

Divide-and-Conquer and Reduce

1 Divide-and-Conquer

You have already seen several algorithms that use divide-and-conquer in previous classes, such as Mergesort and Quicksort, but it is useful to step back and make the general pattern explicit. What exactly makes an algorithm “divide and conquer”?

Definition: Divide-and-Conquer

A **divide-and-conquer** algorithm solves a problem by **dividing** it into smaller instances, solving those **recursively**, then **combining** those solutions to solve the original instance.

Divide-and-conquers algorithm are recursive algorithms therefore have two components, the latter of which contains the three real ingredients of divide and conquer:

- **Base case:** For sufficiently small problem sizes, solve it directly without further recursion.
- **Inductive step:**
 1. **Divide** the instance into some number of smaller instances.
 2. **Recur** to solve the smaller instances.
 3. **Combine** the recursive solutions to solve the original instance.

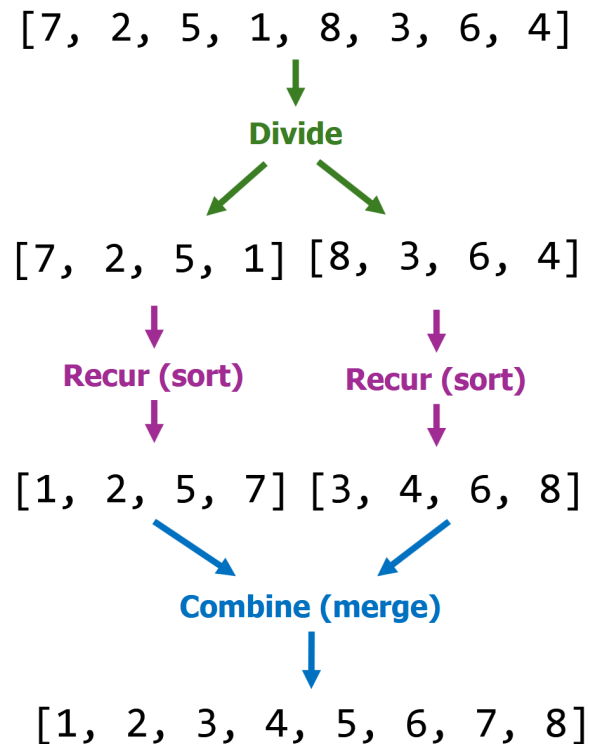
Importantly, divide-and-conquer does **not** require dividing the input into exactly *two* subproblems. The problem can be split into three subproblems, $\log n$ subproblems, $\sqrt{n}$ subproblems, etc. As long as its more than one subproblem. Splitting into two halves is arguably the most common, and will be the main cases we see today, but it is not part of the definition.

1.1 A Familiar Example: MergeSort

A familiar example from 15-122 is **MergeSort**. Given a sequence, it divides the sequence into two halves, recursively sorts each half, then combines the two sorted answers using merge.

Algorithm: MergeSort

```
fun mergesort(S : sequence<T>) -> sequence<T>:
  if |S| <= 1: return S
  L, R = split_mid(S)
  Lsort = mergesort(L)
  Rsort = mergesort(R)
  return merge(Lsort, Rsort)
```



Unlike the version you probably saw in 15-122, this one is written in a *purely functional* style using immutable sequences. This mergesort returns a *new* sequence consisting of the elements of *S* in sorted order, rather than sorting *S* in place.

However, the important point for today is not the details of MergeSort itself or exactly what style it's implemented in, but the **pattern**:

- `split_mid` is the **divide** step,
- the two calls to `mergesort` are the recursive step,
- and `merge` is the **combine** step.

Last but not least, we can add one major improvement that you didn't see in 15-122. Since the two recursive calls have no side effects, it is safe to run the two of them at once in parallel!

Algorithm: MergeSort

```

fun mergesort(S : sequence<T>) -> sequence<T>:
  if |S| <= 1: return S
  L, R = split_mid(S)
  Lsort, Rsort = parallel (mergesort(L), mergesort(R))
  return merge(Lsort, Rsort)
  
```

This is one of the (many) reasons that divide-and-conquer algorithms are important and prevalent in parallel algorithms.

1.2 Why Divide-and-Conquer is Useful

There are many advantages of using divide and conquer:

- The correctness proof is usually a **proof by induction**. We prove the correctness of the base case directly, then as the inductive hypothesis we assume that the solutions to the subproblems are correct and use those assumptions to prove that the combine step produces the correct answer for the original instance.
- Second, the cost of divide-and-conquer algorithms can typically be described **using recurrence relations**, which we have a good toolbox for solving.
- Finally, **divide and conquer is naturally parallel**. If the recursive subproblems are independent, we can solve them simultaneously. In particular, if our algorithm is written in pure functional style (with no side effects), the recursive calls are necessarily independent and therefore safe to run in parallel.

1.3 Cost of Parallel Divide and Conquer

Suppose the divide step on a problem of size n has cost $W_{\text{divide}}(n)$, the recursive subproblems have sizes $n_1 + \dots + n_k = n$, and the combine step has cost $W_{\text{combine}}(n)$. Then the recurrence relation for the work of the algorithm has the form

$$W(n) = W_{\text{divide}}(n) + \sum_{i=1}^k W(n_i) + W_{\text{combine}}(n).$$

Since the recursive calls run in parallel, and the span of a set of parallel recursive calls is defined to be the maximum of the spans among them, the recurrence relation for the span is

$$S(n) = S_{\text{divide}}(n) + \max_{i=1}^k S(n_i) + S_{\text{combine}}(n).$$

The brick method will be useful for solving these kinds of recurrence relations that arise when analyzing divide-and-conquer algorithms.

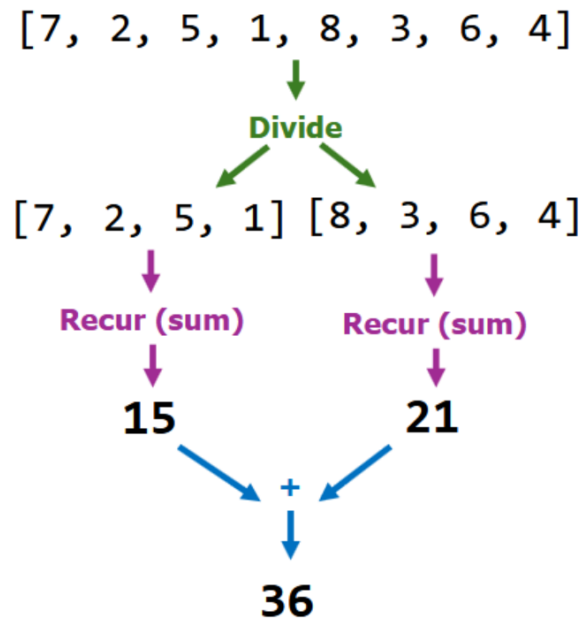
2 Reductions

Lets start with one of the simplest problems you might see when you first learn computer science. One that you probably solved with a for loop the first time you saw it.

Problem: Sum

Given a sequence of integers, compute their total sum.

This problem is naturally suited to divide and conquer; if I know the sum of half of the numbers in the list and the sum of the other half, I can get the total sum by adding those together.



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)
    Lsum, Rsum = parallel (sum(L), sum(R))
    return Lsum + Rsum
  
```

This is a particularly simple divide-and-conquer algorithm. The divide step, `split_mid`, costs $O(1)$ work and span, and the combine step, addition, also costs $O(1)$ work and span. There are two recursive calls on subproblems of size $n/2$.

The work recurrence is therefore

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

This is leaf dominated since the work at every call is 1, and hence the work at the children of a given call is 2, which is a constant factor larger. There are n leaves and each leaf does $O(1)$ work, so by the brick method the total work is $O(n)$.

Since the two recursive calls run in parallel, the span recurrence is

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

This is balanced since at each call the local cost is 1 and each call has one child, hence the cost of the children adds to 1. There are $\log_2 n + 1$ levels, each of which has $O(1)$ span, so the total span is $O(\log n)$.

2.1 Generalizing Parallel Sum

As it stands, this algorithm only computes sums, but the same divide-and-conquer structure can compute many other aggregates. Consider finding the maximum value in a sequence:

Algorithm: Parallel Sequence Maximum

```
fun maxv(S : sequence<int>) -> int:
  match length(S) with:
  case 0: return -infinity
  case 1: return S[0]
  case _:
    L, R = split_mid(S)
    Lmax, Rmax = parallel (maxv(L), maxv(R))
    return max(Lmax, Rmax)
```

This is almost exactly the same algorithm! The only differences are the value returned for an empty sequence and the operation used to combine the two recursive answers.

The same pattern could be used for many different aggregates over a sequence:

- sum and product,
- minimum and maximum,
- Boolean **or** and Boolean **and**.

In each case, the only thing that changes is the operation used to combine the answers and the value used for the empty-sequence base case. This general primitive is called **reduce**.

2.2 Reduce

Given a type T , a combining function $f : (T, T) \rightarrow T$, and a value I to use for an empty sequence, reduce computes an aggregate of a sequence of values of type T .

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(f, I, L), reduce(f, I, R))
    return f(Lres, Rres)
```

For example, parallel sum is `reduce(+, 0, S)`, while parallel max is `reduce(max, -∞, S)`. Parallel min would be `reduce(min, ∞, S)` because the minimum value in an empty sequence is considered to be ∞ , for important reasons we will see shortly.

2.3 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 and most common case is when f can be evaluated in constant time.

Theorem: Cost of Reduce

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

Proof. The work and span recurrences are exactly the same as for parallel sum:

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

and

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

The work recurrence is leaf dominated: it has n leaves which each do $O(1)$ work, giving $O(n)$ total work. The span recurrence is balanced and has $O(\log n)$ levels with $O(1)$ span per level, giving $O(\log n)$ total span. $\square$

2.4 Preconditions for Reduce

There is one important catch: we cannot use just any combining function.

A parallel reduce will not apply the function on the elements of the sequence from left to right; it is free to group applications of f according to its divide-and-conquer recursion. For example, addition behaves sensibly because

$$(x + y) + z = x + (y + z).$$

Subtraction does not:

$$(x - y) - z \neq x - (y - z)$$

in general. Therefore, if changing the grouping changes the answer, the result of reduce would depend on how the implementation happens to divide the input.

The property we need is called **associativity**.

Definition: Associative

A binary operator $f : (T, T) \rightarrow T$ is associative if for all values $x, y, z \in T$,

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

Furthermore, the empty-sequence value I must be an **identity** for f .

Definition: Identity

A value $I \in T$ is an identity for a binary operator $f : (T, T) \rightarrow T$ if for every value x ,

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

For now, simple operators like addition, multiplication, minimum, maximum, Boolean **or**, and Boolean **and** are enough to give us useful applications of reduce. We will return to associativity next lecture, where it will let us build considerably more interesting algorithms.

3 Merge

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

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.

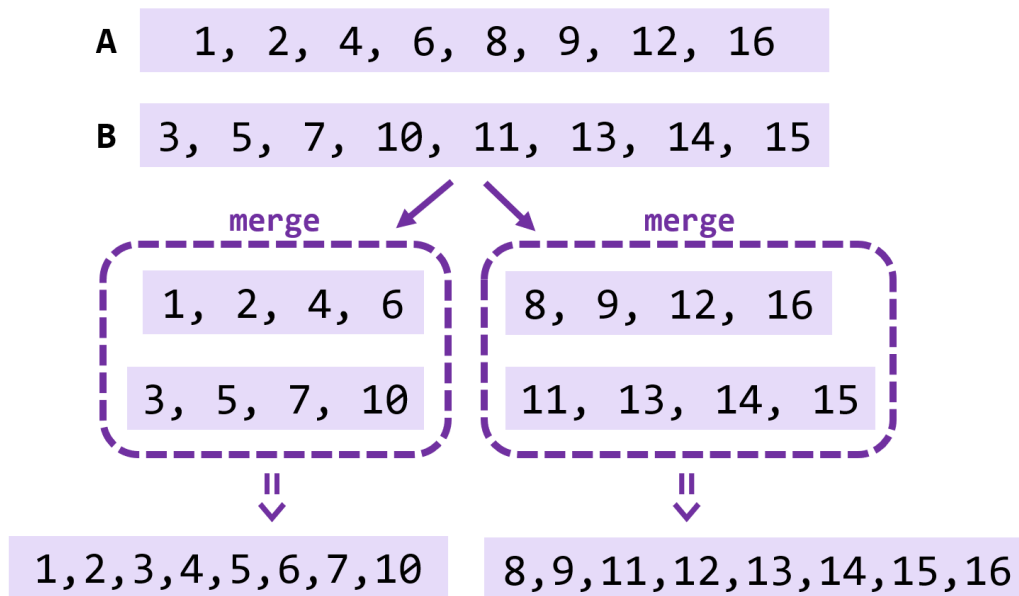
As the theme of today's lecture suggests, we are going to use divide-and-conquer!

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.

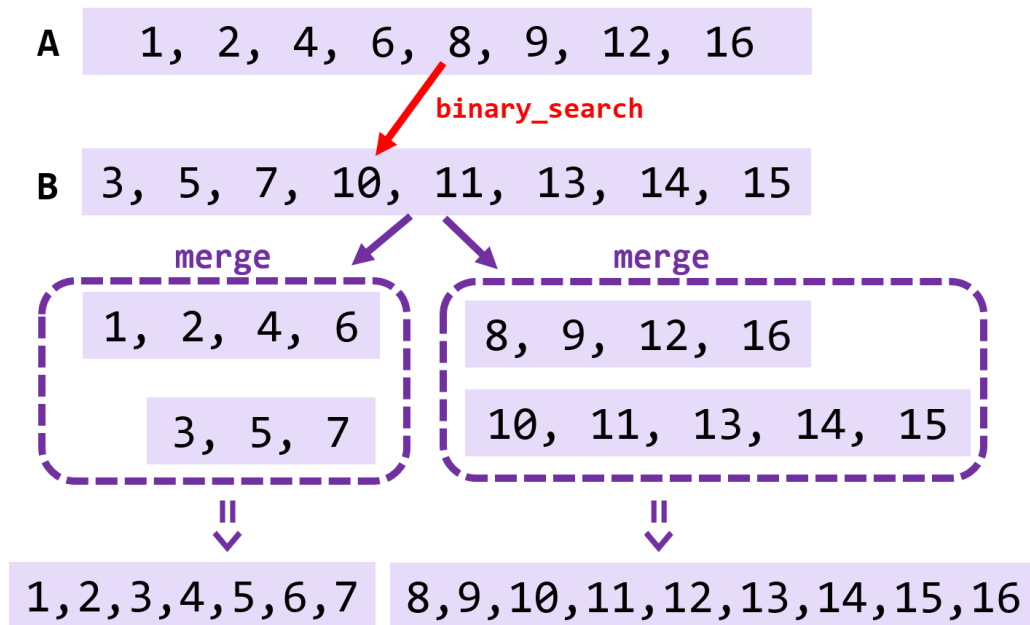
The issue is that we chose to split both sequences down the middle, but this resulted in two sequences that were still interleaved. Instead, we want to split the sequences so that the left one is *strictly smaller* than the right one—so that when we are done the combine step can simply be to append the 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 B 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, it is not guaranteed 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 possibly imbalanced way, we can just *always split the larger sequence in half*.

This gives us an important balance guarantee. Let

$$n = |A| + |B|$$

be the total problem size, and WLOG suppose $|A| \geq |B|$ so that A is the sequence we split in half. Let $n_L = |L_A| + |L_B|$ and $n_R = |R_A| + |R_B|$ be the sizes of the two recursive subproblems.

Lemma: Even Split

Both recursive subproblem sizes decrease geometrically:

$$\frac{n}{4} \leq n_L, n_R \leq \frac{3n}{4}.$$

Proof. Since A is the larger sequence,

$$|A| \geq \frac{n}{2}.$$

Both recursive subproblems get half of A . The smallest a subproblem can be is therefore when it gets none of B :

$$\frac{|A|}{2} \geq \frac{1}{2} \cdot \frac{n}{2} = \frac{n}{4}.$$

For the upper bound, the largest a subproblem can be is when it gets all of B . Since $|A| = n - |B|$,

$$\frac{|A|}{2} + |B| = \frac{n - |B|}{2} + |B| = \frac{n}{2} + \frac{|B|}{2} \leq \frac{3n}{4},$$

where the last inequality uses $|B| \leq n/2$. □

Thus neither recursive call can retain almost the entire input. Each recursive problem shrinks by a constant factor, which will be important in both our work and span analyses.

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>:
    // Split the larger one in half; WLOG assume A is larger than B
    if |B| > |A|: swap(A, B)

    // Base cases: one is empty or both have only one element
    if |B| == 0: return A
    if |A| == |B| == 1: return [min(A[0], B[0]), max(A[0], B[0])]

    // Recursive case: split A in half and find a split point in B so
    // that every element on the left <= every element on the right
    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, MR = parallel (merge(LA, LB), merge(RA, RB))
    return append(ML, MR)
```

3.2.1 Cost Analysis

Claim: Cost of Inefficient Merge

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

Proof. The expensive combine step is `append(ML, MR)`, which costs $\Theta(n)$ work at a problem of total size n . Therefore the work recurrence is

$$W(n) = W(n_L) + W(n_R) + \Theta(n).$$

Using the Brick Method, compare the cost of the parent brick with the total cost of its two children. The parent costs n , and the children cost $n_L + n_R = n$ also. Therefore the recurrence is **balanced**. By the Even Split Lemma, the recursive problem sizes decrease geometrically, so the recursion has $\Theta(\log n)$ levels. Each level performs $\Theta(n)$ work, giving

$$W(n) = \Theta(n \log n).$$

□

3.3 Making Merge Work-Efficient

The problem with the previous implementation is the `append` in the combine step. It feels natural in a purely functional implementation: first construct the two recursive answers, then append them together. However, the `append` is actually **entirely redundant work**.

Before making the recursive calls, we already know exactly how large each recursive result will be: the left recursive merge contains $n_L = |LA| + |LB|$ elements, while the right recursive merge contains $n_R = |RA| + |RB|$ elements.

We also know exactly where those results belong in the final output: the left result occupies the positions 0 to n_L and the right result occupies the remaining positions n_L to n .

Idea: Write Directly to the Output

Allocate the output just once, and have each recursive call write each element of its answer directly into the correct final location. Then there is no work left for the combine step: when the recursive calls complete, the answer is already there!

This requires us to leave the blissful safety of pure functions and allow the recursive merge helper to mutate a pre-allocated output array. Of course, we can still hide this impurity behind a safe pure interface which allocates the output, calls the imperative helper to populate it, and returns the completed result. 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.

Instead of returning anything, the imperative helper takes three sequences as input: A and B, the sorted sequences to merge, and Out, a mutable array sequence of length $|A| + |B|$ in which the answer will be written.

Algorithm: Efficient (Imperative) Merge

```
fun merge(A: sequence<T>, B : sequence<T>, Out : mutable array_sequence<T>):
  if |B| > |A|: swap(A, B)

  if |B| == 0:
    Out[0...|A|] <- A
    return
  if |A| == |B| == 1:
    Out[0...2] <- [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))
```

The two recursive calls write to disjoint portions of Out, so there are no conflicts between them and they can safely execute in parallel. Notice that there is also no combine operation after the recursive calls since they are “combined” automatically when written into the output array!

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
```

3.4 Analysis of Work-Efficient Merge

Claim: Cost of Merge

The imperative merge implementation costs $\Theta(n)$ work and $\Theta(\log^2 n)$ span.

The analysis has one subtlety that is easy to miss. At an *internal* node, the combine step now does nothing, and the dominant non-recursive cost is the binary search, which costs $\Theta(\log n)$. However, the base case, `Out[0...|A|] <- A`, may copy more than one element, so leaves do not necessarily all have the same constant cost.

Proof. First consider the work at internal nodes. The recurrence is

$$W(n) = W(n_L) + W(n_R) + \Theta(\log n).$$

We can classify the internal-node recurrence using the Brick Method. The parent brick has cost $\log n$. By the Even Split Lemma, both children have size at least $n/4$, so the total cost of their bricks is at least

$$\log n_L + \log n_R \geq \log\left(\frac{n}{4}\right) + \log\left(\frac{n}{4}\right) = 2 \log n - 4.$$

Thus, for sufficiently large n , the total cost of the children is a constant factor larger than the cost of the parent. The internal-node work is therefore **leaf dominated**.

Ordinarily, for a leaf-dominated recurrence we would count the number of leaves and multiply by the work per leaf. That does not work here because both of those quantities can vary, but we do not actually need either one individually. Every base case writes into a disjoint portion of Out, and together the base cases write exactly the n elements of the output. Therefore the *total* work across all leaves is $\Theta(n)$.

For the span, the two recursive calls execute in parallel, so only the larger subproblem appears on the critical path. By the Even Split Lemma, that subproblem has size at most $3n/4$:

$$S(n) \leq S\left(\frac{3n}{4}\right) + \Theta(\log n).$$

This recurrence is **balanced** under the Brick Method. The parent brick costs $\log n$, while its single child brick costs

$$\log\left(\frac{3n}{4}\right) = \log n - \log(4/3).$$

The cost decreases only by an additive constant, not geometrically. The recursion has $\Theta(\log n)$ levels because the problem size decreases by a constant factor, and the maximum span at any level is $\Theta(\log n)$. Therefore

$$S(n) = \Theta(\log n) \cdot \Theta(\log n) = \Theta(\log^2 n).$$

The base-case copies do not change this bound: copying a sequence can be performed in linear work and constant span. $\square$

Remark: Efficient Pure Merge

If you prefer the comfort of functional programming, it *is* still possible to implement a purely functional parallel merge with no imperative code in $O(n)$ work, but it requires representing the inputs as balanced binary search trees instead of array sequences, since balanced BSTs can be efficiently appended in $O(\log n)$ time.

Remark: An Even Better Merge

We can improve the span of merge to $\Theta(\log n)$ while retaining $\Theta(n)$ work, but the details are somewhat more complicated.

The key idea is that divide-and-conquer does not have to create only two recursive subproblems. Instead, the input is divide into roughly $\sqrt{n}$ subproblems which can all recurse in parallel. This makes the recursive problem size decrease much faster, lowering the span without increasing the asymptotic work.

The idea is simple; making all of the split points work out correctly is the tricky part, so we omit the full algorithm here.