Merge Sort

Fred Rossi. 12/1/2022.

Merge

The `Merge` procedure merges two (pre) sorted lists into one sorted list.

```python
def merge(xs, ys):
    i, j = 0, 0

    res = []
    while i < len(xs) and j < len(ys):
        x, y = xs[i], ys[j]

        if x <= y:
            res.append(x)
            i += 1
        else:
            res.append(y)
            j += 1

    while i < len(xs):
        res.append(xs[i])
        i += 1

    while j < len(ys):
        res.append(ys[j])
        j += 1

    return res
```

Questions:

- What is its running time?
- Is that best case or worst case? (Does it matter?)
- If I asked you to prove correctness, could you?

Merge Sort

The `MergeSort` procedure recursively sorts a list. It works by:

1. Splitting the list into lower and upper halves
2. Recursively sorting each half by a call to _MergeSort
3. Combining the sorted halves with `Merge`.

```python
def merge_sort(xs):
    n = len(xs)
    if n < 2:
        return xs

    m = n // 2 # midpoint

    xs_low = merge_sort(xs[:m])
    xs_high = merge_sort(xs[m:])
    return merge(xs_low, xs_high)
```

Let's visualize the recursion of `merge_sort` on 8, 4, 9, 8,7, 1.

```
             ┌───────────────────────┐
             │ 8 | 4 | 9 | 8 | 7 | 1 │
             └───────────────────────┘
[split]        /                    \
         ┌───────────┐          ┌───────────┐
         │ 8 | 4 | 9 │          │ 8 | 7 | 1 │
         └───────────┘          └───────────┘
[split]  /      \                  /      \
      ┌───┐  ┌───────┐           ┌───┐ ┌───────┐
      │ 8 │  │ 4 | 9 │           │ 8 │ │ 7 | 1 │
      └───┘  └───────┘           └───┘ └───────┘
[split] |     /     \              |     /    \
      ┌───┐ ┌───┐  ┌───┐         ┌───┐ ┌───┐ ┌───┐
      │ 8 │ │ 4 │  │ 9 │         │ 8 │ │ 7 │ │ 1 │
      └───┘ └───┘  └───┘         └───┘ └───┘ └───┘
[merge] |     \     /              |     \    /
      ┌───┐  ┌───────┐           ┌───┐ ┌───────┐
      │ 8 │  │ 4 | 9 │           │ 8 │ │ 1 | 7 │
      └───┘  └───────┘           └───┘ └───────┘
[merge]  \      /                   \     /
         ┌───────────┐          ┌───────────┐
         │ 4 | 8 | 9 │          │ 1 | 7 | 8 │
         └───────────┘          └───────────┘
[merge]         \                    /
             ┌───────────────────────┐
             │ 1 | 4 | 7 | 8 | 8 | 9 │
             └───────────────────────┘
```

Questions:

- What is its running time?
- Is that best case or worst case? (Does it matter?)

How do we figure out the running time?

First, Let $T(n)$ denote the time it takes process a list of length n. Observe that, ignoring recursive subcalls, the running time of a single call to `merge_sort` is $\Theta(n)$ due to call to `merge`. Therefore, $T(n)$ is equal to the time to handle two recursive sub-calls of size $n/2$, plus $\Theta(n)$. That is, $T(n) \leq 2T(n/2) + \Theta(n)$.

In other words, there exists $c_1, c_2 \in \mathbb{R}^+$, and $n_0 \in \mathbb{N}_0$, such that for all $n \geq n_0$, we have

$$2T(n/2) + c_1 n \leq T(n) \leq 2T(n/2) + c_2 n.$$

Now let's find an upper bound on the running time. How? by plugging this thing into itself over and over. Let's see what I mean.

First, Focusing on the right – where it says $T(n) \leq 2T(n/2) + c_2 n$ – let's apply the inequality to itself. Plugging in our inequality for T(n) into $T(n/2)$, we see

$$T(n) \leq 2T(n/2) + c_2 n \leq 2\left(2T(n/2^2) + c_2 \frac{n}{2}\right) + c_2 n = 2^2 T(n/2^2) + 2c_2 n.$$

What did that do? You might not be able to see it yet, but it peeled away one level of recursion. Let's do it again and go for another.

We now see with a second substitution that

$$T(n) \leq 2^2 \left(2T(n/2^3) + c_2 n/2^2\right) + 2c_2 n.$$

That is,

$$T(n) \leq 2^3 T(n/2^3) + 3c_2 n$$

It therefore follows, if we did this $k-1$ times, that after $k-1$ substitutions,

$$T(n) \leq 2^k T(n/2^k) + kc_2 n.$$

If you look closely each time we substitute we are unwrapping a recursive "layer". The time to process a list of length n is no more than the time to process 2^k subcalls of size $n/2^k$ plus k times $c_2 n$.

That's interesting, but it appears we can unwrap to arbitrary depth in the sense that we plug in any k we want to peel away $k-1$ layers. We could make k infinitely big.

Is that right? Well sort of. On one hand there's no limit to the number of times merge sort could recurse. On the other, eventually when the list is small enough the recursion needs to bottom out (check the source for `merge_sort`). When then happens, it takes constant time. But when does that happen? It depends on how long the list is. That is, it's a function of n.

The next step to making progress is relating k and n. This happens by noting that for a fixed size n, the recursion must terminate at/before k becomes big enough that we've divided the input into single elements. That is, at/before $n/2^k \leq 1$. Solving for k we see this is once $k \geq \log_2(n)$.

This is good. We don't want the k's, we want an answer in terms of n. Let's plug this in and simplify as best we can. Returning to the inequality, it follows that at the time of termination $T(n) \leq 2^{\log_2(n)} T(n/2^{\log_2(n)}) + \log_2(n)c_2 n$. And, further simplifying, $T(n) \leq nT(1) + \log_2(n)c_2 n$.

Now observe that $T(1) \in \Theta(1)$ since the time it takes to process a constant-sized list is, at worst, constant. It follows there exists some $c \in \mathbb{R}^+$ such that $T(n) \leq nc + \log_2(n)c_2 n \in \Theta(n\log(n))$. In other words, $T(n) \in O(n \log n)$.

Wow, okay, so it's no slower than $n\log(n)$-time (i.e. no more than something proportional to $n\log(n)$ steps).

However, aren't we expecting a Theta? What happened?

Oh! We forgot about the lower bound!

Now observe that a symmetric argument works from below. That is, after $k-1$ subsitutions, we have time taking at least $2^k T(n/2^k) + kc_1 n$, so that once $n/2^k \leq 1$, we have $k = \log_2(n)$ and $nc' + \log_2(n)c_1 n \leq T(n)$. It follows immediately that $T(n) \in \Omega(n \log n)$.

Therefore, `merge_sort` takes $\Theta(n \log n)$-time. Cool!