Chapter 6

Sequences

So far in class we have defined several "problems" and discussed algorithms for solving them. The idea is that the problem is an abstract definition of what we want in terms of a function specification, and the algorithms are particular ways to solve/implement the problem. There can be many algorithms for solving the same problems, and these algorithms can have different costs.

In addition to abstract functions we also often need to define abstractions over data. In such an abstraction we define a set of functions (abstractly) over a common data type. We will refer to the abstractions as abstract data types (ADTs) and their implementations as data structures. The ADT just defines the functionality of the data type but says nothing about the cost. The data structure defines a specific way to implement the ADT, which gives a specific cost. Often it is useful, however, to have a cost specification for an ADT without having to know how it is implemented. In fact there might be many different implementations that match the same cost specification. There can also be different implementations that have different cost specifications. These specifications might pose tradeoffs where some functions are more expensive while others are cheaper.

In this chapter we will see that an array-based implementations of sequences allow for constant work to access the n^{th} element of a sequence but linear work (in the sizes of the inputs) to append two sequences. On the other hand a tree-based implementation takes logarithmic work for both accessing the nth element and appending. This leads to two different cost specifications, where neither is strictly better than the other. We can describe these cost specifications without having to know how the implementation works.

As such, data types can be described at three levels of abstraction:

1. **The abstract data type**: The definition of the interface for the purpose of describing functionality and correctness criteria.
2. **The cost specification**: The definition of the costs for each of the functions in the interface. There can be multiple different cost specifications for an ADT depending on the type of implementation.

3. **The implementation as a data structure**: This is the particular data structure used to implement the ADT. There might be multiple data structures that match the same cost specification. If you are a user of the ADT you don't need to know what this is, although it is good to be curious.

6.1 The Sequence ADT

The first ADT we consider in some detail is one for sequences. A sequence is a mathematical notion representing an ordered list of elements, such as $\langle a, b, c, d \rangle$. In mathematics a sequence can be either of finite or infinite length. In this book, however, we will only consider finite-length sequences. It is important you do not associate a sequence with a particular implementation—e.g. with an array of contiguous locations in the memory of the machine. Instead you should be thinking of it abstractly in terms of the mathematical definition and what functions on the data type it supports. Indeed we will consider two different implementations of sequences, one based on arrays and the other on trees.

We first do a quick review of set theory to more formally define a sequence. A *relation* is a set of ordered pairs. In particular for two sets α and β, ρ is a relation from α to β if $\rho \subseteq \alpha \times \beta$. Here $\alpha \times \beta$ indicates the set of all ordered pairs made from taking the first element from α and the second from β, also called the Cartesian product of α and β. A *mapping* (or *function*) is a relation ρ such that for every a in the domain of ρ there is only one b such that $(a, b) \in \rho$. A *sequence* is a mapping whose domain is $\{0, \ldots, n-1\}$ for some $n \in \mathbb{N}$ (we use $\mathbb{N}$ to indicate the natural numbers, including zero).[1] For example the sequence $\langle a, b, c, d \rangle$ corresponds to the mapping $\{(0, a), (1, b), (2, c), (3, d)\}$, or equivalently $\{(1, b), (3, d), (2, c), (0, a)\}$ since the ordering of a set does not matter.

Based on this representaiton in terms of sets, the sequence ADT is defined in Abstract Data Type 6.1. For each function in the interface the definition specifies the type of the function (the middle column) along with a definition of what the function computes (the right column). In the pseudocode in this book we use a notation for sequences using angle brackets $\langle\ \rangle$ instead of curly brackets $\{\}$ used for sets. The translation of this notation to the functions in the sequence definition is given in figure 6.1. The sequence notation basically hides the index from the set notation, and is less clunky than using the function names.

We now briefly describe the functions in words instead of as sets. The `empty`, `length` and `singleton` functions should be self explanatory. The `nth` function extracts the i^{th} element from a sequence. If the i is out of range it returns a special value $\perp$, which is called *bottom*, and means the result is undefined. The `map` function is a higher-order function that applies a function f to each element of a sequence and returns a new equal length sequence. As we will see in the cost specification, shortly, the map can be done in parallel across all elements of the sequence. Also, as we will see in a later chapter, `map` generalizes to arbitrary mappings. The `tabulate` function is similar to `map` but maps over the sequence of indices $\langle 0, \ldots, n-1 \rangle$.

[1] Traditionally sequences are indexed from 1 not 0, but being computer scientists, we violate the tradition here.

Abstract Data Type 6.1 (Sequences). *A* Sequence *is a type* $\mathbb{S}_\alpha$ *representing a mapping from* $\mathbb{N}$ *(the natural numbers) to* α *with domain* $\{0, \ldots, n-1\}$ *for some* $n \in \mathbb{N}$*, and supporting the following values and functions:*

$$
\begin{array}{lcll}
\texttt{empty} & : & \mathbb{S}_\alpha & = \{\} \\
\texttt{length}(A) & : & \mathbb{S}_\alpha \to \mathbb{N} & = |A| \\
\texttt{singleton}(v) & : & \alpha \to \mathbb{S}_\alpha & = \{(0, v)\} \\
\texttt{nth}(A, i) & : & \mathbb{S}_\alpha \times \mathbb{N} \to (\alpha \cup \{\bot\}) & = \begin{cases} v & (i, v) \in A \\ \bot & \textit{otherwise} \end{cases} \\
\texttt{map}(f, A) & : & (\alpha \to \beta) \times \mathbb{S}_\alpha \to \mathbb{S}_\beta & = \{(i, f(v)) : (i, v) \in A\} \\
\texttt{tabulate}(f, n) & : & (\mathbb{N} \to \alpha) \times \mathbb{N} \to \mathbb{S}_\alpha & = \{(i, f(i)) : 0 \le i < n\} \\
\texttt{subseq}(A, s, l) & : & \mathbb{S}_\alpha \times \mathbb{N} \times \mathbb{N} \to \mathbb{S}_\alpha & = \{(i - s, v) \\
 & & & \quad : (i, v) \in A \mid s \le i < s + l\} \\
\texttt{append}(A, B) & : & \mathbb{S}_\alpha \times \mathbb{S}_\alpha \to \mathbb{S}_\alpha & = A \cup \{(i + |A|, v) : (i, v) \in B\} \\
\texttt{filter}(f, A) & : & (\alpha \to \mathbb{B}) \times \mathbb{S}_\alpha \to \mathbb{S}_\alpha & = \{(|\{(j, x) \in A \mid j < i \wedge f(x)\}|, v) \\
 & & & \quad : (i, v) \in A \mid f(v)\} \\
\texttt{flatten}(A) & : & \mathbb{S}_{\mathbb{S}_\alpha} \to \mathbb{S}_\alpha & = \{(i + \sum_{(k,X) \in A, k<j} |X|, v) \\
 & & & \quad : (i, v) \in Y, (j, Y) \in A\} \\
\texttt{inject}(p, A) & : & \mathbb{S}_{\mathbb{N} \times \alpha} \times \mathbb{S}_\alpha \to \mathbb{S}_\alpha & = \{(i, x) : (i, v) \in A\}, \\
 & & & \quad x = \begin{cases} y & (j, i, y) \in p \\ v & \textit{otherwise} \end{cases}
\end{array}
$$

where $\mathbb{B} = \{true, false\}$*. The additional functions* `iter`*,* `iterh`*,* `reduce`*, and* `scan` *are defined later.*

The $\texttt{subseq}(A, s, i)$ function extracts a contiguous subsequence starting at s and with length l. If the subsequence is out of bounds of A, only the part within A is returned. The `append` function should be self explanatory.

The `filter` function takes as an argument a predicate (i.e. a function that returns true or false). It applies this predicate to each element of the input sequence and returns a new sequence only containing elements which returned true. The order is maintained.

S_i	`nth` S i
$\|S\|$	`length`(S)
$\langle\,\rangle$	`empty`$()$
$\langle\, v \,\rangle$	`singleton`(v)
$\langle\, i, \ldots, j \,\rangle$	`tabulate` (**fn** $k \Rightarrow i + k$) $(j - i + 1)$
$S\,\langle\, i, \ldots, j \,\rangle$	`subseq` S $(i, j - i + 1)$
$\langle\, e : p \in S \,\rangle$	`map` (**fn** $p \Rightarrow e$) S
$\langle\, e : 0 \leq i < n \,\rangle$	`tabulate` (**fn** $i \Rightarrow e$) n
$\langle\, p \in S \mid e \,\rangle$	`filter` (**fn** $p \Rightarrow e$) S
$\langle\, e_1 : p \in S \mid e_2 \,\rangle$	`map` (**fn** $p \Rightarrow e_1$) (`filter` (**fn** $p \Rightarrow e_2$) S)
$\langle\, e : p_1 \in S_1, p_2 \in S_2 \,\rangle$	`flatten`(`map` (**fn** $p_1 \Rightarrow$ `map` (**fn** $p_2 \Rightarrow e$) S_2) S_1)
$\langle\, e_1 : p_1 \in S_1, p_2 \in S_2 \mid e_2 \,\rangle$	`flatten`(`map` (**fn** $p_1 \Rightarrow \langle\, e_1 : p_2 \in S_2 \mid e_2 \,\rangle$) S_1)
$\sum_{p \in S} e$	`reduce add` 0 (`map` (**fn** $p \Rightarrow e$) S)
$\sum_{i=k}^{n} e$	`reduce add` 0 (`map` (**fn** $i \Rightarrow e$) $\langle\, k, \ldots, n \,\rangle$)

Figure 6.1: Translation from sequence notation to the sequence functions. The $\sum$ can be replaced with $\min$, $\max$, $\cup$ and $\cap$ with the presumed meanings, and by replacing `add` and 0 with the appropriate values.

Example 6.2. *The expression:*

$$\langle\, x \in \langle\, 9, -1, 4, 11, 13, 2 \,\rangle \mid x > 5 \,\rangle$$

is equivalent to:

$$\texttt{filter}\ (\textbf{fn}\ x \Rightarrow x > 5)\ \langle\, 9, -1, 4, 11, 13, 2 \,\rangle$$

and when evaluated returns the sequence:

$$\langle\, 9, 11, 13 \,\rangle\ .$$

The `flatten` function takes a sequence of sequences and flattens them—i.e. if the input is a sequence $\langle\, S_1, S_2, \ldots, S_n \,\rangle$ it appends all the S_i together one after the other. It is often used when we map over multiple sequences or sets of indices.

Example 6.3. *Let's say we want to generate all contiguous subsequences of a sequence A. Each sequence can start at any position $0 \leq i < |A|$, and end at any position $i \leq j < |A|$. We can do this with the following pseudocode:*

$$\langle\, A\,\langle\, i, \ldots, j\,\rangle : 0 \leq i < |A|, i \leq j < |A|\,\rangle\ ,$$

which is equivalent to:

```
flatten(tabulate
        (fn i ⇒ tabulate
                (fn l ⇒ subseq(A, i, l + 1))
                (length(A) − i))
        (length A))
```

From a theoretical point of view, the exact list of functions in the interface does not matter that much. This is because many of the functions can be implemented with others.

Example 6.4. *We can implement* `map` *using* `tabulate` *as follows:*

$$\textbf{function}\ \texttt{map}\ f\ S = \texttt{tabulate}\ (\textbf{fn}\ i \Rightarrow f(\texttt{nth}\ S\ i))\ (\texttt{length}\ S)$$

or equivalently in our sequence notation as

$$\textbf{function}\ \texttt{map}\ f\ S = \langle\, f(S_i) : 0 \leq i < |S|\,\rangle$$

However, we have to be careful that the implementation preserves costs in the cost specification.

We are now ready to consider the cost specifications for the sequence ADT. We consider two cost specifications, one which we refer to as *ArraySequence*, and the other as *TreeSequence* . These are given in figure 6.2 for a few of the functions. A full list of costs can be found in the Cost Specifications part of http://www.cs.cmu.edu/~15210/docs/cost/. The names of the cost specifications roughly indicate the class of implementation that can achieve these cost bounds, but there might be many specific implementations that match the bounds. For examples for the TreeSequence there are many types of trees that might be used. To use the cost bounds, you don't need to know the specifics of how these implementations work.

Cost Specification 6.5 (Sequences).

	ArraySequence		TreeSequence	
	Work	Span	Work	Span
`length`(T)	1	1	1	1
`nth`(T)	1	1	$\log n$	$\log n$
`tabulate` f n	$\sum_{i=0}^{n} W(f(i))$	$\max_{i=0}^{n} S(f(i))$	$\sum_{i=0}^{n} W(f(i))$	$\log n + \max_{i=0}^{n} S(f(i))$
`map` f S	$\sum_{s\in S} W(f(s))$	$\max_{s\in S} S(f(s))$	$\sum_{s\in S} W(f(s))$	$\log \lvert S\rvert + \max_{s\in S} S(f(s))$
`filter` p S	$\sum_{s\in S} W(p(s))$	$\log \lvert S\rvert + \max_{s\in S} S(p(s))$	$\sum_{s\in S} W(p(s))$	$\log \lvert S\rvert + \max_{s\in S} S(p(s))$
`subseq`(S, s, l)	1	1	$\log(\lvert S\rvert)$	$\log(\lvert S\rvert)$
`append`(S_1, S_2)	$\lvert S_1\rvert + \lvert S_2\rvert$	1	$\log(\lvert S_1\rvert + \lvert S_2\rvert)$	$\log(\lvert S_1\rvert + \lvert S_2\rvert)$
`flatten`(S)	$\lvert\lvert S\rvert\rvert + \lvert S\rvert$	1	$\log(\lvert\lvert S\rvert\rvert) \log(\lvert S\rvert)$	$\log(\lvert\lvert S\rvert\rvert + \lvert S\rvert)$

Figure 6.2: Sample *cost specifications* for the Array and Tree based implementations of Sequences. $||S|| = \sum_{X\in S} |S|$. All are big-$O$.

Example 6.6. *Consider the code from Example 6.3. We have for the ArraySequence cost specification for $n = |A|$:*

$$W(n) = O(n^2) \ .$$

This is because there are $O(n^2)$ total calls to `subseq` *($A\langle i, \ldots, j\rangle$), each of which take $O(1)$ work. Furthermore the* `flatten` *takes $O(n^2)$ work since the total length is $O(n^2)$. We also have*

$$S(n) = O(1)$$

This is because the `subseq` *has $O(1)$ span, and we take a maximum of the spans across each of the* `tabulate`*s. At the end we do a* `flatten`*, which also has $O(1)$ span.*

Exercise 6.7. *What is the work and span for the code from Example 6.3 if we use the TreeSequence cost specification?*

There are a couple things about the cost specifications we should note. Firstly, note that the `tabulate`, `map`, and `filter` are all parallel. In particular the span takes the maximum of the span of the subcalls at each position. The `append` for the ArraySequence cost specification is also parallel since the span is only constant, while the work is linear. This is so since the elements can be copied in parallel. It might not be obvious that filter, for example, can be parallelized. We will get to this later.

Secondly, note that neither specification dominates the other, in the sense that for some functions ArraySequence has better bounds (e.g. `nth`) but for others TreeSequence has better bounds (e.g. `append`). Intuitively, for an array implementation we can access the `nth` element using a direct access into the array costing constant work, but to `append` two arrays we need

to put one after the others, which requires moving them, requiring linear work in the sizes. On the other hand, for a (balanced binary) tree implementation we need to traverse $O(\lg n)$ levels of the tree to get to the `nth` element, but an append does not require moving the whole tree, so it can also be done in $O(\log n)$ work. More details on these implementations will be given later in this book. This sort of tradeoff is common in data types. The user can decide to use an implementation that matches either specification, and this decisions should be based on which specification leads to better asymptotic performance for their algorithm. For example if making many calls to `nth` but no calls to `append`, then the user might want to use the ArraySequence specification, and when running the code use an implementation of Sequences that matches that specification.

Exercise 6.8. *Example 6.4 showed how to implement `map` using `tabulate`. Decide whether this implementation preserves asymptotic costs for an ArraySequence, and then for TreeSequence.*

We will now cover some of the sequence functions in more detail, including `reduce`, `iter`, `scan` and `iterh`, `tokens`, `fields`, and `collect`.

6.2 The Scan Operation

A function closely related to `reduce` is `scan`. It has the interface:

$$\texttt{scan}\ f\ I\ S : (\alpha \times \alpha \to \alpha) \to \alpha \to \alpha\ \texttt{seq} \to (\alpha\ \texttt{seq} \times \alpha)$$

As with `reduce`, when the function f is associative, the scan function returns the sum with respect to f of each prefix of the input sequence S, as well as the total sum of S. Hence the operation is often called the *prefix sums* operation. For a function f which is associative it can be defined as follows:

```
1  function scan f I S =
2    (⟨ reduce f I (S⟨0,...,l−1⟩ : 0 ≤ l < n)⟩,
3     reduce f I S)
```

This uses our pseudocode notation and the $\langle\, \texttt{reduce}\ f\ I\ (\texttt{take}(S,i)) : i \in \langle 0,\ldots,n-1\rangle\rangle$ indicates that for each i in the range from 0 to $n-1$ apply `reduce` to the first i elements of S. For example,

$$\begin{aligned}\texttt{scan} + 0\ \langle 2,1,3\rangle &= (\langle\ \texttt{reduce} + 0\ \langle\,\rangle,\ \texttt{reduce} + 0\ \langle 2\rangle,\ \texttt{reduce} + 0\ \langle 2,1\rangle\ \rangle \\ &\quad\ \texttt{reduce} + 0\ \langle 2,1,3\rangle) \\ &= (\langle 0,2,3\rangle, 6)\end{aligned}$$

Using a bunch of reduces, however, is not an efficient way to calculate the partial sums.

Exercise 6.9. *What is the work and span for the* `scan` *code shown above, assuming* f *takes constant work.*

We will soon see how to implement a scan with the following bounds:

$$\begin{aligned} W(\texttt{scan}\ f\ I\ S) &= O(|S|) \\ S(\texttt{scan}\ f\ I\ S) &= O(\log|S|) \end{aligned}$$

assuming that the function f takes constant work. For now we will consider some useful applications of scans.

Note that the scan operation takes the "sum" of the elements before the position i. Sometimes it is useful to include the value at position i. We therefore also will use a version of such an *inclusive scan*.

$$\texttt{scanI}\ +\ 0\ \langle 2,1,3\rangle\ =\ \langle 2,3,6\rangle$$

This version does not return a second result since the total sum is already included in the last position.

6.2.1 The MCSS Problem Algorithm 5: Using Scan

Let's consider how we might use scan operations to solve the Maximum contiguous subsequence (MCSS) problem. Recall, this problem is given a sequence S to find:

$$\max_{0\le i\le j\le n}(\sum_{k=i}^{j-1} S_k)\ .$$

As a running example for this section consider the sequence

$$S = \langle 1,-2,3,-1,2,-3\rangle\ .$$

What if we do an inclusive scan on our input S using addition? i.e.:

$$X = \texttt{scanI}\ +\ 0\ S = \langle 1,-1,2,1,3,0\rangle$$

Now for any j^{th} position consider all positions $i < j$. To calculate the sum from immediately after i to j all we have to do is return $X_j - X_i$. This difference represents the total sum of the subsequence from $i+1$ to j since we are taking the sum up to j and then subtracting off the sum up to i. For example to calculate the sum between the -2 (location $i+1=1$) and the 2 (location $i=4$) we take $X_4 - X_0 = 3-1 = 2$, which is indeed the sum of the subsequence $\langle -2,3,-1,2\rangle$.

Now consider how for each j we might calculate the maximum sum that starts at any $i \leq j$ and ends at j. Call it R_j. This can be calculated as follows:

$$\begin{aligned} R_j &= \max_{i=0}^{j} \sum_{k=i}^{j} S_k \\ &= \max_{i=0}^{j} (X_j - X_{i-1}) \\ &= X_j + \max_{i=0}^{j} (-X_{i-1}) \\ &= X_j + \max_{i=0}^{j-1} (-X_i) \\ &= X_j - \min_{i=0}^{j-1} X_i \end{aligned}$$

The last equality is because the maximum of a negative is the minimum of the positive. This indicates that all we need to know is X_j and the minimum previous $X_i, i < j$. This can be calculated with a scan using the minimum operation. Furthermore the result of this scan is the same for everyone, so we need to calculate it just once. The result of the scan is:

$$(M, \_) = \texttt{scan}\ \min\ 0\ X = (\langle\, 0, 0, -1, -1, -1, -1 \,\rangle, -1)\ ,$$

and now we can calculate R:

$$R = \langle\, X_j - M_j : 0 \leq j < |S| \,\rangle = \langle\, 1, -1, 3, 2, 4, 1 \,\rangle\ .$$

You can verify that each of these represents the maximum contiguous subsequence sum ending at position j.

Finally, we want the maximum string ending at any position, which we can do with a reduce using $\max$. This gives 4 in our example.

Putting this all together we get the following very simple algorithm:

Algorithm 6.10 (Scan-based MCSS)**.**

```
1 function MCSS(S) =
2 let
3    val X = scanI + 0 S
4    val (M,_) = can min 0 X
5    val Y = ⟨ X_j − M_j : 0 ≤ j < |S| ⟩
6 in
7    max(Y)
8 end
```

Given the costs for scan and the fact that addition and minimum take constant work, this algorithm has $O(n)$ work and $O(\log n)$ span.

6.2.2 Copy Scan

Previously, we used `scan` to compute partial sums to solve the maximum contiguous subsequence sum problem and to match parentheses. Scan is also useful when you want pass information along the sequence. For example, suppose you have some "marked" elements that you would like to copy across to their right until they reach another marked element. One way to mark the elements is to use options.

That is, suppose you are given a sequence of type α `option seq`. For example

$$\langle\, \mathit{NONE},\ \mathit{SOME}(7),\ \mathit{NONE},\ \mathit{NONE},\ \mathit{SOME}(3),\ \mathit{NONE}\,\rangle$$

and your goal is to return a sequence of the same length where each element receives the previous SOME value. For the example:

$$\langle\, \mathit{NONE},\ \mathit{NONE},\ \mathit{SOME}(7),\ \mathit{SOME}(7),\ \mathit{SOME}(7),\ \mathit{SOME}(3)\,\rangle$$

Using a sequential loop or `iter` would be easy. How would you do this with `scan`?

If we are going to use a `scan` directly, the combining function f must have type

$$\alpha\ \texttt{option} \times \alpha\ \texttt{option} \to \alpha\ \texttt{option}$$

How about

```
1  function copy(a, b) =
2    case b of
3      SOME(_) ⇒ b
4    | NONE ⇒ a
```

What this function does is basically pass on its right argument if it is `SOME` and otherwise it passes on the left argument. To be used in a scan it needs to be associative. In particular we need to show that $\texttt{copy}(x, \texttt{copy}(y, z)) = \texttt{copy}(\texttt{copy}(x, y), z)$ for all x, y and z. There are eight possibilities corresponding to each of x, y and z being either `SOME` or `NONE`. For the cases that $z = \mathit{SOME}(c)$ it is easy to verify that that either ordering returns z. For the cases that $z = \mathit{NONE}$ and $y = \mathit{SOME}(b)$ one can verify that both orderings give y, for the cases that $y = z = \mathit{NONE}$ and $x = \mathit{SOME}(a)$ they both return x, and for all being `NONE` either ordering returns `NONE`.

There are many other applications of scan in which more involved functions are used. One important case is to simulate a finite state automaton.

6.2.3 Contraction and Implementing Scan

Now let's consider how to implement `scan` efficiently and at the same time apply one of the algorithmic techniques from our toolbox of techniques: *contraction*. Throughout the following

discussion we assume the work of the binary operator is $O(1)$. As described earlier a brute force method for calculating scans is to apply a reduce to all prefixes. This requires $O(n^2)$ work and is therefore not work-efficient since we can do it in $O(n)$ work sequentially.

Beyond the wonders of what it can do, a surprising fact about `scan` is that it can be accomplished efficiently in parallel, although on the surface, the computation it carries out appears to be sequential in nature. At first glance, we might be inclined to believe that any efficient algorithms will have to keep a cumulative "sum," computing each output value by relying on the "sum" of the all values before it. It is this apparent dependency that makes *scan* so powerful. We often use *scan* when it seems we need a function that depends on the results of other elements in the sequence, for example, the copy scan above.

Suppose we are to run `plus_scan` (i.e. `scan (op +)`) on the sequence $\langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$. What we should get back is

$$(\langle 0, 2, 3, 6, 8, 10, 15, 19 \rangle, 20)$$

We will use this as a running example.

Divide and Conquer: We first consider a divide-and-conquer solution. We can do this by splitting the sequence in half, solving each half and then trying to put the results together. The question is how do we put the results together. In particular lets say the scans on the two halves return (S_l, t_l) and (S_r, t_r), which would be $(\langle 0, 2, 3, 6 \rangle, 8)$ and $(\langle 0, 2, 7, 11 \rangle, 12)$ in our example. Note that S_l already gives us the first half of the solution.

Question 6.11. *How do we get the second half?*

To get the second half, note that in calculating S_r in the second half we started with the identity instead of the sum of the first half, t_l. Therefore if we add the sum of the first half, t_l, to each element of S_r, we get the desired result. This leads to the following algorithm:

Algorithm 6.12 (Scan using divide and conquer)**.**

```
1  function scan f I S =
2    case showt(S) of
3      EMPTY ⇒ (⟨⟩, I)
4    | ELT(v) ⇒ (⟨I⟩, v)
5    | NODE(L, R) ⇒ let
6          val ((S_l, t_l), (S_r, t_r)) = (scan f I L ‖ scan f I R)
7          val X_r = ⟨f(t_l, y) : y ∈ S_r⟩
8        in
9          (append(S_l, X_r), t_l + t_r)
10       end
```

One caveat about this algorithm is that it only works if I is really the "identity" for f, i.e. $f(I, x) = x$, although it can be fixed to work in general.

We now consider the work and span for the algorithm. Note that the joining step requires a map to add t_l to each element of S_r, and then an append. Both these take $O(n)$ work and $O(1)$ span, where $n = |S|$. This leads to the following recurrences for the whole algorithm:

$$W(n) = 2W(n/2) + O(n) \in O(n \log n)$$

$$S(n) = S(n/2) + O(1) \in O(\log n)$$

Although this is much better than $O(n^2)$ work, we can do better.

Contraction: To compute `scan` in $O(n)$ work in parallel, we introduce a new inductive technique common in algorithms design: contraction. It is inductive in that such an algorithm involves solving a smaller instance of the same problem, much in the same spirit as a divide-and-conquer algorithm. But with contraction, there is only one subproblem. In particular, the contraction technique involves the following steps:

1. Contract the instance of the problem to a smaller instance (of the same sort).
2. Solve the smaller instance recursively.
3. Use the solution to help solve the original instance.

The contraction approach is a useful technique in algorithm design in general but for various reasons it is more common in parallel algorithms than in sequential algorithms. This is usually because both the contraction and expansion steps can be done in parallel and the recursion only goes logarithmically deep because the problem size is shrunk by a constant fraction each time.

We'll demonstrate this technique first by applying it to a slightly simpler problem, `reduce`. To begin, we have to answer the following question: *How do we make the input instance smaller in a way that the solution on this smaller instance will benefit us in constructing the final solution?*

The idea is simple: We apply the combining function pairwise to adjacent elements of the input sequence and recursively run `reduce` on it. In this case, the third step is a "no-op"; it does nothing. For example on input sequence $\langle\, 2, 1, 3, 2, 2, 5, 4, 1 \,\rangle$ with addition, we would contract the sequence to $\langle\, 3, 5, 7, 5 \,\rangle$. Then we would continue to contract recursively to get the final result. There is no expansion step.

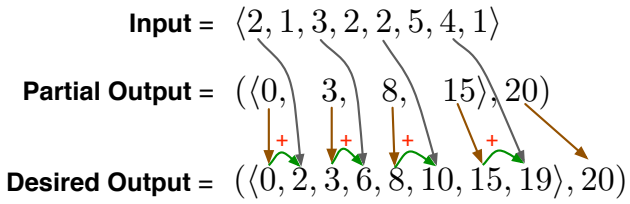
Thought Experiment II: How can we use the same idea to implement scan? What would be the result after the recursive call? In the example above it would be

$$(\langle\, 0, 3, 8, 15 \,\rangle, 20).$$

But notice, this sequence is every other element of the final scan sequence, together with the final sum—and this is enough information to produce the desired final output. This time, the

third expansion step is needed to fill in the missing elements in the final scan sequence: Apply the combining function element-wise to the even elements of the input sequence and the results of the recursive call to `scan`.

To illustrate, the diagram below shows how to produce the final output sequence from the original sequence and the result of the recursive call:



This leads to the following code. The algorithm we present works for when n is a power of two.

Algorithm 6.13 (Scan Using Contraction, for powers of 2)**.**

```
1  function scanPow2 f i s =
2    case |s| of
3      0 ⇒ (⟨ ⟩, i)
4    | 1 ⇒ (⟨ i ⟩, s[0])
5    | n ⇒ let
6             val s' = ⟨ f(s[2i], s[2i+1]) : 0 ≤ i < n/2 ⟩
7             val (r, t) = scanPow2 f i s'
8           in
9             (⟨ p_i : 0 ≤ i < n ⟩, t),  where p_i = { r[i/2]            if even(i)
                                                    { f(r[i/2], s[i-1])  otherwise.
10          end
```

6.3 Reduce Operation

Recall that `reduce` function has the interface

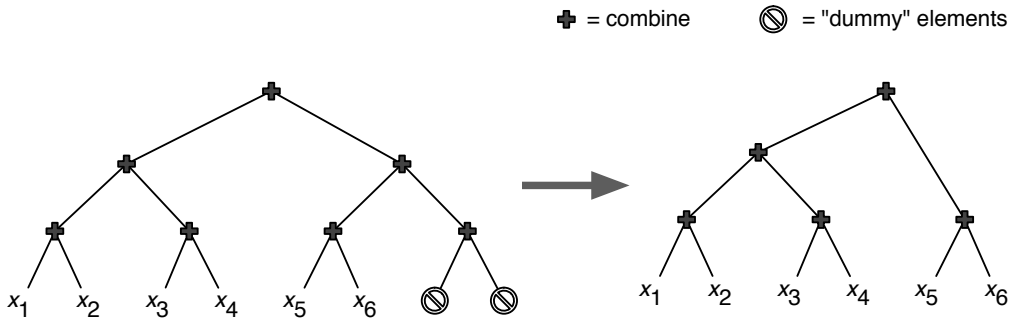
$$\texttt{reduce}\ f\ I\ S : (\alpha \times \alpha \to \alpha) \to \alpha \to \alpha\ \texttt{seq} \to \alpha$$

When the combining function f is associative—i.e., $f(f(x, y), z) = f(x, f(y, z))$ for all x, y and z of type α—`reduce` returns the sum with respect to f of the input sequence S. It is the same result returned by `iter` $f\ I\ S$. The reason we include `reduce` is that it is parallel, whereas `iter` is strictly sequential. Note, though, `iter` can use a more general combining function with type: $\beta \times \alpha \to \beta$.

The results of `reduce` and `iter`, however, may differ if the combining function is non-associative. In this case, the order in which the reduction is performed determines the result; because the function is non-associative, different orderings will lead to different answers. While we might try to apply reduce to only associative operations, unfortunately even some functions that seem to be associative are actually not. For instance, floating point addition and multiplication are not associative. In SML/NJ, integer addition is also not associative because of the overflow exception.

To properly deal with combining functions that are non-associative, it is therefore important to specify the order that the combining function is applied to the elements of a sequence. This order is part of the specification of the ADT `Sequence`. In this way, every (correct) implementation returns the same result when applying `reduce`; the results are deterministic regardless of what data structure and algorithm are used.

For this reason, we define a specific combining tree, which is defined quite carefully in the library documentation for `reduce`. This tree is the same as if we rounded up the length of the input sequence to the next power of 2, i.e., $|x| = 2^k$, and then put a perfectly balanced binary tree[2] over the sequence with 2^k leaves. Wherever we are missing children in the tree, we don't apply the combining function. An example is shown in the following figure.



6.3.1 Divide and Conquer with Reduce

Now, let's look back at divide-and-conquer algorithms you have encountered so far. Many of these algorithms have a "divide" step that simply splits the input sequence in half, proceed to solve the subproblems recursively, and continue with a "combine" step. This leads to the following structure where everything except what is in boxes is generic, and what is in boxes is specific to the particular algorithm.

[2] A *perfect* binary tree is a tree in which every node other than the leaves have exactly 2 children.

```
1  function myDandC(S) =
2    case showt(S) of
3      EMPTY ⇒ emptyVal
4    | ELT(v) ⇒ base(v)
5    | NODE(L, R) ⇒ let
6        val (L′, R′) = ( myDandC(L) ‖ myDandC(R) )
7      in
8        someMessyCombine(L′, R′)
9      end
```

Algorithms that fit this pattern can be implemented in one line using the sequence `reduce` function. Turning a divide-and-conquer algorithm into a reduce-based solution is as simple as invoking `reduce` with the following parameters:

$$\mathit{reduce}\ \boxed{\texttt{someMessyCombine}}\ \boxed{\texttt{emptyVal}}\ (\mathit{map}\ \boxed{\texttt{base}}\ S)$$

We will take a look two examples where `reduce` can be used to implement a relatively sophisticated divide-and-conquer algorithm. Both problems should be familiar to you.

Algorithm 4: MCSS Using Reduce.

The first example is the Maximum Contiguous Subsequence Sum problem from last lecture. Given a sequence S of numbers, find the contiguous subsequence that has the largest sum—more formally:

$$\mathsf{mcss}(s) = \max\left\{\sum_{k=i}^{j} s_k \ : \ 1 \le i \le n, i \le j \le n\right\}.$$

Recall that the divide-and-conquer solution involved strengthening the problem so it returns four values from each recursive call on a sequence S: the desired result $\mathit{mcss}(S)$, the maximum prefix sum of S, the maximum suffix sum of S, and the total sum of S. We will denote these as M, P, S, T, respectively. We refer to this strengthened problem as mcss'. To solve mcss' we can then use the following implementations for $\mathit{combine}$, base, and $\mathit{emptyVal}$:

$$\textbf{function}\ \mathit{combine}((M_L, P_L, S_L, T_L), (M_R, P_R, S_R, T_L)) = \\ (\max(S_L + P_R, M_L, M_R),\quad \max(P_L, T_L + P_R),\quad \max(S_R, S_L + T_R),\quad T_L + T_R)$$

$$\textbf{function}\ \mathit{base}(v) = (v, v, v, v)$$

$$\textbf{val}\ \mathit{emptyVal} = (-\infty, -\infty, -\infty, 0)$$

and then solve the problem with:

function `mcss'`$(S) =$
  `reduce combine emptyVal (map base` S`)`

Question 6.14. *Is the MCSS combine function described above associative?*

It turns out that the combine function for MCSS is associative, as we would expect for the binary function passed to `reduce`. Indeed this is true for the all the combine functions we have used in divide-and-conquer so far. To prove associativity of the MCSS combine function we could go through all the cases. However a more intuitive way to see it is to consider what `combine`$(A,$ `combine`$(B, C))$ and `combine`$($`combine`$(A, B), C)$ should return. In particular for A, B and C appearing in that order, both ways of associating the combines should return the overall maximum contiguous sum, the overall maximum prefix sum, the overall maximum suffix sum, and the overall sum. The divide-and-conquer algorithm would not be correct if this were not the case.

Stylistic Notes. We have just seen that we could spell out the divide-and-conquer steps in detail or condense our code into just a few lines that take advantage of the almighty `reduce`. *So which is preferable*, using the divide-and-conquer code or using reduce? We believe this is a matter of taste. Clearly, your reduce code will be (a bit) shorter, and for simple cases easy to write. But when the code is more complicated, the divide-and-conquer code is easier to read, and it exposes more clearly the inductive structure of the code and so is easier to prove correct.

Restriction. You should realize, however, that this pattern does not work in general for divide-and-conquer algorithms. In particular, it does not work for algorithms that do more than a simple split that partitions their input in two parts in the middle. For example, it cannot be used for implementing quick sort as the divide step partitions the data with respect to a pivot. This step requires picking a pivot, and then filtering the data into elements less than, equal, and greater than the pivot. It also does not work for divide-and-conquer algorithms that split more than two ways, or make more than two recursive calls.

6.4 Analyzing the Costs of Higher Order Functions

In Section 1 we looked at using `reduce` to solve divide-and-conquer problems. In the MCSS problem the combining function f had $O(1)$ cost (i.e., both its work and span are constant). In that case the cost specifications of `reduce` on a sequence of length n is simply $O(n)$ (linear) work and $O(\log n)$ (logarithmic) span.

Question 6.15. *Does `reduce` have linear work and logarithmic span when the binary function passed to it does not have constant cost?*

Unfortunately when the function passed to reduce does not take constant work, then the work of the reduce is not necessarily linear. More generally when using a higher-order function that is passed a function f (or possibly multiple functions) one needs to consider the cost of f.

For `map` it is easy to find its costs based on the cost of the function applied:

$$\begin{aligned} W(\texttt{map } f\ S) &= 1 + \sum_{s \in S} W(f(s)) \\ S(\texttt{map } f\ S) &= 1 + \max_{s \in S} S(f(s)) \end{aligned}$$

Tabulate is similar. But can we do the same for *reduce*?

Merge Sort. As an example, let's consider merge sort. As you have likely seen from previous courses you have taken, merge sort is a popular divide-and-conquer sorting algorithm with optimal work. It is based on a function `merge` that takes two already sorted sequences and returns a sorted sequence containing all elements from both sequences. We can use our reduction technique for implementing divide-and-conquer algorithms to implement merge sort with a `reduce`. In particular, we can write a version of merge sort, which we refer to as reduceSort, as follows:

val *combine* $=$ *merge*$_<$

val *base* $=$ *singleton*

val *emptyVal* $=$ *empty*()

function *reduceSort*(S) $=$ *reduce combine emptyVal* (*map base* S)

where *merge*$_<$ is a merge function that uses an (abstract) comparison operator $<$. Note that merging is an associative function.

Assuming a constant work comparison function, two sequences S_1 and S_2 with lengths n_1 and n_2 can be merged with the following costs:

$$\begin{aligned} W(\textit{merge}_<(S_1, S_2)) &= O(n_1 + n_2) \\ S(\textit{merge}_<(S_1, S_2)) &= O(\log(n_1 + n_2)) \end{aligned}$$

Question 6.16. *What do you think the cost of* `reduceSort` *is?*

6.4.1 Reduce: Cost Specifications

We want to analyze the cost of `reduceSort`. Does the reduction order matter? As mentioned before, if the combining function is associative, which it is in this case, all reduction orders give the same answer so it seems like it should not matter.

To answer this question, let's consider the sequential reduction order that is used by `iter`, as given by the following code.

```
function iterSort(S) =
  iter merge< (empty()) (map singleton S)
```

Since the merge is associative this is functionally the same as `reduceSort`, but will sequentially add the elements in one after the other. On input $x = \langle x_1, x_2, \ldots, x_n \rangle$, the algorithm will first merge $\langle\ \rangle$ and $\langle x_1 \rangle$, then merge in $\langle x_2 \rangle$, then $\langle x_3 \rangle$, etc.

With this order $\texttt{merge}_<$ is called when its left argument is a sequence of varying size between 1 and $n-1$, while its right argument is always a singleton sequence. The final merge combines $(n-1)$-element with 1-element sequences, the second to last merge combines $(n-2)$-element with 1-element sequences, so on so forth. Therefore, the total work for an input sequence S of length n is

$$W(\texttt{iterSort } S) \ \leq \ \sum_{i=1}^{n-1} c \cdot (1+i) \ \in \ O(n^2)$$

since merge on sequences of lengths n_1 and n_2 has $O(n_1 + n_2)$ work.

Question 6.17. *Can you see what algorithm `iterSort` implements?*

Using this reduction order the algorithm is effectively working from the front to the rear, "inserting" each element into a sorted prefix where it is placed at the correct location to maintain the sorted order. This corresponds to the well-known insertion sort.

We can also analyze the span of `iterSort`. Since we iterate adding in each element after the previous, there is no parallelism between merges, but there is parallelism within a merge. We can calculate the span as

$$S(\texttt{iterSort } x) \ \leq \ \sum_{i=1}^{n-1} c \cdot \log(1+i) \ \in \ O(n \log n)$$

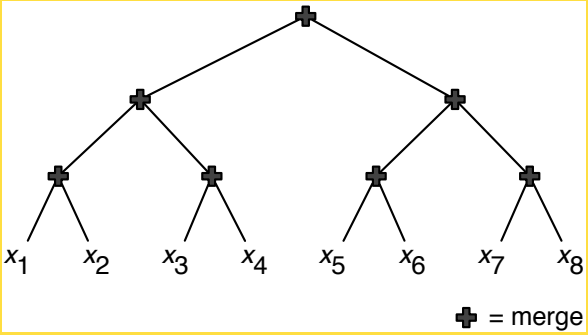
since merge on sequences of lengths n_1 and n_2 has $O(\log(n_1 + n_2))$ span. This means our algorithm does have a reasonable amount of parallelism, $W(n)/S(n) = O(n/\log(n))$, but the real problem is that it does much too much work.

Question 6.18. *Can you think of a way to improve our bound by using a different reduction order with the merge operation?*

In `iterSort`, the reduction tree is unbalanced. We can improve the cost by using a balanced tree instead.

For ease of exposition, let's suppose that the length of our sequence is a power of 2, i.e., $|x| = 2^k$. Now we lay on top the input sequence a perfect binary tree[3] with 2^k leaves and merge according to the tree structure.

Example 6.19. *As an example, the merge sequence for $|x| = 2^3$ is shown below.*



What would the cost be if we use a perfect tree?

At the bottom level where the leaves are, there are $n = |x|$ nodes with constant cost each.

Stepping up one level, there are $n/2$ nodes, each corresponding to a `merge` call, each costing $c(1+1)$. In general, at level i (with $i = 0$ at the root), we have 2^i nodes where each node is a merge with input two sequences of length $n/2^{i+1}$.

Therefore, the work of such a balanced tree of $\texttt{merge}_<$'s is the familiar sum

$$\begin{aligned} &\leq \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^{i+1}} + \frac{n}{2^{i+1}}\right) \\ &= \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^i}\right) \end{aligned}$$

This sum, as you have seen before, evaluates to $O(n \log n)$.

Merge Sort. In fact, this algorithm is essentially the merge sort algorithm. We can use our reduction technique for implementing divide-and-conquer algorithms to implement merge sort with a `reduce`.

In particular, we can write a version of merge sort, which we refer to as reduceSort, as follows:

[3]This is simply a binary tree in which every node either has exactly 2 children or is a leaf, and all leaves are at the same depth.

val $\texttt{combine} = \texttt{merge}_<$

val $\texttt{base} = \texttt{singleton}$

val $\texttt{emptyVal} = \texttt{empty}()$

function $\texttt{reduceSort}(S) = \texttt{reduce combine emptyVal (map base } S)$

where $\texttt{merge}_<$ is a merge function that uses an (abstract) comparison operator $<$.

Summary 6.20. *A brief summary of a few points.*

- *When applying a binary operation in* `reduce`*, if the operation is associative, the order of applications does not matter for the final result.*
- *When applying a binary operation in* `reduce`*, the order of applications does matter when calculating the cost (work and span), regardless of whether the operation is associative or not.*
- *Implementing a "reduce" with* `merge` *with a sequential order leads to insertion sort, while implementing with a balanced tree (parallel order) leads to merge sort.*

The cost of `reduce` in general. In general, how would we go about defining the cost of `reduce` with higher order functions. Given a reduction tree, we'll first define $\mathcal{R}(\texttt{reduce } f \mathbb{I} S)$ as

$$\mathcal{R}(\texttt{reduce } f \mathbb{I} S) = \left\{\text{all function applications } f(a,b) \text{ in the reduction tree}\right\}.$$

Following this definition, we can state the cost of `reduce` as follows:

$$\begin{aligned} W(\texttt{reduce } f \mathbb{I} S) &= O\left(n + \sum_{f(a,b)\in\mathcal{R}(f \mathbb{I} S)} W(f(a,b))\right) \\ S(\texttt{reduce } f \mathbb{I} S) &= O\left(\log n \max_{f(a,b)\in\mathcal{R}(f \mathbb{I} S)} S(f(a,b))\right) \end{aligned}$$

The work bound is simply the total work performed, which we obtain by summing across all combine operations. The span bound is more interesting. The $\log n$ term expresses the fact that the tree is at most $O(\log n)$ deep. Since each node in the tree has span at most $\max_{f(a,b)} S(f(a,b)$, any root-to-leaf path, including the "critical path," has at most $O(\log n \max_{f(a,b)} S(f(a,b))$ span.

This can be used, for example, to prove the following lemma:

Lemma 6.21. *For any combine function $f : \alpha \times \alpha \to \alpha$ and size function $s : \alpha \to \mathbb{R}_+$, if for any x, y,*

1. $s(f(x,y)) \leq s(x) + s(y)$ *and*
2. $W(f(x,y)) \leq c\,(s(x) + s(y))$ *for some constant c,*

then

$$W(\texttt{reduce}\ f\ \mathbb{I}\ S) = O\left(\log|S| \sum_{x \in S}(1 + s(x))\right).$$

Applying this lemma to the merge sort example, we have

$$W(\texttt{reduce}\ \texttt{merge}_<\ \langle\,\rangle\ \ \langle \langle\, a \,\rangle : a \in A \,\rangle) = O(|A| \log |A|)$$

6.5 Collect

Thus far we considered two very important functions on sequences, `scan` and `reduce`.

We now look a third function: `collect`.

Specification of Collect. Let's start with something that you may have heard of.

Question 6.22. *Do you know of key-value stores?*

The term key-value store often refers to a storage systems (which may in on disk or in-memory) that stores pairs of the form "key x value."

Question 6.23. *Can you think of a way of representing a key-value store using a data type that we know?*

We can use a sequence to represent such a store.

Example 6.24. *For example, we may have a sequence of key-value pairs consisting of our students from last semester and the classes they take.*

val `Data` $= \langle$(*"jack sprat"*, *"15-210"*),
(*"jack sprat"*, *"15-213"*),
(*"mary contrary"*, *"15-210"*),
(*"mary contrary"*, *"15-213"*),
(*"mary contrary"*, *"15-251"*),
(*"peter piper"*, *"15-150"*),
(*"peter piper"*, *"15-251"*),
$\ldots\rangle$

Note that key-value pairs are intentionally asymmetric: they map a key to a value. This is fine because that is how we often like them to be.

But sometimes, we often want to put together all the values for a given key.

We refer to this operations as a *collect*.

Example 6.25. *We can determine the classes taken by each student.*

val `classes` = ⟨(*"jack sprat"*, ⟨ *"15-210"*, *"15-213"*, . . . ⟩)
(*"mary contrary"*, ⟨ *"15-210"*, *"15-213"*,*"15-251"*, . . . ⟩)
(*"peter piper"*, ⟨ *"15-210"*,*"15-251"*, . . . ⟩)
. . .⟩

Collecting values together based on a key is very common in processing databases. In relational database languages such as SQL it is referred to as "Group by". More generally it has many applications and furthermore it is naturally parallel.

We will use the function `collect` for this purpose, and it is part of the sequence library. Its interface is:

$$\texttt{collect} : (\alpha \times \alpha \to \textit{order}) \to (\alpha \times \beta)\ \textit{seq} \to (\alpha \times \beta\ \textit{seq})\ \textit{seq}$$

The first argument is a function for comparing keys of type α, and must define a total order over the keys.

The second argument is a sequence of key-value pairs.

The `collect` function collects all values that share the same key together into a sequence, ordering the values in the same order as their appearance in the original sequence.

Example 6.26. *Given sequence of pairs each consisting of a student's name and a course they are taking, we want to collect all entries by course number so we have a list of everyone taking each course. This would give us the roster for each class, which can be viewed as another sequence of key-value pairs. For example,*

val `rosters` = ⟨(*"15-150"*, ⟨ *"peter piper"*, . . . ⟩)
(*"15-210"*, ⟨ *"jack sprat"*, *"mary contrary"*, . . . ⟩)
(*"15-213"*, ⟨ *"jack sprat"*, . . . ⟩)
(*"15-251"*, ⟨ *"mary contrary"*, *"peter piper"* ⟩)
. . .⟩

Question 6.27. *Can you see how to create a roster?*

Example 6.28. *If we wanted to collect the entries of* `Data` *given above by course number to create a roster, we could do the following:*

val `collectStrings` = `collect String.compare`

val `rosters` = `collectStrings`$(\langle (c,n) : (n,c) \in$ `Data`$\rangle)$

This would give something like:

$$\begin{aligned}\textbf{val}\ \texttt{rosters} = \langle &(\text{“15-150”}, \langle \text{“peter piper”}, \ldots \rangle)\\ &(\text{“15-210”}, \langle \text{“jack sprat”, “mary contrary”}, \ldots \rangle)\\ &(\text{“15-213”}, \langle \text{“jack sprat”}, \ldots \rangle)\\ &(\text{“15-251”}, \langle \text{“mary contrary”, “peter piper”} \rangle)\\ &\ldots \rangle\end{aligned}$$

We use a map ($\langle (c,n) : (n,c) \in$ `Data`$\rangle$) to put the course number in the first position in the tuple since that is the position used to collect on.

Cost of Collect. Collect can be implemented by sorting the keys based on the given comparison function, and then partitioning the resulting sequence. In particular, the sort will move the pairs so that all equal keys are adjacent.

A partition function can then identify the positions where the keys change values, and pull out all pairs between each change. Doing this partitioning can be done relatively easily by filtering out the indices where the value changes.

The dominant cost of `collect` is therefore the cost of the sort.

Assuming the comparison has complexity bounded above by W_c work and S_c span, then the costs of `collect` are $O(W_c n \log n)$ work and $O(S_c \log^2 n)$ span.

Exercise 6.29. *Complete the details of this implementation. Better yet, implement collect on your own.*

Question 6.30. *Can you think of another way to implement collect?*

It is also possible to implement a version of collect that runs in linear work using hashing. But hashing would require that a hash function is also supplied and would not return the keys in sorted order. Later we discuss tables which also have a `collect` function. However tables are specialized to the key type and therefore neither a comparison nor a hash function need to be passed as arguments.

6.6 Using Collect in Map Reduce

Some of you have probably heard of the map-reduce paradigm first developed by Google for programming certain data intensive parallel tasks. It is now widely used within Google as well as by many others to process large data sets on large clusters of machines—sometimes up to tens of thousands of machines in large data centers. The map-reduce paradigm is often used to analyze various statistical data over very large collections of documents, or over large log files that track the activity at web sites. Outside Google the most widely used implementation is the Apache Hadoop implementation, which has a free license (you can install it at home). The paradigm is different from the `mapReduce` function you might have seen in 15-150 which just involved a map then a reduce. The map-reduce paradigm actually involves a map followed by a collect followed by a bunch of reduces, and therefore might be better called the map-collect-reduces.

The map-reduce paradigm processes a collection of documents based on a map function f_m and a reduce function f_r supplied by the user. The f_m function must take a document as input and generate a sequence of key-value pairs as output. This function is mapped over all the documents. All key-value pairs across all documents are then collected based on the key. Finally the f_r function is applied to each of the keys along with its sequence of associated values to reduce to a single value.

In ML the types for map function f_m and reduce function f_r are the following:

$$\begin{array}{rcl} f_m & : & (\mathit{document} \to (\mathit{key} \times \alpha)\ \mathit{seq}) \\ f_r & : & (\mathit{key} \times (\alpha\ \mathit{seq}) \to \beta) \end{array}$$

In most implementations of map-reduce the document is a string (just the contents of a file) and the key is also a string. Typically the α and β types are limited to certain types. Also, in most implementations both the f_m and f_r functions are sequential functions. Parallelism comes about since the f_m function is mapped over the documents in parallel, and the f_r function is mapped over the keys with associated values in parallel.

In ML map reduce can be implemented as follows

```
1  function mapCollectReduce fm fr docs =
2    let
3      val pairs = flatten ⟨ fm(s) : s ∈ docs ⟩
4      val groups = collect String.compare pairs
5    in
6      ⟨ fr(g) : g ∈ groups ⟩
7    end
```

The function `flatten` simply flattens a nested sequence into a flat sequence, e.g.:

$$\begin{array}{rl} & \texttt{flatten}\,\langle\,\langle\,a,b,c\,\rangle\,,\langle\,d,e\,\rangle\,\rangle \\ \Rightarrow & \langle\,a,b,c,d,e\,\rangle \end{array}$$

As an example application of the paradigm, suppose we have a collection of documents, and we want to know how often every word appears across all documents. This can be done with the following f_m and f_r functions.

function $f_m(\texttt{doc}) = \langle (w, 1) : \texttt{tokens doc} \rangle$

function $f_r(w, s) = (w, \texttt{reduce} \ + \ 0 \ s)$

Here `tokens` is a function that takes a string and breaks it into tokens by removing whitespace and returning a sequence of strings between whitespace.

Now we can apply `mapCollectReduce` to generate a `countWords` function, and apply this to an example case.

val $\texttt{countWords} = \texttt{mapCollectReduce} \ f_m \ f_r$

$\texttt{countWords} \ \langle$*"this is a document"*,
"this is is another document",
"a last document"$\rangle$

$\Rightarrow \langle$ (*"a"*, 2), (*"another"*, 1), (*"document"*, 3), (*"is"*, 3), (*"last"*, 1), (*"this"*, 2) $\rangle$

6.7 Single-Threaded Array Sequences

In this course we will be using purely functional code because it is safe for parallelism and enables higher-order design of algorithms by use of higher-order functions. It is also easier to reason about formally, and is just cool. For many algorithms using the purely functional version makes no difference in the asymptotic work bounds—for example quickSort and mergeSort use $\Theta(n \log n)$ work (expected case for quickSort) whether purely functional or imperative. However, in some cases purely functional implementations lead to up to a $O(\log n)$ factor of additional work. To avoid this we will slightly cheat in this class and allow for benign "effect" under the hood in exactly one ADT, described in this section. These effects do not affect the observable values (you can't observe them by looking at results), but they do affect cost analysis—and if you sneak a peak at our implementation, you will see some side effects.

The issue has to do with updating positions in a sequence. In an imperative language updating a single position can be done in "constant time". In the functional setting we are not allowed to change the existing sequence, everything is persistent. This means that for a sequence of length n an update can either be done in $\Theta(n)$ work with an arraySequence (the whole sequence has to be copied before the update) or $\Theta(\log n)$ work with a treeSequence (an update involves traversing the path of a tree to a leaf). In fact you might have noticed that our sequence interface does not even supply a function for updating a single position. The reason is both to discourage sequential computation, but also because it would be expensive.

Consider a function `update` (i, v) S that updates sequence S at location i with value v returning the new sequence. This function would have cost $\Theta(|S|)$ in the arraySequence cost specification. Someone might be tempted to write a sequential loop using this function. For example for a function $f : \alpha -> \alpha$, a `map` function can be implemented as follows:

function `map` f $S =$
  `iter` (**fn** $((i, S'), v) \Rightarrow (i + 1,$ `update` $(i, f(v))$ $S'))$
    $(0, S)$
    S

This code iterates over S with i going from 0 to $n - 1$ and at each position i updates the value S_i with $f(S_i)$. The problem with this code is that even if f has constant work, with an `arraySequence` this will do $\Theta(|S|^2)$ total work since every update will do $\Theta(|S|)$ work. By using a `treeSequence` implementation we can reduce the work to $\Theta(|S| \log |S|)$ but that is still a factor of $\Theta(\log |S|)$ off of what we would like.

In the class we sometimes do need to update either a single element or a small number of elements of a sequence. We therefore introduce an ADT we refer to as a *Single Threaded Sequence* (`stseq`). Although the interface for this ADT is quite straightforward, the cost specification is somewhat tricky. To define the cost specification we need to distinguish between the latest "copy" of an instance of an `stseq`, and earlier copies. Basically whenever we update a sequence we create a new "copy", and the old "copy" is still around due to the persistence in functional languages. The cost specification is going to give different costs for updating the latest copy and old copies. Here we will only define the cost for updating the latest copy, since this is the only way we will be using an `stseq`. The interface and costs is as follows:

	Work	Span
`fromSeq(S) : α seq → α stseq` Converts from a regular sequence to a stseq.	$O(\|S\|)$	$O(1)$
`toSeq(ST) : α stseq → α seq` Converts from a stseq to a regular sequence.	$O(\|S\|)$	$O(1)$
`nth ST` i `: α stseq → int → α` Returns the i^{th} element of ST. Same as for seq.	$O(1)$	$O(1)$
`update` (i,v) ST `: (int × α) → α stseq → α stseq` Replaces the i^{th} element of ST with v.	$O(1)$	$O(1)$
`inject` I ST `: (int × α) seq → α stseq → α stseq` For each $(i, v) \in I$ replaces the i^{th} element of ST with v.	$O(\|I\|)$	$O(1)$

An `stseq` is basically a sequence but with very little functionality. Other than converting to and from sequences, the only functions are to read from a position of the sequence (`nth`), update a position of the sequence (`update`) or update multiple positions in the sequence (`inject`). To

use other functions from the sequence library, one needs to covert an `stseq` back to a sequence (using `toSeq`).

In the cost specification the work for both `nth` and `update` is $O(1)$, which is about as good as we can get. Again, however, this is only when S is the latest version of a sequence (i.e. noone else has updated it). The work for `inject` is proportional to the number of updates. It can be viewed as a parallel version of `update`.

Now with an `stseq` we can implement our map as follows:

```
1  function map f S = let
2     val S' = StSeq.fromSeq(S)
3     val R = iter (fn ((i, S''), v) ⇒ (i + 1, StSeq.update (i, f(v)) S''))
4                  (0, S')
5                  S
6  in
7     StSeq.toSeq(R)
8  end
```

This implementation first converts the input sequence to an `stseq`, then updates each element of the `stseq`, and finally converts back to a sequence. Since each update takes constant work, and assuming the function f takes constant work, the overall work is $O(n)$. The span is also $O(n)$ since `iter` is completely sequential. This is therefore not a good way to implement `map` but it does illustrate that the work of multiple updates can be reduced from $\Theta(n^2)$ on array sequences or $O(n \log n)$ on tree sequences to $O(n)$ using an `stseq`.

Implementing Single Threaded Sequences. You might be curious about how single threaded sequences can be implemented so they act purely functional but match the cost specification. Here we will just briefly outline the idea.

The trick is to keep two copies of the sequence (the original and the current copy) and additionally to keep a "change log". The change log is a linked list storing all the updates made to the original sequence. When converting from a sequence to an `stseq` the sequence is copied to make a second identical copy (the current copy), and an empty change log is created. A different representation is now used for the latest version and old versions of an `stseq`. In the latest version we keep both copies (original and current) as well as the change log. In the old versions we only keep the original copy and the change log. Lets consider what is needed to update either the current or an old version. To update the current version we modify the current copy in place with a side effect (non functionally), and add the change to the change log. We also take the previous version and mark it as an old version removing its current copy. When updating an old version we just add the update to its change log. Updating the current version requires side effects since it needs to update the current copy in place, and also has to modify the old version to mark it as old and remove its current copy.

Either updating the current version or an old version takes constant work. The problem is the cost of `nth`. When operating on the current version we can just look up the value in the current

copy, which is up to date. When operating on an old version, however, we have to go back to the original copy and then check all the changes in the change log to see if any have modified the location we are asking about. This can be expensive. This is why updating and reading the current version is cheap ($O(1)$ work) while working with an old version is expensive.

In this course we will use `stseq`s for some graph algorithms, including breadth-first search (BFS) and depth-first search (DFS), and for hash tables.