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

```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:** Split array roughly into half

```
Unsorted
/
Unsorted
```

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

```
<table>
<thead>
<tr>
<th>Sorted</th>
</tr>
</thead>
</table>
```

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

```
Sortend
```

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

```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

5 10 7 2 3 6 2 11

5 10 7 2

3 6 2 11

5 10 7 2

3 6 2 11

5 10 7 2

3 6 2 11

2 5 7 10

2 3 6 11

2 5 7 10

2 3 6 11

2 5 7 10

2 3 6 11
Merge sort: Analysis

**Pseudocode**

```java
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?

7

8

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

9

Quick sort: Divide step

**Stability and In-place**

If we implement the `merge` function correctly, merge sort will be stable.

However, `merge` must construct a new array to contain the output, so merge sort is **not in-place**.

10

Quick sort: Core pieces

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

- $P$: Pivot
- $\leq P$: Numbers $\leq$ pivot
- $> P$: Numbers $> pivot$

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

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

- $\leq P$: Numbers $\leq pivot$
- $P$: Pivot
- $> P$: Numbers $> pivot$

11

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**

```java
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

20  50  70  10  60  40  30

Quick sort: Example

20  50  70  10  60  40  30

10  50  70  60  40  30

10  50  70  60  40  30

20  50  70  10  60  40  30

10  50  70  60  40  30

20  50  70  10  60  40  30

30  60

30  60
Quick sort: Example

10  20  30  40  50  60  70

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

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 \( \Theta(n^2) \).
Quick sort: Analysis

**Stability**

Quick sort is **not stable** – our partition step ends up disregarding and sometimes ignoring the existing relative ordering of duplicate elements.

**In-place?**

Quick sort is **in-place** – see next few slides for details!

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 :(

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: Partitioning (using median-of-three pivot)

Find the lo, med, and hi

<table>
<thead>
<tr>
<th>8</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>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(0)</td>
<td>a(1)</td>
<td>a(2)</td>
<td>a(3)</td>
<td>a(4)</td>
<td>a(5)</td>
<td>a(6)</td>
<td>a(7)</td>
<td>a(8)</td>
<td>a(9)</td>
</tr>
</tbody>
</table>
Quick sort: Partitioning (using median-of-three pivot)

Find the lo, med, and hi

8 1 4 9 0 3 5 2 7 6


Find the median of the three and swap with front

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

24

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 9 0 3 5 2 7 8


low

1 ≤ 6

high

8 > 6

25

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 9 0 3 5 2 7 8


low

9 ≤ 6

high

2 > 6

25
Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

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

Partitioning:

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

<table>
<thead>
<tr>
<th>low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 ≤ 6</td>
<td>9 &gt; 6</td>
</tr>
</tbody>
</table>

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

Array after moving pivot:

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

Partitioning:

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

<table>
<thead>
<tr>
<th>low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 ≤ 6</td>
<td>5 &gt; 6</td>
</tr>
</tbody>
</table>

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

Array after moving pivot:

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

Partitioning:

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

<table>
<thead>
<tr>
<th>low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 ≤ 6</td>
<td>5 &gt; 6</td>
</tr>
</tbody>
</table>

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

Array after moving pivot:

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

Partitioning:

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

<table>
<thead>
<tr>
<th>low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 &gt; 6</td>
<td>9 &lt; 6</td>
</tr>
</tbody>
</table>

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

Array after moving pivot:

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

Partitioning:

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

<table>
<thead>
<tr>
<th>high</th>
<th>low</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 &gt; 6</td>
<td>9 &lt; 6</td>
</tr>
</tbody>
</table>

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

Array after moving pivot:

![Array after moving pivot]

Partitioning:

![Partitioning]

Quick sort: Core pieces revisited

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

Unsorted ≤ P

Unsorted > P

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:

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

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:

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

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

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

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

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

\[ \ldots \]

\[ = n \log(n) \]

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

\[ T(n) \]
The tree method: example

Step 2: Replace with definition

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

Step 3: Stick each recursive call into a subtree

\[ \frac{n}{2} \]

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

Step 4: Replace with definition

Repeat step 3 (move recursive call to subtrees):

\[ \frac{n}{4} \]

\[ T\left(\frac{n}{4}\right) \quad T\left(\frac{n}{4}\right) \]

Repeat step 4 (replace recursive call with definition):

\[ \frac{n}{8} \]

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

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

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

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

Final step: how much work does each base case do?
The tree method: example

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

\[ \begin{align*}
S(n) &= \begin{cases} 
2 & \text{if } n \leq 1 \\
2S(n/3) + n^2 & \text{otherwise}
\end{cases} 
\]

Draw a tree to help you visualize the work done.

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:

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

Draw a tree to help you visualize the work done.

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

1. \( \text{numNodes}(i) = 2^i \)
2. \( \text{workPerNode}(n, i) = \frac{n}{i} \)
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 \( \log_2(n) \)?

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

```
   4
  / \  \
2   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 height.

---

The tree method: practice

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

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

Combine into a single expression representing the total runtime.

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

---
The finite geometric series

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

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

Plug and chug:

\[
\text{totalWork} = n^2 \sum_{i=0}^{\log_b(n)} \left( \frac{2}{9} \right)^i + 2n^{\log_b(2)} \\
= n^2 \sum_{i=0}^{\log_b(n)+1-1} \left( \frac{2}{9} \right)^i + 2n^{\log_b(2)} \\
= n^2 \frac{1 - \left(\frac{2}{9}\right)^{\log_b(n)+1}}{1 - \frac{2}{9}} + 2n^{\log_b(2)}
\]

Applying the finite geometric series

With a bunch of effort...

\[
\text{totalWork} = n^2 \frac{1 - \left(\frac{2}{9}\right)^{\log_b(n)+1}}{1 - \frac{2}{9}} + 2n^{\log_b(2)} \\
= \frac{9}{7} n^2 \left[ 1 - \frac{2}{9} \left( \frac{2}{9} \right)^{\log_b(n)} \right] + 2n^{\log_b(2)} \\
= \frac{9}{7} n^2 - \frac{2}{7} n^2 \left[ \left( \frac{2}{9} \right)^{\log_b(n)} \right] + 2n^{\log_b(2)} \\
= \frac{9}{7} n^2 - \frac{2}{7} n^2 \left( \frac{2}{9} \right)^{\log_b(2)+2} + 2n^{\log_b(2)} \\
= \frac{9}{7} n^2 - \frac{2}{7} n^{\log_b(2)+2} + 2n^{\log_b(2)} \\
= \frac{9}{7} n^2 + 2 \frac{12}{7} n^{\log_b(2)}
\]

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-\(\Theta\) bound, yes.

The master theorem

Given: \[ T(n) = \begin{cases} \\
\quad d & \text{if } n = 1 \\
\quad aT\left(\frac{n}{b}\right) + \Theta(n^c) & \text{otherwise} \\
\end{cases} \]

Then...

\[ T(n) \in \Theta\left(n^c\right) \]

Intuition, the \(\log_b(n) < c\) 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.

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

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

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

1. We 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
   \[ d \cdot \text{height} = d \cdot \log_b(n) = d \cdot n^{\log_b(a)}. \]

**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: \( n \cdot \log_b(n) \).