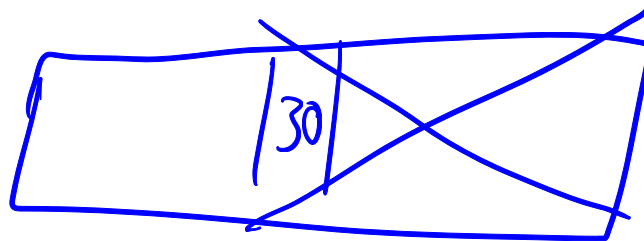


Divide-and-Conquer Algorithms:

1. **Divide:** Break the problem (instance) into subproblems of sizes that are fractions of the original problem size.
2. **Conquer:** Recursively solve each of the subproblems. If the subproblems are “small enough” (base case), then solve the subproblem in a straightforward manner.
3. **Combine:** Put the solutions for each of the subproblems together to obtain a solution for the original problem.

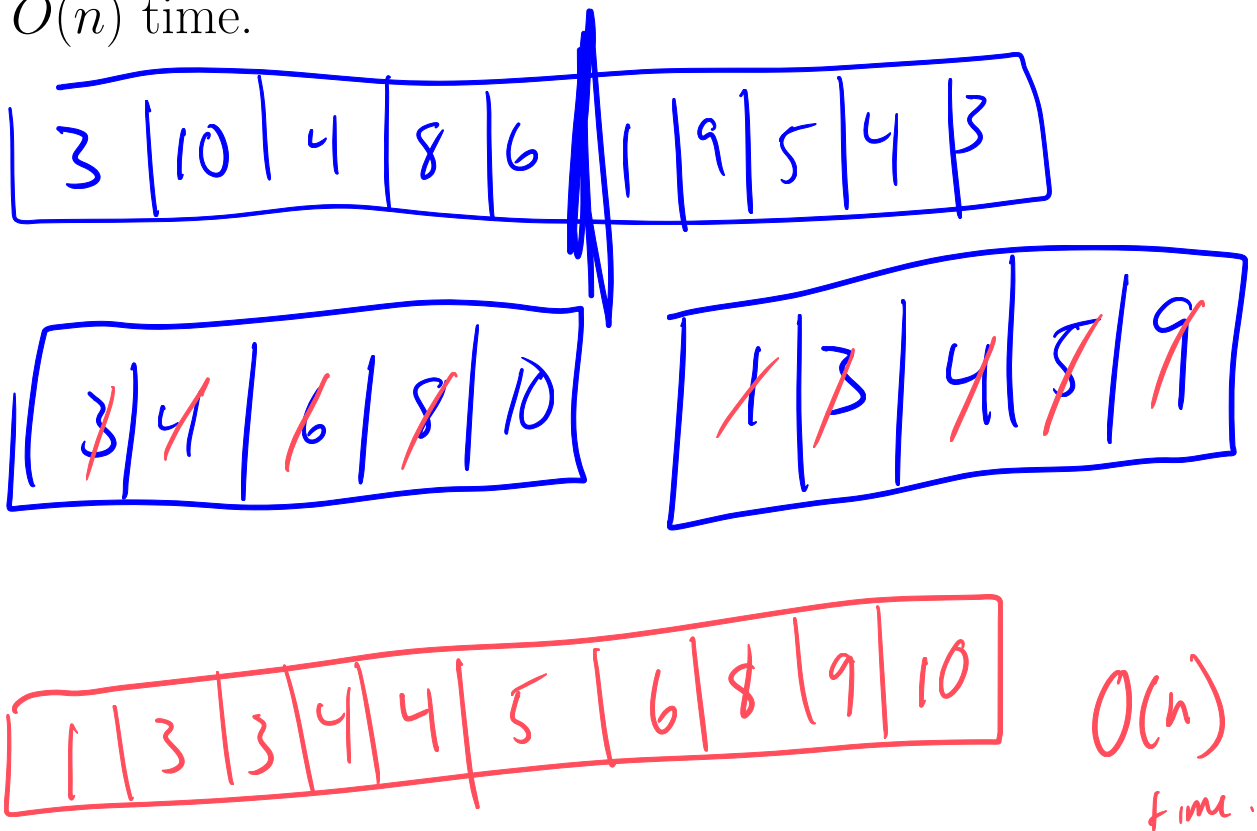
Example: Use binary search to find an element k in a sorted array.

1. **Divide:** Compare the middle element with k .
2. **Conquer:** If the middle element is not k , we can recursively solve the subproblem where we check half of the current array.
3. **Combine:** There is only 1 subproblem in binary search, and so the combine step is trivial.

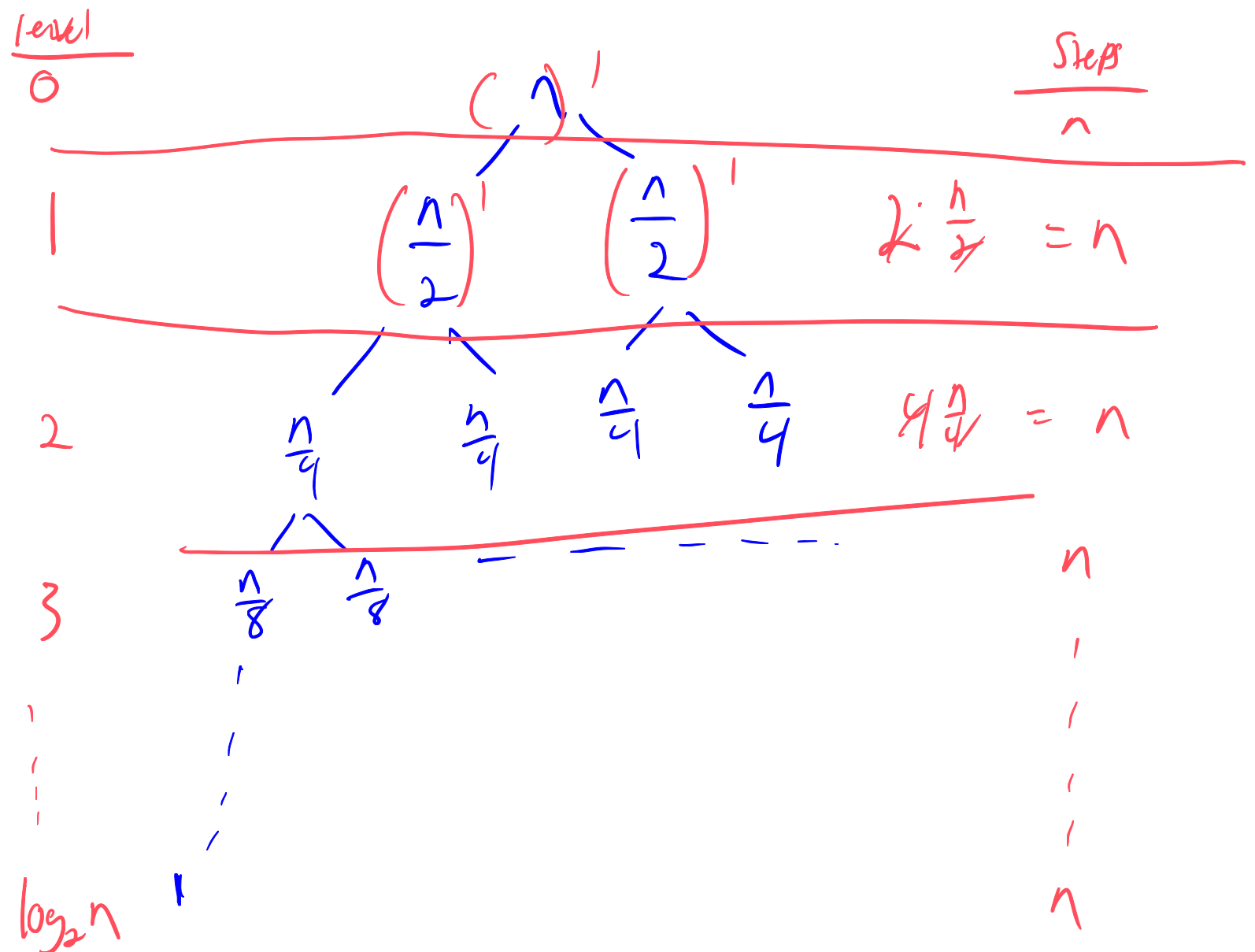


Merge Sort: Given an array of n numbers, sort the numbers in the array in non-decreasing order.

1. **Divide:** Break the problem into 2 subproblems of size $n/2$.
2. **Conquer:** Recursively sort each of the 2 subarrays.
3. **Combine:** Combine the two sorted subarrays with $n/2$ elements into one sorted array of n elements in $O(n)$ time.



Merge Sort Recursion Tree:



answer is
to sum this
column

$$\sum_{i=0}^{\log n} n = \Theta(n \log n).$$

Merge Sort:

- Merge sort runs in $\Theta(n \log n)$ time.
- $\Theta(n \log n)$ grows more slowly than the $\Theta(n^2)$ running time of insertion sort as n approaches infinity.
- Therefore, merge sort asymptotically beats insertion sort in the worse case.
- In practice, merge sort performs better than insertion sort on arrays with 30 or more numbers.

Recursion Trees:

- A recursion tree models the running time of a recursive algorithm, but in some cases can be unreliable and/or difficult to analyze.
- It is good for generating guesses of what the running time could be.
- In such cases, we may need to verify our guess is correct via a proof by induction.

Guess: Merge Sort runs in $O(n \log n)$ time,

$T(n) \leq C \cdot n \log n$ for some constant $C > 0$ for all $n \geq n_0$ for some $n_0 > 0$.

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

of Subproblems \nwarrow size of Subproblem \nwarrow
time to divide & combine

Base Case: $T(1) = d'$
 $C \cdot 1 \log 1 \geq 0$

$$T(2) = 2T(1) + d2 = 2d' + d2 = 2(d' + d)$$

$$C2 \log 2 \geq 2C. \quad T(2) \leq C2 \log 2 \text{ if } C \geq d' + d$$

Inductive Step: Assume $T(k) \leq C \cdot k \log k$ for all $k < n$.

$$T(n) = 2T\left(\frac{n}{2}\right) + dn \leq 2 \left[C \frac{n}{2} \log \frac{n}{2} \right] + dn$$

$$\log n - \log 2$$

$$= Cn[\log n - \log 2] + dn$$

$$= \underbrace{Cn \log n}_{\text{desired}} - \underbrace{Cn + dn}_{\text{residual}}$$

true when residual ≤ 0

$$-Cn + dn \leq 0 \Rightarrow dn \leq Cn \Rightarrow \boxed{d \leq C}$$

$$\leq \underbrace{Cn \log n}_{\text{desired}}$$

$$x + y \leq x$$

$$y \leq 0$$

Example: computing a^n for some non-negative integer n .

Naive algorithm: Compute $a \cdot a \cdot a \cdots a$. Takes $\Theta(n)$ time.

Divide-and-conquer algorithm:

Suppose I know $a^{\frac{n}{2}}$, I can get a^n by squaring $a^{\frac{n}{2}}$.

If n is even: $a^n = a^{\frac{n}{2}} \cdot a^{\frac{n}{2}}$

If n is odd: $a^n = a^{\lfloor \frac{n}{2} \rfloor} \cdot a^{\lfloor \frac{n}{2} \rfloor} \cdot a$

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

levels
0
1
2
⋮
 $\log_2 n$

$(n)^0$
|
 $\left(\frac{n}{2}\right)^0$
|
 $\left(\frac{n}{4}\right)^0$
⋮
|

Steps
1
1
1
⋮
1

$$\sum_{i=0}^{\log n} 1 = \Theta(\log n)$$

$\text{pow}(a, b)$

Example: Matrix Multiplication: Given two $n \times n$ matrices A and B , compute the $n \times n$ matrix $C = A \cdot B$.

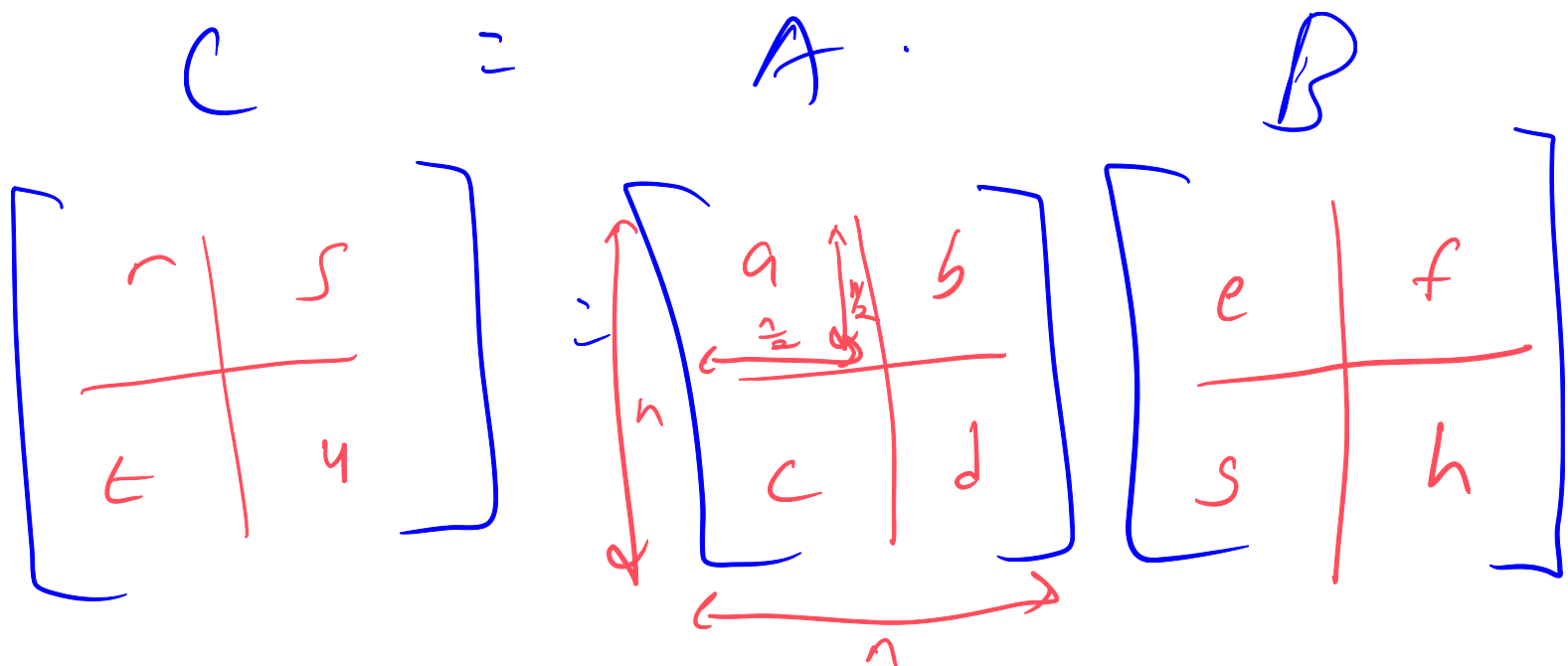
$$\begin{matrix} & C & & & \\ & & & A & & & B \\ \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} & = & \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} & \cdot & \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}
 \end{matrix}$$

Possible algorithm:

```

for  $i = 1; i \leq n; i = i + 1$  do
  for  $j = 1; j \leq n; j = j + 1$  do
     $\underline{c_{ij}} = 0$ 
    for  $k = 1; k \leq n; k = k + 1$  do
       $c_{ij} = c_{ij} + \underline{a_{ik} b_{kj}}$ 
    end for
  end for
end for
  
```

Divide-and-conquer algorithm:



$$r = a \odot e \oplus b \odot g$$

$$s = a \odot f \oplus b \odot h$$

$$t = c \odot e \oplus d \odot g$$

$$u = c \odot f \oplus d \odot h$$

8 recursive calls, $8T(\frac{n}{2})$

4 additions: $O(n^2)$

$$T(n) = 8T(\frac{n}{2}) + O(n^2).$$

This runs in $O(n^3)$ time.

Strassen's idea:

- Recursively compute the following matrices (note only 7 multiplications):

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

- We can then compute C as so:

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

- Running time of first divide-and-conquer algorithm:
 - $T(n) = 8T(n/2) + \Theta(n^2) = \Theta(\underline{n^{\log 8}}) = \Theta(n^3)$
- Running time of Strassen's algorithm:
 - $T(n) = 7T(n/2) + \Theta(n^2) = \Theta(\underline{n^{\log 7}}) = o(n^3)$
- Strassen's algorithm beats ordinary iterative algorithm in practice for $n \geq 30$ or so.