1 Analyzing Running Time

We will use merge sort to demonstrate how to analyze the running time of a divide-and-conquer algorithm.

1.1 The Algorithm

MergeSort(a[]):
0) base case
1) split a[] into b[] and c[]
2) MergeSort(b[]), MergeSort(c[])
3) Merge(b[], c[])

The running time (or cost) of merge sort is consists of merge cost (the cost of step 0, 1 and 3) and recursion cost (the cost of step 2). The merge cost can be analyzed directly.

1.2 The Recurrence Relation $T(n)$

Let $T(n)$ be the running time of the algorithm for an input of size $n$. We will analyze the merge cost, which will be a function of $n$, as well as the recursion cost, which will be written as $T(k)$ for some $k < n$. From there, $T(n)$ is simply the sum of merge cost and recursion cost.

For merge sort, the merge cost is $O(n)$, as we go through and combine two sorted sublists in linear time. The recursion cost is $2T(n/2)$, since there are 2 recursive calls, and the input size for each call is $n/2$. Therefore, we have $T(n) = 2T(n/2) + O(n)$. Since the base case is a list of size 1 that does not need to be sorted, i.e. $T(1) = 0$, we can be more precise and write $T(n) = 2T(n/2) + n$.

1.3 The Analysis

So how do we solve the recurrence relation $T(n)$? There are 2 methods in general to upper-bound the running time.
1.3.1 Guess-and-Verify

A guess: \( T(n) \leq cn\log_2 n \). We will verify this guess by proving \( T(n) \leq cn\log_2 n \) for some \( c \), by strong induction:

**Proof.** Induction hypothesis: \( T(n) \leq cn\log_2 n \) for some \( c \).

Base case: when \( n = 1 \), \( T(1) = 0 \leq c \cdot 1 \log_2 1 \) is true for every \( c \).

Induction step: suppose IH is true for all \( k < n \). We will show that IH is also true for \( n \).

\[
T(n) = 2T(n/2) + n \quad \text{(by recurrence relation)}
\]
\[
\leq 2(cn/2 \log_2 n/2) + n \quad \text{(by IH)}
\]
\[
= 2cn/2 \log_2 n - 1 + n
\]
\[
= cn\log_2 n - cn + n
\]

And \( T(n) \leq cn\log_2 n - cn + n \leq cn\log_2 n \) is true whenever \( c \geq 1 \).

Therefore, \( T(n) \leq cn\log_2 n \) for some \( c \). Hence \( T(n) = O(n\log n) \)

1.3.2 Recursion Tree

We will draw a tree of all recursive calls: each node represents a recursive call; each edge represents one call calling another; each leaf is a base case. From there, \( T(n) \) can be calculated by summing the merge cost over every node in the recursion tree. For merge sort, we have:

Below is a summary of this tree.

<table>
<thead>
<tr>
<th>Depth</th>
<th>Number of nodes</th>
<th>Problem size (each node)</th>
<th>Total problem size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>( n/2 )</td>
<td>( n )</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>( n/4 )</td>
<td>( n )</td>
</tr>
<tr>
<td></td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>( \log_2(n) - 1 )</td>
<td>( 2^{\log_2(n)} = n )</td>
<td>1</td>
<td>( n )</td>
</tr>
</tbody>
</table>
We are now ready to sum the merge cost over every node in the recursion tree, which is simply

\[
T(n) = \sum_{i=0}^{\log_2 n - 1} \text{merge cost for level } i
\]

\[
= \sum_{i=0}^{\log_2 n - 1} n
\]

\[
= n \log_2 n
\]

One way to interpret the recursion tree method is that we are substituting in the expression for lower levels. That is:

\[
T(n) = 2(T/2) + n
\]

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

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

\[\ldots\]

From right to left, we can see that we are essentially summing the merge cost for layer 0, layer 1, layer 2, etc. Hence, we are finding

\[
\sum_{i=0}^{\text{# layers}} \text{merge cost for layer } i
\]