CSE 373: Data Structures and Algorithms

Sorting and recurrence analysis techniques

Autumn 2018

Shrirang (Shri) Mare
shri@cs.washington.edu

Thanks to Kasey Champion, Ben Jones, Adam Blank, Michael Lee, Evan McCarty, Robbie Weber, Whitaker Brand, Zora Fung, Stuart Reges, Justin Hsia, Ruth Anderson, and many others for sample slides and materials ...

Sorting problem statement

Review

Given n comparable elements, rearrange them in an increasing order.

Input

- An array A that contains n elements
- Each element has a key k and an associated data
- Keys support a comparison function (e.g., keys implement a Comparable interface)

Expected output

- An output array A such that for any i and j,
- $A[i] \le A[j]$ if $i < j$ (increasing order)
- Array A can also have elements in reverse order (decreasing order)

Desired properties in a sorting algorithm

Review

Stable
- In the output, equal elements (i.e., elements with equal keys) appear in their original order

In-place
- Algorithm uses a constant additional space, $O(1)$ extra space

Adaptive
- Performs better when input is almost sorted or nearly sorted
- (Likely different big-O for best-case and worst-case)

Fast. $O\ (n \log n)$

No algorithm has all of these properties. So choice of algorithm depends on the situation.

Sorting algorithms – High-level view

- $O(n^2)$
  - Insertion sort
  - Selection sort
  - Quick sort (worst)
- $O(n \log n)$
  - Merge sort
  - Heap sort
  - Quick sort (avg)
- $\Omega(n \log n)$ -- lower bound on comparison sorts
- $O(n)$ – non-comparison sorts
  - Bucket sort (avg)

A framework to think about sorting algos

Some questions to consider when analyzing a sorting algorithm.

Loop/step invariant: What is the state of the data during each step while sorting

Runtime: Worst ______ Average ______ Best ______

Input: Worst ______ Best ______

Stable Yes/No/Can-be In-place Yes/No/Can-be Adaptive Yes/No

Operations: Comparisons > or < or approx. equal Moves

Data structure Which data structure is better suited for this algo

Visualization: https://visualgo.net/en/sorting

Insertion sort

Idea: At step i, insert the i^{th} element in the correct position among the first i elements.

0	1	2	3	4	5	6	7

Loop/step invariant: ____

Runtime: Worst ____________ Average ____________ Best ____________

Input: Worst ____________________ Best ____________________

Stable __________________ In-place __________________ Adaptive __________________

Operations: Comparisons ________________________ Moves

Data structure ____

Visualization: https://visualgo.net/en/sorting

Selection sort

Idea: At step i, find the smallest element among the not-yet-sorted elements (i ... n) and swap it with the element at i.

0	1	2	3	4	5	6	7

Loop/step invariant: ____

Runtime: Worst ______________ Average ______________ Best ______________

Input: Worst ______________________ Best ______________________

Stable ____________________ In-place ____________________ Adaptive ____________________

Operations: Comparisons __________________________ Moves

Data structure ____

Heap sort

Idea: $buildHeap$ with all n elements

for i = 0 to n **do**

$A[i] = removeMin()$

end for

Loop/step invariant: ______________________________

Runtime: Worst ____________ Average ____________ Best ____________

Input: Worst ________________ Best ________________

Stable ________________ In-place ________________ Adaptive ________________

Operations: Comparisons ____________________ Moves

Data structure ______________________________

In-place heap sort

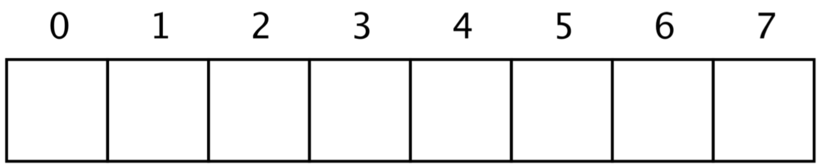
Idea:

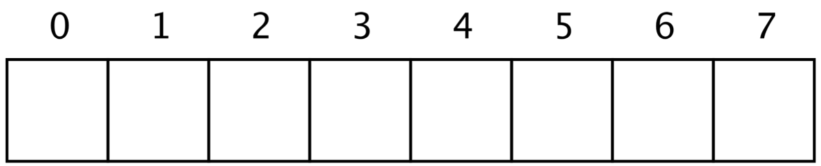
1. Treat initial array as a heap

2. When you call `removeMin()`, that frees up a slot towards the end in the array. Put the extract `min` element there.

3. More specifically, when you remove the i^{th} element, put it at $A[n-i]$

4. This gives you a reverse sorted array. But easy to fix in-place.





Design technique: Divide-and-conquer

Very important technique in algorithm to attack problems

Three steps:

1. Divide: Split the original problem into smaller parts
2. Conquer: Solve individual parts independently (think recursion)
3. Combine: Put together individual solved parts to produce an overall solution

Merge sort and Quick sort are classic examples of sorting algorithms that use this technique

Visualization: https://visualgo.net/en/sorting

Merge sort

To sort a given array,

Divide: Split the input array into two halves

Conquer: Sort each half independently

Combine: Merge the two sorted halves into one sorted whole (HW3 Problem 6!)

```
function mergeSort(A)
    if A.length == 1 then
        return A;
    else
        mid = A.length / 2
        firstHalf = mergeSort(new [0, ... mid])
        secondHalf = mergeSort(new [mid+1, ... ])
        return merge(firstHalf, secondHalf)
    end if
end function
```

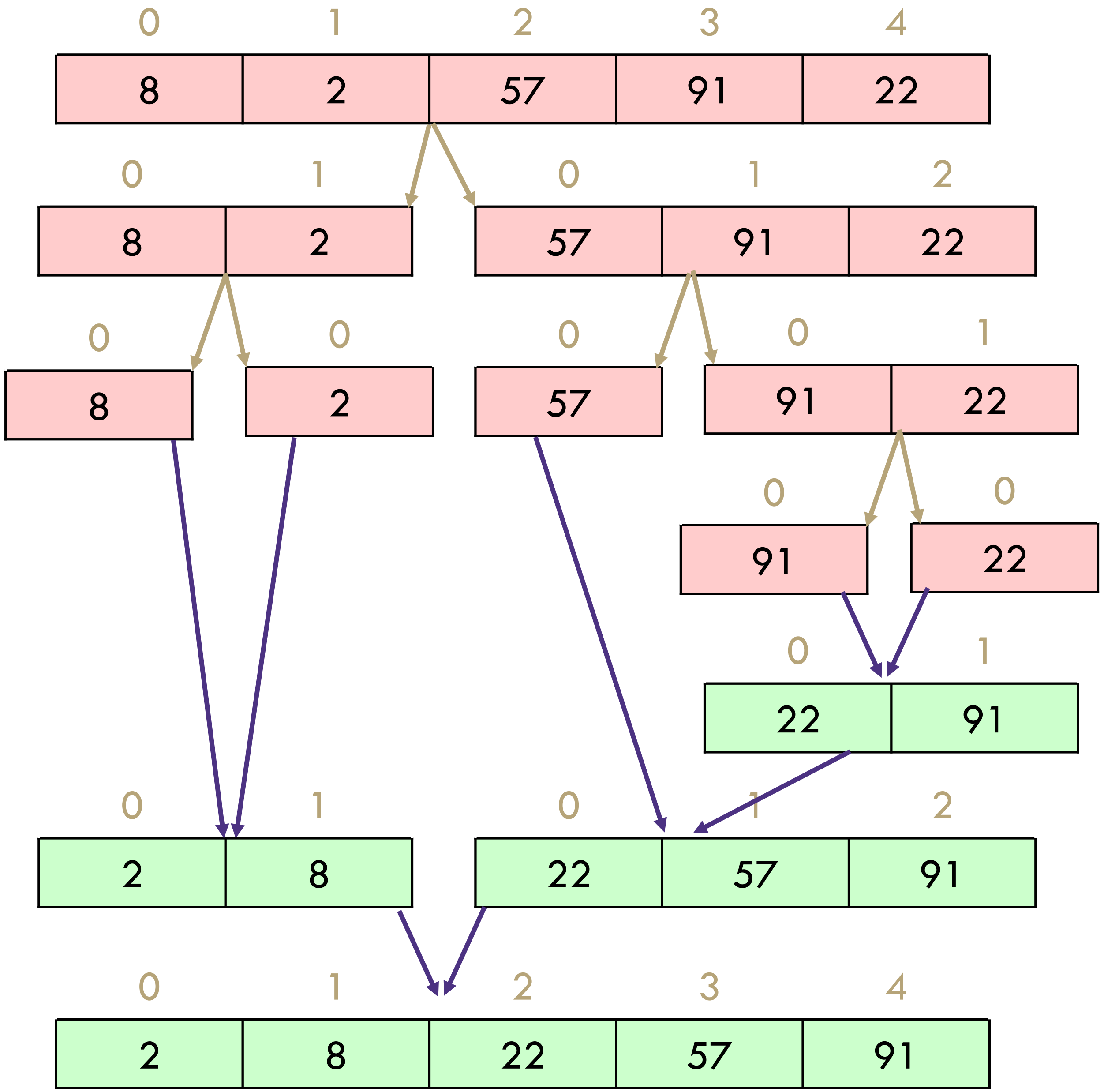
Merge sort

Split array in the middle

Sort the two halves

Merge them together

$$T(n) = \begin{cases} c_1 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + c_2 n & \text{otherwise} \end{cases}$$



Review: Unfolding (technique 1)

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

$$\begin{aligned}
T(n) &= 2T(n/2) + c_2 n \\
&= 2\left(2T\left(\frac{n}{2\cdot 2}\right) + c_2\frac{n}{2}\right) + c_2 n \\
&= 2^2 T\left(\frac{n}{2\cdot 2}\right) + c_2 n + c_2 n \\
&= 2^2\left(2T\left(\frac{n}{2^3}\right) + c_2\frac{n}{2^2}\right) + c_2 n + c_2 n \\
&= 2^3 T\left(\frac{n}{2^3}\right) + c_2 n + c_2 n + c_2 n \\
&= \dots \\
&= 2^{\log n} T(1) + \underbrace{c_2 n + c_2 n + \cdots + c_2 n}_{\text{about } \log(n) \text{ times}} \\
&= c_1 n + c_2 n \log n
\end{aligned}$$

Technique 2: Tree method

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

Level	Number of Nodes at level	Work per Node	Work per Level
0			
1			
2			
i			
base			

Last recursive level:

Technique 2: Tree method

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

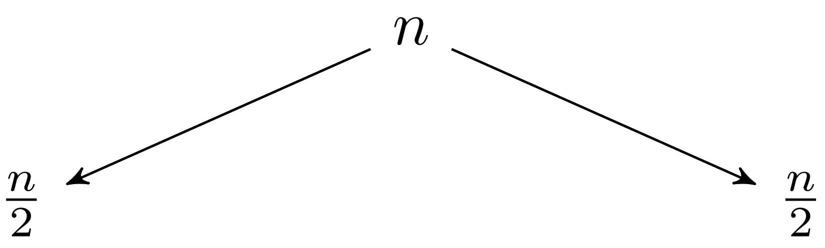
n

Level	Number of Nodes at level	Work per Node	Work per Level
0			
1			
2			
i			
base			

Last recursive level:

Technique 2: Tree method

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

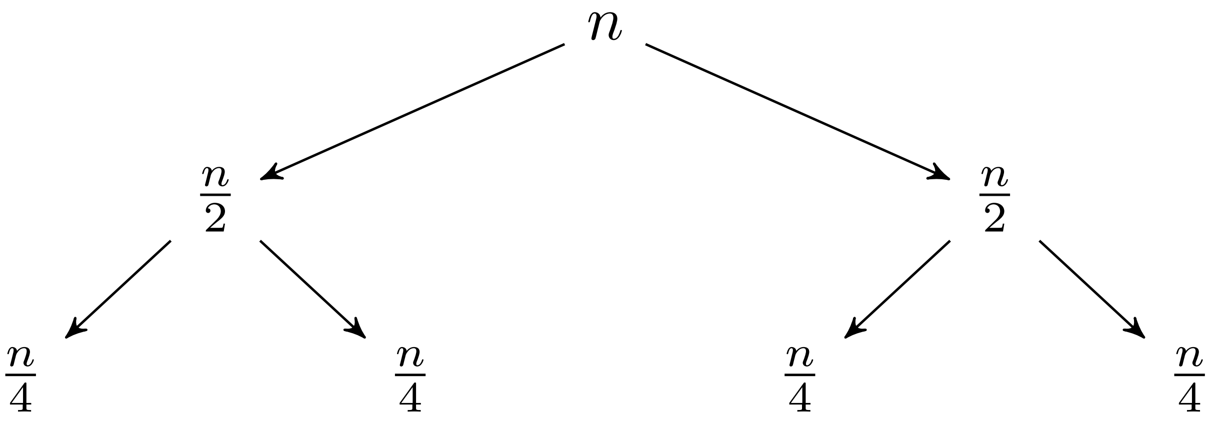


Level	Number of Nodes at level	Work per Node	Work per Level
0			
1			
2			
i			
base			

Last recursive level:

Technique 2: Tree method

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

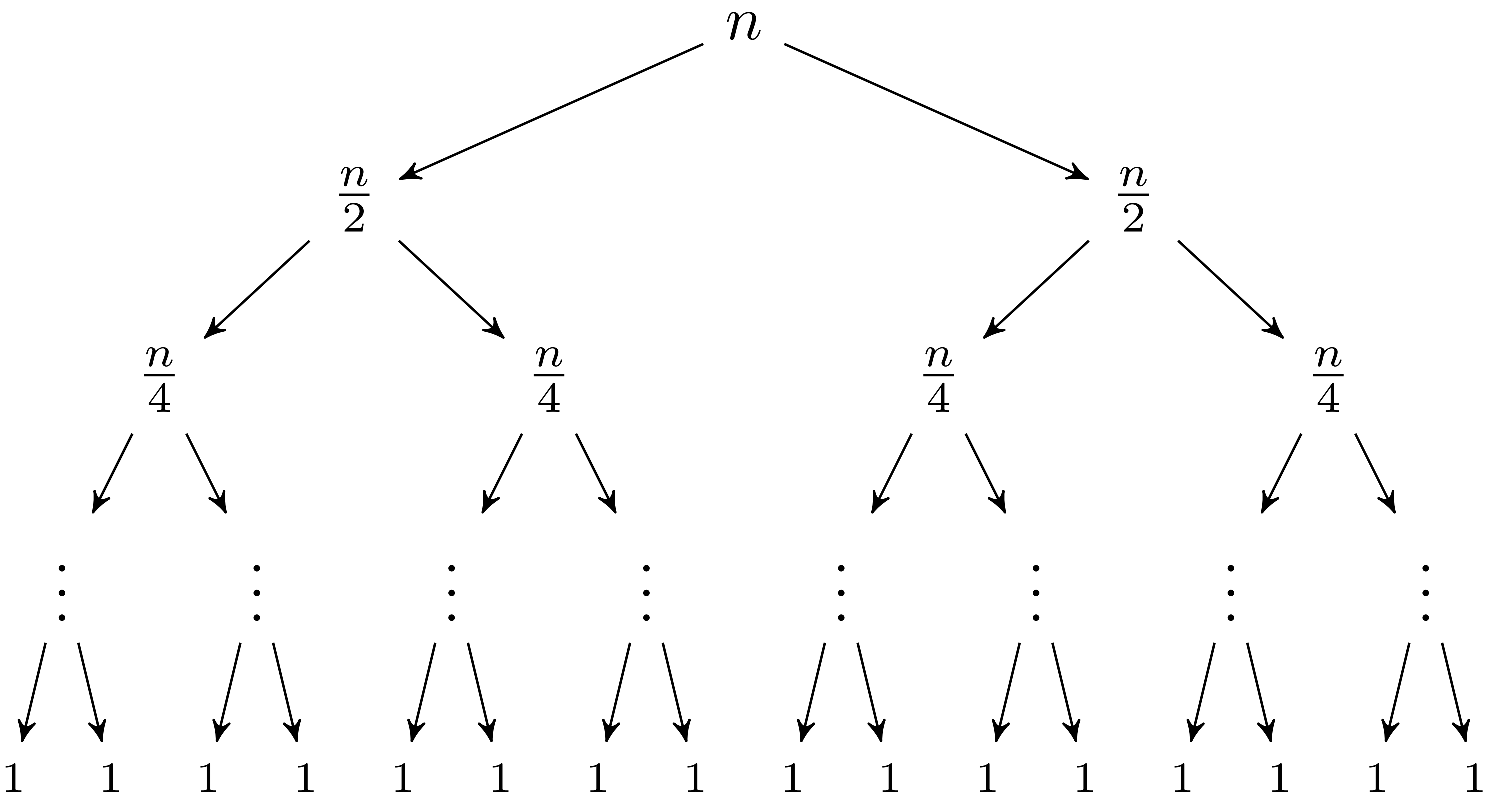


Level	Number of Nodes at level	Work per Node	Work per Level
0			
1			
2			
i			
base			

Last recursive level:

Technique 2: Tree method

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

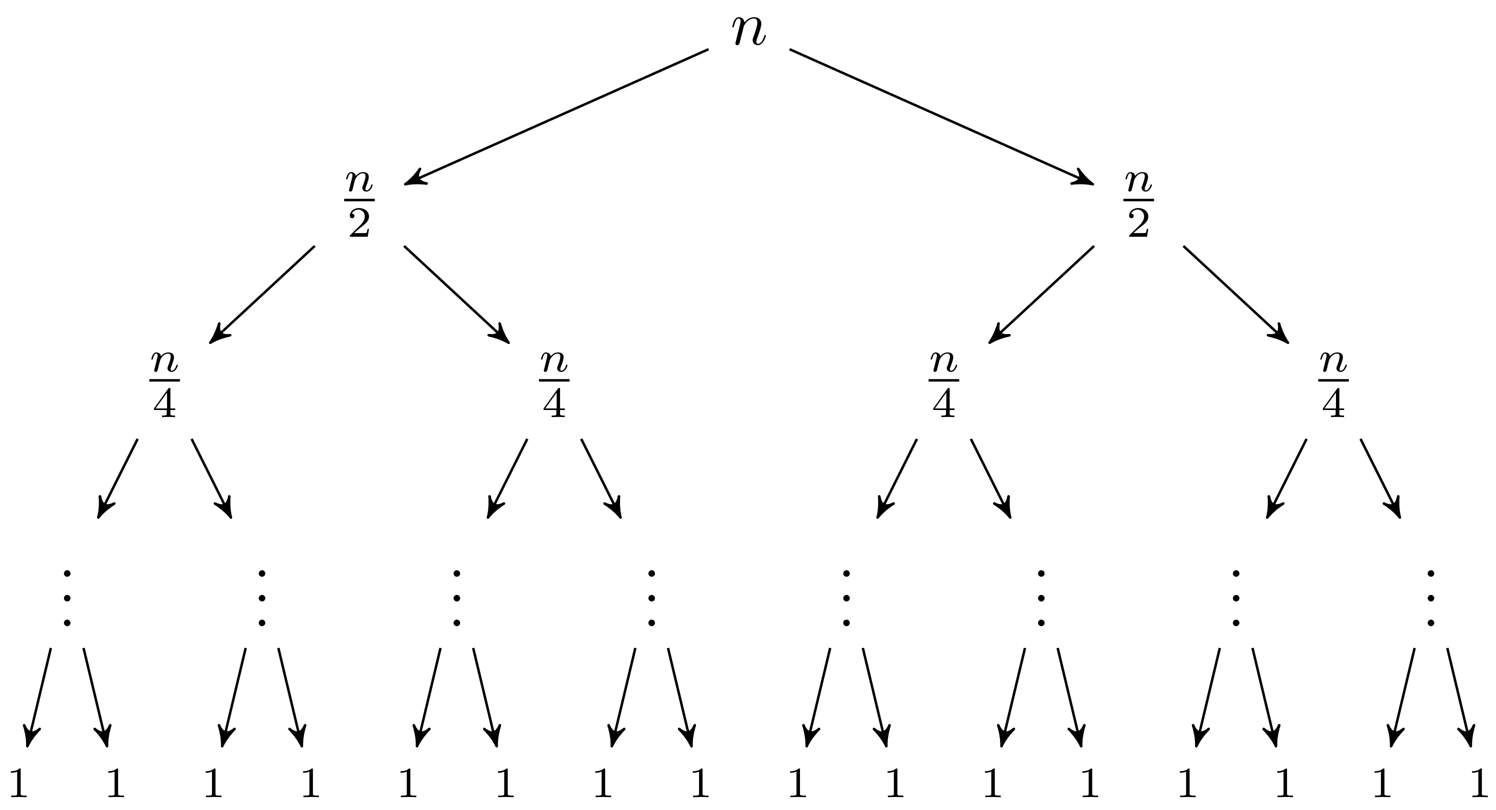


Level	Number of Nodes at level	Work per Node	Work per Level
0			
1			
2			
i			
base			

Last recursive level:

Technique 2: Tree method

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



Level	Number of Nodes at level	Work per Node	Work per Level
0	1	n	n
1	2	$\frac{n}{2}$	n
2	4	$\frac{n}{4}$	n
i	2^i	$\frac{n}{2^i}$	n
base	$2^{\log_2 n}$	1	n

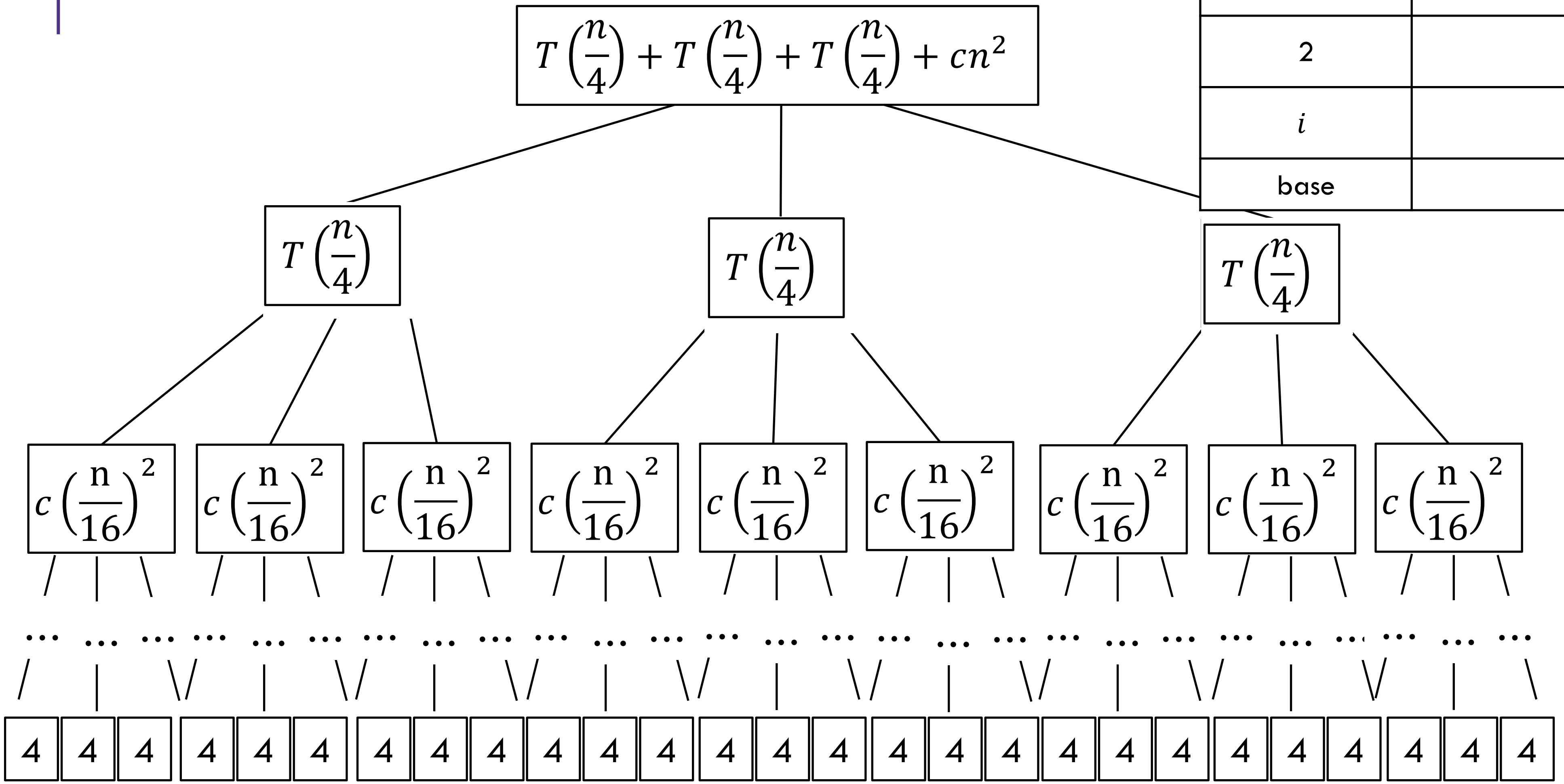
Last recursive level: $\log n - 1$

Combining it all together…

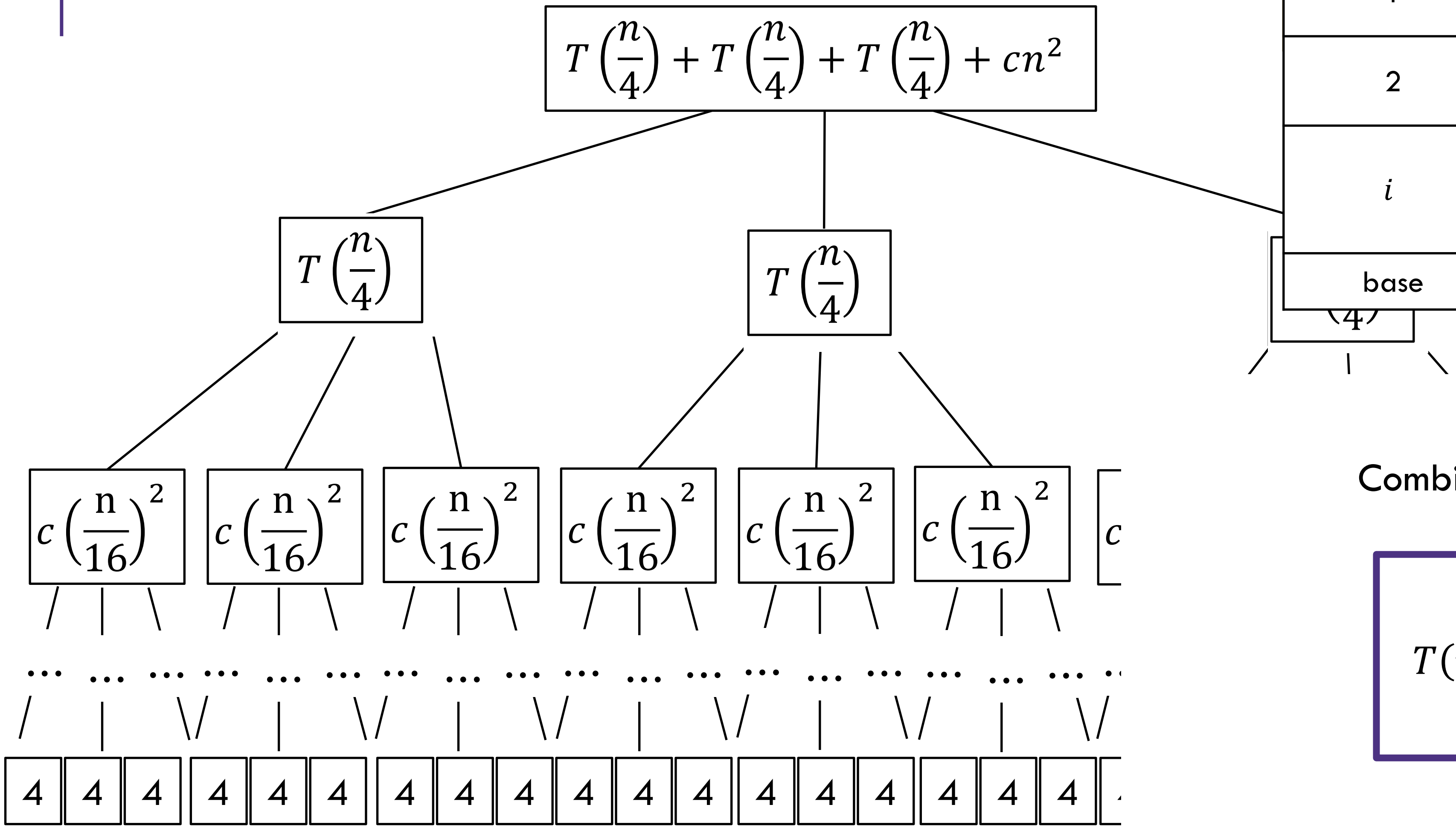
$$T(n) = n + \sum_{i=0}^{\log_2 n - 1} n = n + n \log n$$

Tree Method Practice

Level (i)	Number of Nodes	Work per Node	Work per Level
0	1		
1			
2			
i			
base			



Tree Method Practice



Level (i)	Number of Nodes	Work per Node	Work per Level
0	1	cn^2	cn^2
1	3	$c\left(\frac{n}{4}\right)^2$	$\frac{3}{16}cn^2$
2	9	$c\left(\frac{n}{16}\right)^2$	$\frac{9}{256}cn^2$
i	3^i	$c\left(\frac{n}{4^i}\right)^2$	$\left(\frac{3}{16}\right)^i cn^2$
base	$3^{\log_4 n}$	4	$4 \cdot 3^{\log_4 n}$

Last recursive level: $\log_4 n - 1$

Combining it all together…

$$T(n) = 4\, n^{\log_4 3} + \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2$$

Technique 3: Master Theorem

Given a recurrence of the following form:

$$T(n) = \begin{cases} d & when\ n = 1 \\ aT\left(\frac{n}{b}\right) + n^c & otherwise \end{cases}$$

Where a, b, and c are constants, then $T(n)$ has the following asymptotic bounds

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log_2 n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

Apply Master Theorem

Given a recurrence of the form:

$$T(n) = \begin{cases} d \text{ } when \text{ } n = 1 \\ aT\left(\frac{n}{b}\right) + n^c \text{ } otherwise \end{cases}$$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log_2 n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

$$T(n) = \begin{cases} 1 \text{ } when \text{ } n \le 1 \\ 2T\left(\frac{n}{2}\right) + n \text{ } otherwise \end{cases}$$

a = 2
b = 2
c = 1
d = 1

$$\log_b a = c \Rightarrow \log_2 2 = 1$$

$$T(n) \in \Theta(n^c \log_2 n) \Rightarrow \Theta(n^1 \log_2 n)$$

Reflecting on Master Theorem

Given a recurrence of the form:

$$T(n) = \begin{cases} d \text{ when } n = 1 \\ aT\left(\frac{n}{b}\right) + n^c \text{ otherwise} \end{cases}$$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log_2 n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

$height \approx \log_b a$

$branchWork \approx n^c \log_b a$

$leafWork \approx d\left(n^{\log_b a}\right)$

The $\log_b a < c$ case

- Recursive case conquers work more quickly than it divides work
- Most work happens near "top" of tree
- Non recursive work in recursive case dominates growth, n^c term

The $\log_b a = c$ case

- Work is equally distributed across call stack (throughout the "tree")
- Overall work is approximately work at top level x height

The $\log_b a > c$ case

- Recursive case divides work faster than it conquers work
- Most work happens near "bottom" of tree
- Leaf work dominates branch work

Recurrence analysis techniques

1. Unfolding method
   - more of a brute force method
   - Tedious but works
2. Tree methods
   - more scratch work but less error prone
3. Master theorem
   - quick, but applicable only to certain type of recurrences
   - does not give a closed form (gives big-Theta)