1. Time analysis of algorithms

Beyond being correct, we would like a good algorithm to be efficient. In order to determine efficiency, we’ll discuss how we determine how long an algorithm takes.

2. Merge sort

Let’s start by looking at algorithm for merge sort. Recall that merge sort is a sorting algorithm which works by taking a list, splitting it in half, recursively sorting the two halves, and then merging them together.

Here’s Python code for merge sort:

```python
1 def merge_sort(L):
2     '''
3     Given a non-empty list, returns the list in non-descreasing order.
4     '''
5     if len(L) == 1:
6         return L
7     else:
8         middle = ((len(L) - 1) / 2) + 1
9         left = L[:middle]
10        right = L[middle:]
11        left = merge_sort(left)
12        right = merge_sort(right)
13        return merge(left, right)
14
15 def merge(L1, L2):
16     '''
17     Given two sorted lists, return a new list of all their elements again
18     in sorted order.
19     '''
20     i = 0
21     j = 0
22     result = []
23     while i < len(L1) and j < len(L2):
24         if L1[i] < L2[j]:
25             result.append(L1[i])
26             i += 1
27         else:
28             result.append(L2[j])
29             j += 1
30     while i < len(L1):
31         result.append(L1[i])
32         i += 1
33     while j < len(L2):
34         result.append(L2[j])
35         j += 1
36     return result
```

3. Running time of merge

First, let’s talk about the running time of `merge`. Before we can determine the running time, we have to decide what we are counting. There are a variety of ways of doing it (e.g. number of operations), but we’ll do the number of comparisons, meaning the number of times we compare two list elements.

Note that this only happens in line 24 of `merge`. So to determine the running time of `merge`, we need to count the number of times the while loop in lines 23-29 is run. Note that `i` starts at the beginning of `L1` and `j` starts
at the beginning of $L_2$. The loop exits when either of them makes it to the end of their respective lists. In each iteration, either $i$ or $j$ is incremented.

The fewest number of times through the loop possible is $\min(L_1, L_2)$ (this happens when all the elements of one list are smaller than all the elements of the other list). The greatest number of times through the loop is $L_1 + L_2 - 1$ (this happens if the the lists alternate, or, more generally, if the second largest element is in a different list from the largest element).

4. Recurrence for merge sort

To compute the running time of recursive algorithms like merge sort, we develop a recurrence. This is a recursive function that represents the running time of the algorithm. If we’re not at the base case, we can think of the running time of merge as follows:

$$
time \text{ of merge sort}(L) = time \text{ of merge sort}(\text{left})$$
$$+ time \text{ of merge sort}(\text{right})$$
$$+ time \text{ of merge}(\text{left, right})$$

We let $T(n)$ be the maximum running time of merge sort on a list of length $n$.

$$T(n) = \begin{cases} 
c & \text{if } n = 1 \\
T \left( \left\lceil \frac{n}{2} \right\rceil \right) + T \left( \left\lfloor \frac{n}{2} \right\rfloor \right) + dn & \text{if } n > 1
\end{cases}$$

where $c$ is the amount of time it takes for the base case (when the list is of size 1) and $dn$ is the amount of time it takes to merge (note that this is not the same for all lists, but it only varies by a constant multiple). $T \left( \left\lceil \frac{n}{2} \right\rceil \right)$ is the running time of merge sort(left) and $T \left( \left\lfloor \frac{n}{2} \right\rfloor \right)$ is the running time of merge sort(right).

5. Solving the recurrence for powers of 2

To simplify things, we first will just deal with the case that $n$ is a power of 2. We do this so that $n$ will always divide easily and we don’t have to worry about taking the floor and ceiling. Here’s our simplified recurrence in this case:

$$T(n) = \begin{cases} 
c & \text{if } n = 1 \\
2T \left( \frac{n}{2} \right) + dn & \text{if } n > 1
\end{cases}$$

Now we want to find a closed form for $T(n)$. This is a non-recursive definition for $T(n)$. We can discover it by doing repeated substitution. This is exactly what it sounds like, we repeatedly substitute in using the recursive definition of $T$ until we see the pattern. Note that this is not a proof, but just a method of determining what we want to prove.

$$T(n) = 2T \left( \frac{n}{2} \right) + dn$$
$$= 2^2 \left( 2T \left( \frac{n}{2^2} \right) + d \frac{n}{2^2} \right) + dn = 2^2T \left( \frac{n}{2^2} \right) + 2dn$$
$$= 2^3 \left( 2T \left( \frac{n}{2^3} \right) + d \frac{n}{2^3} \right) + dn = 2^3T \left( \frac{n}{2^3} \right) + 3dn$$
$$= 2^4 \left( 2T \left( \frac{n}{2^4} \right) + d \frac{n}{2^4} \right) + dn = 2^4T \left( \frac{n}{2^4} \right) + 4dn$$
$$\vdots$$
$$= 2^i T \left( \frac{n}{2^i} \right) + idn$$

When does this stop? Well, when we reach our base case. That’ll happen when $\frac{n}{2^i} = 1$ (when $i = \log_2 n$). So what does the above yield when $i = \log_2 n$?

$$T(n) = 2^i T \left( \frac{n}{2^i} \right) + idn$$
$$= 2^{\log_2 n} T \left( \frac{n}{2^{\log_2 n}} \right) + (\log_2 n)dn$$
$$= nT(1) + dn \log_2 n$$
$$= cn + dn \log_2 n$$

Okay, so now let’s formally prove that this formula we got for $T(n)$ is correct.

Lemma 1. $\forall i \in \mathbb{N}, ((\forall n \in \mathbb{N}, \text{n is a power of 2}) \land i \leq \log_2 n) \rightarrow T(n) = 2^iT \left( \frac{n}{2^i} \right) + idn.$

Proof. Proof by induction on $i$. For all $i \in \mathbb{N}$, define our predicate:

$$[(\forall n \in \mathbb{N}, \text{n is a power of 2}) \land i \leq \log_2 n) \rightarrow T(n) = 2^iT \left( \frac{n}{2^i} \right) + idn]$$

Base Case: Say $i = 0$. Let $n \in \mathbb{N}$ be a power of 2. Then $2^0T \left( \frac{n}{2^0} \right) + idn = 2^0T \left( \frac{n}{2^0} \right) + 0dn = T(n)$.
Induction Step: Let \( i \in \mathbb{N} \) and let \( n \in \mathbb{N} \) be a power of 2 such that \( i + 1 \leq \log_2 n \). Assume \( P(i) \).

\[
T(n) = 2^i T\left(\frac{n}{2^i}\right) + idn \text{ by IH}
\]

\[
= 2^i \left(2T\left(\frac{n}{2^{i+1}}\right) + d\left(\frac{n}{2^i}\right)\right) + idn \text{ by definition of } T \text{ (since } i + 1 \leq \log_2 n, \text{ we have } \frac{n}{2^i} > 1)\]

\[
= 2^{i+1} T\left(\frac{n}{2^{i+1}}\right) + (i+1)dn
\]

From this lemma and the calculations before the lemma, we can conclude that \( T(n) = cn + dn \log_2 n \) (at least when \( n \) is a power of 2).

### 6. Running time of merge sort in general

We’ll do this by showing that \( T(n) \) is nondecreasing.

**Lemma 2.** \( T(n) \) is nondecreasing.

**Proof.** For all \( n \in \mathbb{N} \), define the predicate \( T(n) \leq T(n+1) \). Let \( n \in \mathbb{N} \). Assume \( \forall k \in \mathbb{N}, k < n \rightarrow P(k) \).

**CASE 1:** Say \( n = 1 \). Then we have \( T(n) = T(1) = c \) and \( T(n+1) = T(2) = 2c + 2d \), so \( T(1) \leq T(2) \).

**CASE 2:** Now say \( n > 1 \). Then:

\[
T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + T\left(\left\lceil \frac{n}{2} \right\rceil \right) + dn \text{ by IH and some properties of integers}
\]

\[
\leq T\left(\left\lfloor \frac{n+1}{2} \right\rfloor \right) + T\left(\left\lceil \frac{n+1}{2} \right\rceil \right) + dn \text{ by IH and } n \text{ odd, } n \text{ even.}
\]

(If you don’t see how we got from step 1 to 2, look at 2 cases: \( n \) even and \( n \) odd.)

Now we’re ready to show that the running time of merge sort is \( n \log_2 n \) (times some constant).

**Theorem.** \( \forall k \in \mathbb{N}, n \geq 2 \rightarrow T(n) \leq kn \log_2 n \).

**Proof.** Let \( n' \) be the smallest power of 2 that’s larger than or equal to \( n \). So \( \frac{n'}{2} < n < n' \). Then \( T(n) \leq T(n') \) by the nondecreasing lemma. So,

\[
T(n) \leq T(n') \leq cn' + dn' \log_2 n' \leq 2cn + 2dn \log_2 n - 1 < 2cn + 2dn \log_2 n
\]

If we let \( k = 2c + 2d \), then \( T(n) \leq k \log_2 n \).

### 7. Divide and conquer algorithms

Merge sort is an example of what we call a *divide and conquer algorithm*. These are algorithms that fit this model:

- Divide up an instance of size \( n \) into \( a \) instances of the same problem, each of size approximately \( n/b \).
- Recursively solve each of the smaller instances.
- Combine the solutions.

For merge sort, \( a = b = 2 \).

First, let’s look at the simpler recurrence that doesn’t have rounding issues. We would like \( n/b \) to always be an integer, so we assume that \( n \) is a power of \( b \). Our recurrence looks like this:

\[
T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  aT\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + f(n) & \text{if } n > 1 
\end{cases}
\]

\( f(n) \) is the time for combining the solutions (for merge sort, this was \( dn \)).

For a general \( n \), \( T(n) \) looks like this slightly uglier recurrence:

\[
T(n) = \begin{cases} 
  c & \text{if } 1 \leq n < b \\
  a_1 T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + a_2 T\left(\left\lceil \frac{n}{2} \right\rceil \right) + f(n) & \text{if } n > 1 
\end{cases}
\]

(note that \( a_1 + a_2 = a \).)

But, in your head, you should think of it like the first. The rounding only changes things by a negligible amount, and we can always solve the simpler case and then deal with the \( n \)'s that aren’t powers of 2 like we did with merge sort (show that \( T(n) \) is nondecreasing).