

Algorithms for Sequences

1 Sequences

Recall from Lecture One, the mathematical definition of a **sequence**: an ordered collection of elements where each position $0 \leq i < n$ has exactly one corresponding value.

Definition: Sequence

Mathematically, a finite **sequence** of length n over elements of type T is a function

$$\{0, 1, \dots, n-1\} \rightarrow T.$$

When working with abstractions, it is useful to separate three questions: what operations a type must provide so algorithms can consume it, how that type is represented internally, and what higher-level library operations can be built generically on top of those requirements.

Definition: Sequence type requirements

For a sequence with value type T , a `sequence<T>` must support:

Sequence Access:

- `nth : (S : sequence<T>, i : int) -> T`
Return the i^{th} element (zero-indexed) of the sequence S .
- `length : (S : sequence<T>) -> int`
Return the length (number of elements in) the sequence S .

The `nth` function is usually abbreviated as `S[i]`, and `length` is abbreviated as `|S|`.

Subsequence Views:

- `subseq : (S : sequence<T>, i : int, k : int) -> sequence<T>`
Returns a *view* of the subsequence of S starting at index i with length k .

The function `subseq` returns a view rather than copying the elements of the subsequence.

The **sequence library** provides many additional operations for constructing, transforming, and combining sequences. Some important ones are:

- `nth`, `length`
- `empty`, `singleton`
- `tabulate`, `map`, `zip`
- `subseq`, `split_mid`
- `filter`, `flatten`
- `reduce`, `scan`
- `sort`, `merge`
- `collect`, `compare`

Importantly, the abstract notion of a sequence does not prescribe one particular representation. The same sequence requirements could be provided by arrays, trees, or other data structures, with different implementations providing different cost bounds.

In this lecture, we will look under the hood at one particularly important implementation, `ArraySequence`, and use it to implement several fundamental sequence algorithms efficiently.

2 Arrays and Imperative Parallelism

2.1 Arrays

Arrays work slightly differently in different programming languages, but for the most part they support the following set of fundamental operations with some syntax in some form or another:

- `allocate<T>(n)`: allocate an array of length n of type T
- `A[i]`: return the i^{th} element of A
- `|A|`: return the length of A
- `A[i] ← x`: write the value x to the i^{th} element of A

All of these operations runs in $O(1)$ time. In most languages, `|A|` would be written as `A.length` or `A.size()` or something similar, but we'll keep this shorthand to make our pseudocode nicer.

And of course, to make everything work under the hood, we are going to need to assume that arrays are *mutable*. The fourth operation in our list of primitive array operations allows us to write a value to an element of the array.

2.2 Imperative Parallel Loops

Thus far, we have written and described parallelism in algorithms using only functional constructs; either parallel list comprehension (our pseudocode for `tabulate`), which evaluates a (pure) function over many indices in parallel and returns the results as a sequence, or parallel tuple evaluation, which evaluates several expressions in parallel and returns their results.

By default we try to avoid side-effects since this makes code harder to reason about. To implement the low level parallel operations that our algorithms rely on, we will however need to write *imperative* parallel code, i.e., parallel code with side effects.

The standard abstraction used and supported by most imperative parallel programming libraries is the **parallel for loop**. A parallel for loop evaluates the iterations of its loop body in parallel.

Definition: Parallel For Loop

A parallel for loop executes its body for every value of its range in parallel. In our pseudocode, we denote a parallel for loop, where $e(i)$ is some (impure) function of i , as:

```
parallel for i in ...:  
   $e(i)$ 
```

Since the iterations of the parallel for loop execute in parallel, its work and span are

$$W = \sum_i W(e(i)), \quad S = \max_i S(e(i)).$$

Now that we have a parallel construct that can execute imperative code, we are vulnerable to a new kind of bug that we must be extremely careful to avoid, a **data race**.

Definition: Data Race

A *data race* occurs when there are two unsynchronized parallel operations on the same memory location, and at least one of the operations is a write.

Note that two parallel *reads* of the same memory location is not a data race. Any access that involves a write must be treated with care.

The precise meaning of “synchronization” depends on the programming language, its memory model, and sometimes even the hardware. These details are beyond the scope of this class. As a result, we will adopt a deliberately conservative rule: *All parallel operations in this class are treated as unsynchronized.*

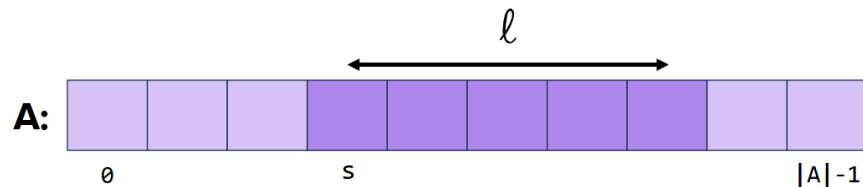
Under this rule, **any** pair of parallel operations that access the same memory location, where at least one is a write, constitutes a data race. This cannot happen in a purely functional program, since values are never mutated. However, it *can* happen in imperative parallel constructs such as parallel **for** loops.

In this class, the presence of a data race means the algorithm is *incorrect*. Programs with data races are considered to have undefined behavior and are not valid algorithms.

3 A Data Type for ArraySequence

Because we want our implementation of `subseq` (extract a contiguous subsequence) to be fast, our data structure for `ArraySequence` will keep the actual array, along with two additional integers. These are the starting point of the subsequence, and the length of it. So we have:

```
type ArraySequence<T> =
{
  A: Array<T>
  s: int
  l: int
}
```



s is the starting index of the subsequence in the array A , and ℓ is the length of the subsequence. When we create an `ArraySequence` initially, s will be zero and ℓ will be the length of the array.

3.1 nth and length

`nth` and `length` are straightforward to implement given the `ArraySequence` data type.

Algorithm: *nth*

```
fun nth (S : ArraySequence<T>, i : int) -> T:  
  (A,s,_) = S  
  return A[s+i]
```

Algorithm: *length*

```
fun length(S : ArraySequence<T>) -> int:  
  (_,_, l) = S  
  return l
```

Both of these run in constant time (work and span).

3.2 subseq

Since our `ArraySequence` data type stores an offset and length, implementing `subseq` trivial; we just add to the start and set the new length while keeping the same underlying array.

Algorithm: *Subseq*

```
fun subseq (S: ArraySequence<T>, s: int, l: int) -> sequence<T>:  
  (A,s',l') = S  
  return {A = A, s = s + s', l = l}
```

`subseq` also runs in constant time (work and span).

3.3 tabulate

Since `tabulate` constructs a new sequence, our `ArraySequence` implementation must allocate an array and fill it with the desired values. We can do this using our assumed `allocate` primitive for creating arrays, and a `parallel for` loop.

Algorithm: *Tabulate*

```
fun tabulate (f : (int -> T), n : int) -> sequence<T>:  
  R = allocate<T>(n)  
  parallel for i in 0...n-1:  
    R[i] ← f(i)  
  return {A = R, s = 0, l = n}
```

Is there a data race? No! Because each of the parallel executions of the loop body operates on a different element of `R`. (We assume that the function `f` is pure, i.e., has no side effects.)

The work and span of `tabulate` are

$$W = \sum_{0 \leq i < n} W(f(i)), \quad S = \max_{0 \leq i < n} S(f(i)).$$

4 Sequence Library Functions

With `ArraySequence` as one concrete implementation of the `sequence` abstraction, we can now implement several useful sequence algorithms efficiently. Some are straightforward compositions of existing sequence operations, while others require more sophisticated techniques and manually allocating and filling the array.

We begin with `map` and `append`, which are simple applications of `tabulate`, then progress to `filter` and `flatten`, which are interesting applications of `scan` that also manage their own array. Finally we implement `collect`, and lexicographic comparison, which make use of `sort`, `filter`, `zip`, `map`, and `reduce`.

4.1 Map

The `map` function applies a given **pure** function to every element of a sequence and returns the resulting sequence. It can be implemented directly using `tabulate`: the element at output position i is simply $f(S[i])$.

Algorithm: Map

```
fun map (f : (T -> U), S : sequence<T>) -> sequence<U>:
  return tabulate(fn i => f(S[i]), |S|)
```

The work and span of `map` are

$$W = \sum_{x \in S} W(f(x)), \quad S = \max_{x \in S} S(f(x)).$$

4.2 Append

The `append` function concatenates two sequences. This is also a straightforward application of `tabulate`: for each output index i , we determine whether it belongs to the portion copied from A (indices less than $|A|$) or from B (indices greater than or equal to $|A|$).

Algorithm: Append

```
fun append (A : sequence<T>, B : sequence<T>) -> sequence<T>:
  return tabulate(fn i => A[i] if i < |A| else B[i-|A|], |A|+|B|)
```

The work and span of `append` are

$$W = O(|A| + |B|), \quad S = O(1).$$

4.3 Filter

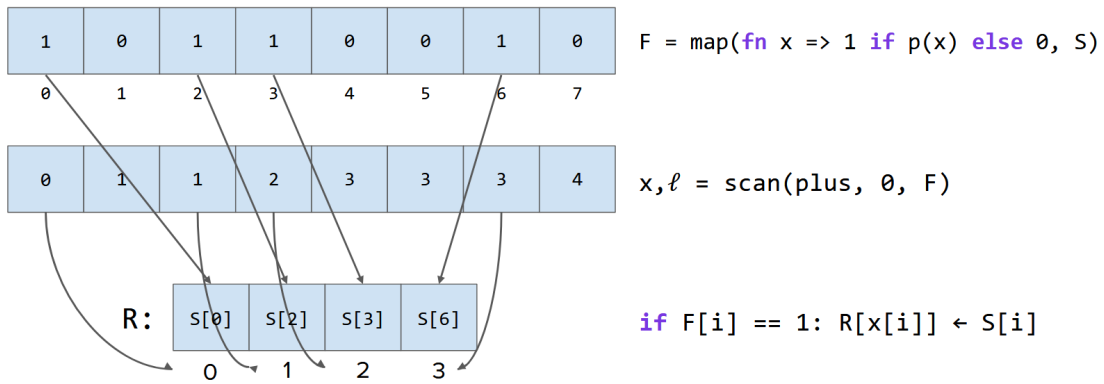
The filter function returns the elements of a sequence that satisfy a predicate, preserving their relative order. Its type is:

`filter : (p : (T -> bool), S : sequence<T>) -> sequence<T>`

For example `filter (fn x => x < 5, [7,1,3,11,7,2])` returns `[1,3,2]`

Our goal is to implement filter in linear work and logarithmic span. The main challenge is determining where each surviving element belongs in the output.

The key observation is that the output position of a surviving element is exactly the number of surviving elements that occur before it. That is a prefix-sum problem, so we can use scan!



First, map each element x to 1 if $p(x)$ is true and 0 otherwise. Scanning these indicator values gives, at every position, the number of surviving elements that occur before it. These prefix sums therefore tell each surviving element exactly where to write itself in the output array.

Because two distinct surviving elements have different prefix sums, they write to different output positions. Therefore these writes can be performed in parallel without a data race.

Algorithm: Filter

```

fun filter (p : (T -> bool), S : sequence<T>) -> sequence<T>:
  F = map(fn x => 1 if p(x) else 0, S)
  X, l = scan(plus, 0, F)
  R = allocate<T>(l)
  parallel for i in 0...|S|-1:
    if F[i] == 1:
      R[X[i]] ← S[i]
  return {A = R, s = 0, l = l}

```

The work and span of filter depend on the cost of the predicate p :

$$W = \sum_{x \in S} W(p(x)), \quad S = O(\log |S|) + \max_{x \in S} S(p(x)).$$

4.4 Flatten

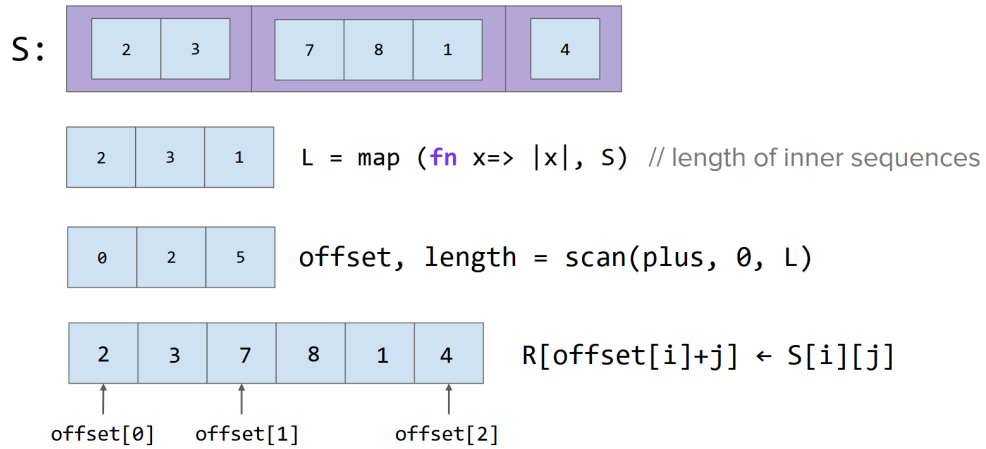
The `flatten` function takes a sequence of sequences and concatenates all of the inner sequences, preserving their relative order. Its type is:

`flatten : (S : sequence<sequence<T>>) -> sequence<T>`

For example, `flatten([[2,3],[7,8,1],[4]])` returns `[2,3,7,8,1,4]`.

As with `filter`, the main challenge is determining where each input element belongs in the output array. For an element at position j of the i^{th} inner sequence, its output position is j plus the total length of all inner sequences that occur before i .

Those are prefix sums of the lengths of the inner sequences, so once again we can use `scan`!



First, we map each inner sequence to its length, then scan those lengths to compute the starting offset of every inner sequence in the output. Once these offsets are known, all of the elements can be copied to their destinations in parallel.

No two input elements are assigned the same output position, so these parallel writes do not cause a data race, so the algorithm is safe and correct.

Algorithm: Flatten

```
fun flatten (S : sequence<sequence<T>>) -> sequence<T>:
  L = map(fn x => |x|, S)
  offset, length = scan(plus, 0, L)
  R = allocate<T>(length)
  parallel for i in 0...|S|-1:
    parallel for j in 0...L[i]-1:
      R[offset[i]+j] <- S[i][j]
  return {A = R, s = 0, l = length}
```

The work and span of `flatten` are

$$W = O\left(\sum_{x \in S} (1 + |x|)\right), \quad S = O(\log |S|).$$

4.5 Collect

The `collect` function takes a sequence of key-value pairs and groups together all values that have the same key. The distinct keys appear in sorted order, while the values associated with each key remain in their original relative order. Its type is:

```
collect : (S : sequence<(K,V)>) -> sequence<(K,sequence<V>)>
```

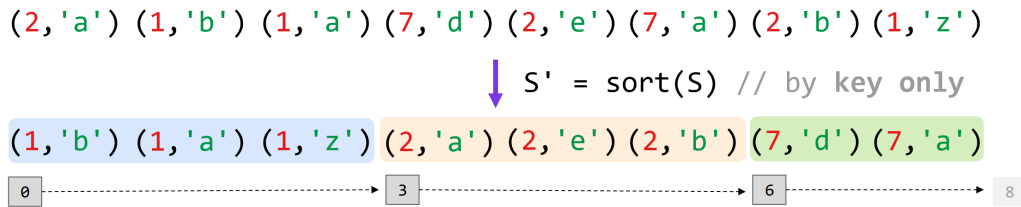
For simplicity, we assume here that keys support `<`; a more general version of `collect` would take a comparison function on keys. For example, with integer keys supporting comparison via `<`,

```
collect([(1,'a'), (2,'b'), (1,'c'), (2,'a'), (2,'d')])
```

returns

```
[(1,['a','c']), (2,['b','a','d'])]
```

The first step is to *stably sort* the pairs by key only. Stability is important: it ensures that pairs with the same key remain in their original relative order, so the values in the final groups will also appear in their original relative order. After sorting, all pairs with the same key occur consecutively. The remaining problem is therefore to find where each new key begins.



We can find these boundaries with `filter`. Consider the sequence of indices $0, \dots, |S|$. An index i is a boundary if it is the beginning or end of the sequence, or if the key at position i differs from the key immediately before it.

If the resulting boundary indices are $i_0, i_1, \dots, i_k$, then the pairs in the group corresponding to the j^{th} distinct key are exactly `subseq(S', ij, ij+1 - ij)`.

For each such group, the output key is the key of its first pair, `S'[ij][0]`, and the output value sequence is obtained by mapping each pair in the group to just its value. Since every group is independent, we can construct all of them in parallel with `tabulate`.

Algorithm: Collect

```
fun collect (S : sequence<(K,V)>) -> sequence<(K,sequence<V>)>:
  S' = sort(S) // stable sort by key only
  ids = filter(
    fn i => i == 0 || i == |S| || S'[i-1][0] < S'[i][0],
    [0, ..., |S|])
  return tabulate(
    fn i => (S'[ids[i]][0],
              map(fn p => p[1],
                  subseq(S', ids[i], ids[i+1]-ids[i]))),
    |ids|-1)
```

The sorting dominates the cost. Sorting costs $O(|S|\log|S|)$ work and $O(\log^2|S|)$ span. The call to `filter` costs $O(|S|)$ work and $O(\log|S|)$ span, and the `tabulate` performs $O(|S|)$ total work because the subsequences corresponding to different keys are disjoint. Therefore,

$$W = O(|S|\log|S|), \quad S = O(\log^2|S|).$$

4.6 Lexicographic Comparison

Lexicographic comparison generalizes the usual dictionary ordering of strings to arbitrary sequences. Given a comparison function on individual elements, we compare two sequences using the first position at which they differ. If no such position exists, the shorter sequence comes first.

We represent the result of a comparison using the type

type `order` = LESS | EQUAL | GREATER

The type of lexicographic comparison is therefore:

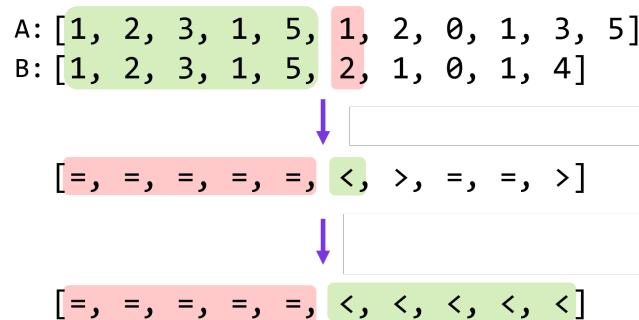
`compare` : ((T,T) -> order, `sequence`<T>, `sequence`<T>) -> order

For example,

```
compare(cmp, [1,5,1,2,2], [1,5,2,1,0]) -> LESS
compare(cmp, [1,5,2,1,0], [1,5,2,1])   -> GREATER
compare(cmp, [1,5,2], [1,5,2])         -> EQUAL
```

The key observation is that the result is determined by the *first mismatching pair*. We can compare all corresponding pairs in parallel by zipping the two sequences and mapping the element comparison function over them, i.e., writing `comps = map(f, zip(a, b))`.

The sequence `comps` contains EQUAL until the first mismatch, followed potentially by arbitrary comparison results. We therefore need to find the *leftmost* value that is not EQUAL.



We can do this with the following associative combining function, which is closely related to the `copy_right` function from the scan lecture:

```
fun copy_notequal(x, y):
  return y if x == EQUAL else x
```

The function keeps its left argument once a non-EQUAL value has appeared, so reducing with it propagates the leftmost non-EQUAL comparison to the final result.

If the reduction instead returns EQUAL, then every position shared by the two sequences is equal. In that case, their lengths determine the lexicographic ordering.

Algorithm: Lexicographic Comparison

```
fun compare(f : (T,T) -> order, a : sequence<T>, b : sequence<T>)
  -> order:
  comps = map(f, zip(a, b))

  fun copy_notequal(x, y):
    return y if x == EQUAL else x

  res = reduce(copy_notequal, EQUAL, comps)
  match res with:
    case EQUAL:
      return LESS if |a| < |b|
      else GREATER if |a| > |b|
      else EQUAL
    case _:
      return res
```

The zip contains only the positions shared by both sequences, so it has length $\min(|a|, |b|)$. Assuming the element comparison function has constant cost, the work and span are therefore

$$W = O(\min(|a|, |b|)), \quad S = O(\log \min(|a|, |b|)).$$