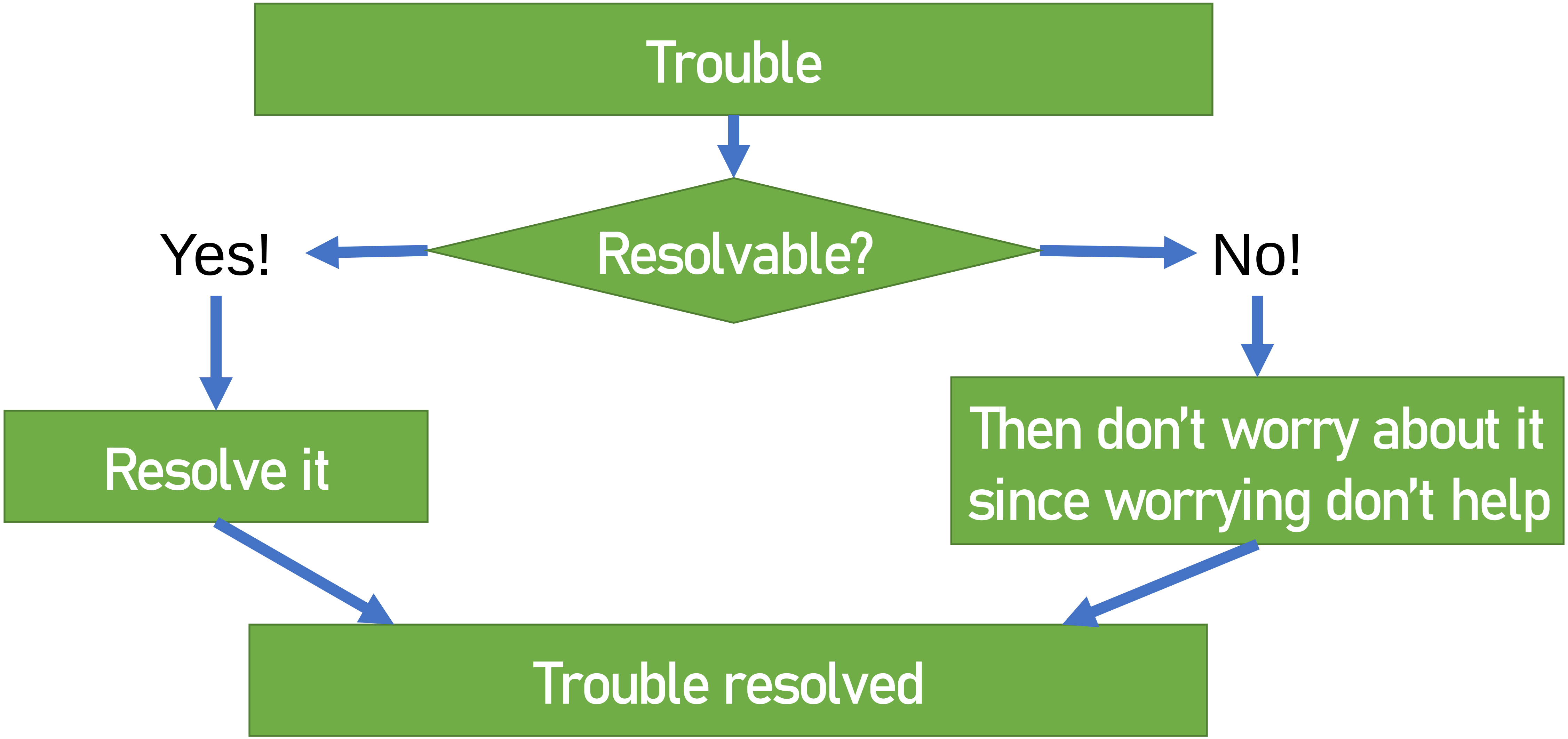
Yan Gu

This lecture covers Section 4 of the CLRS textbook

We use divide-and-conquer to solve real-world problems already

- Your friend: I’m not happy recently…
- You: Why?
- Your friend: I have a lot of problems to worry about recently…
- You: Are there problems you cannot resolve?
- Your friend: Yes…
- You: Then worrying cannot help, right? Other things are solvable, right?
- Your friend: Yeah…
- You: Then take your time and resolve them ☺
- Your friend: Great! I feel much better now!

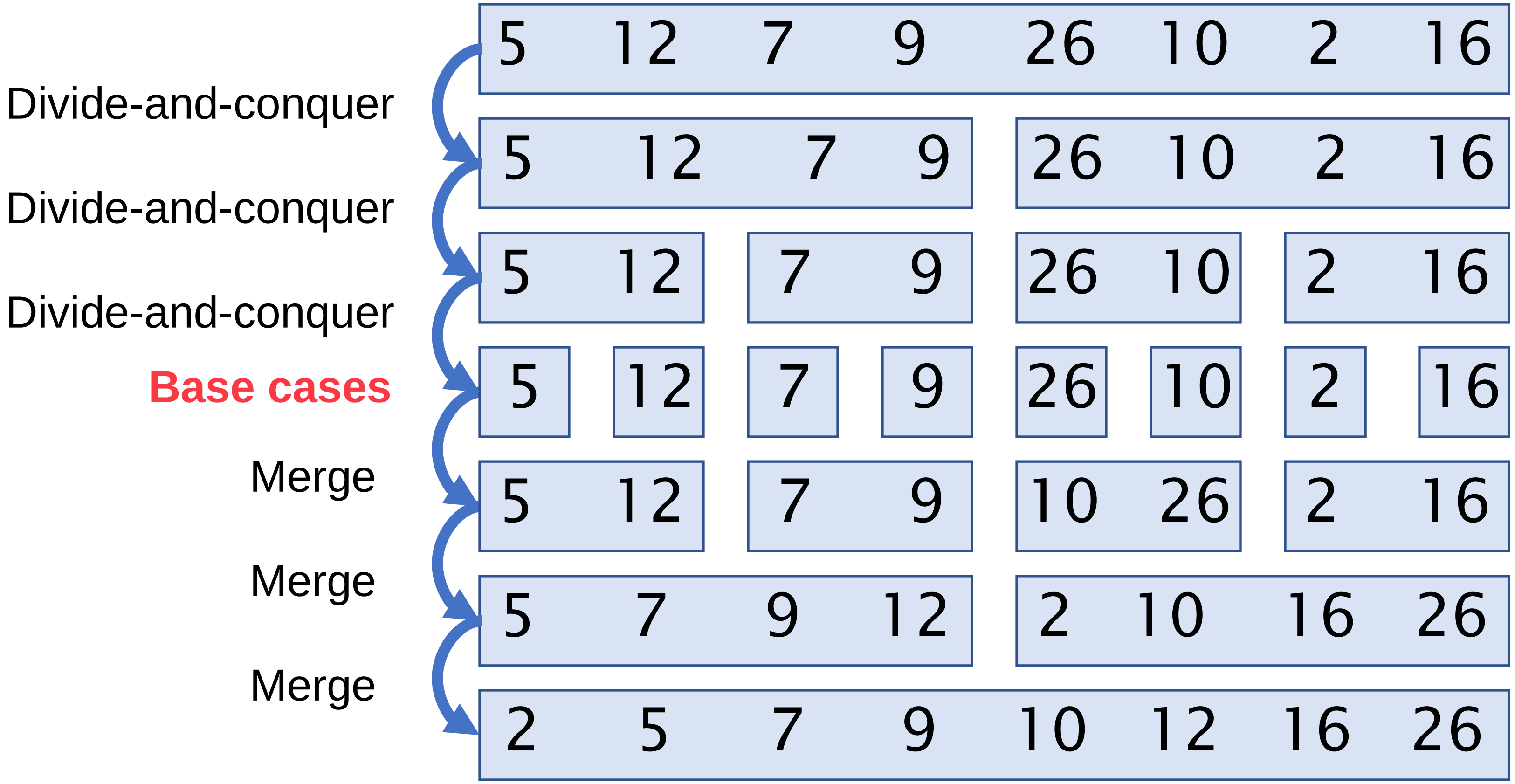
We use divide-and-conquer to solve real-world problems already



Reason 1 to use divide-and-conquer

- **Sometimes the subproblems are easier than the entire problem**
  - Smaller
  - Simpler
  - Fit into the cache
  - Etc.

Merge sort




```
void mergesort(int *A, int n) {
  if (n <= 1) return; else {
    mergesort(A, n/2);
    mergesort(A+n/2, n-n/2);
    A = merge(A, n/2, A+n/2, n-n/2);  }}
```

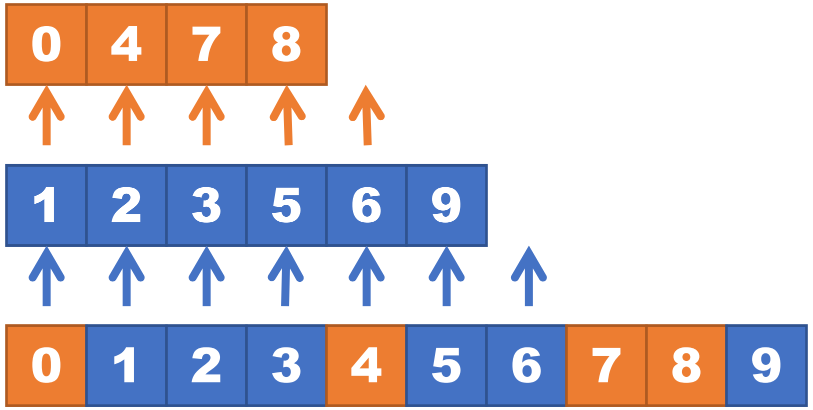
← Divide

← Conquer

← Combine

Merge two sorted arrays

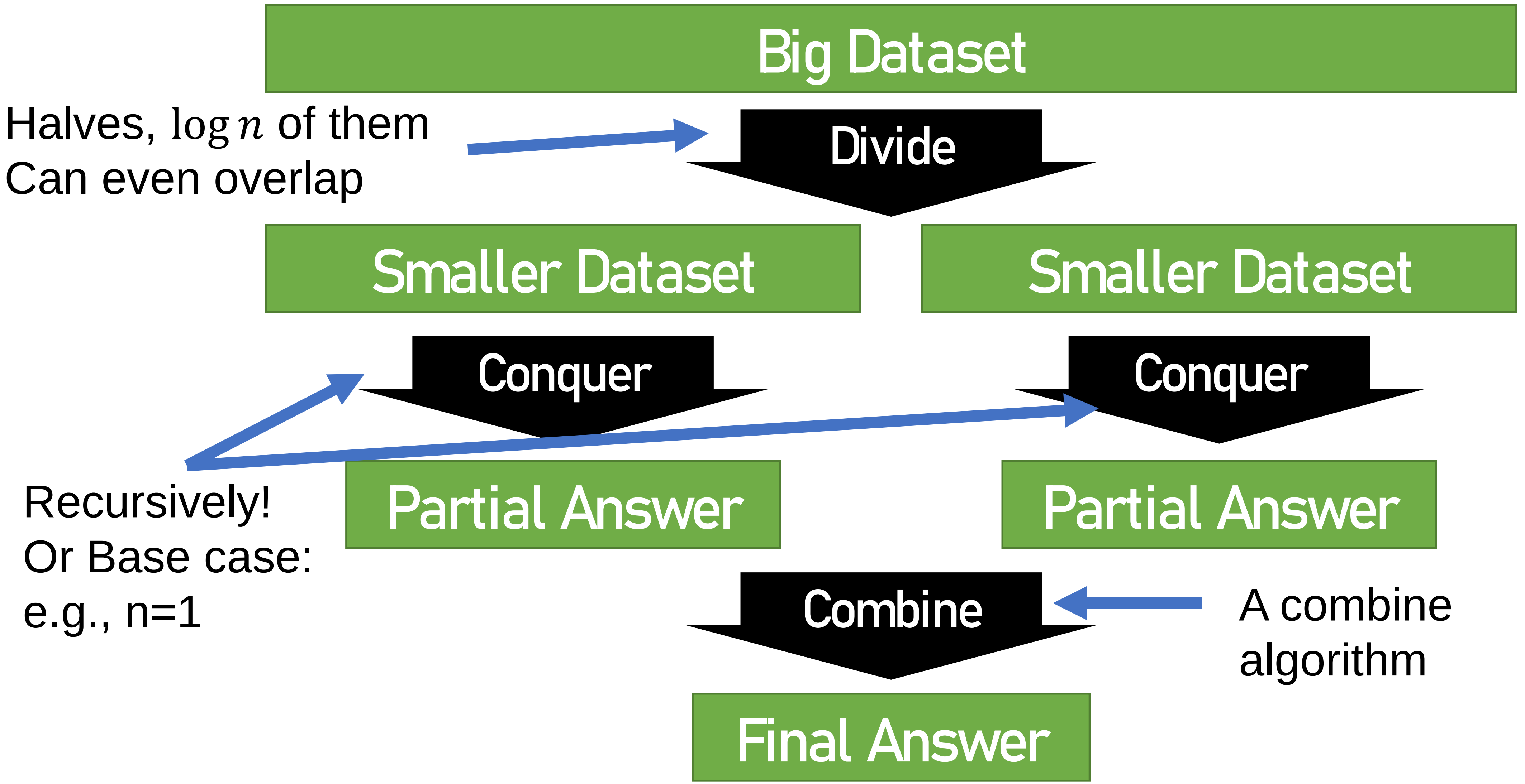
- **Given two sorted arrays**
- **Combine them into one sorted one**




```
merge(A, na, B, nb) {
  p1 = 0; p2 = 0; p3 = 0;
  while ((p1 < na) && (p2< nb)) {
    if (A[p1]<B[p2]) {
      C[p3++] = A[p1]; p1++;
    } else {
      C[p3++] = B[p2]; p2++;
    } }
  //copy the rest of the unfinished array
  return C;
}
```

- **What's the cost of merging two arrays of size n?**
  - **$\Theta(n)$ time**

Divide-and-Conquer



Divide and conquer and combine

- **Divide the problem into multiple subproblems with smaller sizes**
  - E.g., divide into halves
- **Conquer each of them recursively**
  - Can just call the same algorithm on the sub problems (recursively solve them)
  - Base case: when n=1 (or is small)
- **Combine results from the recursive calls**
  - Usually the hardest part in the algorithm design

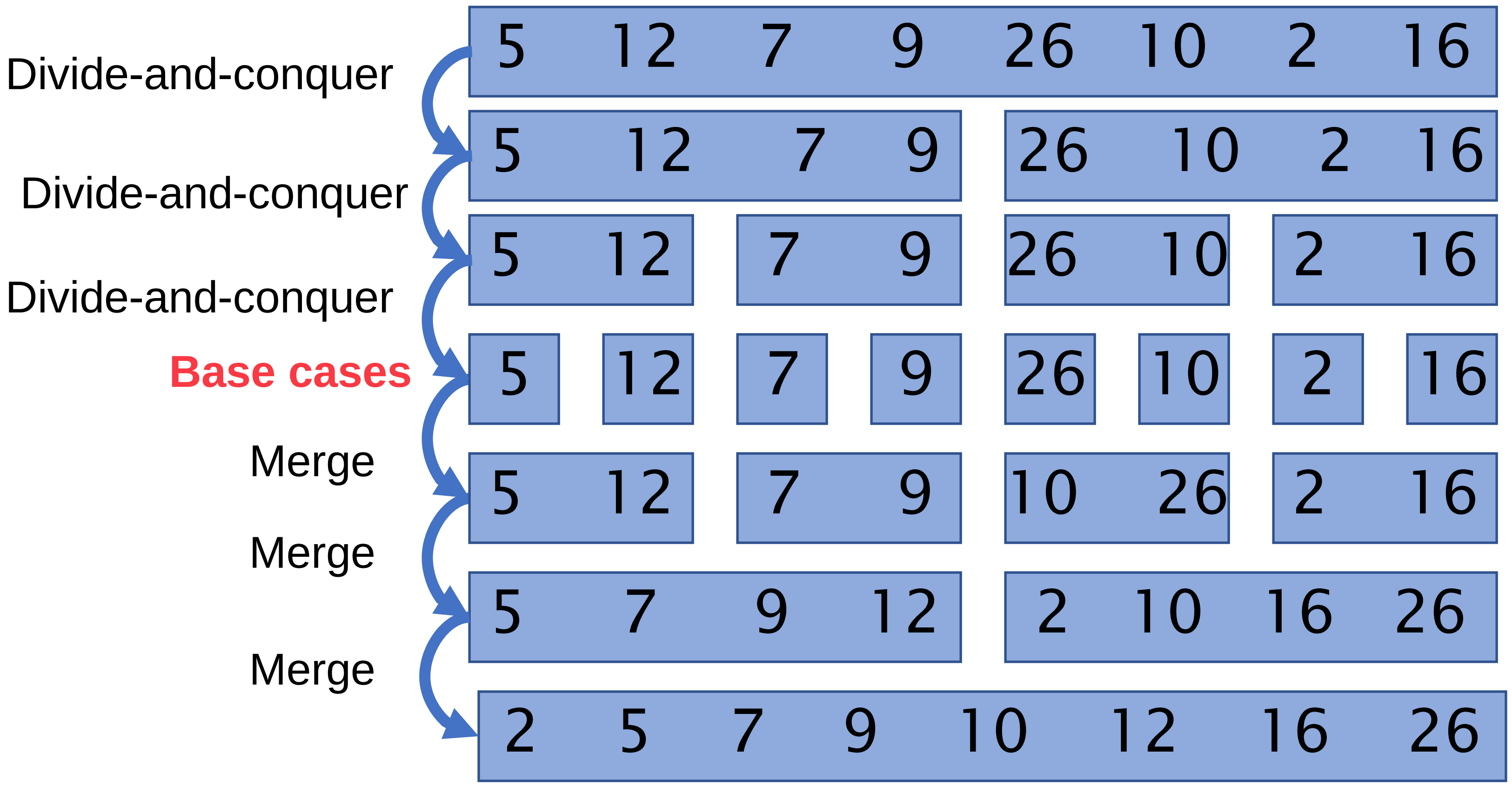
In this lecture

- Review:
- **Merge sort**: Algorithm and analysis
- **Quicksort**: Algorithm

- New algorithm:
- **Matrix multiplication**: Algorithm and analysis

- Practice: sum up an array in a divide-and-conquer manner

Merge sort




```
void mergesort(int *A, int n) {
  if (n <= 1) return; else {
    mergesort(A, n/2);
    mergesort(A+n/2, n-n/2);
    A = merge(A, n/2, A+n/2, n-n/2);  }}
```

← Divide

← Conquer

← Combine

Merge sort

$$T(n) = \begin{cases} c & \text{if } n \leq 1, \\ 2T(n/2) + c \cdot n & \text{otherwise} \end{cases}$$

```
void mergesort(int *A, int n) {
  if (n <= 1) return; else {
    mergesort(A, n/2);
    mergesort(A+n/2, n-n/2);
    A = merge(A, n/2, A+n/2, n-n/2);  }}
```

Recursion Tree

$$T(n) = \begin{cases} c & \text{if } n \le 1, \\ 2T(n/2) + c \cdot n & \text{otherwise} \end{cases}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n$$

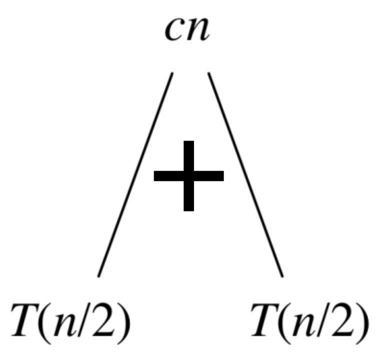
$$T(n)$$

Recursion Tree

$$T(n) = \begin{cases} c & \text{if } n \le 1, \\ 2T(n/2) + c \cdot n & \text{otherwise} \end{cases}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2}$$



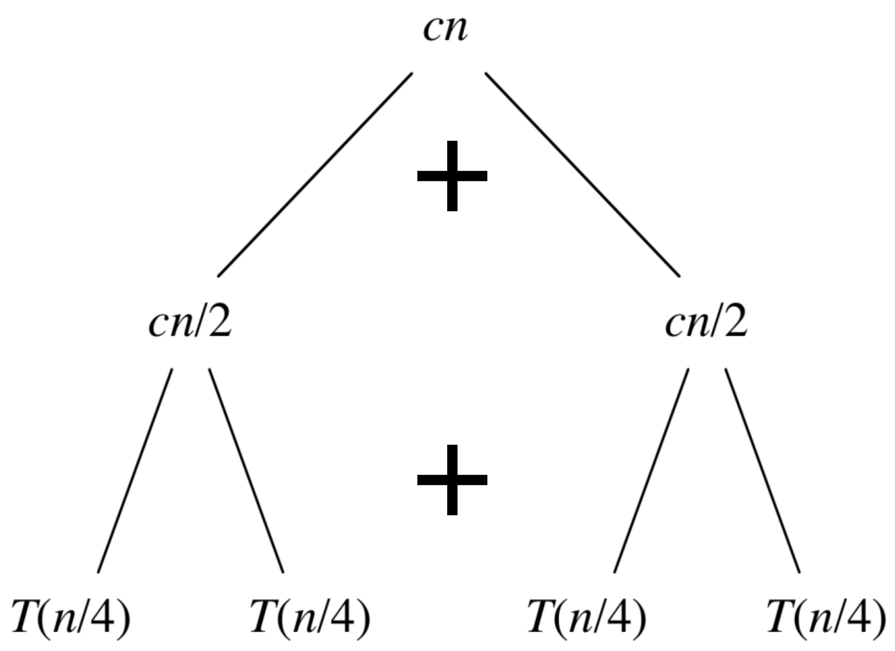
Recursion Tree

$$T(n) = \begin{cases} c & \text{if } n \le 1, \\ 2T(n/2) + c \cdot n & \text{otherwise} \end{cases}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n$$

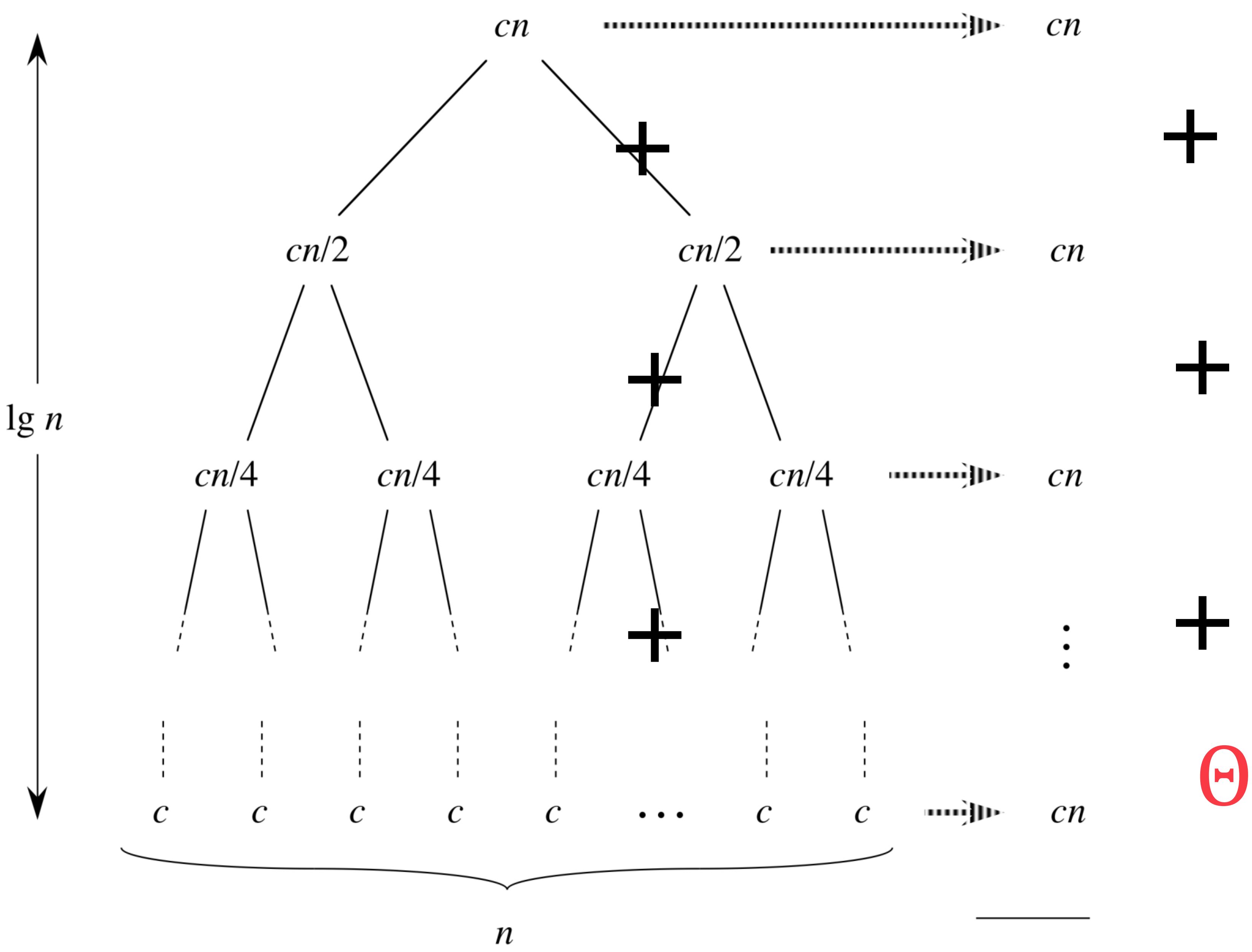
$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2}$$

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + c \cdot \frac{n}{4}$$



Recursion Tree

$$T(n) = \begin{cases} c & \text{if } n \le 1, \\ 2T(n/2) + c \cdot n & \text{otherwise} \end{cases}$$



$\Theta(n \log n)$

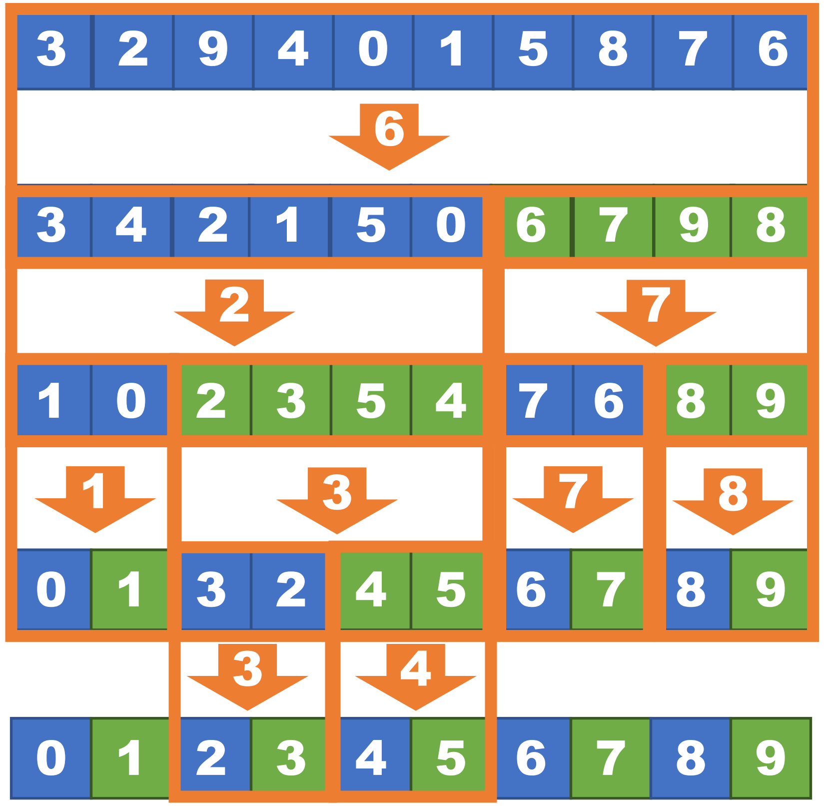
Quicksort

Quicksort

- **Another sorting algorithm that uses divide and conquer**
- **Divide (different from directly dividing into halves):**
  - Find a pivot p
  - Put all elements $\leq p$ on the left, call them L
  - Put all elements $\geq p$ on the right, call them R
  - (Those $= p$ can be put either on the right or left, or in the middle)
- **Conquer**
  - Sort $\boldsymbol{L}$ and $\boldsymbol{R}$ recursively
- **Combine**
  - No need to do anything

Quicksort

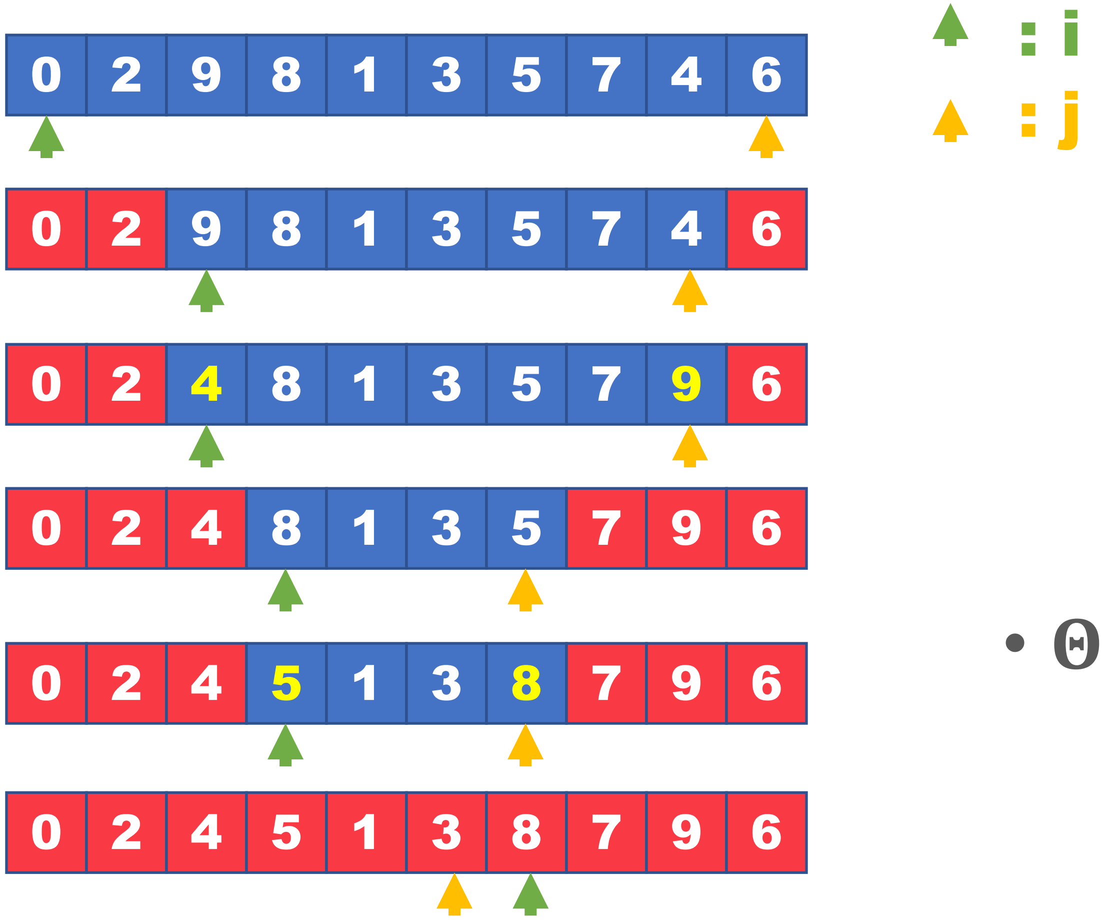
- Find a random pivot x in the array
- Put all elements in A that are smaller than p on the left of x, and all elements in A that are greater than x on the right



- The hardest part is in how to divide!

Divide: Partition the array

- **How to move elements around? (using 6 as a pivot)**

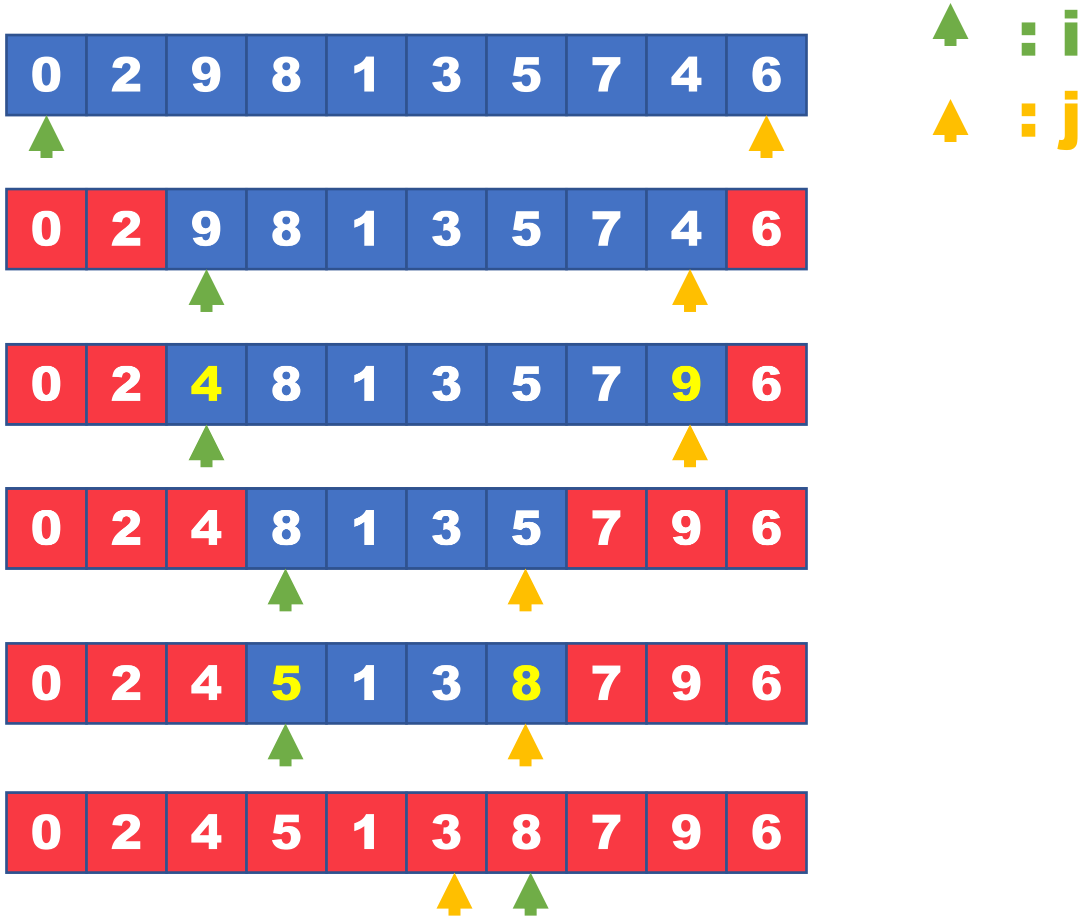



```
Partition(A, n, x) {
  i = 0; j = n-1;
  while (i < j) {
    while (A[i] < x) i++;
    while (A[j] > x) j--;
    if (i < j) {
      swap A[i] and A[j];
      i++; j--;
    }
  }
}
```

- **$\Theta(n)$ time for one round**

Divide: Partition the array

- **How to move elements around? (using 6 as a pivot)**




```
qsort(A, n) {
  i = 0; j = n-1; x = A[rand(0,n)];
  while (i < j) {
    while (A[i] < x) i++;
    while (A[j] > x) j--;
    if (i < j) {
      swap A[i] and A[j];
      i++; j--;
    }
  }
  if (i < n-1) qsort(A+i, n-i);
  if (0 < j) qsort (A, j+1);
}
```

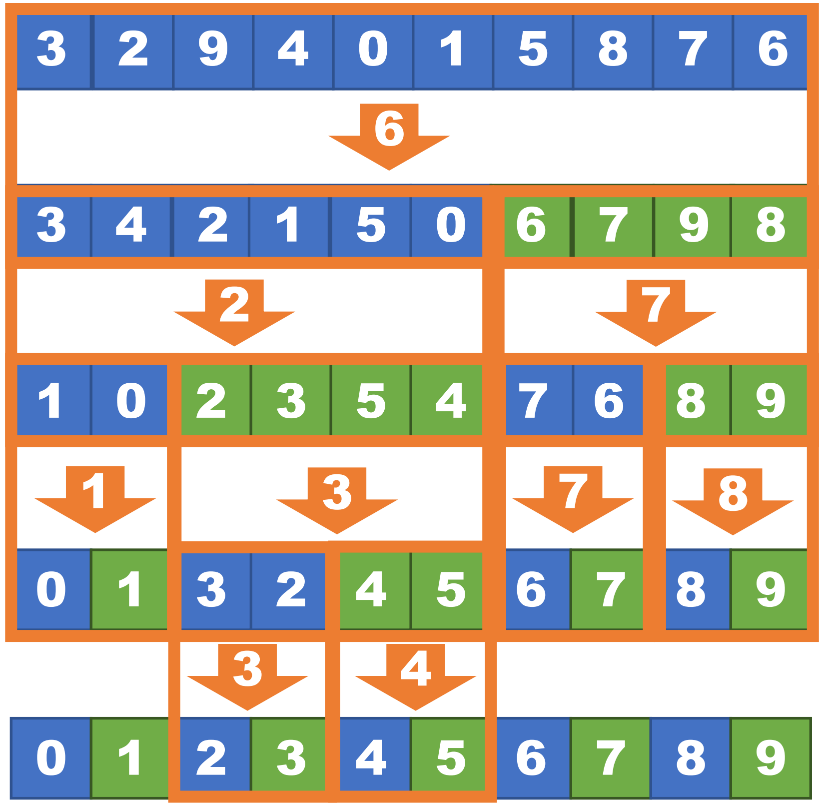
Divide (partition)

Conquer (recurse)

Quicksort – cost analysis

- **If every time we can partition the array perfectly in halves**
  - $O(\log n)$ rounds, $O(n \log n)$ time in total
- **But in the worst case, it is $O(n^2)$**
  - What is the worst case?

- **Does that mean it has similar performance as bubble sort/selection sort/insertion sort?**
- **The average cost is $O(n \log n)$!**
  - The analysis will be given CS 218
  - More analysis will be given in CS 219



Sorting algorithms

- **Both quicksort and merge sort takes $O(n \log n)$ time (in expectation for quicksort)**
  - Deterministic for merge sort, randomized for quicksort
- **However, quicksort is usually "quicker" than other sorting algorithms in practice**
  - Merge sort need additional space, and quicksort is in-place
  - Each recursive call in quicksort is dealing with a consecutive chunk in the input – more cache friendly
  - (Take CS142 / 214 for more details! ☺)

Reason 1 to use divide-and-conquer

- **Sometimes the subproblems are easier than the entire problem**
  - Smaller
  - Simpler
  - Fit into the cache
  - Etc.

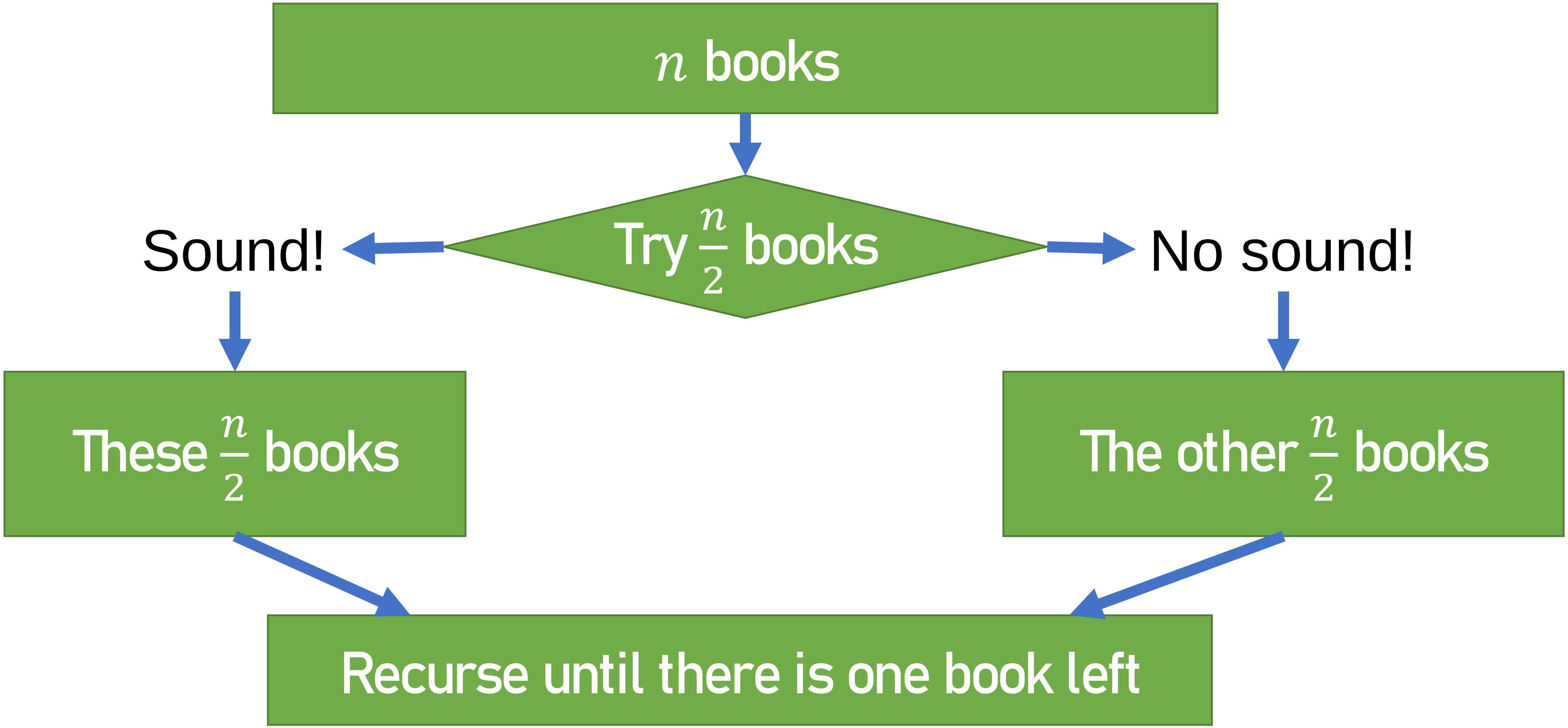
We use divide-and-conquer to solve real-world problems

- You went to the library, and borrowed quite a few books. Unfortunately, one of the books was not checked out, so the alarm started.

- You can try each book and see if this book was not checked out.

- Any better solution?

We use divide-and-conquer to solve real-world problems already



Another example: COVID testing in China

- **It's common in China to run COVID test for all citizens in a city**
- **Chinese cities are large: 20 cities with more than 5M population, and 50 cities more than 2M**
- **Assume we know that there are only 100 positive cases, how can we test it efficiently?**
- **Solution:**
  - They mix the parts of every 100 samples and test them (round 1)
  - For all positive samples, they check each of the samples in those groups (round 2)
  - Assume the city has 1 million people, how many tests do they need in the worst case?
  - 10^6/100 (round 1) + 100*100 (round 2, worst case) = 20000, saved 98% tests

Reason 2 to use divide-and-conquer

- **It saves the run time of the algorithms**
  - Fewer operations
  - Each operation gives you more information than doing it straightforwardly
  - Etc.

Matrix Multiplication

Matrix Multiplication

Consider standard iterative matrix-multiplication algorithm

$$C := A \quad B$$

- Where A, B, and C are $N \times N$ matrices

```
for i = 1 to N do
    for j = 1 to N do
        for k = 1 to N do
            C[i][j] += A[i][k] * B[k][j]
```

- $\Theta(N^3)$ computation in RAM model.

Recursive Matrix Multiplication

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} := \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Compute 8 submatrix products recursively

$$\begin{aligned} C_{11} &:= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &:= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &:= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &:= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

- **8-way divide-and-conquer**
  - $T(N) = \Theta(N^2) + 8T\left(\frac{N}{2}\right)$

Matrix Multiplication

- $\boldsymbol{T(N) = \Theta(N^2) + 8T\left(\frac{N}{2}\right)}$
- **Omit constant:**
  - $T(N) = N^2 + 8T\left(\frac{N}{2}\right)$
  - $T\left(\frac{N}{2}\right) = \left(\frac{N}{2}\right)^2 + 8T\left(\frac{N}{4}\right)$
  - $T\left(\frac{N}{4}\right) = \left(\frac{N}{4}\right)^2 + 8T\left(\frac{N}{8}\right)$
  - ……

$$
\begin{aligned}
T(N) &= N^2 + 8T\left(\frac{N}{2}\right) \\
&= N^2 + 8\left(\left(\frac{N}{2}\right)^2 + 8T\left(\frac{N}{4}\right)\right) \\
&= N^2 + 8\left(\frac{N}{2}\right)^2 + 64\left(\left(\frac{N}{4}\right)^2 + 8T\left(\frac{N}{8}\right)\right) \\
&= \cdots \text{ (}\log_2 N\text{ rounds)} \\
&= N^2 + 2N^2 + 4N^2 + \cdots + 2^{\log_2 N} N^2 \\
&= N^2(1 + 2 + 4 + \cdots + N) \\
&= \Theta(N^3)
\end{aligned}
$$

Strassen's Algorithm: cost analysis

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} := \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Step 1: Compute S matrices:

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

Step 2: Compute P matrices:

- $P_1 = A_{11} \cdot S_1$ $\quad P_2 = S_2 \cdot B_{22}$
- $P_3 = S_3 \cdot B_{11}$ $\quad P_4 = A_{22} \cdot S_4$
- $P_5 = S_5 \cdot S_6$ $\quad P_6 = S_7 \cdot S_8$
- $P_7 = S_9 \cdot S_{10}$

Only 7 of them!!

Step 3: Compute C matrices:

- $C_{11} = P_5 + P_4 - P_2 + P6$
- $C_{12} = P_1 + P_2$
- $C_{21} = P_3 + P_4$
- $C_{22} = P_5 + P_1 - P_3 - P_7$

Strassen's Algorithm: cost analysis

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} := \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Step 1: Compute S matrices:

- $S_1 = B_{12} - B_{22}$ $\quad S_6 = B_{11} + B_{22}$
- $S_2 =$ … $- A_{22}$
- $S_3 =$ … $+ B_{22}$
- $S_4 =$ … $- A_{21}$
- $S_5 = A_{11} + A_{22}$ $\quad S_{10} = B_{11} + B_{12}$

10 +/- of matrices of size $\frac{N}{2}$: $c_1 N^2$ $(\Theta(N^2))$

Step 2: Compute P matrices:

- $P_1 =$ … B_{22}
- $P_3 =$ … $\cdot S_4$
- $P_5 =$ … S_8
- $P_7 = S_9 \cdot S_{10}$

7 multiplications of matrices of size $\frac{N}{2}$: $7T(\frac{N}{2})$

Step 3: Compute C matrices:

- C … $P6$
- C …
- C …
- C … P_7

9 +/- of matrices of size $\frac{N}{2}$: $c_2 N^2$ $(\Theta(N^2))$

$$T(N) = 7T\left(\frac{N}{2}\right) + cN^2$$

Strassen's Algorithm: cost analysis

- $T(N) = 7T\left(\frac{N}{2}\right) + cN^2$

- **How to solve this? We'll discuss it later.**

- **Solution:**
- $T(N) = \Theta\left(N^{\log_2 7}\right) \approx \Theta(N^{2.8074})$
  - Smaller than N^3!
  - Computing the multiplication of two matrices of size N doesn't need $\Theta(N^3)$ operations!

Matrix Multiplication

- **It doesn't need to be $\times$ and $+$**
- **If elements are Boolean values:**
  - Use $\times$ as "**and**"
  - Use $+$ as "**or**"

- **The same matrix multiplication algorithm applies**
  - We will see this again in the all-pair shortest path algorithm later in this course

- **This does not apply to Strassen's algorithm**
  - It needs "minus" (inverse of "$+$"), which does not exist for "**or**"

How to solve a recurrence in general?

Master Theorem

Solving recurrences – Master Theorem

- The Master Method for solving divide-and-conquer recurrences applies to the recurrences in the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where $a \geq 1, b > 1$, and f is asymptotically positive (positive for sufficiently large n).

Base case: $T(c)$ is a constant when c is a constant

Review of the simpler version in CS 111

- You have learned how to solve recurrence in this form:

$$T(n) = aT\left(\frac{n}{b}\right) + n^y$$

Where $a \geq 1, b > 1$

Base case: $T(c)$ is a constant when c is a constant

- Case 1: $y < \log_b a, T(n) = \Theta\left(n^{\log_b a}\right)$

- Case 2: $y = \log_b a, T(n) = \Theta(n^y \log n)$

- Case 3: $y > \log_b a, T(n) = \Theta(n^y)$

Merge sort

$$T(n) = \begin{cases} c & \text{if } n \le 1, \\ 2T(n/2) + c \cdot n & \text{otherwise} \end{cases}$$

- $\boldsymbol{a = b = 2}$, $\log_{\boldsymbol{b}} \boldsymbol{a = y = 1}$
- **Case 2:** $\boldsymbol{T(n)} = \Theta(\boldsymbol{n}^{\boldsymbol{y}} \log \boldsymbol{n}) = \Theta(\boldsymbol{n} \log \boldsymbol{n})$

Strassen's Algorithm

- $T(N) = 7T\left(\frac{N}{2}\right) + cN^2$

- $\boldsymbol{a = 7, b = 2,} \log_{\boldsymbol{b}} \boldsymbol{a} = \log_{\boldsymbol{2}}\boldsymbol{7} \approx \boldsymbol{2.81 > y = 2}$

- **Case 1:** $\boldsymbol{T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{2.81}\right)}$

Now let's extend it to a more general case

- You have learned how to solve recurrence in this form:

$$T(n) = aT\left(\frac{n}{b}\right) + n^y$$

Where $a \geq 1, b > 1$
Base case: $T(c)$ is a constant when c is a constant

- Case 1: $y < \log_b a, T(n) = \Theta\left(n^{\log_b a}\right)$

- Case 2: $y = \log_b a, T(n) = \Theta(n^y \log n)$

- Case 3: $y > \log_b a, T(n) = \Theta(n^y)$

Master Theorem – General case

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where $a \geq 1, b > 1$, and f is asymptotically positive (positive for sufficiently large n)

- Case 1: $f(n) = O\left(n^{(\log_b a)-\epsilon}\right)$ for constant $\epsilon > 0$, $T(n) = \Theta\left(n^{\log_b a}\right)$
- Case 2: $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$ for $k \geq 0$, $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- Case 3: $f(n) = \Omega\left(n^{(\log_b a)+\epsilon}\right)$ for constant $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leq cf(n)$ **for some constant $c < 1$ and all sufficiently large n,** $T(n) = \Theta\left(f(n)\right)$

Recursion Tree: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$T\left(\frac{n}{b}\right) = aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)$

$b^h = n$

$h = \log_b n$

a $f(n)$ — 1 task, each $f(n)$

a $f(n/b)$ $f(n/b)$ … $f(n/b)$ — a tasks, each $f\left(\frac{n}{b}\right)$

$f(n/b^2)$ $f(n/b^2)$ … $f(n/b^2)$ — a^2 tasks, each $f\left(\frac{n}{b^2}\right)$

…

$\vdots$

$f(O(1))$ — ? tasks, each $f(O(1))$

Idea: Compare $n^{\log_b a}$ with $f(n)$

Recursion Tree: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$n^{\log_b a} > f(n)$

Case 1, $f(n) = O(n^{y'})$, where $y' < \log_b a = y$ **(Let $y = \log_b a$)**

e.g., when $\log_b a = 3$ but $f(n) = n^2$

$b^h = n$

$h = \log_b n$

1 task, each $f(n)$ $\quad f(n) \le n^{y'}$

$\Uparrow$

a tasks, each $f(\frac{n}{b})$ $\quad af\left(\frac{n}{b}\right) \le a\left(\frac{n}{b}\right)^{y'} = a \cdot \frac{n^{y'}}{b^{y'}} = \frac{a}{b^{y'}} n^{y'}$

$\Uparrow$

a^2 tasks, each $f(\frac{n}{b^2})$ $\quad a^2 f\left(\frac{n}{b^2}\right) \le a^2\left(\frac{n}{b^2}\right)^{y'} \le \left(\frac{a}{b^{y'}}\right)^2 n^{y'}$

$\vdots$ $\quad \vdots$

$\Uparrow$

$a^{\log_b n}$ tasks, each $f(O(1))$
$= \Theta(n^{\log_b a})$ $\quad \Theta(n^{\log_b a})$

$\log_b n$ terms

Grow geometrically,

common ratio is $\frac{a}{b^{y'}} > 1$

(because $y' < \log_b a$)

GEOMETRICALLY INCREASING (LEAF-DOMINATED)

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

Dominated by the last term!

Recursion Tree: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$n^{\log_b a} > f(n)$

Case 1, $f(n) = O(n^{y'})$, where $y' < \log_b a = y$ **(Let $y = \log_b a$)**

e.g., when $\log_b a = 3$ but $f(n) = n^2$

GEOMETRICALLY INCREASING (LEAF-DOMINATED)

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

Example 1:

$$T(n) = 8T\left(\frac{n}{2}\right) + n^2$$

(Matrix multiplication!)

$$b = 2, a = 8$$
$$f(n) = n^2$$

$$y = \log_2 8 = 3$$

$f(n) = n^2$, so $y' = 2 < 3$

Leaf cost: $\Theta(n^3)$, root cost: n^2

$$T(n) = \Theta\left(n^{\log_2 8}\right) = \Theta(n^3)$$

Example 2:

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2$$

(Strassen's Algorithm!)

$$b = 2, a = 7$$
$$f(n) = n^2$$

$$y = \log_2 7 \approx 2.807$$

$f(n) = n^2$ so $y' = 2 < 2.807$

Leaf cost: $\Theta(n^{2.807})$, root cost: n^2

$$T(n) = \Theta\left(n^{\log_2 7}\right) = \Theta(n^{2.807})$$

Recursion Tree: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$n^{\log_b a} \approx f(n)$

Case 2, $f(n) = \Theta(n^y \log^k n)$, where $y = \log_b a$, $k \geq 0$ is a constant

e.g., when $\log_b a = 1$ and $f(n) = n$ **Example: k=0, $f(n) = n^y$**

$b^h = n$

$h = \log_b n$

1 task, each $f(n)$ $\qquad f(n) = n^y$

≀

a tasks, each $f(\frac{n}{b})$ $\qquad af\left(\frac{n}{b}\right) = a\left(\frac{n}{b}\right)^y = \overset{1}{\boxed{\frac{a}{b^y}}} n^y = f(n)$

≀

a^2 tasks, each $f(\frac{n}{b^2})$ $\qquad a^2 f\left(\frac{n}{b^2}\right) \leq a^2\left(\frac{n}{b^2}\right)^y = \overset{1}{\boxed{\left(\frac{a}{b^y}\right)^2}} n^y = f(n)$

⋮ $\qquad$ ⋮

≀

$a^{\log_b n}$ tasks, each $f(O(1)) = \Theta(n^{\log_b a})$ $\qquad \Theta(n^{\log_b a}) = \overset{1}{\boxed{\left(\frac{a}{b^y}\right)^{\log_b n}}} n^y = f(n)$

$\log_b n$ terms

Same order of magnitude,

$f(n) \cdot h = \Theta(f(n) \log n)$

ARITHMETICALLY INCREASING

$T(n) =$

$\boldsymbol{\Theta(f(n)\log n)} =$

$\Theta\left(n^{\log_b a} \log^{k+1} n\right)$

Recursion Tree: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$n^{\log_b a} \approx f(n)$

Case 2, $f(n) = \Theta(n^y \log^k n)$, where $y = \log_b a$, k is a constant

e.g., when $\log_b a = 1$ and $f(n) = n$

ARITHMETICALLY INCREASING

$T(n) = \Theta(f(n)\log n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$

Example 1:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

(Merge sort!)

$$b = 2, a = 2$$

$$f(n) = n$$

$$y = \log_2 2 = 1$$

$$f(n) = \Theta(n \log^0 n)$$

So $k = 0$

Leaf cost = $\Theta(n)$, root cost = n

$$T(n) = \Theta(n \log^{0+1} n)$$

$$= \Theta(n \log n)$$

Example 2:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \log n$$

$$b = 2, a = 2$$

$$f(n) = n \log n$$

$$y = \log_2 2 = 1$$

$$f(n) = n \log n = \Theta(n^1 \log^1 n)$$

so $k = 1$

Leaf cost = $\Theta(n)$, root cost = $\Theta(n \log n)$

$$T(n) = \Theta(n \log^{1+1} n) = \Theta(n \log^2 n)$$

Recursion Tree: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$n^{\log_b a} < f(n)$

Case 3, $f(n) = \Omega(n^{y'})$, where $y' > y = \log_b a$

e.g., when $\log_b a = 2$ but $f(n) = n^3$

$b^h = n$

$h = \log_b n$

1 task, each $f(n)$

a tasks, each $f(\frac{n}{b})$

a^2 tasks, each $f(\frac{n}{b^2})$

$\vdots$

$a^{\log_b n}$ tasks, each $f(1)$
$= \Theta(n^{\log_b a})$

Generally case 3 does not imply anything. But if $f(n)$ satisfies the regularity condition that $af(n/b) \le cf(n)$ for some constant $c < 1$, then

GEOMETRICALLY DECREASING (ROOT-DOMINATED)
$T(n) = \Theta(f(n))$

Recursion Tree: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$b^h = n$
$h = \log_b n$

$a \quad f(n)$ — 1 task, each $f(n)$

$a \quad f(n/b) \quad f(n/b) \quad \dots \quad f(n/b)$ — a tasks, each $f(\frac{n}{b})$

$f(n/b^2) \quad f(n/b^2) \quad \dots \quad f(n/b^2)$ — a^2 tasks, each $f(\frac{n}{b^2})$

$\dots$

$\vdots$

$f(O(1))$ — $a^{\log_b n}$ tasks, each $f(1)$
$= \Theta(n^{\log_b a})$

You can always analyze a specific algorithm using the tree

Master Theorem

- Solve $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where $a \geq 1$ and $b > 1$, f is asymptotically positive
- Let $y = \log_b a$ and constant $k \geq 0$. The leaf cost is $\Theta(n^y)$. The root cost is $f(n)$

- Case 1: $f(n) = O(n^{y'})$ for $y' < y$

leaf cost $\gg$ root cost $f(n) \Rightarrow$ Leaf dominated (differ by at least n^ϵ)

$\Rightarrow T(n) = \Theta(n^y) =$ leaf cost

- Case 2: $f(n) = \Theta(n^y \log^k n)$

leaf cost $\approx$root cost $f(n)$ (can differ by at most a factor of $\log^k n$)

$\Rightarrow T(n) = \Theta\left(n^y \log^{k+1} n\right) = \Theta(f(n) \log n) =$ #levels $\times$ root cost

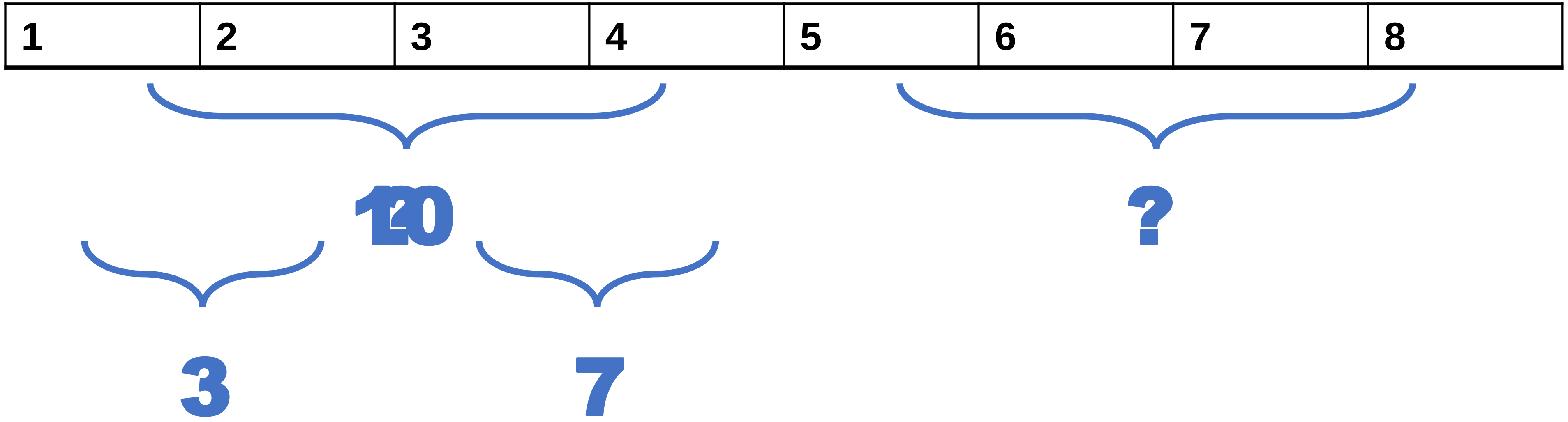
- Case 3: $f(n) = \Omega(n^{y'})$ for $y' > y$ and *regularity condition*

leaf cost $\ll f(n) \Rightarrow$ Root dominated (differ by at least n^ϵ)

$\Rightarrow T(n) = \Theta\left(f(n)\right) =$ root cost

Practice: Sum up an array

Practice: Sum up an array



- **Let's do it in a divide-and-conquer algorithm**

```
sum(A, n) {
  if (n == 1) return A[0];
  L = sum(A, n/2);
  R = sum(A + n/2, n-n/2);
  return L+R;
}
```

Practice: Sum up an array

- **Let's do it in a divide-and-conquer algorithm**

```
sum(A, n) {
  if (n == 1) return A[0];
  L = sum(A, n/2);
  R = sum(A + n/2, n-n/2);
  return L+R;
}
```

$$T(n) = \begin{cases} c_1 & \text{if } n \le 1, \\ 2T(n/2) + c_2 & \text{otherwise} \end{cases}$$

$a = b = 2, \log_b a = 1 > y = 0$ → **Case 1:** $\boldsymbol{T(n)} = \Theta\left(\boldsymbol{n^{\log_b a}}\right) = \Theta(n)$

Reasons to use divide-and-conquer

- **Sometimes the subproblems are easier than the entire problem**
- **It saves the run time of the algorithms**
- **It allows for parallelism, and better locality for memory accesses**

- **However, divide-and-conquer is not a specific algorithm, but a general idea to solve problems**
  - We will see similar methodologies such as greedy and dynamic programming in the rest of this course

Recap

- **Divide-and-conquer (then combine) is a general way to design algorithms**
  - In this lecture we reviewed mergesort and quicksort, and discussed 8-way DAC matrix multiplication, and Strassen's (7-way) matrix multiplication, and DAC reduce
  - For those of you who are interested in algorithms, you can read the "linear-time selection" algorithm in CLRS Section 9.3, which is a divide-and-conquer algorithm

- **Master theorem is a useful tool to analyze recurrences for DAC algorithms**
  - We reviewed the basic form in CS 111, and the full version in CLRS Section 5

- **The next lecture: Greedy Algorithms**

Class announcement

- **Programming homework 1 due tomorrow**
  - Scoreboard is frozen, so you don't see any update on that
  - Protect your privacy (if you want your names to show on the scoreboard, then solve the problems two days before the ddl)
- **Hints: all problems can be solved using knowledge from CS 10A/B/C, so review the content if needed**