CSE 373: More sorts, tree method, the master method

Michael Lee
Wednesday, Feb 7, 2018
Divide-and-conquer is a useful technique for solving many kinds of problems. It consists of the following steps:
Divide-and-conquer is a useful technique for solving many kinds of problems. It consists of the following steps:

1. *Divide* your work up into smaller pieces (recursively)
Technique: Divide-and-Conquer

Divide-and-conquer is a useful technique for solving many kinds of problems. It consists of the following steps:

1. *Divide* your work up into smaller pieces (recursively)
2. *Conquer* the individual pieces (as base cases)
Technique: Divide-and-Conquer

Divide-and-conquer is a useful technique for solving many kinds of problems. It consists of the following steps:

1. *Divide* your work up into smaller pieces (recursively)
2. *Conquer* the individual pieces (as base cases)
3. *Combine* the results together (recursively)
Divide-and-conquer is a useful technique for solving many kinds of problems. It consists of the following steps:

1. **Divide** your work up into smaller pieces (recursively)
2. **Conquer** the individual pieces (as base cases)
3. **Combine** the results together (recursively)

**Example template**

```plaintext
algorithm(input) {
    if (small enough) {
        CONQUER, solve, and return input
    } else {
        DIVIDE input into multiple pieces
        RECURSE on each piece
        COMBINE and return results
    }
}
```
Merge sort: Core pieces

Divide:

Unsorted
**Merge sort: Core pieces**

**Divide:** Split array roughly into half

```
Unsorted

Unsorted  Unsorted
```

**Conquer:** Return array when length ≤ 1

**Combine:** Combine two sorted arrays using merge
Merge sort: Core pieces

**Divide:** Split array roughly into half

- Unsorted
- Unsorted

**Conquer:**

- Sorted
- Sorted
- Sorted

3
**Divide:** Split array roughly into half

- Unsorted

**Conquer:** Return array when length $\leq 1$

- Unsorted
Merge sort: Core pieces

**Divide:** Split array roughly into half

- Unsorted

**Conquer:** Return array when length $\leq 1$

**Combine:**

- Sorted
- Sorted
Merge sort: Core pieces

**Divide:** Split array roughly into half

- Unsorted
  - Unsorted
  - Unsorted

**Conquer:** Return array when length $\leq 1$

**Combine:** Combine two sorted arrays using merge

- Sorted
  - Sorted
  - Sorted
Core idea: split array in half, sort each half, merge back together. If the array has size 0 or 1, just return it unchanged.

**Pseudocode**

```plaintext
sort(input) {
    if (input.length < 2) {
        return input;
    } else {
        smallerHalf = sort(input[0, ..., mid]);
        largerHalf = sort(input[mid + 1, ...]);
        return merge(smallerHalf, largerHalf);
    }
}
```
Merge sort: Example

\[
\begin{array}{cccccccc}
5 & 10 & 7 & 2 & 3 & 6 & 2 & 11 \\
\end{array}
\]
Merge sort: Example

```
5  10  7  2  3  6  2  11
```

```
5  10  7  2  
```

```
3  6  2  11
```

```
```
### Merge sort: Example

<table>
<thead>
<tr>
<th>5</th>
<th>10</th>
<th>7</th>
<th>2</th>
<th>3</th>
<th>6</th>
<th>2</th>
<th>11</th>
</tr>
</thead>
</table>

#### First level of merge:

<table>
<thead>
<tr>
<th>5</th>
<th>10</th>
<th>7</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a[0] )</td>
<td>( a[1] )</td>
<td>( a[2] )</td>
<td>( a[3] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>6</th>
<th>2</th>
<th>11</th>
</tr>
</thead>
</table>

#### Second level of merge:

<table>
<thead>
<tr>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a[0] )</td>
<td>( a[1] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a[2] )</td>
<td>( a[3] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a[4] )</td>
<td>( a[5] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a[6] )</td>
<td>( a[7] )</td>
</tr>
</tbody>
</table>
Merge sort: Example
Merge sort: Example
Merge sort: Example

5 10
a[0] a[1]

2 7

3 6

2 11

5 10 7 2

3 6 2 11
Merge sort: Example
Merge sort: Example

2  2  3  5  6  7  10  11

2  5  7  10

5  10
a[0] a[1]

2  7

3  6

2  11
Pseudocode

sort(input) {
    if (input.length < 2) {
        return input;
    } else {
        smallerHalf = sort(input[0, ..., mid]);
        largerHalf = sort(input[mid + 1, ...]);
        return merge(smallerHalf, largerHalf);
    }
}

Best case runtime?                      Worst case runtime?
Best and worst case

We always subdivide the array in half on each recursive call, and merge takes $O(n)$ time to run. So, the best and worst case runtime is the same:

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

Best and worst case

We always subdivide the array in half on each recursive call, and merge takes $O(n)$ time to run. So, the best and worst case runtime is the same:

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

Spoiler alert: this is $\Theta(n \log(n))$
Quick sort: Divide step

<table>
<thead>
<tr>
<th>6</th>
<th>10</th>
<th>7</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>2</th>
<th>11</th>
</tr>
</thead>
</table>
Quick sort: Divide step

6 10 7 2 3 5 2 11

Numbers ≤ pivot:
6 5 3 2

Numbers > pivot:
10 11 7

Pivot: 6
Quick sort: Divide step

Numbers ≤ pivot

Pivot

6 10 7 2 3 5 2 11

Quick sort: Divide step

Numbers \leq \text{pivot}

Numbers > \text{pivot}
Quick sort: Core pieces

**Divide**: Pick a pivot, partition into groups

- **P**
- Unsorted
- $\leq P$
- $> P$

- Return array when length $\leq 1$

**Combine**: Combine sorted portions and the pivot
Quick sort: Core pieces

**Divide:** Pick a pivot, partition into groups

![Diagram](image)

- Unsorted
  - \( \leq P \)
  - \( > P \)

**Conquer:**
Quick sort: Core pieces

**Divide:** Pick a pivot, partition into groups

- $P$
- $\leq P$
- $> P$

**Conquer:** Return array when length $\leq 1$
Quick sort: Core pieces

**Divide:** Pick a pivot, partition into groups

- \[ P \]
- Unsorted
  - \[ \leq P \]
  - \[ > P \]

**Conquer:** Return array when length \( \leq 1 \)

- \[ \leq P \]
- \[ P \]
- \[ > P \]

**Combine:**
Quick sort: Core pieces

**Divide:** Pick a pivot, partition into groups

- **Unsorted**
  - \( \leq P \)
  - \( > P \)

**Conquer:** Return array when length \( \leq 1 \)

**Combine:** Combine sorted portions and the pivot

- \( \leq P \)
- \( P \)
- \( > P \)
- **Sorted**
Quick sort: Summary

Core idea: Pick some item from the array and call it the **pivot**. Put all items **smaller** in the pivot into one group and all items **larger** in the other and recursively sort. If the array has size 0 or 1, just return it unchanged.

**Pseudocode**

```plaintext
sort(input) {
    if (input.length < 2) {
        return input;
    } else {
        pivot = getPivot(input);
        smallerHalf = sort(getSmaller(pivot, input));
        largerHalf = sort(getBigger(pivot, input));
        return smallerHalf + pivot + largerHalf;
    }
}
```
Quick sort: Example

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>50</td>
<td>70</td>
<td>10</td>
<td>60</td>
<td>40</td>
<td>30</td>
</tr>
</tbody>
</table>
Quick sort: Example

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>50</td>
<td>70</td>
<td>10</td>
<td>60</td>
<td>40</td>
<td>30</td>
</tr>
</tbody>
</table>
Quick sort: Example

```
20  50  70  10  60  40  30
```

```
10
```

```
50  70  60  40  30
```

```
a[0]  a[0]  a[0]  a[0]  a[0]  a[0]
```
Quick sort: Example
Quick sort: Example

20, 50, 70, 10, 60, 40, 30

10, 50, 70, 60, 40, 30

40, 30

70, 60

10, 50, 70, 60, 40, 30
Quick sort: Example

20 50 70 10 60 40 30

10

50 70 60 40 30

40 30

70 60
Quick sort: Example

```
[20 50 70 10 60 40 30]
```

```
[10]
[a[0]]
```

```
[50 70 60 40 30]
```

```
[40 30]
[a[0] a[1]]
```

```
[70 60]
[a[0] a[1]]
```

```
[30 60]
[a[0] a[0]]
```
Quick sort: Example

20

10

40

30

70

60

40

30
Quick sort: Example

10

20

30

40

50

60

70


30

60

30

60

a[0]

a[0]
### Quick sort: Example

<table>
<thead>
<tr>
<th></th>
<th>50</th>
<th>70</th>
<th>10</th>
<th>60</th>
<th>40</th>
<th>30</th>
</tr>
</thead>
</table>
Quick sort: Example
Quick sort: Analysis

**Pseudocode**

```plaintext
sort(input) {
    if (input.length < 2) {
        return input;
    } else {
        pivot = getPivot(input);
        smallerHalf = sort(getSmaller(pivot, input));
        largerHalf = sort(getBigger(pivot, input));
        return smallerHalf + pivot + largerHalf;
    }
}
```

Best case runtime?  
Worst case runtime?
Quick sort: Analysis

Best case analysis

In the **best** case, we always pick the **median** element.

\[ T(n) = \begin{cases} 
2T(n/2) + n & \text{if } n > 1 \\
1 & \text{otherwise} 
\end{cases} \]
### Best case analysis

In the **best** case, we always pick the **median** element.

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

(Spoiler alert: this is \(\Theta(n \log(n))\))
Worst case analysis

In the **worst** case, we always end up picking the **minimum** or **maximum** element.

\[
T(n) = \begin{cases} 
T(n - 1) + n & \text{if } n > 1 \\
1 & \text{otherwise}
\end{cases}
\]

So, the worst-case runtime is \( \Theta(n^2) \).
### Best case analysis

In the **best** case, we always pick the **median** element, so the best-case runtime is $\Theta (n \log(n))$.

### Worst case analysis

In the **worst** case, we always end up picking the **minimum** or **maximum** element, so, the worst-case runtime is $\Theta (n^2)$.

### Average case runtime

Usually, we’ll pick a **random** element, which makes the runtime $\Theta (n \log(n))$. 
Quick sort: Unresolved questions

How do we pick a pivot?

How do we partition?
How do we pick a pivot?

How do we partition?
Quick sort: Unresolved questions

How do we pick a pivot?

- Worst case? Pick the minimum or the maximum. The work will shrink by only 1 on each recursive call.

How do we partition?
Quick sort: Unresolved questions

How do we pick a pivot?

- Worst case? Pick the minimum or the maximum. The work will shrink by only 1 on each recursive call.
- Ideally? Pick the median. The work will split in half on each recursive call.

How do we partition?
Quick sort: Picking a pivot

How do we find the median?

Idea: pick the first item in the array

Problem: what if the array is already sorted?

(Real world data often is partially sorted)

But hey, it's speedy (\( O(1) \))

Idea: try finding it by looping through the array

Problem: hard to implement, and expensive (\( O(n) \))

These seem like bad ideas :(
Quick sort: Picking a pivot

How do we find the median?

▶ Idea: pick the first item in the array
  ▶ Problem: what if the array is already sorted?
  ▶ (Real world data often is partially sorted)
  ▶ But hey, it’s speedy ($O(1)$)
Quick sort: Picking a pivot

How do we find the median?

- Idea: pick the first item in the array
  - Problem: what if the array is already sorted?
  - (Real world data often is partially sorted)
  - But hey, it’s speedy ($O(1)$)
- Idea: try finding it by looping through the array
Quick sort: Picking a pivot

How do we find the median?

- Idea: pick the first item in the array
  - Problem: what if the array is already sorted?
  - (Real world data often is partially sorted)
  - But hey, it’s speedy ($O(1)$)

- Idea: try finding it by looping through the array
  - Problem: hard to implement, and expensive ($O(n)$)
How do we find the median?

- Idea: pick the first item in the array
  - Problem: what if the array is already sorted?
  - (Real world data often is partially sorted)
  - But hey, it’s speedy ($O(1)$)

- Idea: try finding it by looping through the array
  - Problem: hard to implement, and expensive ($O(n)$)

These seem like bad ideas :(
Other ideas:
Quick sort: Picking a pivot

Other ideas:

- Idea: pick a random element

...but works well in practice, and is efficient

These seem like good ideas :)

22
Quick sort: Picking a pivot

Other ideas:

- Idea: pick a random element
  - On average, guaranteed to do well – no easy worst case
  - Random number generation can sometimes be expensive/fraught with peril

These seem like good ideas!
Other ideas:

- Idea: pick a random element
  - On average, guaranteed to do well – no easy worst case
  - Random number generation can sometimes be expensive/fraught with peril
- Idea: pick the median of first, middle, and last
Quick sort: Picking a pivot

Other ideas:

- Idea: pick a random element
  - On average, guaranteed to do well – no easy worst case
  - Random number generation can sometimes be expensive/fraught with peril
- Idea: pick the median of first, middle, and last
  - Adversary could still construct malicious input
  - ...but works well in practice, and is efficient
Quick sort: Picking a pivot

Other ideas:

▶ Idea: pick a random element
  ▶ On average, guaranteed to do well – no easy worst case
  ▶ Random number generation can sometimes be expensive/fraught with peril
▶ Idea: pick the median of first, middle, and last
  ▶ Adversary could still construct malicious input
  ▶ ...but works well in practice, and is efficient

These seem like good ideas :}
Quick sort: Unresolved questions

How do we pick a pivot?

How do we partition?
Find the lo, med, and hi

8 1 4 9 0 3 5 2 7 6

Final result: pivot is now at index 0

6 1 4 9 0 3 5 2 7 8

Quick sort: Partitioning (using median-of-three pivot)

Find the lo, med, and hi

![Array diagram showing partitioning process]

Find the median of the three and swap with front

![Updated array diagram with pivot at index 0]
Quick sort: Partitioning (using median-of-three pivot)

Find the lo, med, and hi

Find the median of the three and swap with front

Final result: pivot is now at index 0
Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

Partitioning:

low
1 ≤ 6

high
8 > 6
Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

![Array diagram]

Partitioning:

![Partitioning diagram]
Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

Partitioning:

low
9 \leq 6

high
2 > 6
Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

<table>
<thead>
<tr>
<th>6</th>
<th>1</th>
<th>4</th>
<th>9</th>
<th>0</th>
<th>3</th>
<th>5</th>
<th>2</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>

Partitioning:

<table>
<thead>
<tr>
<th>6</th>
<th>1</th>
<th>4</th>
<th>9</th>
<th>0</th>
<th>3</th>
<th>5</th>
<th>2</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>

SWAP

low: 9 ≤ 6

high: 2 > 6
Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

Partitioning:

low
2 \leq 6

high
9 > 6
Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

Partitioning:
Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

```
<table>
<thead>
<tr>
<th>6</th>
<th>1</th>
<th>4</th>
<th>9</th>
<th>0</th>
<th>3</th>
<th>5</th>
<th>2</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>
```

Partitioning:

```
<table>
<thead>
<tr>
<th>6</th>
<th>1</th>
<th>4</th>
<th>9</th>
<th>0</th>
<th>3</th>
<th>5</th>
<th>2</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>
```

low: 3 ≤ 6  
high: 5 > 6
Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

```
6 1 4 9 0 3 5 2 7 8
```

Partitioning:

```
6 1 4 2 0 3 5 9 7 8
```

low \( \leq \) 6 \hspace{1cm} 5 \( > \) 6
Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

Partitioning:

low = 9 ≤ 6
high = 5 > 6
Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

Partitioning:

SWAP

high

low

5 > 6

9 ≤ 6
Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

Partitioning:

Unsorted $\leq 6$

Unsorted $> 6$
Quick sort: Core pieces revisited

Divide: Pick a pivot, partition in-place into groups

Unsorted

\[ \leq P \quad P \quad > P \]
Quick sort: Core pieces revisited

**Divide:** Pick a pivot, partition in-place into groups

- Unsorted
- \(\leq P\)
- \(P\)
- \(> P\)

**Conquer:** When subarray is length \(\leq 1\), do nothing
Quick sort: Core pieces revisited

**Divide:** Pick a pivot, partition in-place into groups

![Partition Diagram]

**Conquer:** When subarray is length \( \leq 1 \), do nothing

**Combine:** Do nothing; already done!
So, merge sort and quick sort are both:

\[
T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases}
\]
So, merge sort and quick sort are both:

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

I claim \( T(n) \in \Theta(n\log(n)) \). How can we show this?
We could try unfolding, but it’s annoying:
We could try unfolding, but it’s annoying:

\[ T(n) = n + 2T\left(\frac{n}{2}\right) \]
We could try unfolding, but it’s annoying:

\[
T(n) = n + 2T \left( \frac{n}{2} \right) \\
= n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right)
\]
We could try unfolding, but it’s annoying:

\[ T(n) = n + 2T \left( \frac{n}{2} \right) \]
\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right) \]
\[ = n + 2 \left( \frac{n}{2} + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right) \right) \]
Analyzing recurrences, part 2

We could try unfolding, but it’s annoying:

\[ T(n) = n + 2T \left( \frac{n}{2} \right) \]
\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right) \]
\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right) \]
\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} + 2T \left( \frac{n}{8} \right) \right) \right) \]
We could try unfolding, but it's annoying:

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

\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right) \]

\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right) \]

\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right) \]

\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} + 2T \left( \frac{n}{8} \right) \right) \right) \]

\[ = n + n + 4T \left( \frac{n}{4} + 2T \left( \frac{n}{8} \right) \right) \]
We could try unfolding, but it’s annoying:

\[ T(n) = n + 2T \left( \frac{n}{2} \right) \]
\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right) \]
\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right) \]
\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right) \]
\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} + 2T \left( \frac{n}{8} \right) \right) \right) \]
\[ = n + n + 4T \left( \frac{n}{4} + 2T \left( \frac{n}{8} \right) \right) \]
\[ = n + n + n + 8T \left( \frac{n}{8} \right) \]
We could try unfolding, but it’s annoying:

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

\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right) \]

\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right) \]

\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} \right) \right) \]

\[ = n + 2 \left( \frac{n}{2} + 2T \left( \frac{n}{4} + 2T \left( \frac{n}{8} \right) \right) \right) \]

\[ = n + n + 4T \left( \frac{n}{4} + 2T \left( \frac{n}{8} \right) \right) \]

\[ = n + n + n + 8T \left( \frac{n}{8} \right) \]

\[ = n + n + \cdots + n + n \]

about \( \log(n) \) times
We could try unfolding, but it’s annoying:

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

\[
= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right)
\]

\[
= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right)
\]

\[
= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right)
\]

\[
= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4} + 2T\left(\frac{n}{8}\right)\right)\right)
\]

\[
= n + n + 4T\left(\frac{n}{4} + 2T\left(\frac{n}{8}\right)\right)
\]

\[
= n + n + 8T\left(\frac{n}{8}\right)
\]

\[
= \underbrace{n + n + \cdots + n + n}_{\text{about } \log(n) \text{ times}}
\]

\[
= n \log(n)
\]
The tree method: overview

Core idea:

1. Draw what the work looks like visually, as a tree
Core idea:

1. Draw what the work looks like visually, as a tree
2. Use the visualization to help us analyze the overall behavior
The tree method: overview

Core idea:

1. Draw what the work looks like visually, as a tree
2. Use the visualization to help us analyze the overall behavior
3. Either find the closed form, or construct a summation that we can simplify to get the closed form
The tree method: example

Step 1: Start with the function, let $n$ be the input value

$T(n)$
Step 2: Replace with definition

$$T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n$$
Step 3: Stick each recursive call into a *subtree*

```
  n
 / \
T(n/2) T(n/2)
```

Final step: how much work does each base case do?

\[
n, n^2, n^4, \ldots, 1
\]
The tree method: example

Step 4: Replace with definition

\[ T \left( \frac{n}{4} \right) + T \left( \frac{n}{4} \right) + \frac{n}{2} \]
The tree method: example

Repeat step 3 (move recursive call to subtrees):

```
      n
    /   \
  n/2   n/2
 /    /    \
T(n/4) T(n/4) T(n/4) T(n/4)
```
The tree method: example

Repeat step 4 (replace recursive call with definition):

\[
\begin{align*}
2T\left(\frac{n}{8}\right) + \frac{n}{4} & & 2T\left(\frac{n}{8}\right) + \frac{n}{4} & & 2T\left(\frac{n}{8}\right) + \frac{n}{4} & & 2T\left(\frac{n}{8}\right) + \frac{n}{4}
\end{align*}
\]
The tree method: example

Repeat...

\[
\frac{n}{2} \quad \frac{n}{2} \\
\frac{n}{4} \quad \frac{n}{4} \\
\frac{n}{4} \quad \frac{n}{4} \\
\vdots \quad \vdots \\
\vdots \quad \vdots
\]
Final step: how much work does each base case do?
The tree method: analysis

Now, let’s add everything up!
Now, let’s add everything up!

How much work is done per level?

$\begin{align*}
\text{Height is roughly } & \log_2(n), \text{ so total work is about } n \log_2(n). \\
\end{align*}$
The tree method: analysis

Now, let’s add everything up!

How much work is done per level?

Height is roughly $\log_2(n)$, so total work is about $n \log_2(n)$. 

31
Now, let’s add everything up!

How much work is done per level?

Height is roughly $\log_2(n)$, so total work is about $n \log_2(n)$. 

$n$ work

$n$ work

$n$ work

$n$ work

$n$ work

$n$ work

$n$ work

$n$ work

$n$ work
Now, let’s add everything up!

How much work is done per level?

Height is roughly $\log_2(n)$, so total work is about $n \log_2(n)$. 
Consider the following recurrence:

\[ S(n) = \begin{cases} 
  2 & \text{if } n \leq 1 \\
  2S(n/3) + n^2 & \text{otherwise} 
\end{cases} \]
Consider the following recurrence:

\[
S(n) = \begin{cases} 
2 & \text{if } n \leq 1 \\
2S\left(\frac{n}{3}\right) + n^2 & \text{otherwise}
\end{cases}
\]

Draw a tree to help you visualize the work done.
Step 1: Start with the function, let $n$ be the input value

$$S(n)$$
Step 2: Replace with definition

\[ S \left( \frac{n}{3} \right) + S \left( \frac{n}{3} \right) + n^2 \]
Step 3: Stick each recursive call into a subtree

\[ n^2 \]

\[ S \left( \frac{n}{3} \right) \quad S \left( \frac{n}{3} \right) \]
The tree method: practice

Step 4: Replace with definition

\[ n^2 \]

\[ S\left(\frac{n}{9}\right) + S\left(\frac{n}{9}\right) + \frac{n^2}{9} \]

\[ S\left(\frac{n}{9}\right) + S\left(\frac{n}{9}\right) + \frac{n^2}{9} \]
The tree method: practice

Repeat step 3 (move recursive call to subtrees):

\[
\begin{align*}
\text{n}^2 & \\
\frac{n^2}{9} & \quad \frac{n^2}{9} \\
S\left(\frac{n}{9}\right) & \quad S\left(\frac{n}{9}\right) & \quad S\left(\frac{n}{9}\right) & \quad S\left(\frac{n}{9}\right)
\end{align*}
\]
Repeat step 4 (replace recursive call with definition):

\[
\begin{align*}
\text{n}^2 \\
\text{n}^2/9 \\
2S\left(\frac{n}{27}\right) + \frac{n^2}{81} \\
2S\left(\frac{n}{27}\right) + \frac{n^2}{81} \\
2S\left(\frac{n}{27}\right) + \frac{n^2}{81} \\
2S\left(\frac{n}{27}\right) + \frac{n^2}{81}
\end{align*}
\]
The tree method: practice

Repeat...

\[
\begin{array}{c}
\text{n}^2 \\
\downarrow \\
\text{\(\frac{n^2}{9}\)} & \text{\(\frac{n^2}{9}\)} \\
\downarrow & \downarrow \\
\text{\(\frac{n^2}{81}\)} & \text{\(\frac{n^2}{81}\)} & \text{\(\frac{n^2}{81}\)} & \text{\(\frac{n^2}{81}\)} \\
\downarrow & \downarrow & \downarrow & \downarrow \\
\vdots & \vdots & \vdots & \vdots \\
\end{array}
\]
Final step: how much work does each base case do?

\[
\begin{align*}
    n^2 & \quad \rightarrow \quad \frac{n^2}{9} \\
    \frac{n^2}{9} & \quad \rightarrow \quad \frac{n^2}{81} \quad \rightarrow \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2
\end{align*}
\]
Final step: how much work does each base case do?

Now what?
Problem: Need a rigorous way of getting a closed form
The tree method: precise analysis

**Problem:** Need a rigorous way of getting a closed form

We want to answer a few core questions:

1. How many nodes are there on level \( i \)? (\( i = 0 \) is "root" level)
2. At some level \( i \), how much work does a single node do? (Ignoring subtrees)
3. How many recursive levels are there?
4. How much work does the leaf level (base cases) do?

1. How much work does a single leaf node do?
2. How many leaf nodes are there?
The tree method: precise analysis

**Problem:** Need a rigorous way of getting a closed form

We want to answer a few core questions:

**How much work does each recursive level do?**

1. How many nodes are there on level $i$? ($i = 0$ is "root" level)
2. At some level $i$, how much work does a single node do? (Ignoring subtrees)
3. How many recursive levels are there?
4. How much work does the leaf level (base cases) do?
   1. How much work does a single leaf node do?
   2. How many leaf nodes are there?
The tree method: precise analysis

**Problem:** Need a rigorous way of getting a closed form

We want to answer a few core questions:

**How much work does each recursive level do?**

1. How many nodes are there on level $i$? ($i = 0$ is “root” level)
The tree method: precise analysis

**Problem:** Need a rigorous way of getting a closed form

We want to answer a few core questions:

**How much work does each recursive level do?**

1. How many nodes are there on level $i$? ($i = 0$ is “root” level)
2. At some level $i$, how much work does a *single* node do? (Ignoring subtrees)
**Problem:** Need a rigorous way of getting a closed form

We want to answer a few core questions:

**How much work does each recursive level do?**

1. How many nodes are there on level $i$? ($i = 0$ is “root” level)
2. At some level $i$, how much work does a *single* node do? (Ignoring subtrees)
3. How many recursive levels are there?

**How much work does the leaf level (base cases) do?**

1. How much work does a single leaf node do?
2. How many leaf nodes are there?
The tree method: precise analysis

**Problem:** Need a rigorous way of getting a closed form

We want to answer a few core questions:

**How much work does each recursive level do?**

1. How many nodes are there on level $i$? ($i = 0$ is “root” level)
2. At some level $i$, how much work does a *single* node do? (Ignoring subtrees)
3. How many recursive levels are there?

**How much work does the leaf level (base cases) do?**
The tree method: precise analysis

**Problem:** Need a rigorous way of getting a closed form

We want to answer a few core questions:

**How much work does each recursive level do?**

1. How many nodes are there on level $i$? ($i = 0$ is “root” level)
2. At some level $i$, how much work does a *single* node do? (Ignoring subtrees)
3. How many recursive levels are there?

**How much work does the leaf level (base cases) do?**

1. How much work does a single leaf node do?
The tree method: precise analysis

Problem: Need a rigorous way of getting a closed form

We want to answer a few core questions:

How much work does each recursive level do?

1. How many nodes are there on level \( i \)? (\( i = 0 \) is “root” level)
2. At some level \( i \), how much work does a single node do? (Ignoring subtrees)
3. How many recursive levels are there?

How much work does the leaf level (base cases) do?

1. How much work does a single leaf node do?
2. How many leaf nodes are there?
**Problem:** Need a rigorous way of getting a closed form

We want to answer a few core questions:

**How much work does each recursive level do?**

1. How many nodes are there on level $i$? ($i = 0$ is “root” level)
2. At some level $i$, how much work does a *single* node do? (Ignoring subtrees)
3. How many recursive levels are there?

**How much work does the leaf level (base cases) do?**

1. How much work does a single leaf node do?
2. How many leaf nodes are there?
The tree method: precise analysis

\begin{figure}
\centering
\includegraphics[width=\textwidth]{treeDiagram.png}
\end{figure}

1. \text{numNodes}(i) = 2^i
2. \text{workPerNode}(n, i) = n^{2^i}
3. \text{numLevels}(n) = ?
4. \text{workPerLeafNode}(n) = 1
5. \text{numLeafNodes}(n) = ?
The tree method: precise analysis

1. \( \text{numNodes}(i) = ? \)
2. \( \text{workPerNode}(n, i) = ? \)
3. \( \text{numLevels}(n) = ? \)
4. \( \text{workPerLeafNode}(n) = ? \)
5. \( \text{numLeafNodes}(n) = ? \)
The tree method: precise analysis

1. $\text{numNodes}(i) = ?$
2. $\text{workPerNode}(n, i) = ?$
3. $\text{numLevels}(n) = ?$
4. $\text{workPerLeafNode}(n) = ?$
5. $\text{numLeafNodes}(n) = ?$

1 node, $n$ work per
2 nodes, $\frac{n}{2}$ work per
4 nodes, $\frac{n}{4}$ work per
$2^i$ nodes, $\frac{n}{i}$ work per
$2^h$ nodes, 1 work per
The tree method: precise analysis

1. \( \text{numNodes}(i) = 2^i \)
2. \( \text{workPerNode}(n, i) = \frac{n}{2^i} \)
3. \( \text{numLevels}(n) = ? \)
4. \( \text{workPerLeafNode}(n) = 1 \)
5. \( \text{numLeafNodes}(n) = ? \)
The tree method: precise analysis

How many levels are there, exactly? Is it $\log_2(n)$?
How many levels are there, exactly? Is it \( \log_2(n) \)?

Let’s try an example. Suppose we have \( T(4) \). What happens?
How many levels are there, exactly? Is it $\log_2(n)$?

Let’s try an example. Suppose we have $T(4)$. What happens?

![Tree Diagram]

- Height is $\log_2(4) = 2$.
- For this recursive function, the number of recursive levels is the same as the height.
- Important: total levels, counting the base case, is the height + 1.
- Important: for other recursive functions, where the base case doesn’t happen at $n \leq 1$, the number of recursive levels might be different than the height.
How many levels are there, exactly? Is it $\log_2(n)$?

Let’s try an example. Suppose we have $T(4)$. What happens?

```
      4
     / \  /
    2   2
   / \  / \  /
  1   1 1   1 1
```
How many levels are there, exactly? Is it $\log_2(n)$?

Let’s try an example. Suppose we have $T(4)$. What happens?

Height is $\log_2(4) = 2$.

For this recursive function, num recursive levels is same as height.
How many levels are there, exactly? Is it $\log_2(n)$?

Let’s try an example. Suppose we have $T(4)$. What happens?

```
  4
 /\  \
/   \ /   \
2   2
 / \ / \  \
/   /   \ \
1   1   1 1
```

Height is $\log_2(4) = 2$.

For this recursive function, num recursive levels is same as height.

**Important:** total levels, counting base case, is height + 1.
The tree method: precise analysis

How many levels are there, exactly? Is it $\log_2(n)$?

Let’s try an example. Suppose we have $T(4)$. What happens?

```
  4
 / \ / \n2   2
|   |
1   1
```

Height is $\log_2(4) = 2$.

For this recursive function, num recursive levels is same as height.

**Important:** total levels, counting base case, is height + 1.

**Important:** for other recursive functions, where base case doesn’t happen at $n \leq 1$, num recursive levels might be different then
The tree method: precise analysis

We discovered:

1. \( \text{numNodes}(i) = 2^i \)
2. \( \text{workPerNode}(n, i) = \frac{n}{2^i} \)
3. \( \text{numLevels}(n) = \log_2(n) \)
4. \( \text{workPerLeafNode}(n) = 1 \)
5. \( \text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_2(n)} = n \)
The tree method: precise analysis

We discovered:

1. $\text{numNodes}(i) = 2^i$
2. $\text{workPerNode}(n, i) = \frac{n}{2^i}$
3. $\text{numLevels}(n) = \log_2(n)$
4. $\text{workPerLeafNode}(n) = 1$
5. $\text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_2(n)} = n$

Our formulas:

$$\text{recursiveWork} = \sum_{i=0}^{\text{numLevels}(n)} \text{numNodes}(i) \cdot \text{workPerNode}(n, i)$$

$$\text{baseCaseWork} = \text{numLeafNodes}(n) \cdot \text{workPerLeafNode}(n)$$

$$\text{totalWork} = \text{recursiveWork} + \text{baseCaseWork}$$
The tree method: precise analysis

We discovered:

1. \text{numNodes}(i) = 2^i
2. \text{workPerNode}(n, i) = \frac{n}{2^i}
3. \text{numLevels}(n) = \log_2(n)
4. \text{workPerLeafNode}(n) = 1
5. \text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_2(n)} = n

Our formulas:

\[
\text{recursiveWork} = \sum_{i=0}^{\text{numLevels}(n)} \text{numNodes}(i) \cdot \text{workPerNode}(n, i)
\]

\[
\text{baseCaseWork} = \text{numLeafNodes}(n) \cdot \text{workPerLeafNode}(n)
\]
We discovered:

1. \( \text{numNodes}(i) = 2^i \)
2. \( \text{workPerNode}(n, i) = \frac{n}{2^i} \)
3. \( \text{numLevels}(n) = \log_2(n) \)
4. \( \text{workPerLeafNode}(n) = 1 \)
5. \( \text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_2(n)} = n \)

Our formulas:

\[
\text{recursiveWork} = \sum_{i=0}^{\text{numLevels}(n)} \text{numNodes}(i) \cdot \text{workPerNode}(n, i)
\]

\[
\text{baseCaseWork} = \text{numLeafNodes}(n) \cdot \text{workPerLeafNode}(n)
\]

\[
\text{totalWork} = \text{recursiveWork} + \text{baseCaseWork}
\]
Solve for recursive case:

\[
\text{recursiveWork} = \sum_{i=0}^{\log_2(n)} 2^i \cdot \frac{n}{2^i}
\]
Solve for recursive case:

\[
\text{recursiveWork} = \log_2(n) \sum_{i=0}^{\log_2(n)} 2^i \cdot \frac{n}{2^i}
\]

\[
= \sum_{i=0}^{\log_2(n)} n
\]

So exact closed form is \( n \log_2(n) + n \).
The tree method: precise analysis

Solve for recursive case:

\[ \text{recursiveWork} = \sum_{i=0}^{\log_2(n)} 2^i \cdot \frac{n}{2^i} = \log_2(n) \sum_{i=0}^{\log_2(n)} n = n \log_2(n) \]

Solve for base case:

\[ \text{baseCaseWork} = \text{numLeafNodes}(n) \cdot \text{workDonePerLeafNode}(n) = n \cdot 1 = n \]
Solve for recursive case:

\[
\text{recursiveWork} = \sum_{i=0}^{\log_2(n)} 2^i \cdot \frac{n}{2^i}
\]

\[
= \sum_{i=0}^{\log_2(n)} n
\]

\[
= n \log_2(n)
\]

Solve for base case:

\[
\text{baseCaseWork} = \text{numLeafNodes}(n) \cdot \text{workDonePerLeafNode}(n)
\]

\[
= n \cdot 1 = n
\]

So exact closed form is \( n \log_2(n) + n \).
The tree method: practice

Practice: Let’s go back to our old recurrence...

\[ S(n) = \begin{cases} 
2 & \text{if } n \leq 1 \\
2S\left(\frac{n}{3}\right) + n^2 & \text{otherwise} 
\end{cases} \]
The tree method: practice

\begin{align*}
&n^2 \\
&\quad \frac{n^2}{9} \quad \frac{n^2}{9} \\
&\quad \frac{n^4}{81} \quad \frac{n^2}{81} \quad \frac{n^2}{81} \quad \frac{n^2}{81} \\
&\quad \vdots \quad \vdots \quad \vdots \quad \vdots \\
&\quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2
\end{align*}
The tree method: practice

1. \text{numNodes}(i) = ?
2. \text{workPerNode}(n, i) = ?
3. \text{numLevels}(n) = ?
4. \text{workPerLeafNode}(n) = ?
5. \text{numLeafNodes}(n) = ?
The tree method: practice

1 node, \( n^2 \) work per

2 nodes, \( \frac{n^2}{3^2} \) work per

4 nodes, \( \frac{n^2}{3^4} \) work per

\( 2^i \) nodes, \( \frac{n^2}{3^{2i}} \) work per

\( 2^h \) nodes, 1 work per

1. \( \text{numNodes}(i) = ? \)
2. \( \text{workPerNode}(n, i) = ? \)
3. \( \text{numLevels}(n) = ? \)
4. \( \text{workPerLeafNode}(n) = ? \)
5. \( \text{numLeafNodes}(n) = ? \)
The tree method: practice

1. \( \text{numNodes}(i) \) = \( 2^i \)
2. \( \text{workPerNode}(n, i) \) = \( \frac{n^2}{9^i} \)
3. \( \text{numLevels}(n) \) = \( \log_3(n) \)
4. \( \text{workPerLeafNode}(n) \) = \( 2 \)
5. \( \text{numLeafNodes}(n) \) = \( 2^{\text{numLevels}(n)} = 2^{\log_3(n)} = n^{\log_3(2)} \)
The tree method: practice

1. \( \text{numNodes}(i) = 2^i \)
2. \( \text{workPerNode}(n, i) = \frac{n^2}{9^i} \)
3. \( \text{numLevels}(n) = \log_3(n) \)
4. \( \text{workPerLeafNode}(n) = 2 \)
5. \( \text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_3(n)} = n^{\log_3(2)} \)

Combine into a single expression representing the total runtime.
1. \(\text{numNodes}(i) = 2^i\)
2. \(\text{workPerNode}(n, i) = \frac{n^2}{9^i}\)
3. \(\text{numLevels}(n) = \log_3(n)\)
4. \(\text{workPerLeafNode}(n) = 2\)
5. \(\text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_3(n)} = n^{\log_3(2)}\)

Combine into a single expression representing the total runtime.

\[
\text{totalWork} = \left( \sum_{i=0}^{\log_3(n)} 2^i \cdot \frac{n^2}{9^i} \right) + 2n^{\log_3(2)}
\]
The tree method: practice

1. \( \text{numNodes}(i) = 2^i \)
2. \( \text{workPerNode}(n, i) = \frac{n^2}{9^i} \)
3. \( \text{numLevels}(n) = \log_3(n) \)
4. \( \text{workPerLeafNode}(n) = 2 \)
5. \( \text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_3(n)} = n^{\log_3(2)} \)

Combine into a single expression representing the total runtime.

\[
\text{totalWork} = \left( \sum_{i=0}^{\log_3(n)} 2^i \cdot \frac{n^2}{9^i} \right) + 2n^{\log_3(2)}
\]

\[
= n^2 \sum_{i=0}^{\log_3(n)} \frac{2^i}{9^i} + 2n^{\log_3(2)}
\]
The tree method: practice

1. \( \text{numNodes}(i) = 2^i \)
2. \( \text{workPerNode}(n, i) = \frac{n^2}{9^i} \)
3. \( \text{numLevels}(n) = \log_3(n) \)
4. \( \text{workPerLeafNode}(n) = 2 \)
5. \( \text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_3(n)} = n^{\log_3(2)} \)

Combine into a single expression representing the total runtime.

\[
\text{totalWork} = \left( \sum_{i=0}^{\log_3(n)} 2^i \cdot \frac{n^2}{9^i} \right) + 2n^{\log_3(2)}
\]

\[
= n^2 \sum_{i=0}^{\log_3(n)} \frac{2^i}{9^i} + 2n^{\log_3(2)}
\]

\[
= n^2 \sum_{i=0}^{\log_3(n)} \left( \frac{2}{9} \right)^i + 2n^{\log_3(2)}
\]
The finite geometric series

We have:

\[ n^2 \sum_{i=0}^{\log_3(n)} \left( \frac{2}{9} \right)^i + 2n^{\log_3(2)} \]
The finite geometric series

We have: \[ n^2 \sum_{i=0}^{\log_3(n)} \left( \frac{2}{9} \right)^i + 2n^{\log_3(2)} \]

The finite geometric series identity: \[ \sum_{i=0}^{n-1} r^i = \frac{1 - r^n}{1 - r} \]
The finite geometric series

We have: \( n^2 \sum_{i=0}^{\log_3(n)} \left( \frac{2}{9} \right)^i + 2n^{\log_3(2)} \)

The finite geometric series identity: \( \sum_{i=0}^{n-1} r^i = \frac{1 - r^n}{1 - r} \)

Plug and chug:

\[
totalWork = n^2 \sum_{i=0}^{\log_3(n)} \left( \frac{2}{9} \right)^i + 2n^{\log_3(2)} = n^2 \sum_{i=0}^{\log_3(n)+1-1} \left( \frac{2}{9} \right)^i + 2n^{\log_3(2)} = n^2 \frac{1 - \left( \frac{2}{9} \right)^{\log_3(n)+1}}{1 - \frac{2}{9}} + 2n^{\log_3(2)}
\]
Applying the finite geometric series

With a bunch of effort...

\[
\text{totalWork} = n^2 \frac{1 - \left(\frac{2}{9}\right) \log_3(n) + 1}{1 - \frac{2}{9}} + 2n \log_3(2)
\]

\[
= \frac{9}{7} n^2 \left(1 - \frac{2}{9} \left(\frac{2}{9}\right) \log_3(n)\right) + 2n \log_3(2)
\]

\[
= \frac{9}{7} n^2 - \frac{2}{7} n^2 \left(\frac{2}{9}\right) \log_3(n) + 2n \log_3(2)
\]

\[
= \frac{9}{7} n^2 - \frac{2}{7} n^2 n^2 \log_3(2/9) + 2n \log_3(2)
\]

\[
= \frac{9}{7} n^2 - \frac{2}{7} n^2 n^2 \log_3(2) + 2n \log_3(2)
\]

\[
= \frac{9}{7} n^2 - \frac{2}{7} n^2 \log_3(2) + 2n \log_3(2)
\]

\[
= \frac{9}{7} n^2 + \frac{12}{7} n \log_3(2)
\]
The master theorem

Is there an easier way?
Is there an easier way?

If we want to find an exact closed form, no. Must use either the unfolding technique or the tree technique.
Is there an easier way?

If we want to find an exact closed form, no. Must use either the unfolding technique or the tree technique.

If we want to find a big-Θ bound, yes.
The master theorem

Suppose we have a recurrence of the following form:

\[ T(n) = \begin{cases} 
    d & \text{if } n = 1 \\
    aT\left(\frac{n}{b}\right) + n^c & \text{otherwise}
\end{cases} \]
The master theorem

<table>
<thead>
<tr>
<th>The master theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suppose we have a recurrence of the following form:</td>
</tr>
</tbody>
</table>
| \[
T(n) = \begin{cases} 
  d & \text{if } n = 1 \\
  aT\left(\frac{n}{b}\right) + n^c & \text{otherwise}
\end{cases}
| |
| Then... |
| ▶ If \( \log_b(a) < c \), then \( T(n) \in \Theta \left(n^c \right) \) |
| ▶ If \( \log_b(a) = c \), then \( T(n) \in \Theta \left(n^c \log(n) \right) \) |
| ▶ If \( \log_b(a) > c \), then \( T(n) \in \Theta \left(n^{\log_b(a)} \right) \) |
The master theorem

Given:

\[ T(n) = \begin{cases} 
  d & \text{if } \log_b(a) < c, \\
  aT\left(\frac{n}{b}\right) + n^c & \text{if } \log_b(a) = c, \\
  aT\left(\frac{n}{b}\right) + n^c & \text{if } \log_b(a) > c,
\end{cases} \]

Then...

- If \( \log_b(a) < c \), then \( T(n) \in \Theta(n^c) \)
- If \( \log_b(a) = c \), then \( T(n) \in \Theta(n^c \log(n)) \)
- If \( \log_b(a) > c \), then \( T(n) \in \Theta(n^{\log_b(a)}) \)

Sanity check: try checking merge sort. We have \( a = 2, b = 2, \text{ and } c = 1 \). We know \( \log_2(2) = 1 = c \), therefore merge sort is \( \Theta(n \log(n)) \).

Sanity check: try checking \( S(n) = 2S\left(\frac{n}{3}\right) + n^2 \). We have \( a = 2, b = 3, \text{ and } c = 2 \). We know \( \log_3(2) \leq 2 < 2 = c \), therefore \( S(n) \in \Theta(n^2) \).
The master theorem

Given:

\[ T(n) = \begin{cases} 
  d & \text{if } \log_b(a) < c, \text{ then } T(n) \in \Theta(n^c) \\
  aT\left(\frac{n}{b}\right) + n^c & \text{if } \log_b(a) = c, \text{ then } T(n) \in \Theta(n^c \log(n)) \\
  & \text{if } \log_b(a) > c, \text{ then } T(n) \in \Theta(n^{\log_b(a)}) 
\end{cases} \]

Then...

Sanity check: try checking merge sort.

Sanity check: try checking \( S(n) = 2S\left(\frac{n}{3}\right) + n^2 \).

We have \( a = 2, b = 3, \text{ and } c = 2 \). We know \( \log_3(2) \leq 1 < 2 = c \), therefore \( S(n) \in \Theta(n^2) \).
The master theorem

Given:

\[
T(n) = \begin{cases} 
  d & \text{if } \log_b(a) < c, \text{ then } T(n) \in \Theta(n^c) \\
  aT\left(\frac{n}{b}\right) + n^c & \text{if } \log_b(a) = c, \text{ then } T(n) \in \Theta(n^c \log(n)) \\
  aT\left(\frac{n}{b}\right) + n^c & \text{if } \log_b(a) > c, \text{ then } T(n) \in \Theta(n^{\log_b(a)}) 
\end{cases}
\]

Then...

Sanity check: try checking merge sort.

We have \(a = 2\), \(b = 2\), and \(c = 1\). We know \(\log_b(a) = \log_2(2) = 1 = c\), therefore merge sort is \(\Theta(n \log(n))\).
The master theorem

Given:

\[ T(n) = \begin{cases} 
  d & \text{if } \log_b(a) < c, \text{ then } T(n) \in \Theta(n^c) \\
  aT\left(\frac{n}{b}\right) + n^c & \text{if } \log_b(a) = c, \text{ then } T(n) \in \Theta(n^c \log(n)) \\
  aT\left(\frac{n}{b}\right) + n^c & \text{if } \log_b(a) > c, \text{ then } T(n) \in \Theta(n^{\log_b(a)}) 
\end{cases} \]

Then...

Sanity check: try checking merge sort.

We have \( a = 2, b = 2, \) and \( c = 1. \) We know \( \log_b(a) = \log_2(2) = 1 = c, \) therefore merge sort is \( \Theta(n \log(n)) \).

Sanity check: try checking \( S(n) = 2S(n/3) + n^2. \)
The master theorem

Given:

\[ T(n) = \begin{cases} 
  d & \text{if } \log_b(a) < c, \text{ then } T(n) \in \Theta(n^c) \\
  aT\left(\frac{n}{b}\right) + n^c & \text{if } \log_b(a) = c, \text{ then } T(n) \in \Theta(n^c \log(n)) \\
  aT\left(\frac{n}{b}\right) + n^c & \text{if } \log_b(a) > c, \text{ then } T(n) \in \Theta(n^{\log_b(a)}) 
\end{cases} \]

Then...

Sanity check: try checking merge sort.

We have \( a = 2, \ b = 2, \text{ and } c = 1 \). We know \( \log_b(a) = \log_2(2) = 1 = c \), therefore merge sort is \( \Theta(n \log(n)) \).

Sanity check: try checking \( S(n) = 2S(n/3) + n^2 \).

We have \( a = 2, \ b = 3, \text{ and } c = 2 \). We know \( \log_3(2) \leq 1 < 2 = c \), therefore \( S(n) \in \Theta(n^2) \).
**Intuition, the $\log_b(a) < c$ case:**

1. We do work more rapidly than we divide.
2. So, more of the work happens near the “top”, which means that the $n^c$ term dominates.
**Intuition, the $\log_b(a) > c$ case:**

1. We divide more rapidly than we do work.

2. So, most of the work happens near the “bottom”, which means the work done in the leaves dominates.

3. Note: Work in leaves is about
   \[
   d \cdot a^\text{height} = d \cdot a^{\log_b(n)} = d \cdot n^{\log_b(a)}.
   \]
Intuition, the $\log_b(a) = c$ case:

1. Work is done roughly equally throughout tree.
2. Each level does about the same amount of work, so we approximate by just multiplying work done on first level by the height: $n^c \log_b(n)$. 