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

Michael Lee

Wednesday, Feb 7, 2018

Technique: Divide-and-Conquer

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

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)

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)

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)

Example template

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

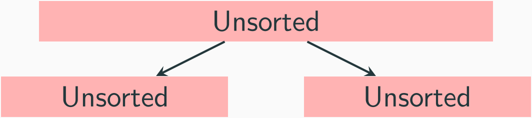
Merge sort: Core pieces

Divide:

Unsorted

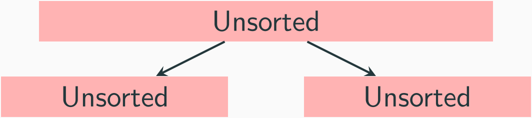
Merge sort: Core pieces

Divide: Split array roughly into half



Merge sort: Core pieces

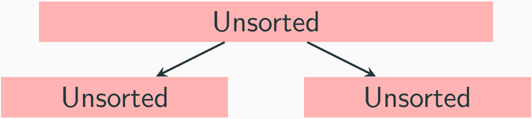
Divide: Split array roughly into half



Conquer:

Merge sort: Core pieces

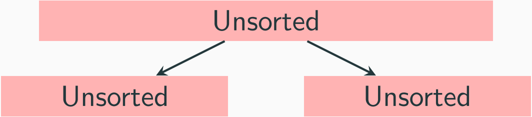
Divide: Split array roughly into half



Conquer: Return array when length ≤ 1

Merge sort: Core pieces

Divide: Split array roughly into half

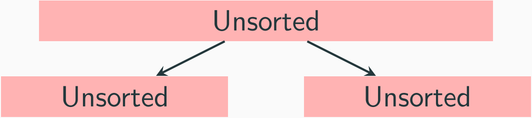


Conquer: Return array when length ≤ 1

Combine:

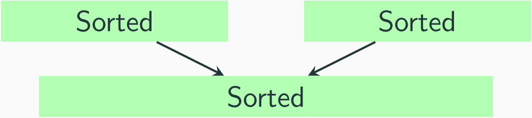
Merge sort: Core pieces

Divide: Split array roughly into half



Conquer: Return array when length ≤ 1

Combine: Combine two sorted arrays using `merge`



Merge sort: Summary

Core idea: split array in half, sort each half, merge back together.

If the array has size 0 or 1, just return it unchanged.

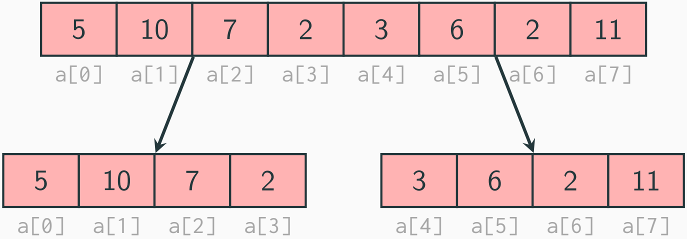
Pseudocode

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

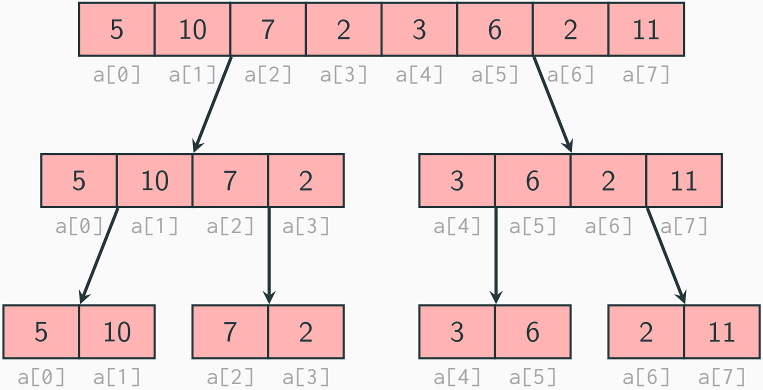
Merge sort: Example

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
5	10	7	2	3	6	2	11

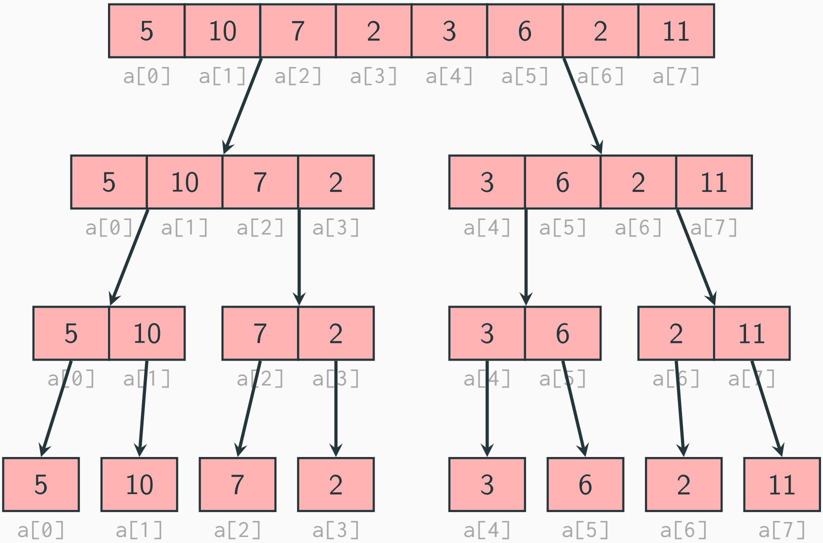
Merge sort: Example



Merge sort: Example



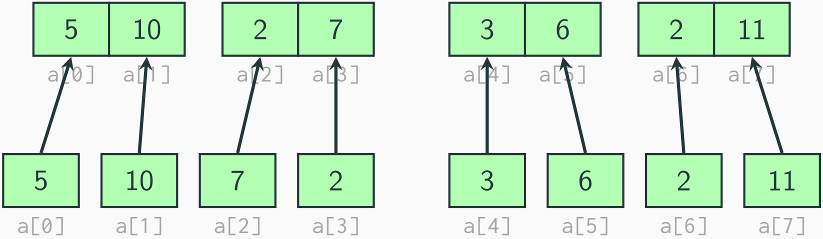
Merge sort: Example



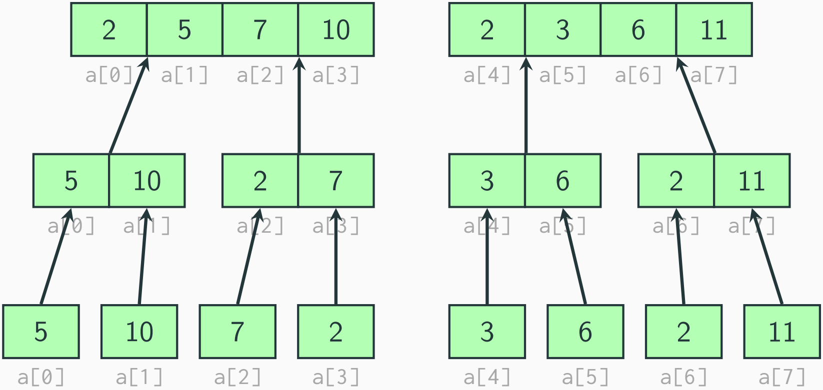
Merge sort: Example

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
5	10	7	2	3	6	2	11

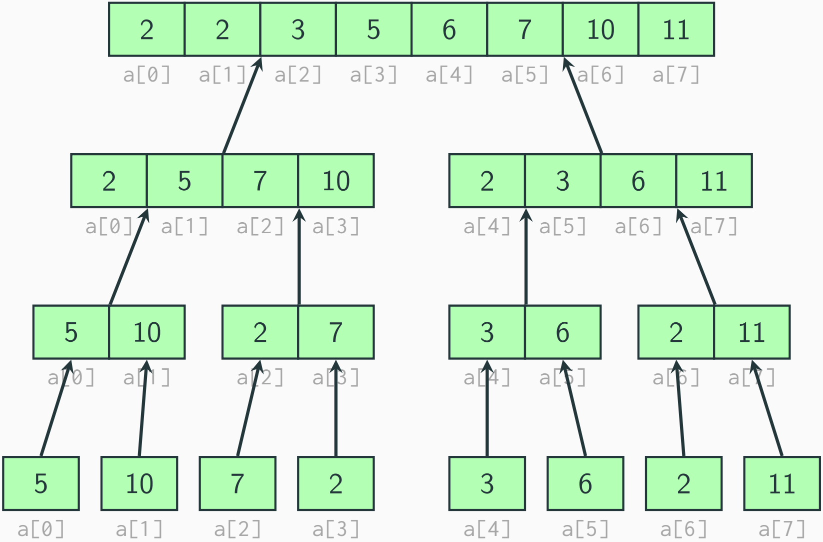
Merge sort: Example



Merge sort: Example



Merge sort: Example



Merge sort: Analysis

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?

Merge sort: Analysis

Best and worst case

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

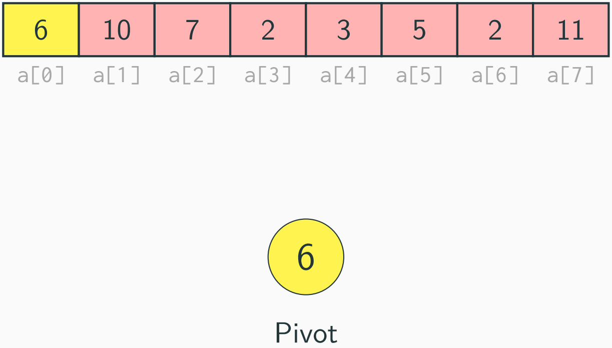
$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

Spoiler alert: this is $\Theta(n \log(n))$

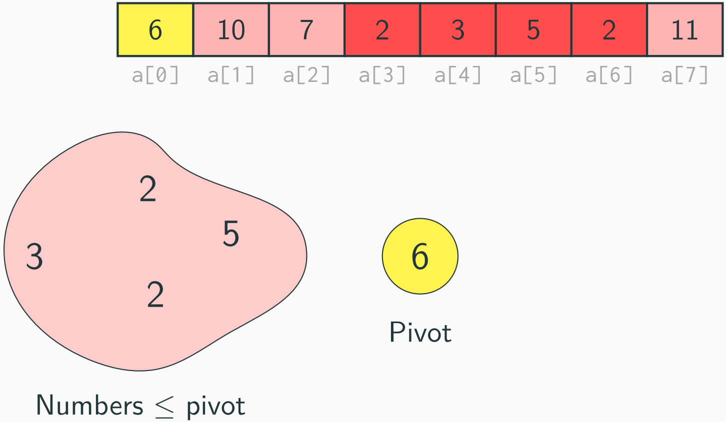
Quick sort: Divide step

6	10	7	2	3	5	2	11
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

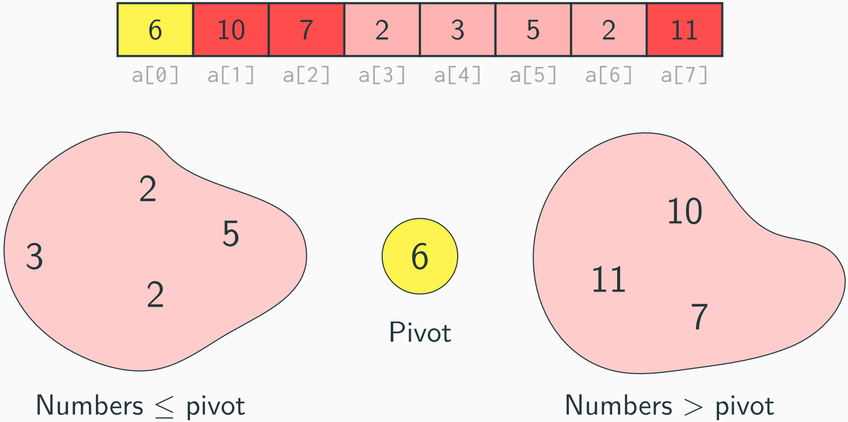
Quick sort: Divide step



Quick sort: Divide step

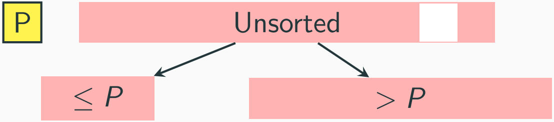


Quick sort: Divide step



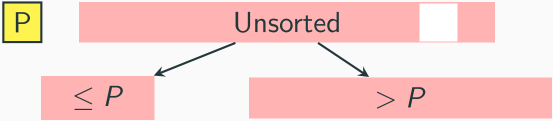
Quick sort: Core pieces

Divide: Pick a pivot, partition into groups



Quick sort: Core pieces

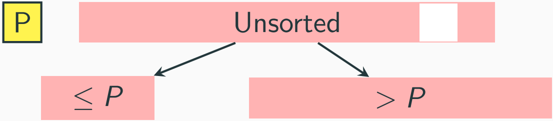
Divide: Pick a pivot, partition into groups



Conquer:

Quick sort: Core pieces

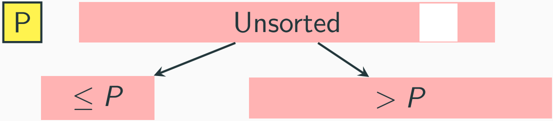
Divide: Pick a pivot, partition into groups



Conquer: Return array when length ≤ 1

Quick sort: Core pieces

Divide: Pick a pivot, partition into groups

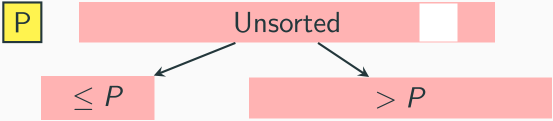


Conquer: Return array when length ≤ 1

Combine:

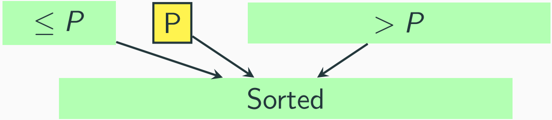
Quick sort: Core pieces

Divide: Pick a pivot, partition into groups



Conquer: Return array when length ≤ 1

Combine: Combine sorted portions and the pivot



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

```
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;
    }
}
```

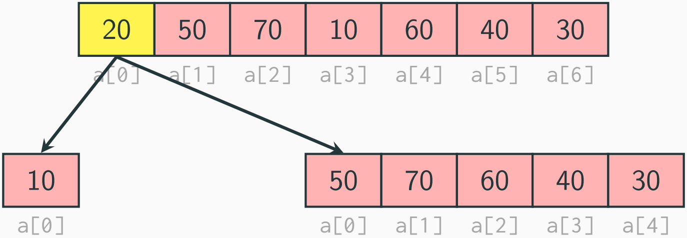
Quick sort: Example

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
20	50	70	10	60	40	30

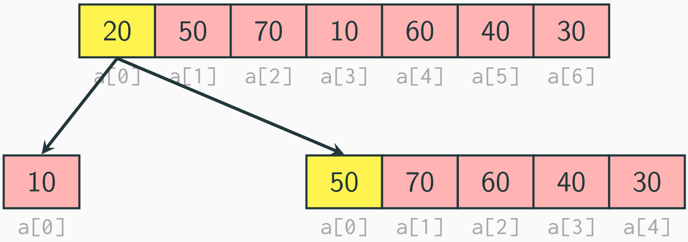
Quick sort: Example

20	50	70	10	60	40	30
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

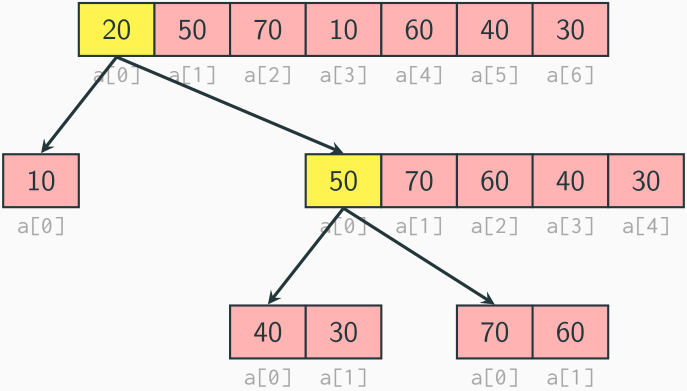
Quick sort: Example



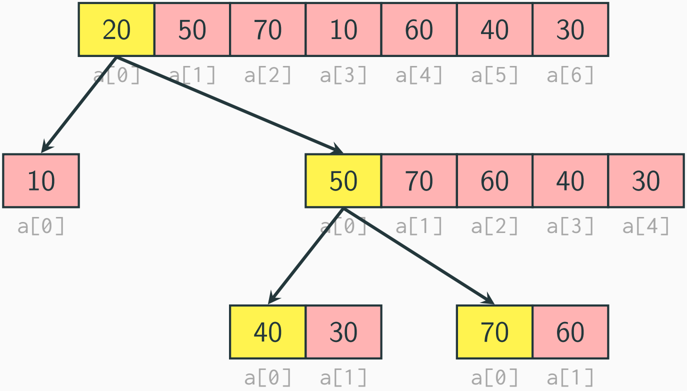
Quick sort: Example



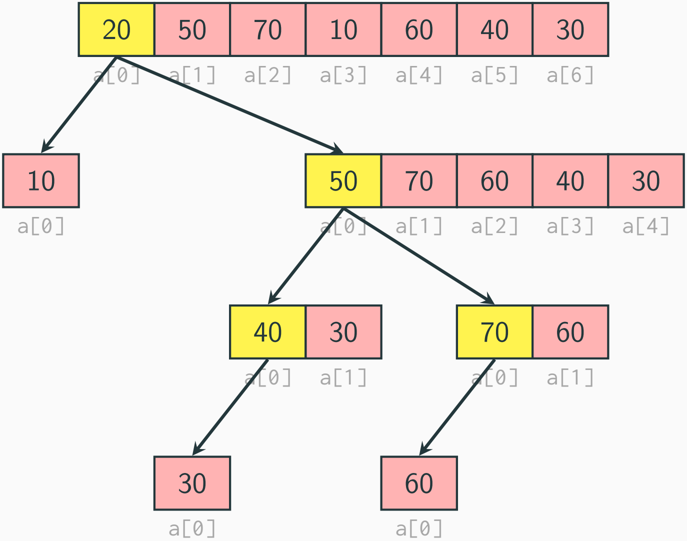
Quick sort: Example



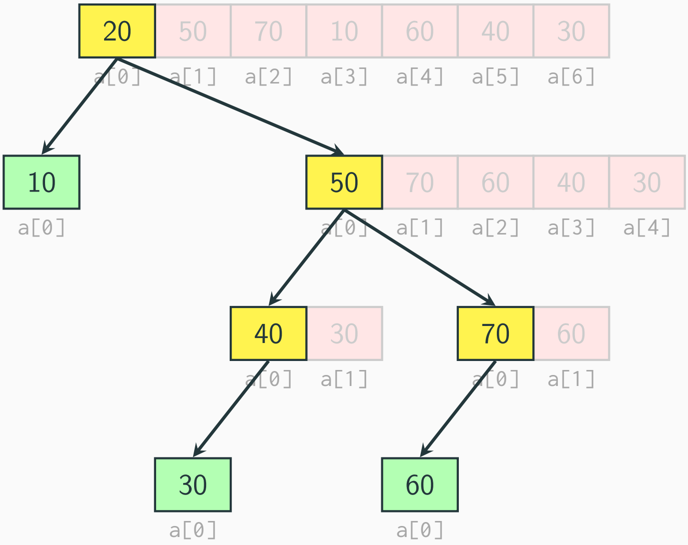
Quick sort: Example



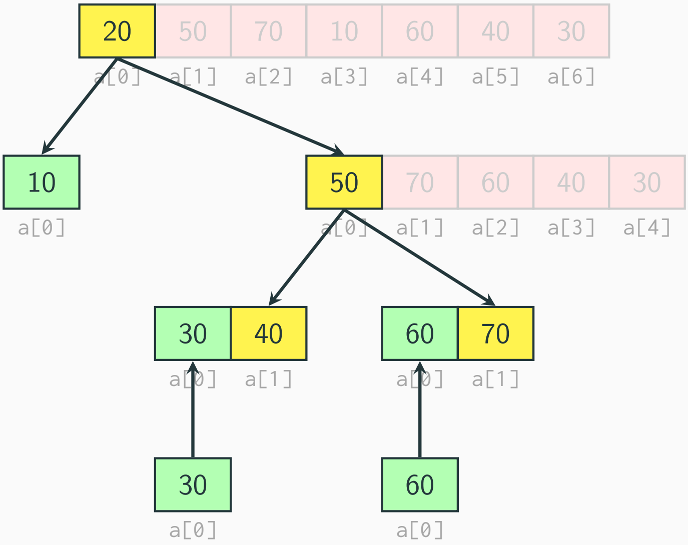
Quick sort: Example



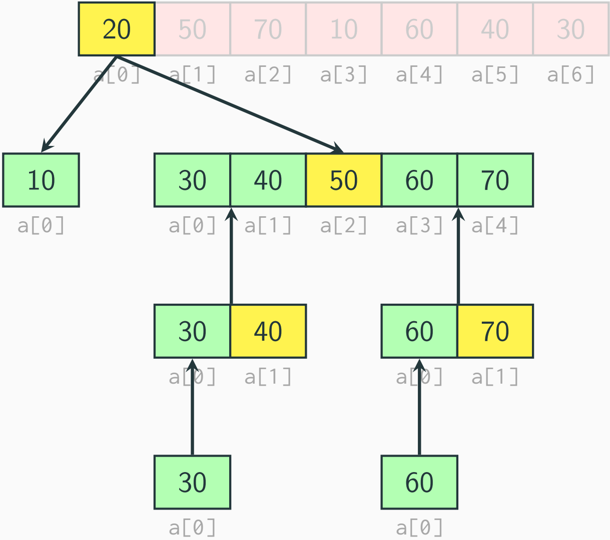
Quick sort: Example



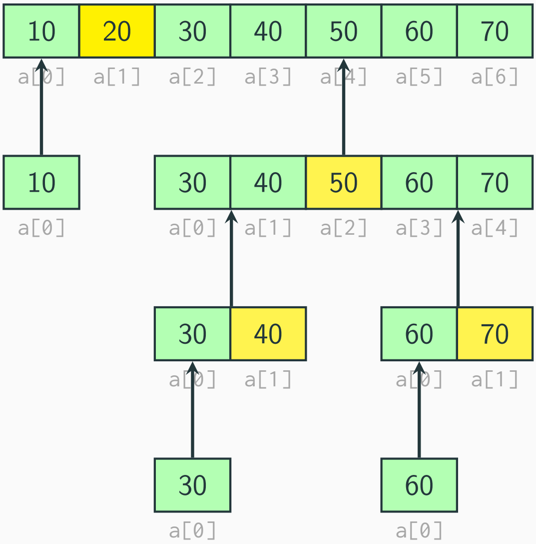
Quick sort: Example



Quick sort: Example



Quick sort: Example



Quick sort: Analysis

Pseudocode

```
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.

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

Quick sort: Analysis

Best case analysis

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

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

(Spoiler alert: this is $\Theta(n \log(n))$)

Quick sort: Analysis

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 $\Theta\left(n^2\right)$.

Quick sort: Analysis

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 $\Theta\left(n^2\right)$.

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?

Quick sort: Unresolved questions

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?

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 ($\mathcal{O}(1)$)

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 ($\mathcal{O}(1)$)
- 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 ($\mathcal{O}(1)$)
- Idea: try finding it by looping through the array
  - Problem: hard to implement, and expensive ($\mathcal{O}(n)$)

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 ($\mathcal{O}(1)$)
- Idea: try finding it by looping through the array
  - Problem: hard to implement, and expensive ($\mathcal{O}(n)$)

These seem like bad ideas :(

Quick sort: Picking a pivot

Other ideas:

Quick sort: Picking a pivot

Other ideas:

- Idea: pick a random element

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

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

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?

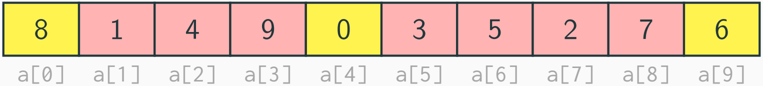
Quick sort: Partitioning (using median-of-three pivot)

Find the lo, med, and hi

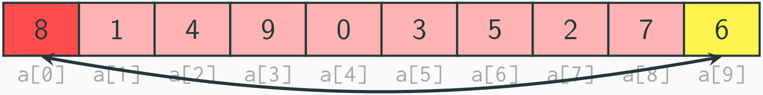
8	1	4	9	0	3	5	2	7	6
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

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

Find the lo, med, and hi



Find the median of the three and swap with front



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

Find the `lo`, `med`, and `hi`

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	1	4	9	0	3	5	2	7	6

Find the median of the three and swap with front

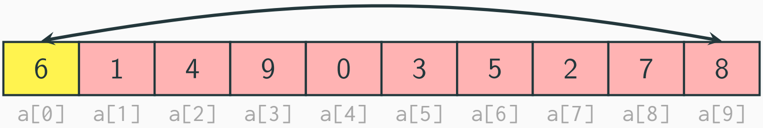
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	1	4	9	0	3	5	2	7	6

Final result: pivot is now at index 0

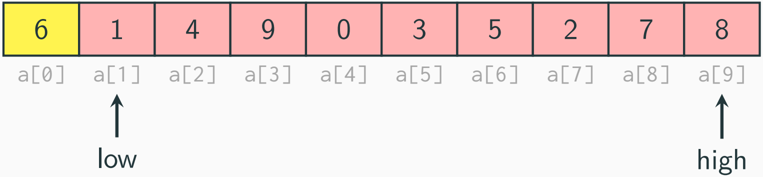
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	1	4	9	0	3	5	2	7	8

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

Array after moving pivot:



Partitioning:

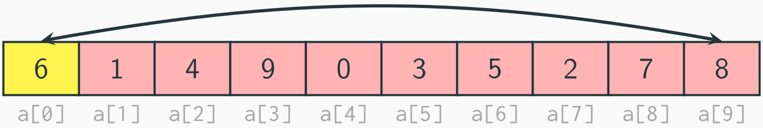


$1 \leq 6$

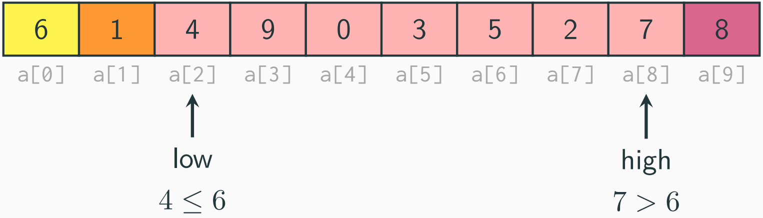
$8 > 6$

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

Array after moving pivot:

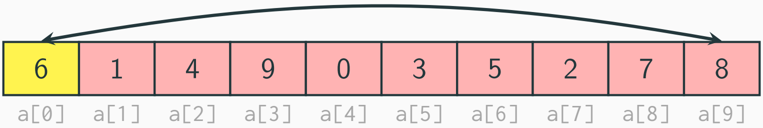


Partitioning:

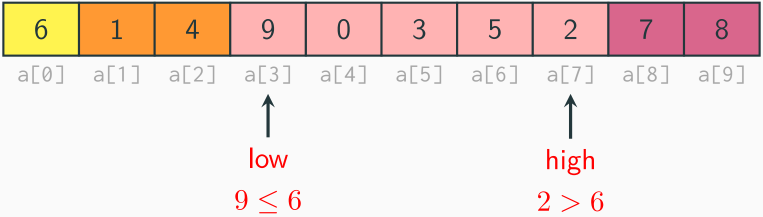


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

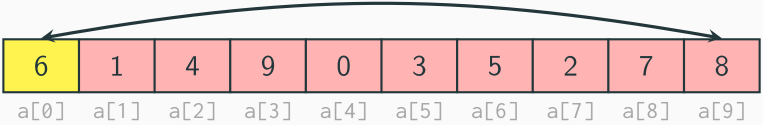


Partitioning:

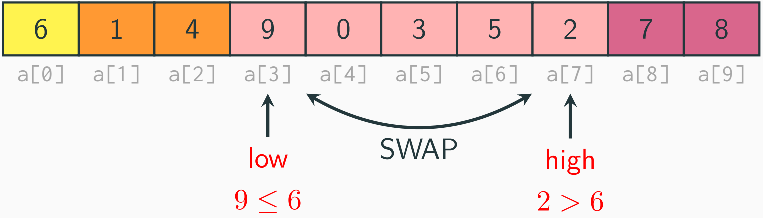


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

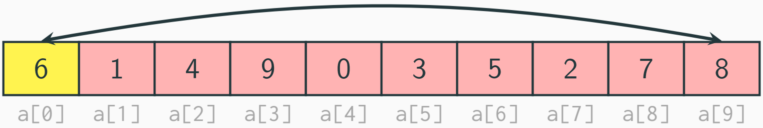


Partitioning:



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

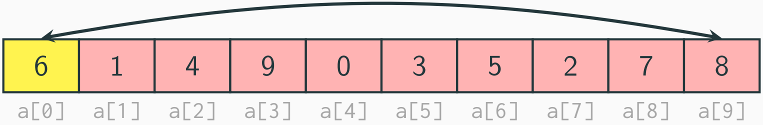
Array after moving pivot:



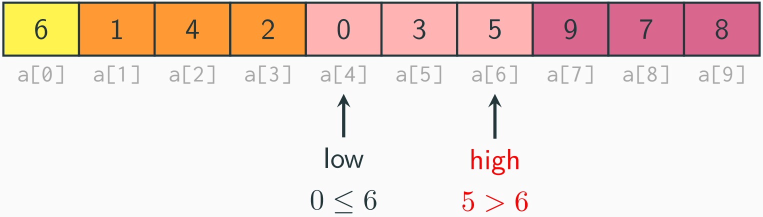
Partitioning:

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

Array after moving pivot:

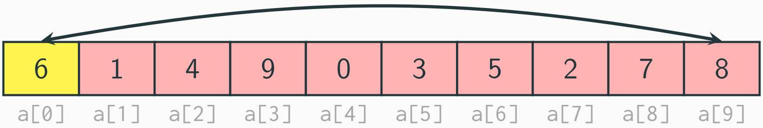


Partitioning:

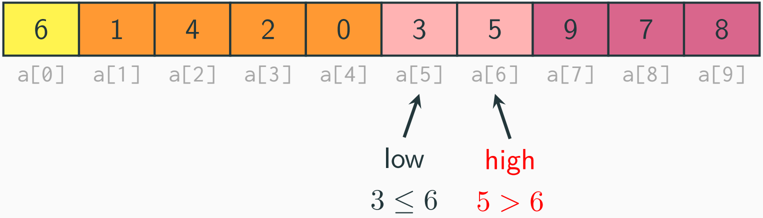


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

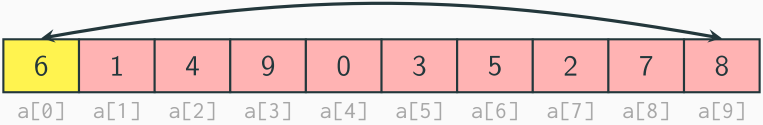


Partitioning:

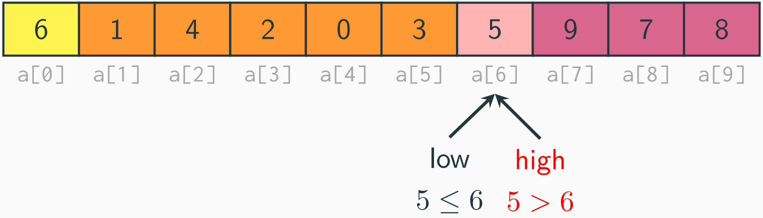


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

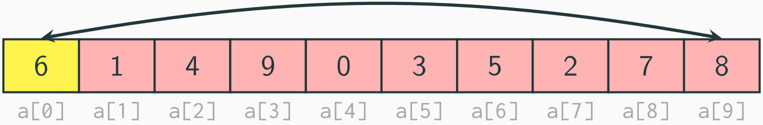


Partitioning:

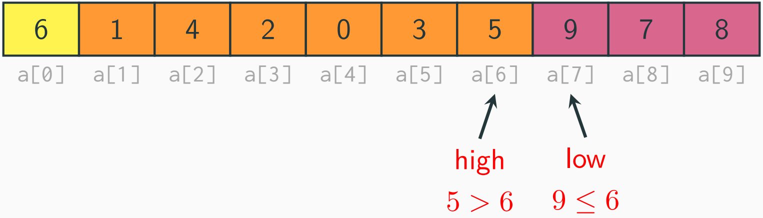


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

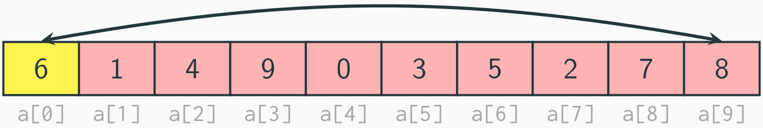


Partitioning:

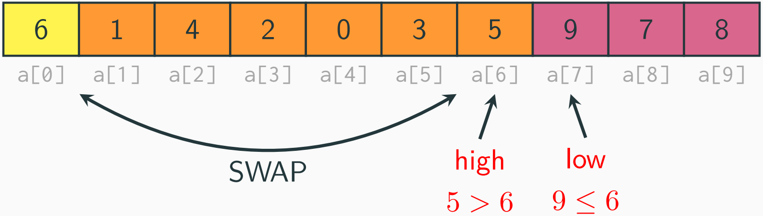


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

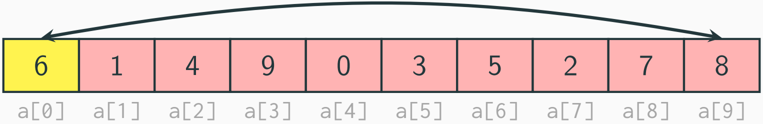


Partitioning:

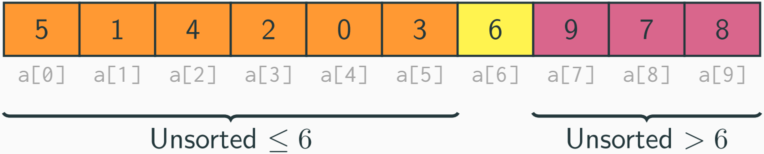


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

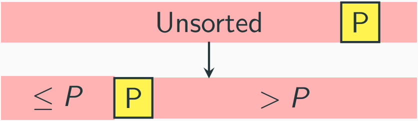


Partitioning:



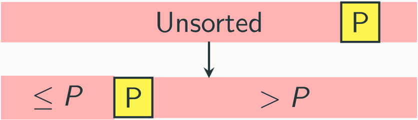
Quick sort: Core pieces revisited

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



Quick sort: Core pieces revisited

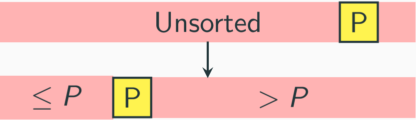
Divide: Pick a pivot, partition in-place into groups



Conquer: When subarray is length ≤ 1, do nothing

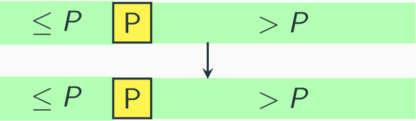
Quick sort: Core pieces revisited

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



Conquer: When subarray is length ≤ 1, do nothing

Combine: Do nothing; already done!



Analyzing recurrences, part 2

So, merge sort and quick sort are both:

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

Analyzing recurrences, part 2

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?

Analyzing recurrences, part 2

We could try unfolding, but it's annoying:

Analyzing recurrences, part 2

We could try unfolding, but it's annoying:

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

Analyzing recurrences, part 2

We could try unfolding, but it's annoying:

$$
\begin{aligned}
T(n) &= n + 2T\left(\frac{n}{2}\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right)
\end{aligned}
$$

Analyzing recurrences, part 2

We could try unfolding, but it's annoying:

$$\begin{aligned} T(n) &= n + 2T\left(\frac{n}{2}\right) \\ &= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \\ &= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \end{aligned}$$

Analyzing recurrences, part 2

We could try unfolding, but it's annoying:

$$\begin{aligned}
T(n) &= n + 2T\left(\frac{n}{2}\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4} + 2T(\frac{n}{8})\right)\right)
\end{aligned}$$

Analyzing recurrences, part 2

We could try unfolding, but it's annoying:

$$\begin{aligned}
T(n) &= n + 2T\left(\frac{n}{2}\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4} + 2T(\frac{n}{8})\right)\right) \\
&= n + n + 4T\left(\frac{n}{4} + 2T\left(\frac{n}{8}\right)\right)
\end{aligned}$$

Analyzing recurrences, part 2

We could try unfolding, but it's annoying:

$$\begin{aligned}
T(n) &= n + 2T\left(\frac{n}{2}\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4} + 2T(\frac{n}{8})\right)\right) \\
&= n + n + 4T\left(\frac{n}{4} + 2T\left(\frac{n}{8}\right)\right) \\
&= n + n + n + 8T\left(\frac{n}{8}\right)
\end{aligned}$$

Analyzing recurrences, part 2

We could try unfolding, but it's annoying:

$$\begin{aligned}
T(n) &= n + 2T\left(\frac{n}{2}\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4} + 2T(\frac{n}{8})\right)\right) \\
&= n + n + 4T\left(\frac{n}{4} + 2T\left(\frac{n}{8}\right)\right) \\
&= n + n + n + 8T\left(\frac{n}{8}\right) \\
&= \underbrace{n + n + \cdots + n}_{\text{about } \log(n) \text{ times}} + n
\end{aligned}$$

Analyzing recurrences, part 2

We could try unfolding, but it's annoying:

$$
\begin{aligned}
T(n) &= n + 2T\left(\frac{n}{2}\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right) \\
&= n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{4} + 2T(\frac{n}{8})\right)\right) \\
&= n + n + 4T\left(\frac{n}{4} + 2T\left(\frac{n}{8}\right)\right) \\
&= n + n + n + 8T\left(\frac{n}{8}\right) \\
&= \underbrace{n + n + \cdots + n}_{\text{about } \log(n) \text{ times}} + n \\
&= n\log(n)
\end{aligned}
$$

The tree method: overview

Core idea:

1. Draw what the work looks like visually, as a *tree*

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

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

$$\boxed{T(n)}$$

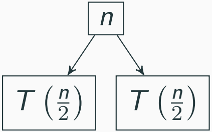
The tree method: example

Step 2: Replace with definition

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

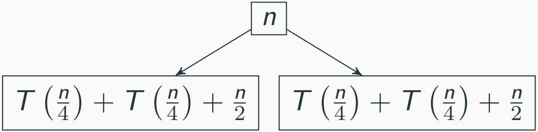
The tree method: example

Step 3: Stick each recursive call into a *subtree*



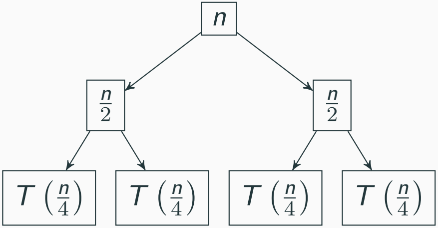
The tree method: example

Step 4: Replace with definition



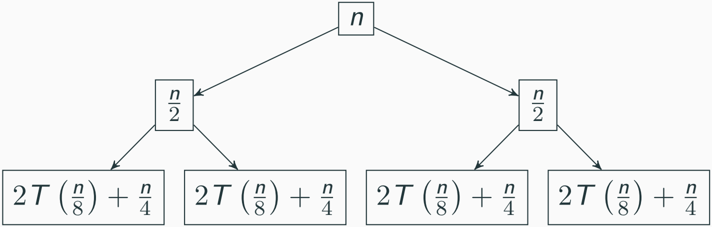
The tree method: example

Repeat step 3 (move recursive call to subtrees):



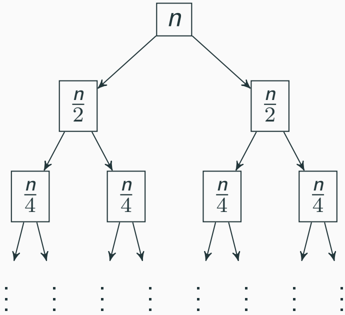
The tree method: example

Repeat step 4 (replace recursive call with definition):



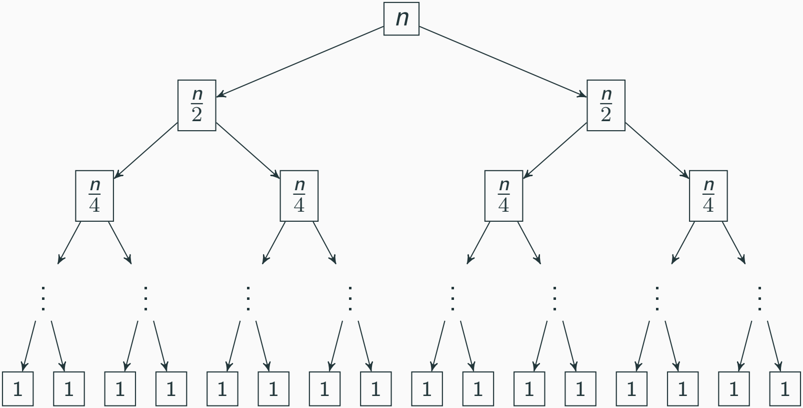
The tree method: example

Repeat...



The tree method: example

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



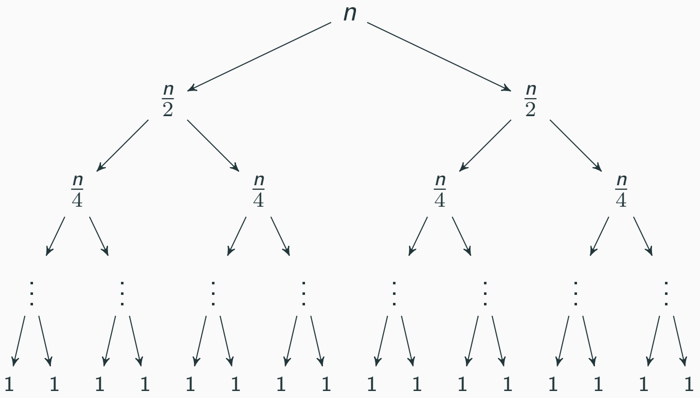
The tree method: analysis

Now, let's add everything up!

The tree method: analysis

Now, let's add everything up!

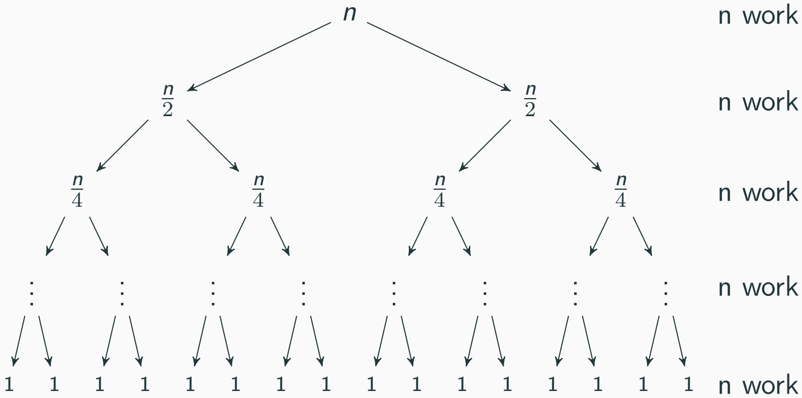
How much work is done per level?



The tree method: analysis

Now, let's add everything up!

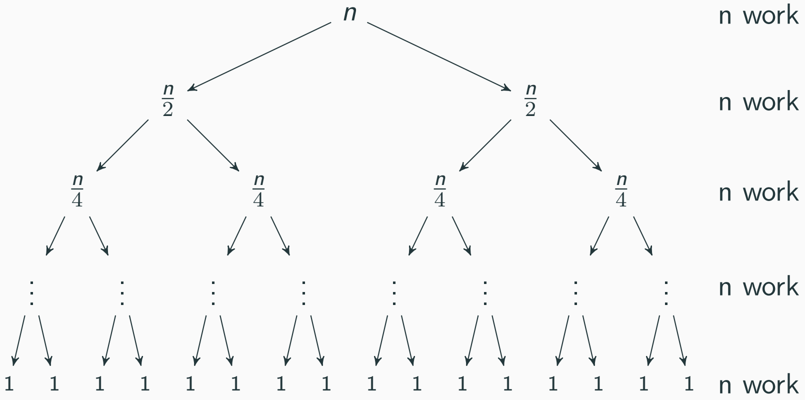
How much work is done per level?



The tree method: analysis

Now, let's add everything up!

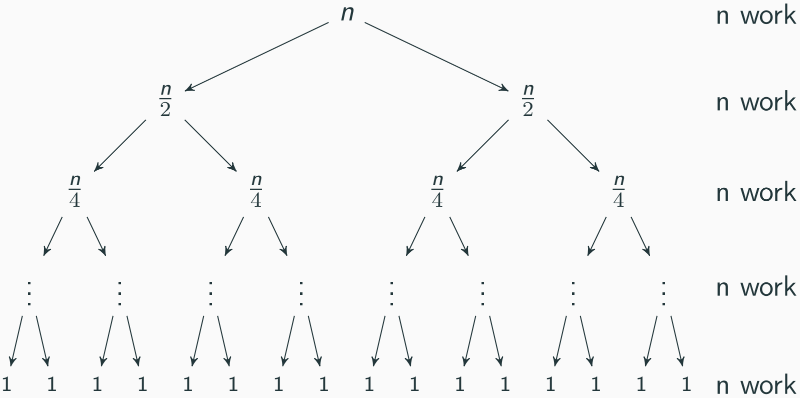
How much work is done per level?



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 $n \log_2(n)$.

The tree method: practice

Consider the following recurrence:

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

The tree method: practice

Consider the following recurrence:

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

Draw a tree to help you visualize the work done.

The tree method: practice

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

$$\boxed{S(n)}$$

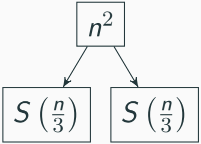
The tree method: practice

Step 2: Replace with definition

$$\boxed{S\left(\frac{n}{3}\right) + S\left(\frac{n}{3}\right) + n^2}$$

The tree method: practice

Step 3: Stick each recursive call into a *subtree*



The tree method: practice

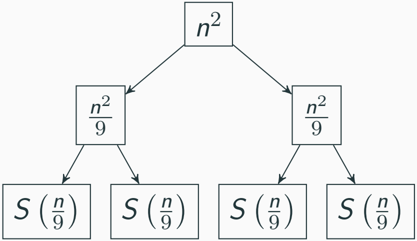
Step 4: Replace with definition

$$n^2$$

$$S\left(\frac{n}{9}\right) + S\left(\frac{n}{9}\right) + \frac{n^2}{9} \qquad S\left(\frac{n}{9}\right) + S\left(\frac{n}{9}\right) + \frac{n^2}{9}$$

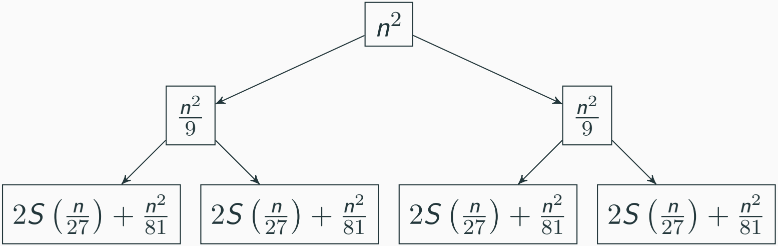
The tree method: practice

Repeat step 3 (move recursive call to subtrees):



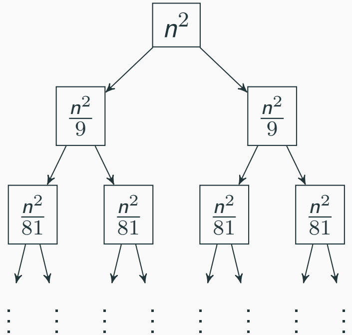
The tree method: practice

Repeat step 4 (replace recursive call with definition):



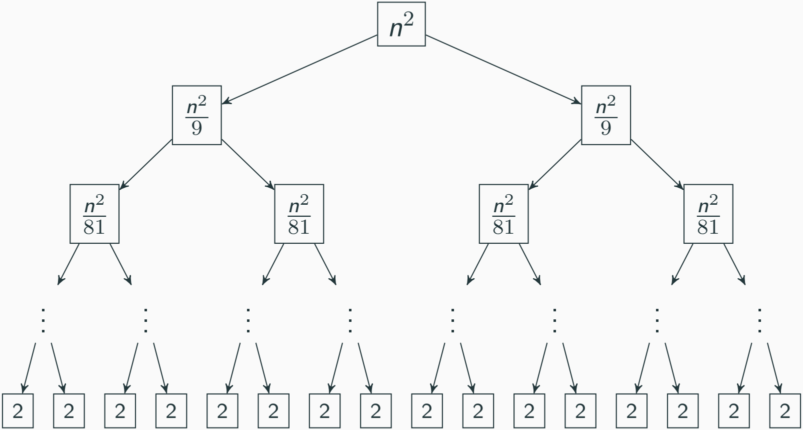
The tree method: practice

Repeat...



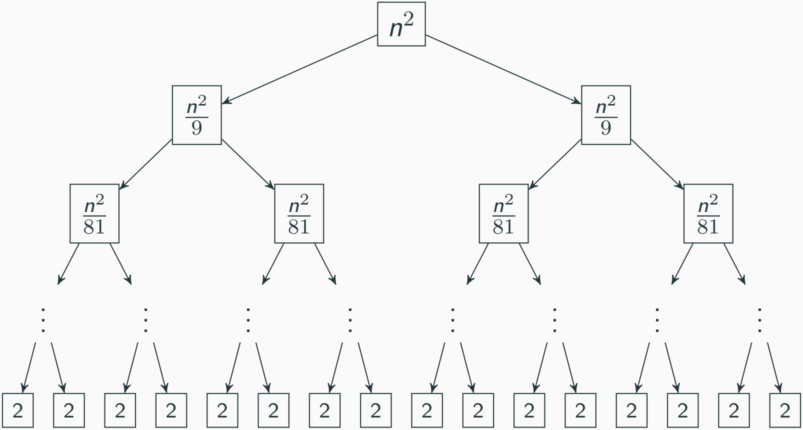
The tree method: practice

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



The tree method: practice

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



Now what?

The tree method: precise analysis

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:

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?

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)

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?

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?

The tree method: precise analysis

Problem: Need a rigorous way of getting a closed form

We want to answer a few core questions:

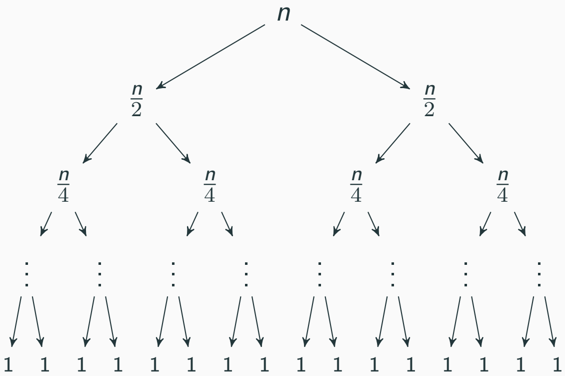
How much work does each recursive level do?

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?

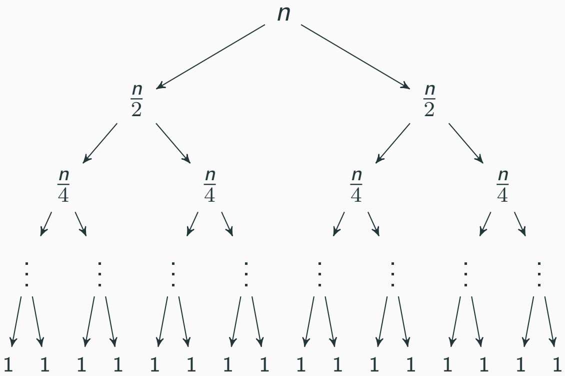
How much work does the leaf level (base cases) do?

1. How much work does a single leaf node do?
2. How many leaf nodes are there?

The tree method: precise analysis

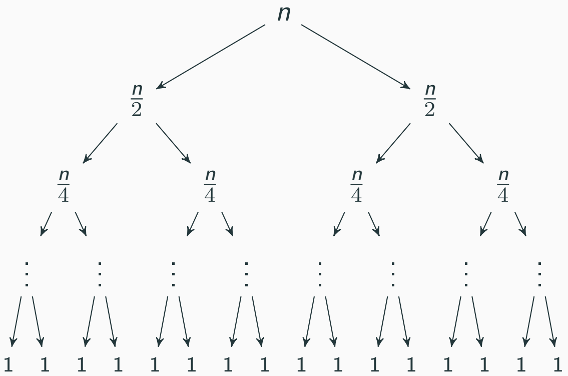


The tree method: precise analysis



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

The tree method: precise analysis



1 node, n work per

2 nodes, $\frac{n}{2}$ work per

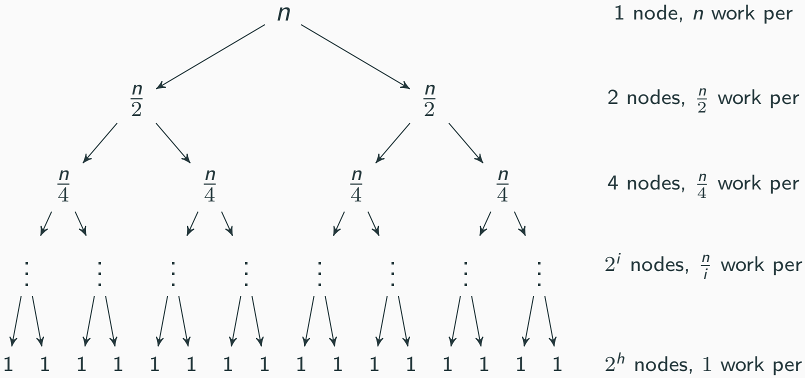
4 nodes, $\frac{n}{4}$ work per

2^i nodes, $\frac{n}{i}$ work per

2^h nodes, 1 work per

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

The tree method: precise analysis

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

The tree method: precise analysis

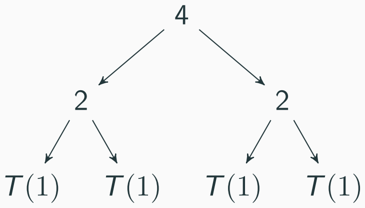
How many levels are there, exactly? Is it $\log_2(n)$?

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

The tree method: precise analysis

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

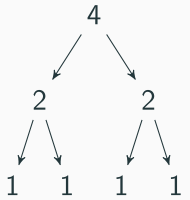
Let's try an example. Suppose we have $T(4)$. What happens?



The tree method: precise analysis

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

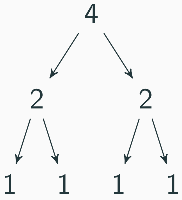
Let's try an example. Suppose we have $T(4)$. What happens?



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?



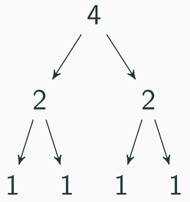
Height is $\log_2(4) = 2$.

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

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?



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

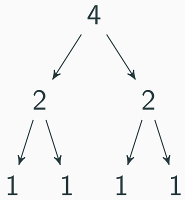
For this recursive function, num recursive levels is same as height.

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?



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:

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

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:

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

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

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

Solve for recursive case:

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

The tree method: precise analysis

Solve for recursive case:

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

The tree method: precise analysis

Solve for recursive case:

$$\begin{aligned} \text{recursiveWork} &= \sum_{i=0}^{\log_2(n)} 2^i \cdot \frac{n}{2^i} \\ &= \sum_{i=0}^{\log_2(n)} n \\ &= n \log_2(n) \end{aligned}$$

Solve for base case:

$$\begin{aligned} \text{baseCaseWork} &= \text{numLeafNodes}(n) \cdot \text{workDonePerLeafNode}(n) \\ &= n \cdot 1 = n \end{aligned}$$

The tree method: precise analysis

Solve for recursive case:

$$\begin{aligned}\text{recursiveWork} &= \sum_{i=0}^{\log_2(n)} 2^i \cdot \frac{n}{2^i} \\ &= \sum_{i=0}^{\log_2(n)} n \\ &= n \log_2(n)\end{aligned}$$

Solve for base case:

$$\begin{aligned}\text{baseCaseWork} &= \text{numLeafNodes}(n) \cdot \text{workDonePerLeafNode}(n) \\ &= n \cdot 1 = n\end{aligned}$$

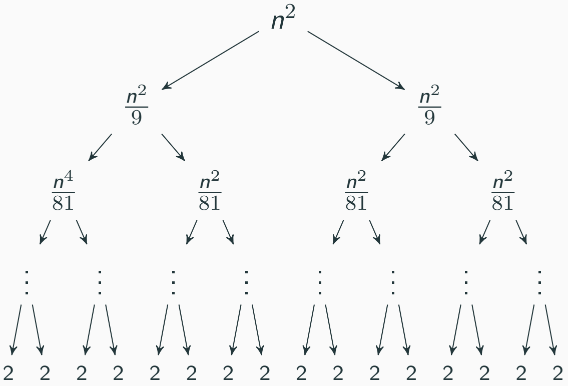
So exact closed form is $n \log_2(n) + n$.

The tree method: practice

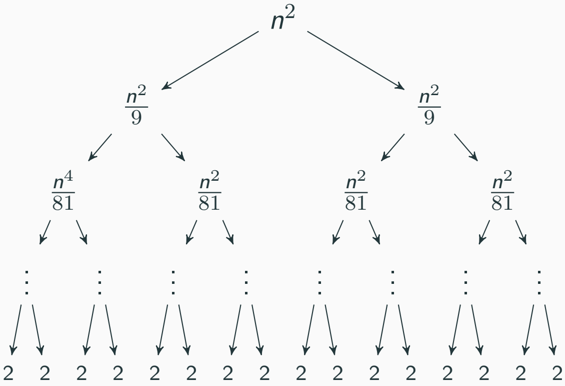
Practice: Let's go back to our old recurrence...

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

The tree method: practice

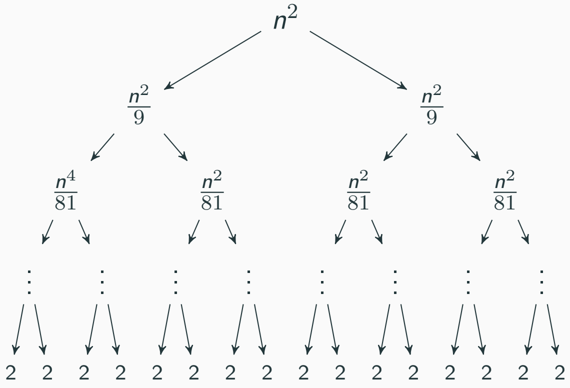


The tree method: practice



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

The tree method: practice



1 node, n^2 work per

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

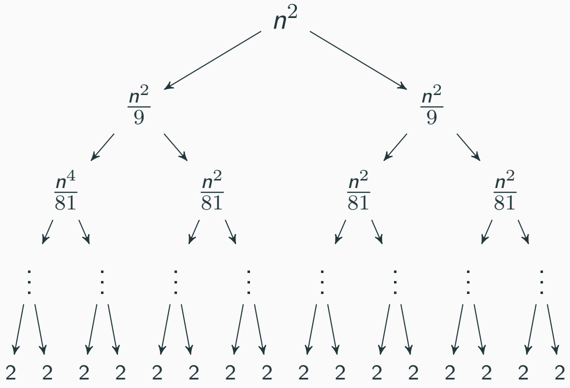
4 nodes, $\frac{n^2}{3^4}$ work per

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

2^h nodes, 1 work per

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

Combine into a single expression representing the total runtime.

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.

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

The tree method: practice

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

$$\begin{aligned}
\text{totalWork} &= n^2 \sum_{i=0}^{\log_3(n)} \left(\frac{2}{9}\right)^i + 2n^{\log_3(2)} \\
&= n^2 \sum_{i=0}^{\log_3(n)+1-1} \left(\frac{2}{9}\right)^i + 2n^{\log_3(2)} \\
&= n^2 \frac{1 - \left(\frac{2}{9}\right)^{\log_3(n)+1}}{1 - \frac{2}{9}} + 2n^{\log_3(2)}
\end{aligned}$$

Applying the finite geometric series

With a bunch of effort...

$$
\begin{aligned}
\text{totalWork} &= n^2 \frac{1 - \left(\frac{2}{9}\right)^{\log_3(n)+1}}{1 - \frac{2}{9}} + 2n^{\log_3(2)} \\
&= \frac{9}{7} n^2 \left(1 - \frac{2}{9}\left(\frac{2}{9}\right)^{\log_3(n)}\right) + 2n^{\log_3(2)} \\
&= \frac{9}{7} n^2 - \frac{2}{7} n^2 \left(\frac{2}{9}\right)^{\log_3(n)} + 2n^{\log_3(2)} \\
&= \frac{9}{7} n^2 - \frac{2}{7} n^2 n^{\log_3(2/9)} + 2n^{\log_3(2)} \\
&= \frac{9}{7} n^2 - \frac{2}{7} n^2 n^{\log_3(2)-2} + 2n^{\log_3(2)} \\
&= \frac{9}{7} n^2 - \frac{2}{7} n^{\log_3(2)} + 2n^{\log_3(2)} \\
&= \frac{9}{7} n^2 + \frac{12}{7} n^{\log_3(2)}
\end{aligned}
$$

The master theorem

Is there an easier way?

The master theorem

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.

The master theorem

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

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

The master theorem

Suppose we have a recurrence of the following form:

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

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:

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

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:

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

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 $T(n) \in \Theta\left(n^{\log_b(a)}\right)$

Sanity check: try checking merge sort.

The master theorem

Given:

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

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 $T(n) \in \Theta\left(n^{\log_b(a)}\right)$

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:

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

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 $T(n) \in \Theta\left(n^{\log_b(a)}\right)$

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 $\Theta\left(n \log(n)\right)$.

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

The master theorem

Given:

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

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 $T(n) \in \Theta\left(n^{\log_b(a)}\right)$

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 $\Theta\left(n \log(n)\right)$.

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

We have $a = 2$, $b = 3$, and $c = 2$. We know $\log_3(2) \leq 1 < 2 = c$, therefore $S(n) \in \Theta\left(n^2\right)$.

The master theorem: intuition

Intuition, the $\log_b(a) < c$ case:

1. We do work more rapidly then we divide.
2. So, more of the work happens near the "top", which means that the n^c term dominates.

The master theorem: intuition

Intuition, the $\log_b(a) > c$ case:

1. We divide more rapidly then 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 $d \cdot a^{\text{height}} = d \cdot a^{\log_b(n)} = d \cdot n^{\log_b(a)}$.

The master theorem: intuition

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: $n^c \log_b(n)$.