Divide-and-Conquer and Reduce

1 Reductions

Earlier that we saw a simple divide-and-conquer algorithm for computing sums.

Algorithm: Parallel Sequence Sum

```
fun sum(S : sequence<int>) -> int:
  match length(S) with:
    case 0: return 0        // Empty sequence
    case 1: return S[0]     // Singleton sequence
    case _:
      L, R = split_mid(S)  // Helper function
      Lsum, Rsum = parallel (sum(L), sum(R))
      return Lsum + Rsum
```

This algorithm runs in $O(n)$ and $O(\log n)$ span. As it stands, it only computes sums, but it turns out we can generalize this idea to compute more, and not just a little bit more (like multiplication), but **a lot more**. In this lecture we will explore the power of generalizing the idea of a sum and how far it can take us in parallel algorithm design.

1.1 Generalizing Sums: Folds

There are many ways to generalize the notion of a "sum". For now, we will think of a sum as addition (for us, of numbers, but if you do a lot of math you might think of adding elements of a group, or if you want to sound fancy, a monoid...).

$$s_0 + s_1 + s_2 + \cdots + s_{n-1} + s_n$$

The first generalization we will consider is a common and powerful tool in functional programming called a **fold**.

Definition: Fold

A (left) fold over a sequence s with a binary operation f and an initial value I computes

$$f(f(f(f(f(I, s_0), s_1), s_2), \ldots, s_{n-1}, s_n).$$

This generalizes a sum because if we pick the operation f to be addition, i.e., $f(x, y) = x + y$, then a fold over this function with initial value 0 is precisely the sum of the input sequence.

We call this a **left fold** specifically because it applies the function f to the elements of s in left-to-right order. One can define a right fold in the same way, except the function is applied right-to-left instead. Note that for an arbitrary function f, applying the function in a different order may result in a completely different output.

Therefore, to compute a fold, this requires an inherently sequential algorithm. If we have no guarantees about the behavior of f, the best we can do to ensure we get the correct answer is apply it left-to-right and compute the fold expression exactly as prescribed by the definition. This leaves no room for parallelism!

1.2 Folds without order: Reduce

We would like to be able to take inspiration from our parallel divide-and-conquer sum algorithm, but for arbitrary operations f, there is essentially no hope. To make this feasible, we need to restrict our attention to values of f for which we can relax the order of evaluation, which will make room for parallelism.

A **reduce** over a binary operator is a fold that can be evaluated **in any order** and always yield the same result. This is equivalent to saying the expression can arbitrarily parenthesized. In terms of sums, it is equivalent to saying we can evaluate left-to-right, right-to-left, or like

$$(s_0 + (s_1 + s_2)) + \cdots + (s_{n-1} + s_n).$$

Definition: Reduce

A reduce over a sequence s with a binary **associative** operation f and an **identity** value I computes a fold of f over s where the order of application of f may be arbitrary.

For instance, the above example of arbitrarily re-parenthesizing a sum is equivalent to evaluating an arbitrary binary operation f like so

$$f\left(f\left(s_0, f(s_1, s_2)\right), \cdots, f(s_{n-1}, s_n)\right).$$

The powerful insight that underlies most of today's lecture is that being allowed to order to evaluations of f in any order gives us the ability to do divide-and-conquer, since splitting the input sequence in half is essentially making the choice to parenthesize the left half and right half and then evaluate them recursively, before finally evaluating their results.

Algorithm: Parallel Reduce

```
fun reduce(f : (T, T) -> T, I : T, S : sequence<T>) -> T:
  match length(S) with:
  case 0: return I
  case 1: return S[0]
  case _:
    L, R = split_mid(S)
    Lres, Rres = parallel (reduce(L), reduce(R))
    return f(Lres, Rres)
```

1.3 Understanding Reduce

Reduce comes with two requirements that are not requirements of an ordinary fold. First, f must be an **associative function**.

Definition: Associative function

An associative function over a type T satisfies for all values x, y, z of type T:

$$f(f(x, y), z) = f(x, f(y, z)).$$

Second, the initial value must be an **identity value**

Definition: Identity

I is an identity for type T if for all values x of type T:

$$f(I, x) = f(x, I) = x.$$

1.4 Cost of Reduce

When analyzing the cost of reduce, we need to make assumptions about the cost of f. In general, if f can have an arbitrary cost, we cannot infer anything about reduce. The simplest case is when f can be evaluated in constant time.

Theorem: Cost of Reduce

Assuming f can be evaluated in $O(1)$ time, reduce costs $O(|S|)$ work and $O(\log |S|)$ span.

Proof. The work recurrence is

$$W(n) = 2W\left(\frac{n}{2}\right) + O(1),$$

which is leaf dominated as has $O(n)$ leaves which do $O(1)$ work.

The span recurrence is

$$S(n) = S\left(\frac{n}{2}\right) + O(1),$$

which unrolls to $O(\log n)$. □

2 "Generic" Divide-and-Conquer

The implementation of reduce should look awfully familiar and remind us of some algorithms that we wrote just recently.

2.1 SMCSS

Recall MCSSLab from Week 1, where we implemented the "strengthened MCSS" problem using divide-and-conquer. The algorithm looked something like this:

```
fun smcss(S : sequence<int>) -> (int,int,int,int):
  match length(S) with:
    case 0: return (0,0,0,0)
    case 1:
      m = max(0, A[0])
      return (m, m, m, A[0])
    case _:
      L, R = split_mid(S)
      (m1,p1,s1,t1), (m2,p2,s2,t2) = parallel (smcss(L), smcss(R))
      return (max(s1 + p2, m1, m2),
              max(p1, t1 + p2),
              max(s2, t2 + s1),
              t1+t2)
```

The code is also divide-and-conquer, but beyond that it has striking similarities to our implementation of reduce. It has an empty sequence case, a length 1 base case, and a general case which splits the input into two halves, recursively solves the problem on those halves, and then combines those answers.

What is remarkable is that `smcss` is actually just a **special case of reduce**. If we factor out the combining-the-two-halves logic into its own function,

```
type sums = (int,int,int,int)

fun combine_smcss((m1,p1,s1,t1) : sums, (m2,p2,s2,t2) : sums) -> sums:
  return (max(s1 + p2, m1, m2),
          max(p1, t1 + p2),
          max(s2, t2 + s1),
          t1+t2)
```

Then we can observe that we can implement `smcss` as a reduce like so:

Algorithm: SMCSS as Reduce

```
fun smcss(S : sequence<int>) -> (int,int,int,int):
  fun base(x : int): return (max(0,x),max(0,x),max(0,x),x)
  return reduce(combine_smcss, (0,0,0,0), map(base, S))
```

That is, we use a **map** to solve the base cases by converting each individual input element into its corresponding base case value, then we perform the recursive divide-and-conquer by simply passing `combine_smcss` to `reduce` as the binary operation!

This is not an isolated coincidence. Let's look at another example.

2.2 Parenthesis Matching

In the *Parenthesis Matching* recitation, we continued to practice divide-and-conquer by giving an efficient parallel implementation of the parenthesis matching problem. Specifically, this strengthened parenthesis matching problem counted the number of excess left and right parenthesis in a given subexpression.

```
fun excessParens (ps : sequence<Paren>) -> (int,int):
  match length(ps) with:
    case 0: return (0, 0)
    case 1:
      if (ps[0] == L): return (0, 1)
      else: return (1, 0)
    case _:
      a, b = split_mid(ps)
      ((i,j),(k,l)) = parallel (excessParens (a), excessParens (b))
      if (j <= k): return (i + k - j, l)
      else: return (i, l + j - k)
```

Again, we see the same pattern emerge: an empty case, a length-1 sequence base case, and a general case which splits the input into two and recursively solves the problem on the two halves before combining those results together into a final result.

We can once again factor out the "combination" logic into its own function.

```
fun combine_paren((i : int, j : int), (k : int, l : int)) -> (int, int):
  return (i + k - j, l) if j <= k else (i, l + j - k)
```

Using this combination function, we can once again rewrite this entire algorithm as a reduce!

Algorithm: exccessParens as Reduce

```
fun excessParens(p : sequence<Paren>) -> (int,int):
  fun base(x : Paren): return (0,1) if x == L else (1,0)
  return reduce(combine_paren, (0,0), map(base, p))
```

2.3 Reduce as "Generic Divide-and-Conquer"

This observation gives us a recipe to convert many (but not all) divide-and-conquer algorithms into much shorter code by using reduce. In general, this technique is applicable whenever the "divide" step of the divide-and-conquer is trivial: it must simply split the input in half and recurse. If the algorithm needs additional steps before recursion, it isn't applicable.

For example, Quicksort doesn't just break the input into two halves and recurse, it partitions the input into those elements less than the pivot and those greater than the pivot, then recurse on those—so its not in the right format. We saw, however, that many algorithms are, like the `smcss` problem and the `excessParen` problem.

Generally, if an algorithm is in the following form:

```
fun algo(S : sequence<T>):
  match length(S) with:
  case 0: return empty
  case 1: return base(S[0])
  case _:
    L, R = split_mid(S)
    Lres, Rres = parallel (algo(L), algo(R))
    return combine(Lsum, Rsum)
```

then we can convert it to an equivalent reduce like so:

Algorithm: "Generic" Divide-and-Conquer

```
fun algo(s : sequence<T>):
  return reduce(combine, empty, map(base, s))
```

Note that `map` being used to handle the base case makes this strictly more generic than a `reduce` on its own which would require that the output of the algorithm be the same as the input type `T` of the sequence. Using `map` allows us to make the output type different, which is powerful and necessary in most applications of this technique.

3 Merge

Recall the merge operation from 15-122 and 15-150, which is often motivated as a subroutine for the sorging algorithm Merge Sort.

Definition: Merge

Given two sorted sequences, return a sorted sequence containing the elements of both.

That is, we want to implement a function of the following form:

```
fun merge(A : sequence<T>, B : sequence<T>) -> sequence<T>
```

where both inputs A and B must be sorted as a precondition and the output must be equivalent to `sort(append(A, B))` (but hopefully computed much more efficiently than that).

We know from 15-122 and 15-150 that merge is easy to implement *sequentially* in $O(n)$ time, with either an array or a linked list. We are interested in a parallel implementation that is just as efficient, i.e., we want to aim for $O(n)$ work and reasonably low span.

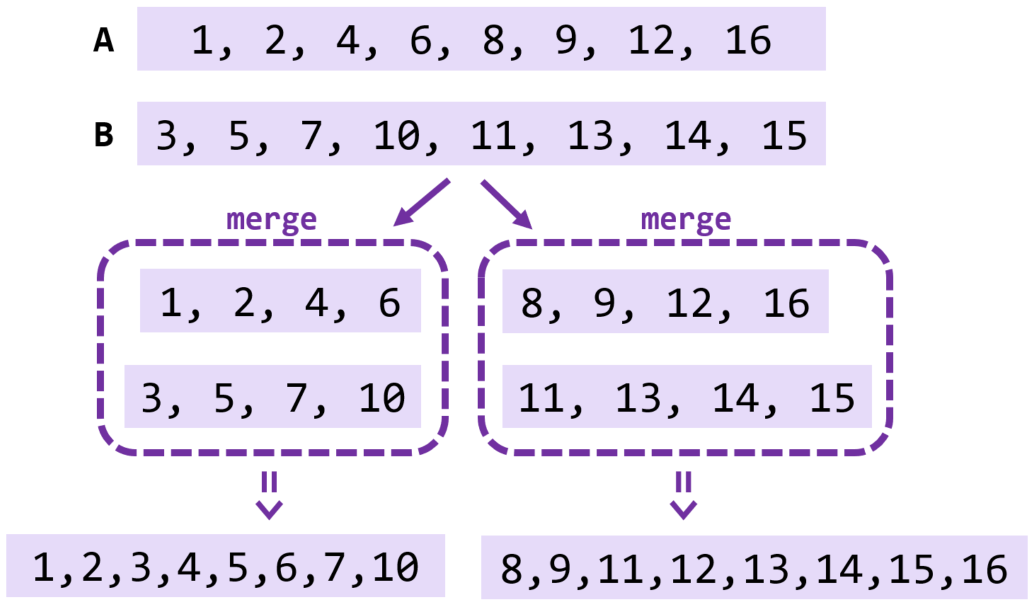
As the theme of today's lecture suggests, we are going to use divide-and-conquer!

3.1 Divide-and-conquer Merge

Merge presents one clear challenge that makes it different from previous divide-and-conquer algorithms we have looked at. The input is not just one sequence, but two! Which one do we split? And how do we split them?

We could start with our usual tried-and-tested method of just splitting everything in half and see what happens, but something will go wrong.

Let's say we start with the following two sequences, which we split in half, recursively merge, and then try to combine back together.



The issue that becomes apparent is that there is no easy way to combine the two recursive results. Indeed, what we have as the two recursive results are two sorted sequences that need merging! Oops, we are right back to the same problem we started with but with absolutely no progress... Clearly something is wrong.

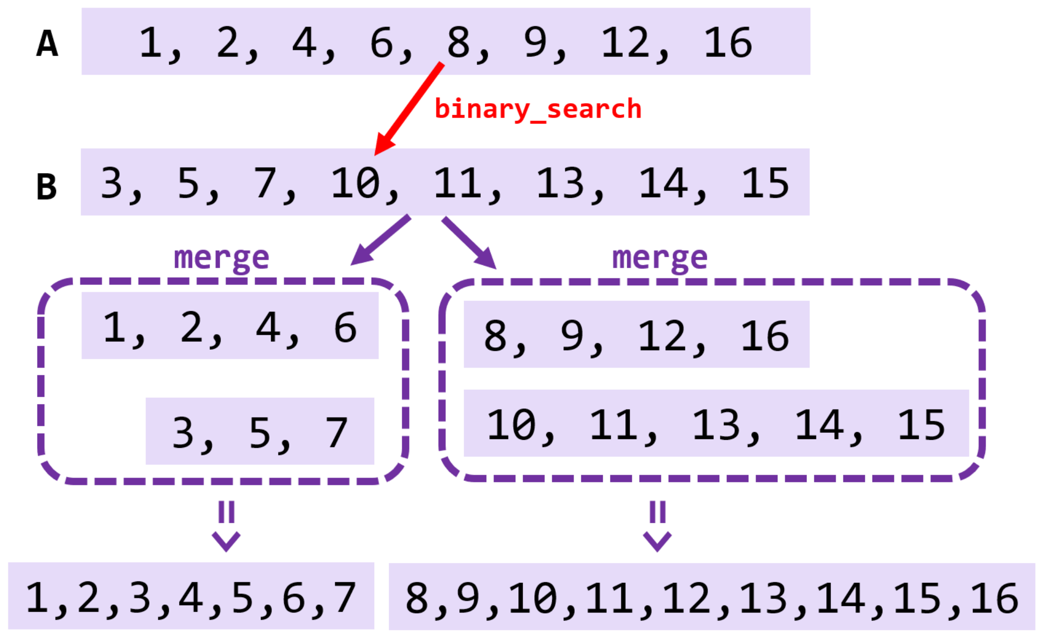
The issue is that we chose to split both sequences down the middle, but this resulted in two sequences that were still interleaved. Instead, we want to split the sequences so that the left one is *strictly smaller* than the right one—so that when we are done the combine step can simply be to append the two sequences and then we are done!

3.1.1 Finding the correct split point

Our goal as stated above is to find a split point such that the left sequence will contain elements smaller than the right sequence. We can still split the first sequence (A) in half—since its sorted, all of the elements on the left are smaller than the elements on the right. We need to figure out the corresponding split point of B, which may not be the middle.

To ensure that the same property is true of the merged sequences, that the left half is smaller than the right half, we need to then split B so that the left side of the split only contains elements that are smaller than the *right half of A*. We can do this by selecting the smallest element in the right half of A, and **binary searching** for that element in B, and using the resulting position as the split point for B. This results in two left halves whose elements are all less than the elements of the two right halves!

Starting with the same two sequences as before, let's see what happens when we split A in half and use binary search to find the corresponding split point in B.



This time, we end up with two sorted sequences which now crucially have the property that the left one contains elements smaller than the right one. So the combine step is to simply append them and we are done!

3.1.2 Balancing the divide-and-conquer

Since we are now splitting not necessarily down the middle, we have to be careful to ensure that our divide-and-conquer actually makes progress in reducing the problem size. If we always split the same sequence in half and leave the other sequence imbalanced, its not clear that this will result in good work and span bounds.

Luckily there is a simple fix to this. Instead of arbitrarily choosing `A` to be the sequence that gets split in half and `B` to be the sequence that gets split in a possible imbalanced way, we can just *always split the larger sequence in half.* Therefore after about $2\log(|A|+|B|)$ levels of recursion we can be assured that both will have been completely split down to single elements!

3.2 Implementing Merge

Armed with this knowledge, we have enough information to implement a reasonable version of `merge`. We just need to add in some base cases. The following should suffice:

- When one of the sequences is empty, just return the other one.
- When both sequences are length one, return a sequence of the min and max of the two.

In all other cases, at least one of the sequences has length at least two and can therefore be split in the recursive case which will bring us closer to a base case, so this is enough for termination.

Unfortunately, this algorithm is still not going to be as efficient as we want! It has a slight problem which will lead to it performing slightly too much work.

Algorithm: Slightly inefficient merge

```
fun merge(A: sequence<T>, B : sequence<T>) -> sequence<T>:
  if |B| > |A|: swap(A, B) // WLOG assume A is larger than B

  if |B| == 0: return A
  if |A| == |B| == 1: return [min(A[0], B[0]), max(A[0], B[0])]

  LA, RA = split_mid(A)
  k = binary_search(B, RA[0])  // smallest index s.t. B[k] >= RA[0]
  LB, RB = subseq(B,0,k), subseq(B,k,|B|-k)
  ML, MB = parallel (merge(LA, LB), merge(RA, RB))
  return append(ML, MB)
```

3.2.1 Cost Analysis

Claim: Cost of Inefficient Merge

This implementation of merge costs $\Theta(n \log n)$ work.

Proof. The issue is `append` at the combine stage. This costs linear work in the size of the sequences, which will make the work recurrence balanced, performing $\Theta(n)$ work per level across $\Theta(\log n)$ levels, for a total of $\Theta(n \log n)$. □

Idea: Removing the Append

Combining the two recursive calls with `append` is the natural pure way to implement merge. However, we can speed it up significantly by allowing ourselves some leeway to write **imperative/impure code**. Instead of appending the results together into a new sequence, we can instead have the recursive calls simply write their answer directly into a *pre-allocated* output array. Therefore zero additional work has to be done by the combine step—when the recursive calls complete, the answer will already be there!

3.3 Efficient (Impure) Merge

Our improved merge algorithm lives outside the blissful safety of pure functions and write directly into the output. This removes the extra work of the `append` and makes it efficient.

Of course, we can *hide* this impurity behind a safe (pure) function which actually allocates the output sequence, calls the impure function to populate it, and then returns it to the caller. Such is the design of many functional libraries—internally they may use side effects and impure functions but expose a pure interface to the outside world.

This implementation of `merge` takes three sequences as input: `A` and `B`, the sequences to merge, and `Out`, a sequence of length `|A| + |B|` in which the output will be written.

Algorithm: Efficient (Impure) Merge

```
fun merge(A: sequence<T>, B : sequence<T>, Out : mutable sequence<T>):
  if |B| > |A|: swap(A, B) // WLOG assume A is larger than B

  if |B| == 0:
    Out[0...|A|] ← A
    return
  if |A| == |B| == 1:
    Out[0...1] ← [min(A[0], B[0]), max(A[0], B[0])]
    return

  LA, RA = split_mid(A)
  k = binary_search(B, RA[0])  // smallest index s.t. B[k] >= RA[0]
  LB, RB = subseq(B,0,k), subseq(B,k,|B|-k)
  Lout, Rout = subseq(Out,0,|LA|+|LB|), subseq(Out,|LA|+|LB|,|RA|+|RB|)
  _, _ = parallel (merge(LA, LB, Lout), merge(RA, RB, Rout))
```

A pure function that calls this one and hides the impurity could then be implemented as:

```
fun merge(A: sequence<T>, B : sequence<T>) -> sequence<T>:
  Out = parallel [None for _ in 1...|A|+|B|]
  merge(A, B, Out)  // impure, mutates Out
  return Out
```

Remark: Efficient Pure Merge

It is still possible to implement a purely functional parallel merge with no side-effects in $O(n)$ work, but it requires representing the inputs as balanced binary search trees instead of sequences, since BSTs can be efficiently appended in $O(\text{height})$.

In practice this would be substantially less efficient due to many extra small memory allocations and cache inefficiency.

3.4 Analysis of Efficient Merge

Claim: Cost of Merge

The efficient merge implementation is $O(n)$ work and $O(\log^2 n)$ span, where $n = |A|+|B|$.

The proof solves the work recurrence using the "substitution", i.e., "guess and check" method. The guess is rather unintuitive, but the proof goes through once you have it.

Proof. The key factor that makes the analysis work is to observe that since WLOG $|A| \geq |B|$ and we split A in half, the size of the subproblems is between $1/4n$ and $3/4n$.

More specifically, Let $W(n)$ denote the work of merging two sequences of total length n, then the work recurrence looks like

$$W(n) = W(\alpha n) + W((1-\alpha)n) + c\log n,$$

where α is between $1/4$ and $3/4$ and technically is not constant but may vary from level to level of the recursion (but it always lies in this range).

We can verify that $W(n) = O(n)$ using the substitution method. The tricky part is canceling out the $c\log n$ term which comes from the cost of the binary search. The trick is to assume that $W(n)$ is of the form

$$W(n) \le c_1 n - c_2 \log n$$

for some constants c_1, c_2 that are large enough to satisfy the base cases and then proceed.

$$\begin{aligned} W(n) &= W(\alpha n) + W((1-\alpha)n) + c\log n, \\ &\le c_1\alpha n - c_2\log(\alpha n) + c_1(1-\alpha)n - c_2\log((1-\alpha)n) + c\log n, \\ &= c_1 n - c_2(\log\alpha + \log n + \log(1-\alpha) + \log n) + c\log n, \\ &= c_1 n - c_2\log n - (c_2(\log n + \log(\alpha(1-\alpha))) - c\log n). \end{aligned}$$

Now pick the constant c_2 such that $c_2 > c$, which makes the rightmost term positive, so

$$\begin{aligned} W(n) &\le c_1 n - c_2\log n - (\text{something positive}), \\ &\le c_1 n - c_2 \log n, \end{aligned}$$

which proves that $W(n) = O(n)$. The span recurrence is much simpler. Based on the previous observation, we know that

$$S(n) \le S(0.75n) + \Theta(\log n).$$

This recurrence has $\log_{4/3} n$ levels which cost $O(\log n)$, so the total span is at most $O(\log^2 n)$. $\square$

Remark: More Efficient Merge

You can improve the span of `merge` to just $O(\log n)$ with a slightly more complicated algorithm. The trick is to divide the input not into two parts, but into a much larger number of parts and recurse on them all in parallel! This gets you to a base case much much quicker, and you can afford to do it because each level of recursion only has to pay for its binary searches which are relatively cheap.

4 Sorting by Folding and Reducing with Merge

Lastly, now that we have the `merge` function, what can we do with it? It turns out, more than you'd think. Although you might not realize it at first, the merge function is associative.

Claim: Merge is Associative

The `merge` operation given two sorted sequence is an associative operation.

Proof. Since `merge` must produce a sorted sequence given any sorted inputs, the result of merging three sequences x, y, z must always be sorted regardless of the order that the merges are applied. If any order produced a different output, it would not be sorted, and hence the output must be the same for any order. □

Since `merge` is associative, given our newfound knowledge from earlier, our first instinct should be to put it in **reduce** and see what happens, because that would be interesting!

4.1 Parallel MergeSort as reduce merge

Since `merge` requires sequences as its input, we can treat the base case as being mapping a single element to a *singleton* sequence, i.e., a sequence containing that one element. A sequence of one element is considered sorted, so this is a valid input for `merge`.

Lastly, the identity value for a merge is the empty sequence, so we could therefore write the following mystery code for any sequence:

```
fn(s : sequence<T>): return reduce(merge, [], map(singleton, s))
```

We argued that this code must be valid since it meets all the requirements of reduce, but what does it actually do? It takes all the elements of a sequence, puts them into their own singleton sequences, and merges them into a single sorted sequence.

Hang on, its just merging the elements of the input together until they are one sorted sequence? That's just **Merge Sort**! Through the magic of reduce, we have managed to write Merge Sort in just one line of code. That's pretty cool.

Theorem: Cost of Parallel MergeSort

Parallel MergeSort, which can be implemented by the following line of code:

```
fn(s : sequence<T>): return reduce(merge, [], map(singleton, s))
```

runs in $O(|S|\log|S|)$ work and $O(\log^2|S|)$ span, assuming we use the $O(\log n)$ span `merge`.

Proof. MergeSort recurses on sequences of size $n/2$ and performs $O(n)$ work at the root for the merge operation. The work recurrence is therefore

$$W(n) = 2W\left(\frac{n}{2}\right) + O(n).$$

This is a balanced recurrence which performs $O(n)$ work per level and does $O(\log n)$ levels of recursion, hence the work is $O(n \log n)$.

Correspondingly the span recurrence (assuming `merge` costs $O(\log n)$ span) is

$$S(n) = S\left(\frac{n}{2}\right) + O(\log n).$$

Unrolling the recurrence gives $S(n) = O(\log^2 n)$. □

4.2 InsertionSort as fold merge

Finally, what if we do a **left fold** instead of a reduce? We know that it must produce the same answer since a fold and a reduce always compute the same result if the function is associative, it just might compute it in a different way.

```
fn(s : sequence<T>): return fold_left(merge, [], map(singleton, s))
```

This function takes a sequence and one-by-one, merges one additional new element into a sorted prefix. Hang on, that's just called **Insertion Sort**!

So, to wrap up today in one sentence: with the magic of higher-order functions, we can write both Insertion Sort and MergeSort in a single line of code (by slightly cheating by also writing merge which is many more than one line of code, but still.) Pretty cool!