SWIFTCRAFT
21-24 May 2024

Sequencing Success

Exploring Swift Sequences in Depth

Adrian Russell

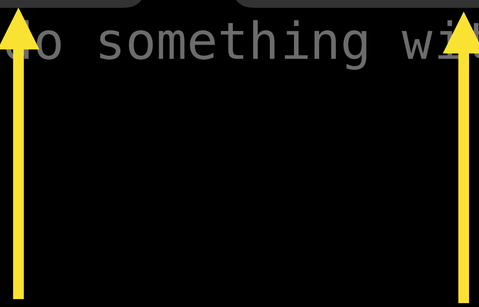
Sequences

Sequence

```
for element in aSequence {
    // do something with element
}
```

Sequence

```
for element in aSequence {
    // do something with element
}
```



Current element

Container

Conforming to `Sequence` *protocol*

Sequence

```
for element in ["a", "b", "c"] {
    // do something with element
}

for i in 0..<10 {
    // do something with i
}

for (key, value) in ["a": 1, "b": 2, "c": 3] {
    // do something with key & element
}
```

Sequence

```
protocol Sequence<Element> {
    associatedtype Element where Self.Element == Self.Iterator.Element

    associatedtype Iterator : IteratorProtocol
    func makeIterator() -> Self.Iterator
}

protocol IteratorProtocol<Element> {
    associatedtype Element

    mutating func next() -> Self.Element?
}
```

IteratorProtocol

Simple example

```
struct AlphabetIterator: IteratorProtocol {
    typealias Element = Character
    private var position = 0

    mutating func next() -> Element? {
        guard position < 26 else { return nil }
        defer { position += 1 }
        return Character(UnicodeScalar(position + 97)!)
    }
}
```

IteratorProtocol

Simple example

```
var iterator = AlphabetIterator()
while let value = counter.next() {
    print(value)
}

// prints a, b, c, d, e, ..., x, y, z
```

Sequence

```
struct AlphabetSequence: Sequence {
    func makeIterator() -> AlphabetIterator {
        AlphabetIterator()
    }
}

for element in AlphabetSequence() {
    print(element)
}

// prints a, b, c, d, e, ..., x, y, z
```

Extension methods

contains(_:) / contains(where:) / allSatisfy(_:) / first(where:) / min() / min(by:)
max() / max(by:) / prefix(_:) / prefix(while:) / suffix(_:) / dropFirst(_:)
dropLast(_:) / drop(while:) / filter(_:) / map<T>(_:) / compactMap(_:)
flatMap(_:) / reduce(_:_:) / reduce(into:_:) / forEach(_:) / sorted()
sorted(by:) / enumerated() / reversed() / shuffled() / shuffled(using:)
split(maxSplits:omittingEmptySubsequences:whereSeparator)
split(separator:maxSplits:omittingEmptySubsequences) / joined()
joined(separator:) / elementsEqual(_:) / elementsEqual(_:by:) / starts(with:)
starts(with:by:) / lexicographicallyPrecedes(_:)
lexicographicallyPrecedes(_:by:)

Extension methods

contains(_:) / contains(where:) / allSatisfy(_:) / first(where:) / min() / min(by:)
max() / max(by:) / prefix(_:) / prefix(while:) / suffix(_:) / dropFirst(_:)
dropLast(_:) / drop(while:) / filter(_:) / map<T>(_:) / compactMap(_:)
flatMap(_:) / reduce(_:_:) / reduce(into:_:) / forEach(_:) / sorted()
sorted(by:) / enumerated() / reversed() / shuffled() / shuffled(using:)
split(maxSplits:omittingEmptySubsequences:whereSeparator)
split(separator:maxSplits:omittingEmptySubsequences) / joined()
joined(separator:) / elementsEqual(_:) / elementsEqual(_:by:) / starts(with:)
starts(with:by:) / lexicographicallyPrecedes(_:)
lexicographicallyPrecedes(_:by:)

Extension methods

contains(_:) / contains(where:) / allSatisfy(_:) / first(where:) / min() / min(by:)
max() / max(by:) / prefix(_:) / prefix(while:) / suffix(_:) / dropFirst(_:)
dropLast(_:) / drop(while:) / filter(_:) / map<T>(_:) / compactMap(_:)
flatMap(_:) / reduce(_:_:) / reduce(into:_:) / forEach(_:) / sorted()
sorted(by:) / enumerated() / reversed() / shuffled() / shuffled(using:)
split(maxSplits:omittingEmptySubsequences:whereSeparator)
split(separator:maxSplits:omittingEmptySubsequences) / joined()
joined(separator:) / elementsEqual(_:) / elementsEqual(_:by:) / starts(with:)
starts(with:by:) / lexicographicallyPrecedes(_:)
lexicographicallyPrecedes(_:by:)

Extension methods

contains(_:) / contains(where:) / allSatisfy(_:) / first(where:) / min() / min(by:)
max() / max(by:) / prefix(_:) / prefix(while:) / suffix(_:) / dropFirst(_:)
dropLast(_:) / drop(while:) / filter(_:) / map<T>(_:) / compactMap(_:)
flatMap(_:) / reduce(_:_:) / reduce(into:_:) / forEach(_:) / sorted()
sorted(by:) / enumerated() / reversed() / shuffled() / shuffled(using:)
split(maxSplits:omittingEmptySubsequences:whereSeparator)
split(separator:maxSplits:omittingEmptySubsequences) / joined()
joined(separator:) / elementsEqual(_:) / elementsEqual(_:by:) / starts(with:)
starts(with:by:) / lexicographicallyPrecedes(_:)
lexicographicallyPrecedes(_:by:)

Extension methods

contains(_:) / contains(where:) / allSatisfy(_:) / first(where:) / min() / min(by:)
max() / max(by:) / prefix(_:) / prefix(while:) / suffix(_:) / dropFirst(_:)
dropLast(_:) / drop(while:) / filter(_:) / map<T>(_:) / compactMap(_:)
flatMap(_:) / reduce(_:_:) / reduce(into:_:) / forEach(_:) / sorted()
sorted(by:) / enumerated() / reversed() / shuffled() / shuffled(using:)
split(maxSplits:omittingEmptySubsequences:whereSeparator)
split(separator:maxSplits:omittingEmptySubsequences) / joined()
joined(separator:) / elementsEqual(_:) / elementsEqual(_:by:) / starts(with:)
starts(with:by:) / lexicographicallyPrecedes(_:)
lexicographicallyPrecedes(_:by:)

Extension methods

contains(_:) / contains(where:) / allSatisfy(_:) / first(where:) / min() / min(by:)

max() / max(by:) / prefix(_:) / prefix(while:) / suffix(_:) / dropFirst(_:)

dropLast(_:) / drop(while:) / filter(_:) / map<T>(_:) / compactMap(_:)

flatMap(_:) / reduce(_:_:) / reduce(into:_:) / forEach(_:) / sorted()

sorted(by:) / enumerated() / reversed() / shuffled() / shuffled(using:)

split(maxSplits:omittingEmptySubsequences:whereSeparator)

split(separator:maxSplits:omittingEmptySubsequences) / joined()

joined(separator:) / elementsEqual(_:) / elementsEqual(_:by:) / starts(with:)

starts(with:by:) / lexicographicallyPrecedes(_:)

lexicographicallyPrecedes(_:by:)

Higher Order Functions

contains(_:) / contains(where:) / allSatisfy(_:) / first(where:) / min() / min(by:)
max() / max(by:) / prefix(_:) / prefix(while:) / suffix(_:) / dropFirst(_:)
dropLast(_:) / drop(while:) / filter(_:) / map<T>(_:) / compactMap(_:)
flatMap(_:) / reduce(_:_:) / reduce(into:_:) / forEach(_:) / sorted()
sorted(by:) / enumerated() / reversed() / shuffled() / shuffled(using:)
split(maxSplits:omittingEmptySubsequences:whereSeparator)
split(separator:maxSplits:omittingEmptySubsequences) / joined()
joined(separator:) / elementsEqual(_:) / elementsEqual(_:by:) / starts(with:)
starts(with:by:) / lexicographicallyPrecedes(_:)
lexicographicallyPrecedes(_:by:)

Reduce

```
let array = [1, 2, 3, 4, 5, 6]
var sum = 0
for element in array {
    sum += element
}
```

```
let sum = [1, 2, 3, 4, 5, 6].reduce(0, { partialResult, element in
    return partialResult + element
})
```

```
let sum = [1, 2, 3, 4, 5, 6].reduce(0, +)
```

Custom extensions

Extending Sequence

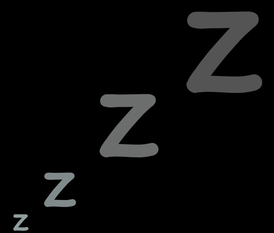
```
extension Sequence where Element: AdditiveArithmetic {
    func sum() -> Element {
        self.reduce(.zero, +)
    }
}

let v = (1...6).sum()
```

EnumeratedSequence

Sequence with an offset count

```
extension Sequence {
     func enumerated() -> EnumeratedSequence<Self>
}

for (index, value) in ["a", "b", "c", "d", "e"].enumerated() {
    print([index, value])
}
// prints [0, "a"], [1, "b"], [2, "c"], [3, "d"], [4, "e"]


for (index, value) in ["a", "b", "c", "d", "e"][1...3].enumerated() {
    print([index, value])
}
// prints [0, "b"], [1, "c"], [2, "d"]
```

Lazy



Lazy

```
extension Sequence<Element> {
     var lazy: LazySequence<Self>
}


struct LazySequence<Base: Sequence>: LazySequenceProtocol { }


protocol LazySequenceProtocol: Sequence {
    associatedtype Elements: Sequence = Self where Elements.Element == Element
    var elements: Elements { get }
}
```

Extension methods

contains(_:) / contains(where:) / allSatisfy(_:) / first(where:) / min() / min(by:)

max() / max(by:) / prefix(_:) / prefix(while:) / suffix(_:) / dropFirst(_:)

dropLast(_:) / drop(while:) / filter(_:) / map<T>(_:) / compactMap(_:)

flatMap(_:) / reduce(_:_:) / reduce(into:_:) / forEach(_:) / sorted()

sorted(by:) / enumerated() / reversed() / shuffled() / shuffled(using:)

split(maxSplits:omittingEmptySubsequences:whereSeparator)

split(separator:maxSplits:omittingEmptySubsequences) / joined()

joined(separator:) / elementsEqual(_:) / elementsEqual(_:by:) / starts(with:)

starts(with:by:) / lexicographicallyPrecedes(_:)

lexicographicallyPrecedes(_:by:)

Extension methods

contains(_:) / contains(where:) / allSatisfy(_:) / first(where:) / min() / min(by:)
max() / max(by:) / prefix(_:) / prefix(while:) / suffix(_:) / dropFirst(_:)
dropLast(_:) / drop(while:) / filter(_:) / map<T>(_:) / compactMap(_:)
flatMap(_:) / reduce(_:_:) / reduce(into:_:) / forEach(_:) / sorted()
sorted(by:) / enumerated() / reversed() / shuffled() / shuffled(using:)
split(maxSplits:omittingEmptySubsequences:whereSeparator)
split(separator:maxSplits:omittingEmptySubsequences) / joined()
joined(separator:) / elementsEqual(_:) / elementsEqual(_:by:) / starts(with:)
starts(with:by:) / lexicographicallyPrecedes(_:)
lexicographicallyPrecedes(_:by:)

Contrived Example

From the first million positive integers, How do you find the 1st & 100th even number which spelled-out contains the letter ' v ' ?

Contrived Example

Non-Lazy

```
let values = (0..<1_000_000)
    .filter { $0.isMultiple(of: 2) }    // (runs 1_000_000 times) Array<Int>(500_000)
    .map { $0.formatted(.spellOut) }    // (runs 500_000 times)   Array<String>(500_000)
    .filter { $0.contains("v") }        // (runs 500_000 times)   Array<String>(300_064)
    .prefix(100)                        // ArraySlice<Element>

print(values.first!)    // prints "twelve"
print(values.last!)     // prints "seven hundred twenty-six"
```

Lazy

```
let values = (0..<1_000_000)
    .lazy                                       // LazySequence<Range<Int>>
    .filter { $0.isMultiple(of: 2) }            // LazyFilterSequence<Range<Int>>
    .map { $0.formatted(.spellOut) }            // LazyMapSequence<LazyFilterSequence<Range<Int>>, String>
    .filter { $0.contains("v") }                // LazyFilterSequence<LazyMapSequence<LazyFilterSequence<Range<Int>>, String>>
    .prefix(100)                                // LazyFilterSequence<LazyMapSequence<LazyFilterSequence<Range<Int>>, String>>

print(values.first!)    // prints "twelve"
print(values.last!)     // prints "seven hundred twenty-six"
```

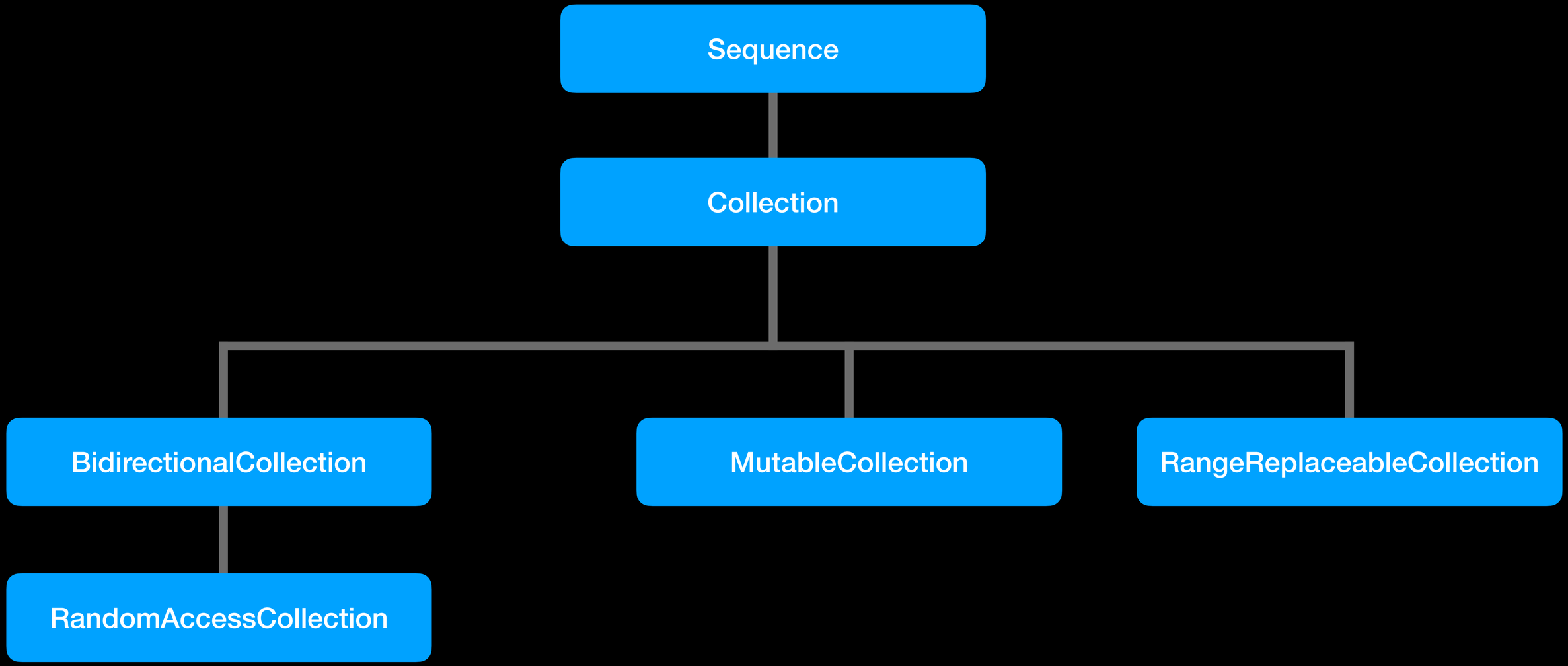
Lazy

```
let values = (0..<1_000_000)
    .lazy
    .filter { $0.isMultiple(of: 2) }    // (runs 13 times) + (runs 737 times)
    .map { $0.formatted(.spellOut) }    // (runs 7 times)  + (runs 369 times)
    .filter { $0.contains("v") }        // (runs 7 times)  + (runs 369 times)
    .prefix(100)

print(values.first!)    // prints "twelve"
print(values.last!)     // prints "seven hundred twenty-six"
```

Lazy

```
let values = Array( (0..<1_000_000)
    .lazy
    .filter { $0.isMultiple(of: 2) }    // (runs 737 times)
    .map { $0.formatted(.spellOut) }    // (runs 369 times)
    .filter { $0.contains("v") }        // (runs 369 times)
    .prefix(100)
)                                       // Array<String>(100)

print(values.first!)    // prints "twelve"
print(values.last!)     // prints "seven hundred twenty-six"
```

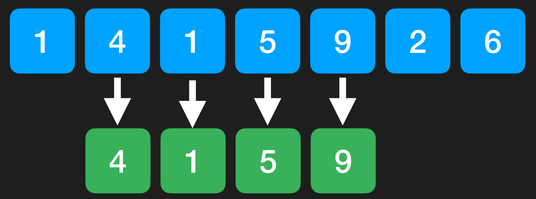
Protocol Structure



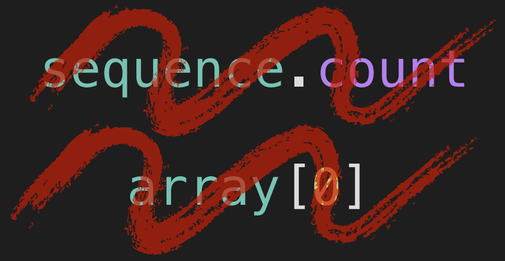
Sequence

Potential Limitations

Repeated consumption
not guaranteed



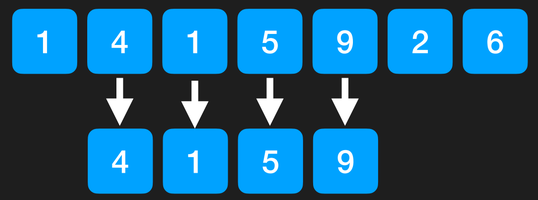
All (non-prefix)
subsequence operations
will produce Array copies



Count & elements
cannot be queried
without iterating

Collection

Differences from Sequence



Efficient SubSequences

Using storage and indices of base collection

```
sequence.count

array[0]
```

Count can be queried & elements can be accessed by index

Repeated consumption guaranteed

Collection

```
protocol Collection<Element>: Sequence {
    associatedtype Element
    associatedtype Iterator: IteratorProtocol = IndexingIterator<Self>

    associatedtype Index: Comparable
    associatedtype Indices: Collection = DefaultIndices<Self>
    associatedtype SubSequence: Collection = Slice<Self>
}
```

Collection

```
protocol Collection<Element>: Sequence {
    subscript(position: Self.Index) -> Self.Element { get }
    Required

    var startIndex: Self.Index { get }
    Required

    var endIndex: Self.Index { get }   // The 'past the end' index.
    Required

    func index(after i: Self.Index) -> Self.Index
    Required

    func formIndex(after i: inout Self.Index)
    Required Default implementation provided.
}
```

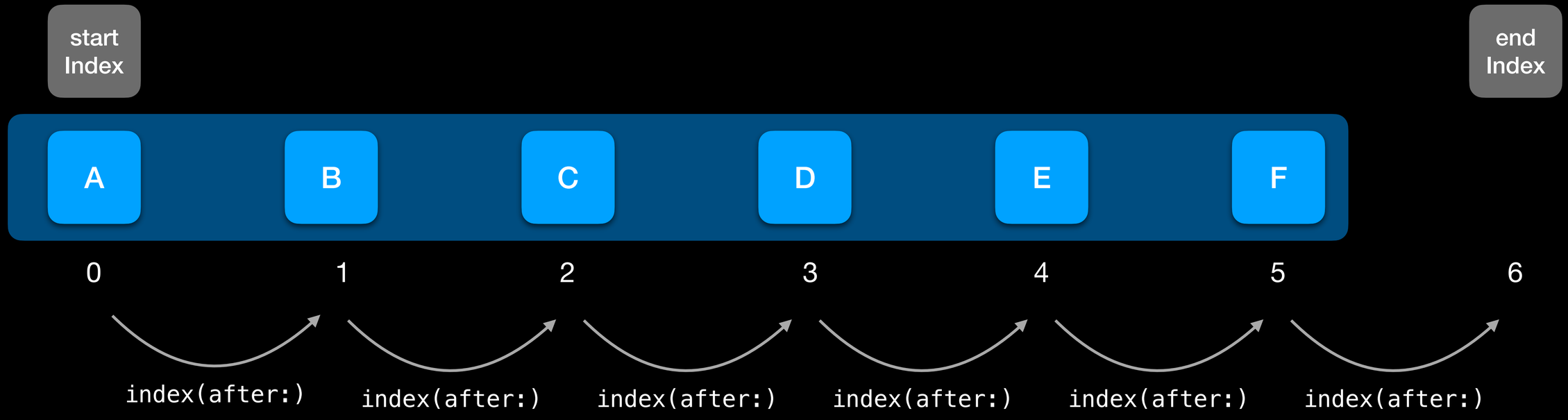
AlphabetSequence

Conforming to Collection

```
extension AlphabetSequence: Collection {
    typealias Element = Character
    typealias Index = Int

    var startIndex: Int { 0 }
    var endIndex: Int { 26 }

    subscript(position: Int) -> Character {
        Character(UnicodeScalar(position + 97)!)
    }

    func index(after i: Int) -> Int {
        assert(i < endIndex, "Index out of range")
        return i + 1
    }
}
```

Index

index(after:)



Index
Dictionary & Set

```
struct Dictionary<Key, Value> {}

extension Dictionary: Collection {
    typealias Element = ??
    typealias Index = ??
}

var dictionary = ["a": 1, "b": 2, "c": 3]
let firstElement = dictionary["a"]
let secondElement = dictionary["b"]
dictionary["d"] = 4

for (key, value) in dictionary { ... }
```

```
struct Set<Element> {}

extension Set: Collection {
    typealias Element = Element
    typealias Index = ??
}

var set: Set = ["a", "b", "c", "d"]
let hasElement = set.contains("a")
set.insert("e")
set.remove("d")

for element in set { ... }
```

Index
Dictionary & Set

```
struct Dictionary<Key, Value> {}

extension Dictionary: Collection {
    typealias Element = (key: Key, value: Value)
    typealias Index = Dictionary.Index
}

var dictionary = ["a": 1, "b": 2, "c": 3]
let firstElement = dictionary["a"]
let secondElement = dictionary["b"]
dictionary["d"] = 4

for (key, value) in dictionary { ... }
```

```
struct Set<Element> {}

extension Set: Collection {
    typealias Element = Element
    typealias Index = Set.Index
}

var set: Set = ["a", "b", "c", "d"]
let hasElement = set.contains("a")
set.insert("e")
set.remove("d")

for element in set { ... }
```

Index
Dictionary & Set

```
struct Dictionary<Key, Value> {}

extension Dictionary: Collection {
    typealias Element = (key: Key, value: Value)
    typealias Index = Dictionary.Index
}

extension Dictionary {
    public struct Index {
        internal enum _Variant {
            case native(_HashTable.Index)
            case cocoa(__CocoaDictionary.Index)
        }
        internal var _variant: _Variant
    }
}
```

```
struct Set<Element> {}

extension Set: Collection {
    typealias Element = Element
    typealias Index = Set.Index
}

extension Set {
    public struct Index {
        internal enum _Variant {
            case native(_HashTable.Index)
            case cocoa(__CocoaSet.Index)
        }
        internal var _variant: _Variant
    }
}
```

Collection

```
protocol Collection<Element>: Sequence {
    var count: Int { get }
    Required Default implementation provided.

    var isEmpty: Bool { get }
    Required Default implementation provided.

    var first: Self.Element? { get }
    Default implementation provided.
}
```

O(n)

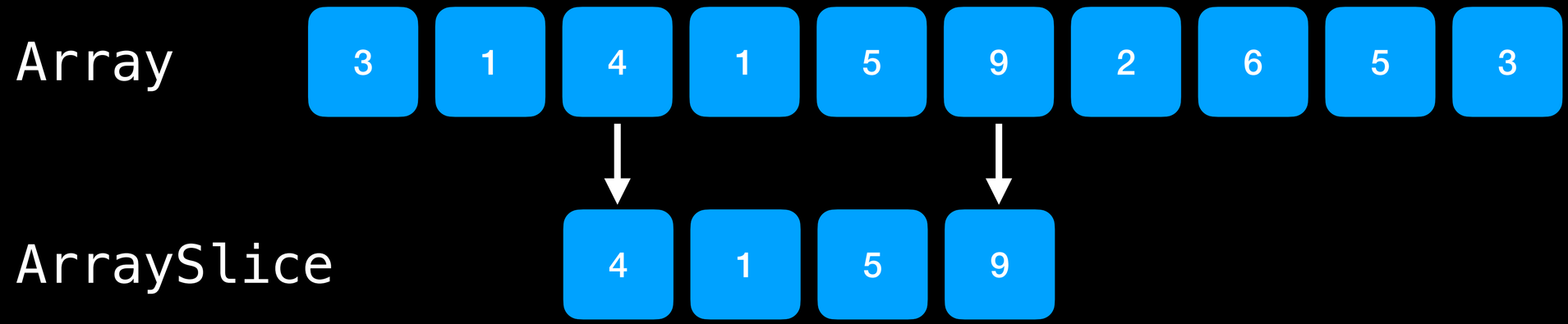
O(1)

O(1)

Subsequence

Represents a portion of a Collection

```
extension Collection {
    subscript(bounds: Range<Self.Index>) -> Self.SubSequence { get }
}

let array = [3,1,4,1,5,9,2,6,5,3]
let slice = array[2...5]
```



Understanding the indices of a subsequence

Using Array & ArraySlice

```
let array = ["A", "B", "C", "D", "E"]
print(array[0])     // prints "A"
print(array.first!) // prints "A"
print(array[1])     // prints "B"
print(array[2])     // prints "C"
```

Understanding the indices of a subsequence

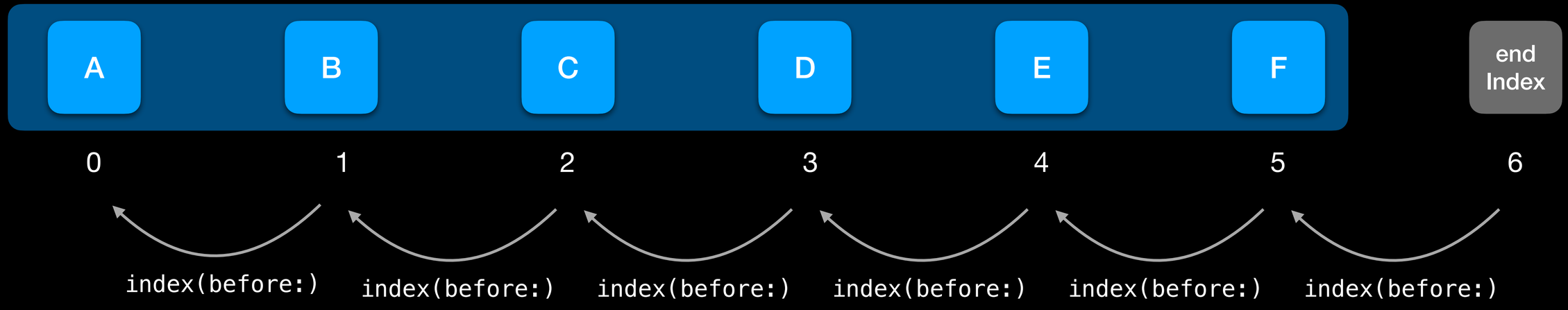
Using Array & ArraySlice

```
let array = ["A", "B", "C", "D", "E"]
print(array[0])     // prints "A"
print(array.first!) // prints "A"
print(array[1])     // prints "B"
print(array[2])     // prints "C"

let sub = array[1..3]
print(sub.first!)  // prints "B"
print(sub[1])      // prints "B"
print(sub[2])      // prints "C"
print(sub[3])      // prints "D"
print(sub[0])
```

Thread 1: Fatal error: Index out of bounds

BidirectionalCollection

```
protocol BidirectionalCollection<Element>: Collection {
    func index(before i: Self.Index) -> Self.Index
    func formIndex(before i: inout Self.Index)
}
```



.last

```
var last: Self.Element? { get }
```

O(1)

```
extension BidirectionalCollection {
    var last: Self.Element? {
        return isEmpty ? nil : self[index(before: endIndex)]
    }
}
```

.reversed()

Sequence / Collection

```
func reversed() -> [Self.Element]
```

O(n)

BidirectionalCollection

```
func reversed() -> ReversedCollection<Self>
```

O(1)

RandomAccessCollection

```
protocol RandomAccessCollection<Element>: BidirectionalCollection { }
```

Sequence / Collection / BidirectionalCollection

```
func distance(from start: Self.Index, to end: Self.Index) -> Int
func index(_ i: Self.Index, offsetBy distance: Int) -> Self.Index
var count: Int { get }
```

O(n)

RandomAccessCollection

```
func distance(from start: Self.Index, to end: Self.Index) -> Int
func index(_ i: Self.Index, offsetBy distance: Int) -> Self.Index
var count: Int { get }
```

O(1)

UnfoldSequence

Sequences from closures

UnfoldSequence

Custom sequences from closures

```
func sequence<T, State>(state: State,
                        next: @escaping (inout State) -> T?) ->
    UnfoldSequence<T, State>



func sequence<T>(first: T,
                 next: @escaping (T) -> T?) ->
    UnfoldFirstSequence<T>
```

Fibonacci

Custom Sequence & Iterator

```
public struct Fibonacci: Sequence {
    typealias Element = Int

    struct Iterator: IteratorProtocol {
        private var state = (0,1)
        mutating func next() -> Element? {
            defer { state = (state.1, state.0 + state.1) }
            return state.0
        }
    }

    func makeIterator() -> Iterator {
        Iterator()
    }
}
```

Fibonacci

Unfold Sequence

```
sequence(state: (0, 1)) { (state) -> Int? in
    defer { state = (state.1, state.0 + state.1) }
    return state.0
}
```

UnfoldFirstSequence

Sequences from closures

```
var element = 1
while (true) {
    print(element)
    element *= 2
}
```

```
for element in sequence(first: 1, next: { $0 * 2 }) {
   // 1, 2, 4, 8, 16, 32, ...
}
```

UnfoldFirstSequence

Sequences from closures

```
var view: UIView
while (view = view.superview) {
    // do something with the superview
}
```

```
for superview in sequence(first: view, next: { $0.superview }) {
    // do something with the superview
}
```

EmptyCollection & CollectionOfOne

Specialised Fixed-length Collections

```
// always empty and 'immutable'
let empty = EmptyCollection<Int>()

// only ever contains one element but mutable
let single = CollectionOfOne(13)
```

EmptyCollection

```
struct EmptyCollection<Element>: RandomAccessCollection, MutableCollection {
    typealias Index = Int
    typealias Indices = Range<Int>
    typealias SubSequence = EmptyCollection<Element>

    @inlinable var startIndex: Index { 0 }
    @inlinable var endIndex: Index { 0 }
    @inlinable var count: Int { 0 }

    @inlinable func index(after i: Index) -> Index {
        fatalError("EmptyCollection can't advance indices")
    }

    @inlinable subscript(position: Index) -> Element {
        get { fatalError("Index out of range") }
        set { fatalError("Index out of range") }
    }

    struct Iterator: IteratorProtocol {
        @inlinable mutating func next() -> Element? { nil }
    }

    @inlinable func makeIterator() -> Iterator {
        Iterator()
    }
}
```

CollectionOfOne

```
struct CollectionOfOne<Element>: RandomAccessCollection, MutableCollection {
    typealias Index = Int
    typealias Indices = Range<Int>
    typealias SubSequence = Slice<CollectionOfOne<Element>>

    @usableFromInline var _element: Element

    @inlinable var startIndex: Index { 0 }
    @inlinable var endIndex: Index { 1 }
    @inlinable var count: Int { 1 }

    @inlinable func index(after i: Index) -> Index {
        assert(i == startIndex)
        return 1
    }

    @inlinable func index(before i: Index) -> Index {
        assert(i == endIndex)
        return 0
    }

    @inlinable subscript(position: Int) -> Element {
        get {
            assert(position == 0, "Index out of range")
            return _element
        }
        set {
            assert(position == 0, "Index out of range")
            _element = newValue
        }
    }
}
```

```
extension CollectionOfOne {
    public struct Iterator {
        @usableFromInline var _elements: Element?

        @inlinable mutating func next() -> Element? {
            let result = _elements
            _elements = nil
            return result
        }
    }

    @inlinable func makeIterator() -> Iterator {
        Iterator(_elements: _element)
    }
}
```

Consuming Sequences

```
func sumOfPrimes(_ input: [Int]) -> Int {
    input.lazy.filter { $0.isPrime }.sum()
}

let sequence = [0,1,2,3,4,5]
sumOfPrimes(sequence)
sumOfPrimes(sequence.reversed())
```

Contrived Example

```swift
func sumOfPrimes(_ input: [Int]) -> Int {
    input.lazy.filter { $0.isPrime }.sum()
}

let sequence = [0,1,2,3,4,5]
sumOfPrimes(sequence)
sumOfPrimes(sequence.reversed())

let reversed = sequence.reversed()
sumOfPrimes(reversed)
```

Cannot convert value of type 'ReversedCollection<[Int]>' to expected argument type '[Int]'

```swift
func sumOfPrimes(_ input: [Int]) -> Int {
    input.lazy.filter { $0.isPrime }.sum()
}

let sequence = [0,1,2,3,4,5]
sumOfPrimes(sequence)
sumOfPrimes(sequence.reversed())

let reversed = sequence.reversed()
sumOfPrimes(reversed)

sumOfPrimes(sequence.prefix(upTo: 4))
```

Cannot convert value of type 'ReversedCollection<[Int]>' to expected argument type '[Int]'

Cannot convert value of type 'Array<Int>.SubSequence' (aka 'ArraySlice<Int>') to expected argument type '[Int]'

```swift
func sumOfPrimes(_ input: [Int]) -> Int {
    input.lazy.filter { $0.isPrime }.sum()
}

let sequence = [0,1,2,3,4,5]

let reversed = sequence.reversed()
sumOfPrimes(reversed)

sumOfPrimes(sequence.prefix(upTo: 4))
```

Cannot convert value of type 'ReversedCollection<[Int]>' to expected argument type '[Int]'

Cannot convert value of type 'Array<Int>.SubSequence' (aka 'ArraySlice<Int>') to expected argument type '[Int]'

```swift
func sumOfPrimes(_ input: [Int]) -> Int {
    input.lazy.filter { $0.isPrime }.sum()
}

let sequence = [0,1,2,3,4,5]

let reversed = sequence.reversed()
sumOfPrimes(Array(reversed))

sumOfPrimes(Array(sequence.prefix(upTo: 4)))
```

```swift
func sumOfPrimes(_ input: [Int]) -> Int {
    input.lazy.filter { $0.isPrime }.sum()
}

let sequence = [0,1,2,3,4,5]

let reversed = sequence.reversed()
sumOfPrimes(reversed)

sumOfPrimes(sequence.prefix(upTo: 4))
```

Cannot convert value of type 'ReversedCollection<[Int]>' to expected argument type '[Int]'

Cannot convert value of type 'Array<Int>.SubSequence' (aka 'ArraySlice<Int>') to expected argument type '[Int]'

```swift
func sumOfPrimes<S: Sequence<Int>>(_ input: S) -> Int {
    input.lazy.filter { $0.isPrime }.sum()
}

let sequence = [0,1,2,3,4,5]

let reversed = sequence.reversed()
sumOfPrimes(reversed)

sumOfPrimes(sequence.prefix(upTo: 4))
```

Cannot convert value of type 'ReversedCollection<[Int]>' to expected argument type '[Int]'

Cannot convert value of type 'Array<Int>.SubSequence' (aka 'ArraySlice<Int>') to expected argument type '[Int]'

```swift
func sumOfPrimes<S: Sequence<Int>>(_ input: S) -> Int {
    input.lazy.filter { $0.isPrime }.sum()
}

let sequence = [0,1,2,3,4,5]

let reversed = sequence.reversed()
sumOfPrimes(reversed)

sumOfPrimes(sequence.prefix(upTo: 4))

sumOfPrimes(EmptyCollection<Int>())
sumOfPrimes(CollectionOfOne(13))
sumOfPrimes(0...5)
```

adjacentPairs() / chain(_:_:) / chunked(by:) / chunked(on:) / chunks(ofCount:)
combinations() / combinations(ofCount:) / compacted() / cycled() / cycled(times:)
firstNonNil(_:) / grouped(by:) / indexed() / interspersed(with:) / joined(by:) / keyed(by:)
keyed(by:resolvingConflictsBy:) / max(count:) / max(count:sortedBy:) / min(count:)
min(count:sortedBy:) / minAndMax() / minAndMax(by:) / partition(by:)
partitioningIndex(where:) / permutations() / permutations(ofCount:) / pro
reductions(into:_:) / rotate(toStartAt:) / rotate(subrange:toStartAt:)
split(whereSeparator:) / split(separator:) / stablePartition(by:)
stablePartition(subrange:by:) / striding(by:) / suffix(while:) / trimming(while:)
trimmingPrefix(while:) / trimmingSuffix(while:) / uniquePermutations()
uniquePermutations(ofCount:) / uniqued() / uniqued(on:) / windows(ofCount:)

Swift Algorithms Package

github.com/apple/swift-algorithms

adjacentPairs() / chain(_:_:) / chunked(by:) / chunked(on:) / chunks(ofCount:)
combinations() / combinations(ofCount:) / compacted() / cycled() / cycled(times:)
firstNonNil(_:) / grouped(by:) / indexed() / interspersed(with:) / joined(by:) / keyed(by:)
keyed(by:resolvingConflictsBy:) / max(count:) / max(count:sortedBy:) / min(count:)
min(count:sortedBy:) / minAndMax() / minAndMax(by:) / partition(by:)
partitioningIndex(where:) / permutations() / permutations(ofCount:) / product(_:_:)
randomSample(count:) / randomSample(count:using:) / randomStableSample(count:)
randomStableSample(count:using:) / reductions(_:) / reductions(_:_:)
reductions(into:_:) / rotate(toStartAt:) / rotate(subrange:toStartAt:)
split(whereSeparator:) / split(separator:) / stablePartition(by:)
stablePartition(subrange:by:) / striding(by:) / suffix(while:) / trimming(while:)
trimmingPrefix(while:) / trimmingSuffix(while:) / uniquePermutations()
uniquePermutations(ofCount:) / uniqued() / uniqued(on:) / windows(ofCount:)

adjacentPairs() / chain(_:_:) / chunked(by:) / chunked(on:) / chunks(ofCount:)
combinations() / combinations(ofCount:) / compacted() / cycled() / cycled(times:)
firstNonNil(_:) / grouped(by:) / indexed() / interspersed(with:) / joined(by:) / keyed(by:)
keyed(by:resolvingConflictsBy:) / max(count:) / max(count:sortedBy:) / min(count:)
min(count:sortedBy:) / minAndMax() / minAndMax(by:) / partition(by:)
partitioningIndex(where:) / permutations() / permutations(ofCount:) / product(_:_:)
randomSample(count:) / randomSample(count:using:) / randomStableSample(count:)
randomStableSample(count:using:) / reductions(_:) / reductions(_:_:)
reductions(into:_:) / rotate(toStartAt:) / rotate(subrange:toStartAt:)
split(whereSeparator:) / split(separator:) / stablePartition(by:)
stablePartition(subrange:by:) / striding(by:) / suffix(while:) / trimming(while:)
trimmingPrefix(while:) / trimmingSuffix(while:) / uniquePermutations()
uniquePermutations(ofCount:) / uniqued() / uniqued(on:) / windows(ofCount:)

Chain

```
for element in arrayOfElements + [anotherElement] {

}
```

Chain

```
for element in arrayOfElements + [anotherElement] {

}

for element in chain(arrayOfElements, CollectionOfOne(anotherElement)) {

}
```

Windowed

```
let windowsOfRange = (0...9).windows(ofCount: 3)
let windowsOfArray = [0,1,2,3,4,5,6,7,8,9].windows(ofCount: 3)
```

Windowed

```
let windowsOfRange = (0...9).windows(ofCount: 3)
let windowsOfArray = [0,1,2,3,4,5,6,7,8,9].windows(ofCount: 3)

type(of: windowsOfRange.first!)     // Slice<ClosedRange<Int>>
type(of: windowsOfArray.first!)     // ArraySlice<Int>
```

Subsequence

Windowed

```
let windows = [0,1,2,3,4,5,6,7,8,9].windows(ofCount: 3)

for window in windows {
    // window of three elements
}
```

Windowed

```
let windows = [0,1,2,3,4,5,6,7,8,9].windows(ofCount: 3)

for window in windows {
    // window of three elements
}
```

Windowed

```
let windows = [0,1,2,3,4,5,6,7,8,9].windows(ofCount: 3)

for window in windows {
    // window of three elements
}
```

Windowed

```
let windows = [0,1,2,3,4,5,6,7,8,9].windows(ofCount: 3)

for window in windows {
    // window of three elements
}
```

Windowed

```
let windows = [0,1,2,3,4,5,6,7,8,9].windows(ofCount: 3)

for window in windows {
    // window of three elements
}
```

Windowed

```
let windows = [0,1,2,3,4,5,6,7,8,9].windows(ofCount: 3)

for window in windows {
    // window of three elements
}
```

Windowed

```
let windows = [0,1,2,3,4,5,6,7,8,9].windows(ofCount: 3)

for window in windows {
    // window of three elements
}
```

Windowed

```
let windows = [0,1,2,3,4,5,6,7,8,9].windows(ofCount: 3)

for window in windows {
    // window of three elements
}
```

AdjacentPairs

```
let windowsOfRange = (0...9).adjacentPairs()
let windowsOfArray = [0,1,2,3,4,5,6,7,8,9].adjacentPairs()

for (first, second) in windowsOfArray {
    // first = 0
    // second = 1
}
```

AdjacentPairs

```
let windowsOfRange = (0...9).adjacentPairs()
let windowsOfArray = [0,1,2,3,4,5,6,7,8,9].adjacentPairs()

for (first, second) in windowsOfArray {
    // first = 1
    // second = 2
}
```

AdjacentPairs

```
let windowsOfRange = (0...9).adjacentPairs()
let windowsOfArray = [0,1,2,3,4,5,6,7,8,9].adjacentPairs()

for (first, second) in windowsOfArray {
    // first = 2
    // second = 3
}
```

IndexedCollection

Like `.enumerated()`, but with the real index

```
for (index, value) in ["a", "b", "c", "d", "e"].indexed() {
    print([index, value])
}
// prints [0, "a"], [1, "b"], [2, "c"], [3, "d"], [4, "e"]


for (index, value) in ["a", "b", "c", "d", "e"][1...3].indexed() {
    print([index, value])
}
// prints [1, "b"], [2, "c"], [3, "d"]
```

Striding Dates

stride

```
func stride<T>(from start: T, to end: T, by stride: T.Stride) ->
    StrideTo<T> where T : Strideable

func stride<T>(from start: T, through end: T, by stride: T.Stride) ->
    StrideThrough<T> where T : Strideable

for i in stride(from: 0.0, to: 100.0, by: 1.0) { ... }
for i in stride(from: 100, to: 0, by: -1) { ... }
for i in stride(from: 0, to: 100, by: 2) { ... }
for i in stride(from: 3.14, to: 314.15, by: 6.28) { ... }
```

stride

```
let start = Date(timeIntervalSince1970: 1713655782)
let end   = Date(timeIntervalSince1970: 1713656082)
for dates in stride(from: start, through: end, by: 60) { ... }
```

Strides by seconds

strideDate

```
let start = Date(timeIntervalSince1970: 1713655782)
let end   = Date(timeIntervalSince1970: 1714174182)


let dates = strideDate(from: start, to: end, by: .days(1))
let dates = strideDate(from: end, to: start, by: .days(-1))
let dates = strideDate(from: end, through: start, by: .weeks(1))
let dates = strideDate(from: end, through: start, by: .weeks(-1))
let dates = strideDate(from: end, to: start, by: .months(1))
let dates = strideDate(from: end, through: start, by: .years(1))

let dates = strideDate(from: start, to: end, by: .days(1))
            .filter { !calendar.isDateInWeekend($0) }
            .map { calendar.startOfDay(for: $0) }
```

strideDate

```
public func strideDate(from: Date, to: Date,
                       by: DateComponents,
                       calendar: Calendar = .current) -> DateStrideSequence {
    DateStrideSequence(_start: from, _end: to, stride: by, calendar: calendar)
}

public func strideDate(from: Date, through: Date,
                       by: DateComponents,
                       calendar: Calendar = .current) -> DateStrideThroughSequence {
    DateStrideThroughSequence(_start: from, _end: through, stride: by, calendar: calendar)
}
```

strideDate

```
public struct DateStrideThroughSequence: Sequence {
    ...
    public struct Iterator: IteratorProtocol {
        var start: Date
        let end: Date
        let stride: DateComponents
        let calendar: Calendar
        let ascending: Bool

        public mutating func next() -> Date? {
            if ascending {
                guard start <= end else { return nil }
            } else {
                guard start >= end else { return nil }
            }
            defer { start = calendar.date(byAdding: stride, to: start)! }
            return start
        }
    }

    public func makeIterator() -> Iterator {
        let isAscending = start < calendar.date(byAdding: stride, to: start)!
        return Iterator(start: start, end: end, stride: stride, calendar: calendar, ascending: isAscending())
    }
}
```

Contrived Example

Reading lines from a file

Synchronously

Contrived Example

Loading from a file
'StackOverflow'

```
let url = URL(filePath: "/usr/share/dict/words")
let lines = try String(contentsOf: url, encoding: .utf8)
                .components(separatedBy: .newlines)

// lines = Array<String>
```

FileReader

Read file by line, using UnfoldSequence

```
func readLines(ofFile fileURL: URL) -> UnfoldSequence<String, UnsafeMutablePointer<FILE>>
```

FileReader

Read file by line, using UnfoldSequence

```
func readLines(ofFile fileURL: URL) -> UnfoldSequence<String, UnsafeMutablePointer<FILE>> {
    sequence(state: fopen(fileURL.path, "r")) { (filePtr) -> String? in
        var linePtr: UnsafeMutablePointer<CChar>?
        var linecap: Int = 0
        if getline(&linePtr, &linecap, filePtr) > 0, let linePtr {
            defer { free(linePtr) }
            return String(cString: linePtr).trimmingCharacters(in: .newlines)
        } else {
            fclose(filePtr)
            return nil
        }
    }
}
```

FileReader

Read file by line, using UnfoldSequence

```
func readLines(ofFile fileURL: URL) -> UnfoldSequence<String, UnsafeMutablePointer<FILE>> {
    sequence(state: fopen(fileURL.path, "r")) { (filePtr) -> String? in
        var linePtr: UnsafeMutablePointer<CChar>?
        var linecap: Int = 0
        if getline(&linePtr, &linecap, filePtr) > 0, let linePtr {
            defer { free(linePtr) }
            return String(cString: linePtr).trimmingCharacters(in: .newlines)
        } else {
            fclose(filePtr)
            return nil
        }
    }
}
```

FileReader

Read file by line, using UnfoldSequence

```
func readLines(ofFile fileURL: URL) -> UnfoldSequence<String, UnsafeMutablePointer<FILE>> {
    sequence(state: fopen(fileURL.path, "r")) { (filePtr) -> String? in
        var linePtr: UnsafeMutablePointer<CChar>?
        var linecap: Int = 0
        if getline(&linePtr, &linecap, filePtr) > 0, let linePtr {
            defer { free(linePtr) }
            return String(cString: linePtr).trimmingCharacters(in: .newlines)
        } else {
            fclose(filePtr)
            return nil
        }
    }
}
```

FileReader

Read file by line, using UnfoldSequence

```
func readLines(ofFile fileURL: URL) -> UnfoldSequence<String, UnsafeMutablePointer<FILE>> {
    sequence(state: fopen(fileURL.path, "r")) { (filePtr) -> String? in
        var linePtr: UnsafeMutablePointer<CChar>?
        var linecap: Int = 0
        if getline(&linePtr, &linecap, filePtr) > 0, let linePtr {
            defer { free(linePtr) }
            return String(cString: linePtr).trimmingCharacters(in: .newlines)
        } else {
            fclose(filePtr)
            return nil
        }
    }
}
```

FileReader

Read file by line, using UnfoldSequence

```
let url = URL(filePath: "/usr/share/dict/words")
for line in readLines(ofFile: url) {
    print(line)
}
```

FileReader

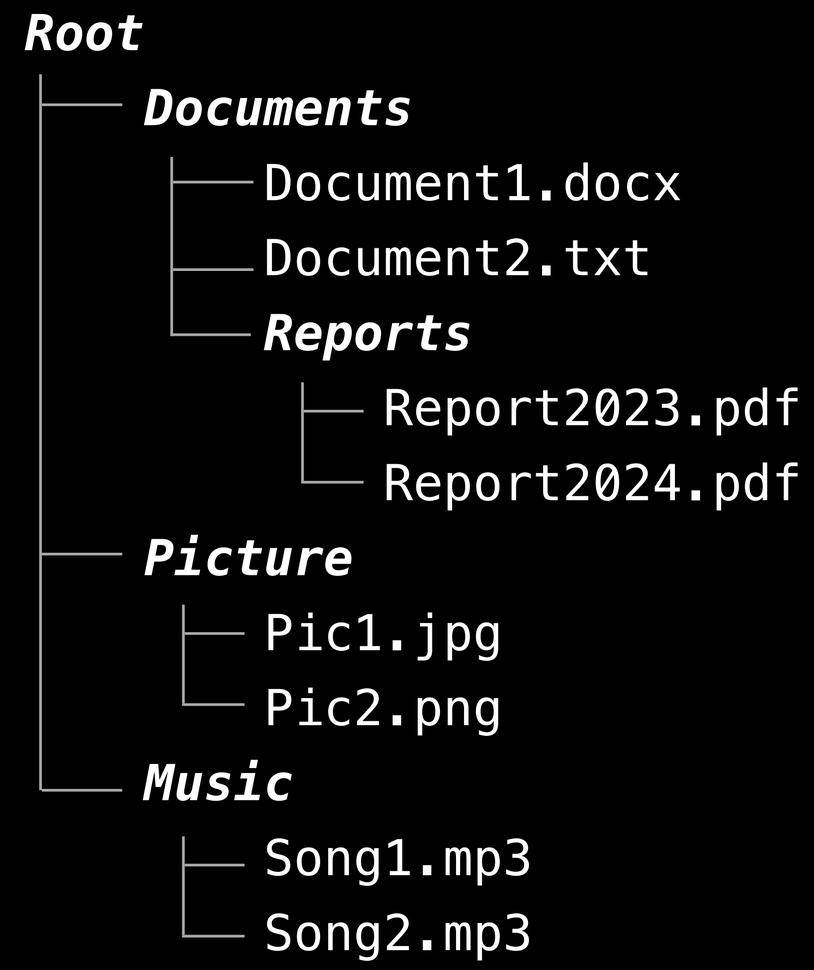
Read file by line, using UnfoldSequence

```
let url = URL(filePath: "/usr/share/dict/words")
readLines(ofFile: url)
    .lazy
    .filter { $0.hasPrefix("ade") }
    .map { $0.capitalized }
    .forEach { print($0) }
```

Recursive

Recursive Example

File System



Recursive Example

File System

```
indirect enum FileSystemItem {
    case file(name: String)
    case folder(name: String, children: [FileSystemItem])

    var children: [FileSystemItem] {
        switch self {
        case .file(_):
            return []
        case .folder(_, let children):
            return children
        }
    }
}
```

Recursive Example

File System

```
let fileDirectory =
FileSystemItem.folder(name: "root", children :[
        FileSystemItem.folder(name: "Documents", children :[
                FileSystemItem.file(name: "Document1.docx"),
                FileSystemItem.file(name: "Document2.txt"),
                FileSystemItem.folder(name: "Reports", children :[
                        FileSystemItem.file(name: "Report2023.pdf"),
                        FileSystemItem.file(name: "Report2024.pdf"),
                ]),
        ]),
        FileSystemItem.folder(name: "Picture", children :[
                FileSystemItem.file(name: "Pic1.jpg"),
                FileSystemItem.file(name: "Pic2.png"),
        ]),
        FileSystemItem.folder(name: "Music", children :[
                FileSystemItem.file(name: "Song1.mp3"),
                FileSystemItem.file(name: "Song2.mp3"),
        ]),
])
```

Recursive

Gather Names Recursively (Depth First)

```
func gatherNames(_ item: FileSystemItem) -> [String] {
    var names = [item.name]
    item.children.forEach {
        names.append(contentsOf: gatherNames($0))
    }
    return names
}

let allNames = gatherNames(fileDirectory)
// Array<String> with all names
```

Recursive

Gather Recursively (DFS) - Type extension

```
extension FileSystemItem {
    var gatherNames: [String] {
        [name] + children.flatMap { $0.gatherNames }
    }
}


let allNames = fileDirectory.gatherNames
// Array<String> with all names
```

Recursive Sequence

Recursive

Using a sequence

```
let allNames = RecursiveSequence(element: fileDirectory, keyPath: \.children)
        .map { $0.name }
// Array<String> with all names


let allNames = RecursiveSequence(element: fileDirectory, keyPath: \.children)
        .lazy
        .map { $0.name }
// LazySequence<String> with all names
```

Custom Recursive Sequence

```
struct RecursiveSequence<Base: Sequence, S1: Sequence>: Sequence where S1.Element == Base.Element {
    typealias Element = Base.Element
    let base: Base
    let keyPath: KeyPath<Element, S1>

    func makeIterator() -> RecursiveIterator {
        RecursiveIterator(stack: Array(base.reversed()), keyPath: keyPath)
    }

    struct RecursiveIterator: IteratorProtocol {
        internal var stack: [Element]
        internal var keyPath: KeyPath<Element, S1>

        mutating func next() -> Element? {
            guard let last = stack.popLast() else { return nil }
            let children = last[keyPath: keyPath]
            stack.append(contentsOf: children.reversed())
            return last
        }
    }
}
```

Recursive

Using a sequence

```
extension RecursiveSequence {
    public init<E>(element: E, keyPath: KeyPath<E, S1>) where Base == CollectionOfOne<E> {
        self.init(CollectionOfOne(element), keyPath: keyPath)
    }
}
```

```
let allNames = RecursiveSequence(element: fileDirectory, keyPath: \.children)
        .map { $0.name }
```

```
let allNames = RecursiveSequence(element: fileDirectory, keyPath: \.children)
        .lazy
        .map { $0.name }
```

Recursive

Conforming to Sequence

```
extension FileSystemItem: Sequence {
    public func makeIterator() -> some IteratorProtocol<Self> {
        RecursiveSequence.Iterator(base: CollectionOfOne(self), keyPath: \.children)
    }
}


let allNames = fileDirectory
        .map { $0.name }


let allNames = fileDirectory
        .lazy
        .map { $0.name }
```

Even further

Conform RecursiveSequence to Collection

Recursive

Extending to a Collection

```
extension RecursiveSequence: Collection where Base: Collection, S1: Collection,
                                              Base.Index == Int, S1.Index == Int {
    typealias Index = IndexPath

    var startIndex: IndexPath { IndexPath(index: 0) }
    var endIndex: IndexPath { IndexPath(index: base.count) }

    subscript(position: IndexPath) -> Element {
        assert(!position.isEmpty, "Position indexPath cannot be empty")
        var indices = position.makeIterator()
        let initial = base[indices.next()!]
        return sequence(first: initial) {
            guard let index = indices.next() else { return nil }
            return $0[keyPath: keyPath][index]
        }
        .reduce(initial, { return $1 })
    }
}
```

Recursive

Extending to a Collection

```
extension RecursiveSequence: Collection where Base: Collection, S1: Collection,
                                              S1.Index == Int, Base.Index == Int {

    func index(after i: IndexPath) -> IndexPath {
        assert(!i.isEmpty, "Position indexPath cannot be empty")
        let children = self[i][keyPath: keyPath]
        if !children.isEmpty { return i.appending(0) }
        var currentPath = i
        while let currentIndex = currentPath.popLast(), !currentPath.isEmpty {
            let children = self[currentPath][keyPath: keyPath]
            if currentIndex < (children.count - 1) {
                return currentPath.appending(currentIndex + 1)
            }
        }
        return IndexPath(index: i[0] + 1)
    }
}
```

Recursive

Then to BidirectionalCollection

```
extension RecursiveSequence: BidirectionalCollection where Base: BidirectionalCollection,
                                                           S1: BidirectionalCollection,
                                                           S1.Index == Int, Base.Index == Int {
    func index(before i: IndexPath) -> IndexPath {
        assert(!i.isEmpty, "Position indexPath cannot be empty")
        assert(i != startIndex, "Must be after start index")
        if i.last == 0 { return i.dropLast() }
        var currentPath = i.dropLast().appending(i.last! - 1)
        var parent = self[currentPath]
        while (!parent[keyPath: keyPath].isEmpty) {
            currentPath.append(parent[keyPath: keyPath].count - 1)
            parent = parent[keyPath: keyPath].last!
        }
        return currentPath
    }
}
```

Recursive

But not RandomAccessCollection

- `index(after:)` & `subscript(position:)` are not O(1), they are O(log n)
- It cannot find the count nor distance between two arbitrary element in O(1) time.
  - count is O(n)
  - Distance is O(n log n)

Expected Performance

Types that conform to `Collection` are expected to provide the `startIndex` and `endIndex` properties and subscript access to elements as O(1) operations. Types that are not able to guarantee this performance must document the departure, because many collection operations depend on O(1) subscripting performance for their own performance guarantees.

The performance of some collection operations depends on the type of index that the collection provides. For example, a random-access collection, which can measure the distance between two indices in O(1) time, can calculate its `count` property in O(1) time. Conversely, because a forward or bidirectional collection must traverse the entire collection to count the number of contained elements, accessing its `count` property is an O(*n*) operation.

Recursive

Printing with depth

```
RecursiveSequence(element: fileDirectory, keyPath: \.children)
    .lazy
    .indexed() 📦
    .map { "\( String(repeating: "\t", count: $0.count-1) )\($1.name)" }
    .forEach { print($0) }
```

```
Root
    Documents
        Document1.docx
        Document2.txt
        Reports
            Report2023.pdf
            Report2024.pdf
    Picture
        Pic1.jpg
        Pic2.png
    Music
        Song1.mp3
        Song2.mp3
```

Recursive
Search

```
let index = RecursiveSequence(element: fileSystem, keyPath: \.children)
    .firstIndex(where: { $0.isFile && $0.name.hasSuffix("png") })!

// index = IndexPath([0, 2, 1])
```

```
Root
    Documents
        Document1.docx
        Document2.txt
        Reports
            Report2023.pdf
            Report2024.pdf
    Picture
        Pic1.jpg
        Pic2.png
    Music
        Song1.mp3
        Song2.mp3
```

Recursive

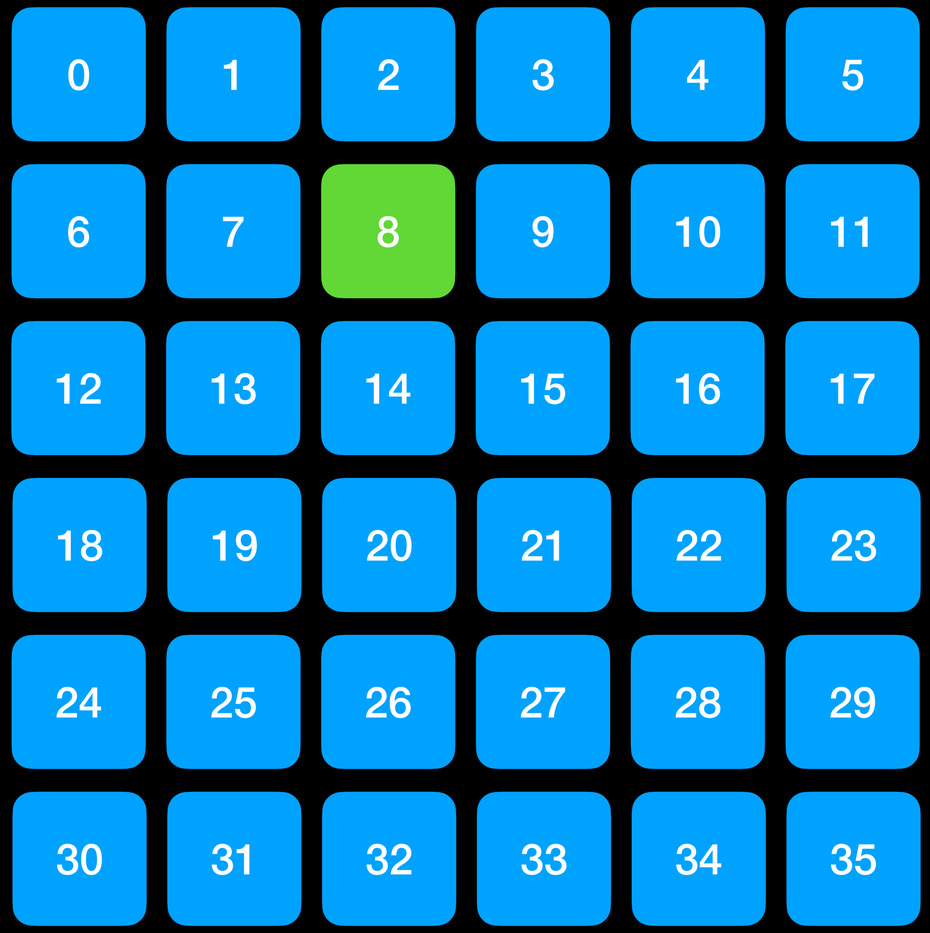
Getting elements along path

```
extension RecursiveSequence where Self: Collection {
    func elements(along path: IndexPath) -> [Element] {
        guard !path.isEmpty else { return [] }
        var indices = path.makeIterator()
        let initial = base[indices.next()!]
        return sequence(first: initial) {
            guard let index = indices.next() else { return nil }
            return $0[keyPath: keyPath][index]
        }.toArray()
    }
}

let pathElements = fileDirectory
        .elements(along: index)
        .map { $0.name }

// pathElements = [ "Root", "Pictures", "Pic2.png" ]
```

Root
- *Documents*
  - Document1.docx
  - Document2.txt
  - *Reports*
    - Report2023.pdf
    - Report2024.pdf
- ***Picture***
  - Pic1.jpg
  - **Pic2.png**
- *Music*
  - Song1.mp3
  - Song2.mp3

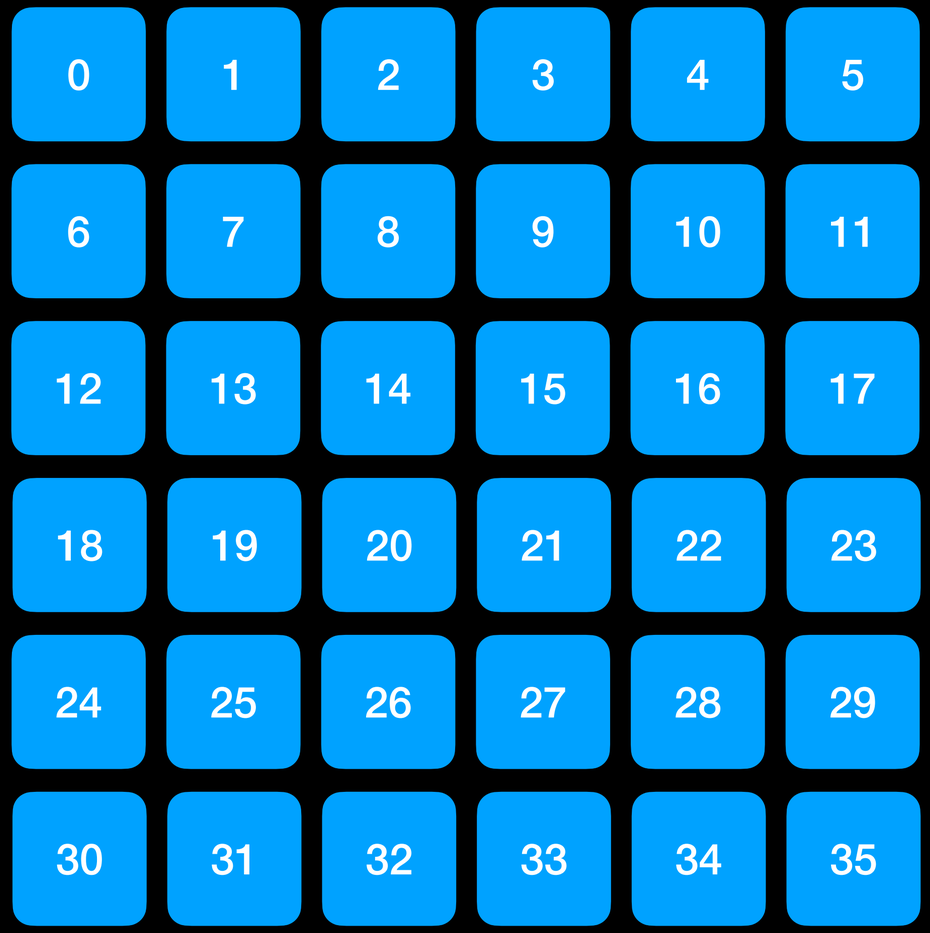
Custom containers

Grid (2D Array)



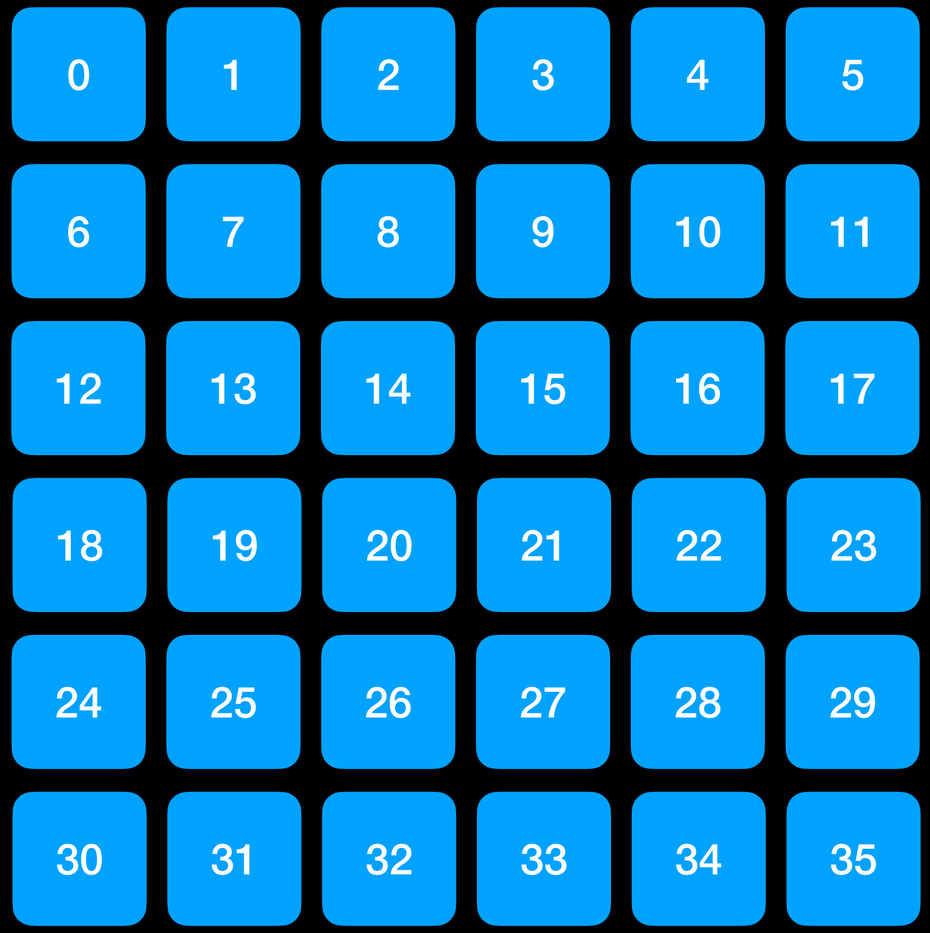

```
let grid = Grid<Int>(0...35, width: 6, height: 6)

let index = Point<Int>(x: 2, y: 1)
let element = grid[index]   // prints 8
```

Grid (2D Array)



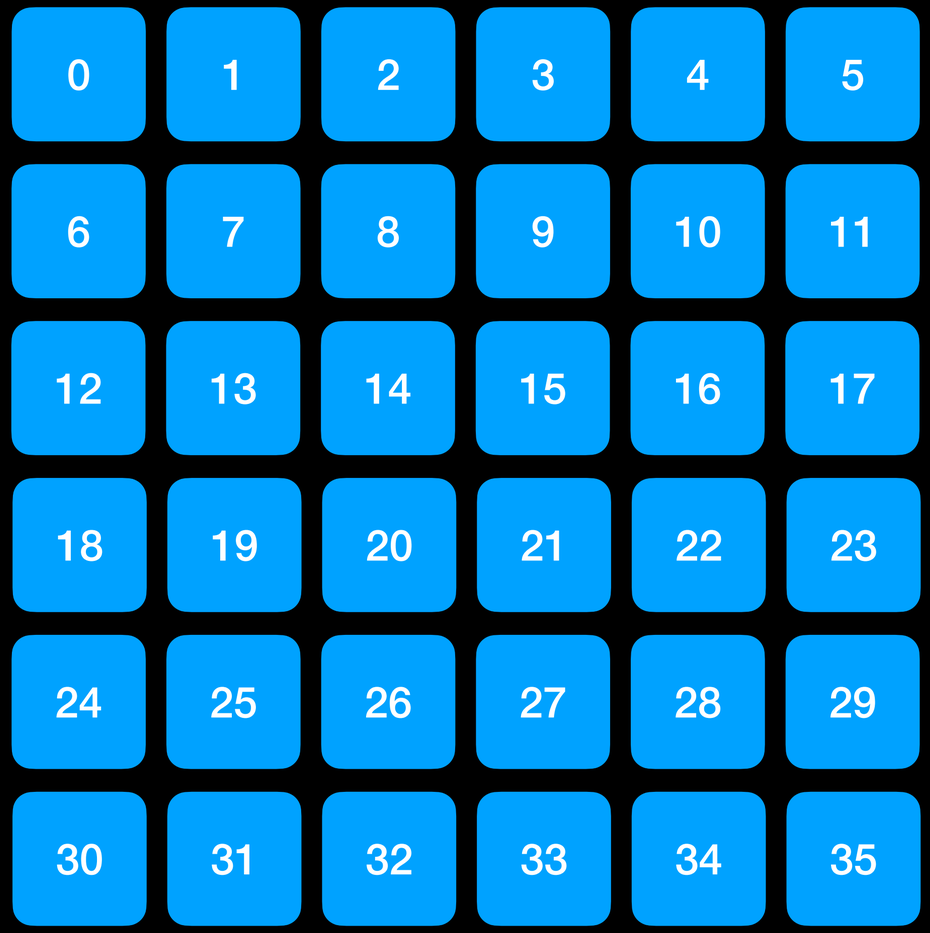

```
struct Grid<Element> {
    typealias Index = Point<Int>

    private var elements: [Element]
    let width: Int
    let height: Int


    @inlinable
    func linearIndex(for index: Index) -> Int {
        index.x + index.y * width
    }


    @inlinable
    func index(from linearIndex: Int) -> Index {
        let y = linearIndex / width
        let x = linearIndex % width
        return .init(x: x, y: y)
    }
}
```

Grid (2D Array)




```
extension Grid: Collection {
    typealias Index = Point<Int>
    typealias Iterator = IndexingIterator<Self>
    typealias Indices = DefaultIndices<Self>
    typealias SubSequence = Slice<Self>

    var count: Int { width * height }
    var isEmpty: Bool { elements.isEmpty }

    var startIndex: Index { .zero }
    var endIndex: Index { index(from: count) }

    subscript(point: Point<Int>) -> Element {
        get { elements[linearIndex(for: point)] }
        set { elements[linearIndex(for: point)] = newValue }
    }

    func index(after i: Index) -> Index {
        let linear = linearIndex(for: i)
        assert(linear < count, "Index out of bounds")
        return index(from: linear + 1)
    }
}
```

Grid (2D Array)

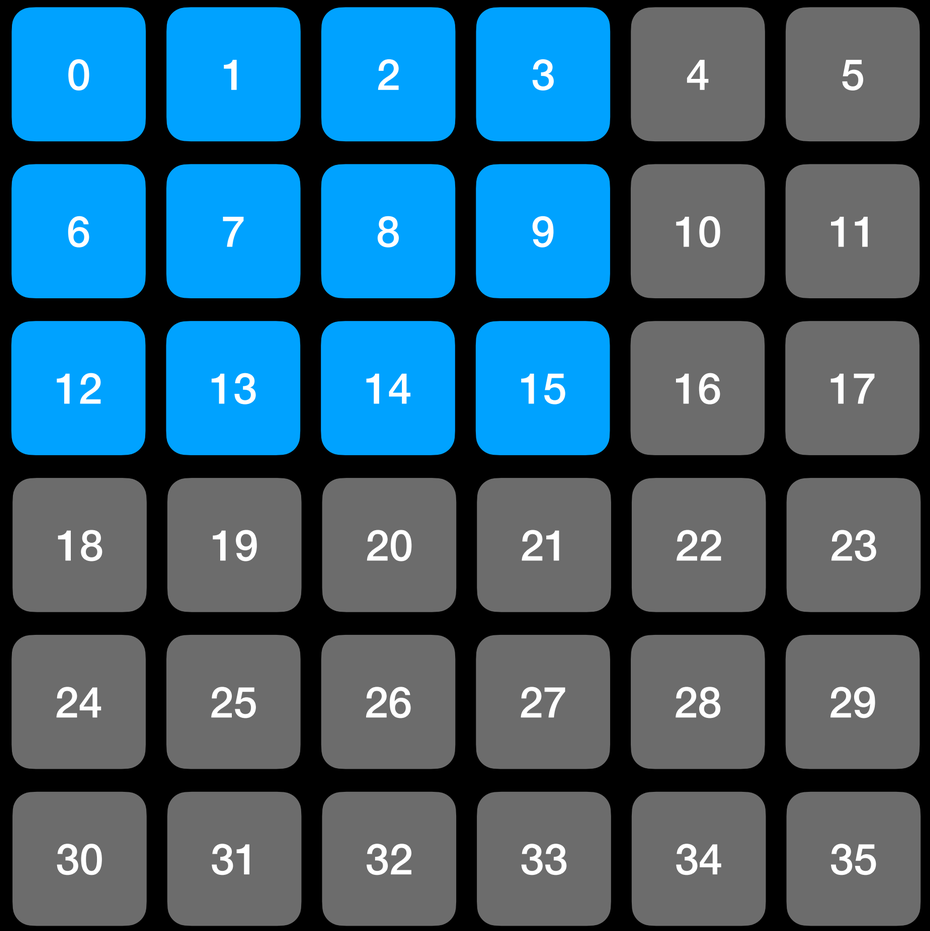



```swift
extension Grid: BidirectionalCollection {
    func index(before i: Index) -> Index {
        assert(i > .zero, "Index out of bounds")
        let linear = linearIndex(for: i)
        return index(from: linear - 1)
    }
}
```

```swift
extension Grid: RandomAccessCollection {

    func distance(from start: Point<Int>, to end: Point<Int>) -> Int {
        return linearIndex(for: end) - linearIndex(for: start)
    }

    func index(_ i: Point<Int>, offsetBy distance: Int) -> Point<Int> {
        let offset = linearIndex(for: i) + distance
        assert(offset < count, "Index out of bounds")
        return index(from: offset)
    }
}
```

Grid (2D Array)

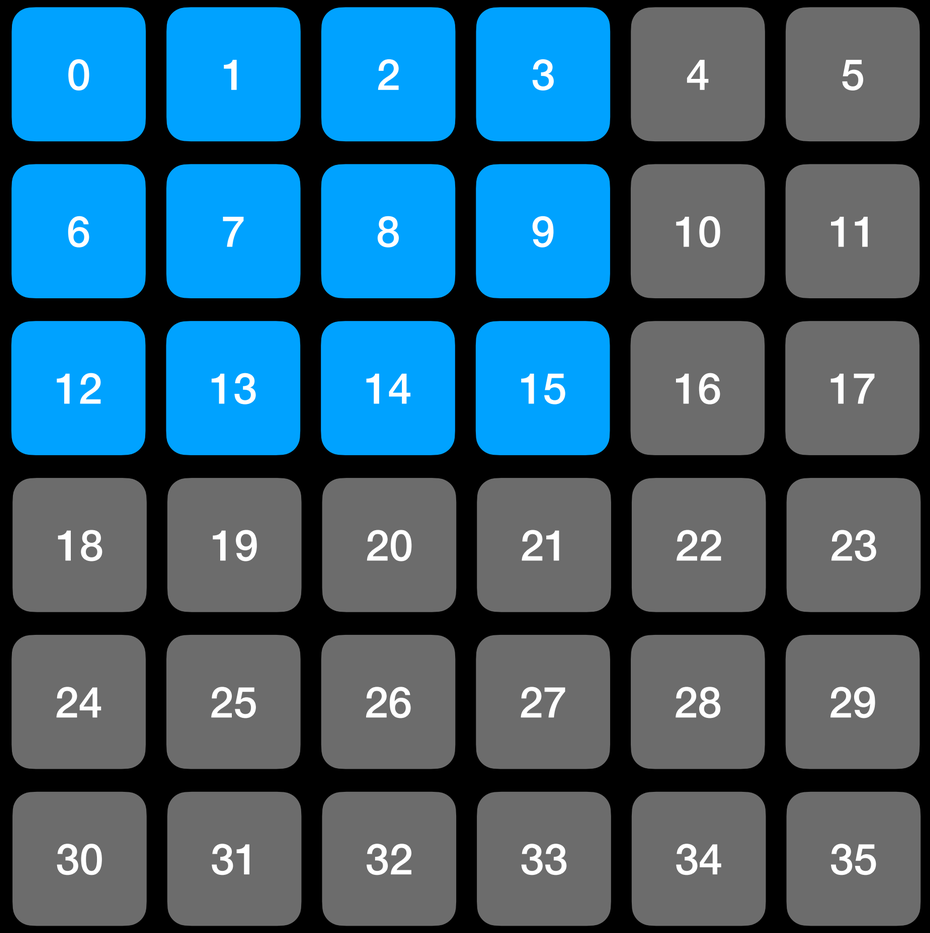
Sub Grid




```
struct GridSubGrid<Element>: Collection {
    typealias Index = Grid<Element>.Index
    let base: Grid<Element>
    let origin: Point<Int>
    let finalIndex: Point<Int>  // origin + (width & height)
    let width: Int
    let height: Int

    var startIndex: Index { origin }
    var endIndex: Index { base.index(after: finalIndex) }
    subscript(position: Index) -> Element { base[position] }

    func index(after: Point<Int>) -> Point<Int> {
        assert(after < endIndex, "Index out of bounds")
        guard after < finalIndex else { return endIndex }
        let x = after.x
        let y = after.y
        let maxX = startIndex.x + (width - 1)
        if x == maxX { return Point(x: startIndex.x, y: y+1) }
        return Point(x: x+1, y: y)
    }
}
```

Grid (2D Array)

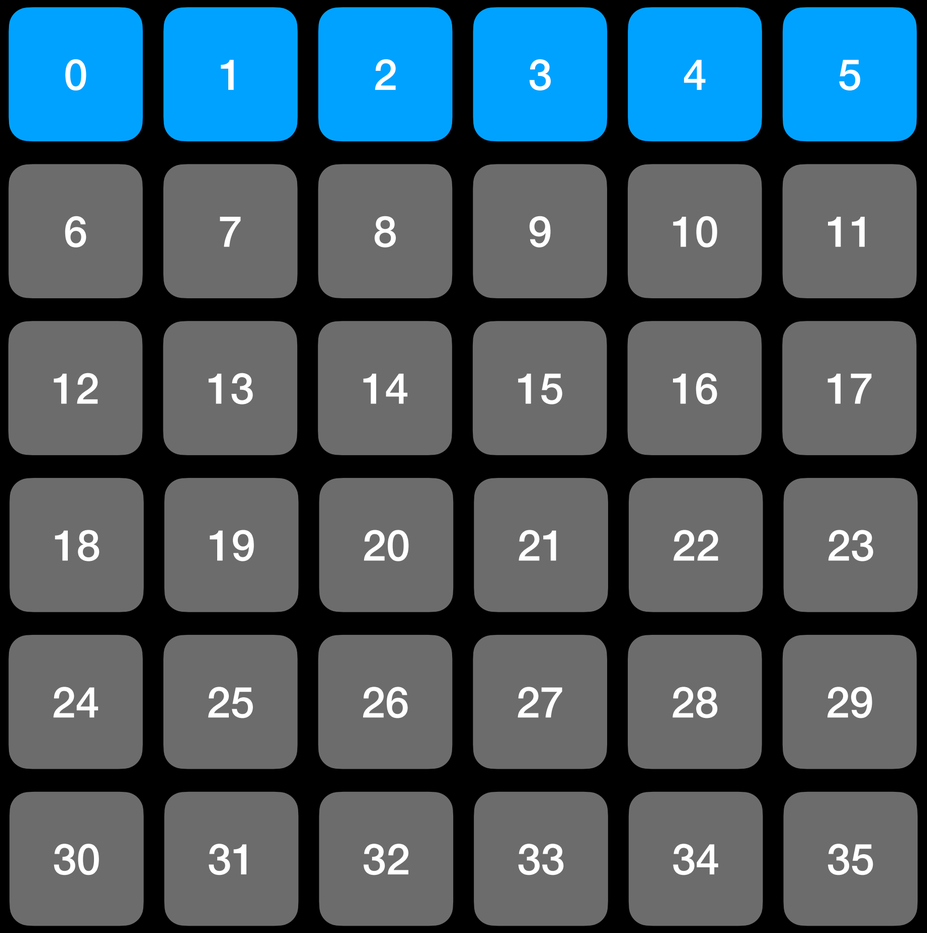
Sub Grid




```
extension Grid {
    func subgrid(
        origin: Index, width: Int, height: Int
    ) -> GridSubGrid<Element> {
        assert(origin.x + width <= self.width, "Too wide")
        assert(origin.y + height <= self.height, "Too high")
        return GridSubSequence(base: self,
                               origin: origin,
                               width: width,
                               height: height)
    }
}

// usage
let grid = Grid<Int>(0...35, width: 6, height: 6)
grid.subgrid(origin: .zero, width: 4, height: 3)
```

Grid (2D Array)

Row

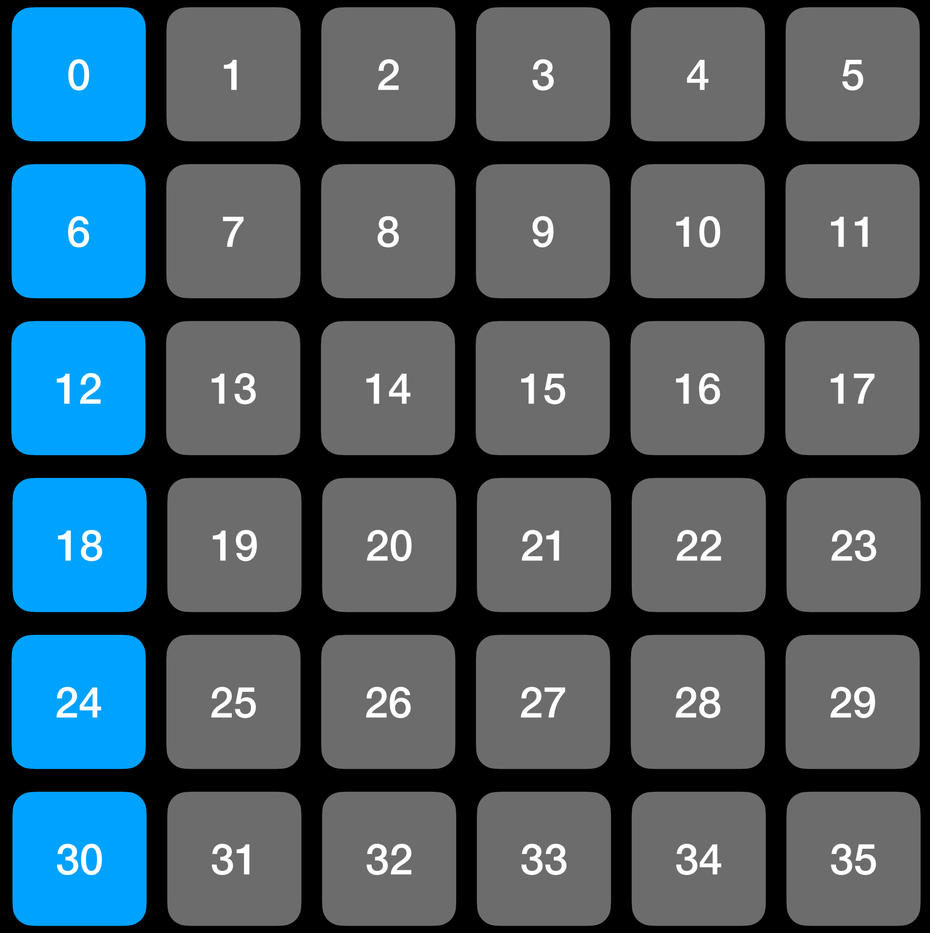



```swift
let grid = Grid<Int>(0...35, width: 6, height: 6)
grid.subgrid(origin: .zero, width: 6, height: 1)
```

```swift
extension Grid {
    @inlinable
    func content(row: Int) -> GridSubGrid<Element> {
        subgrid(origin: .init(x: 0, y: row),
                width: width,
                height: 1)
    }
}
```

Grid (2D Array)

Column

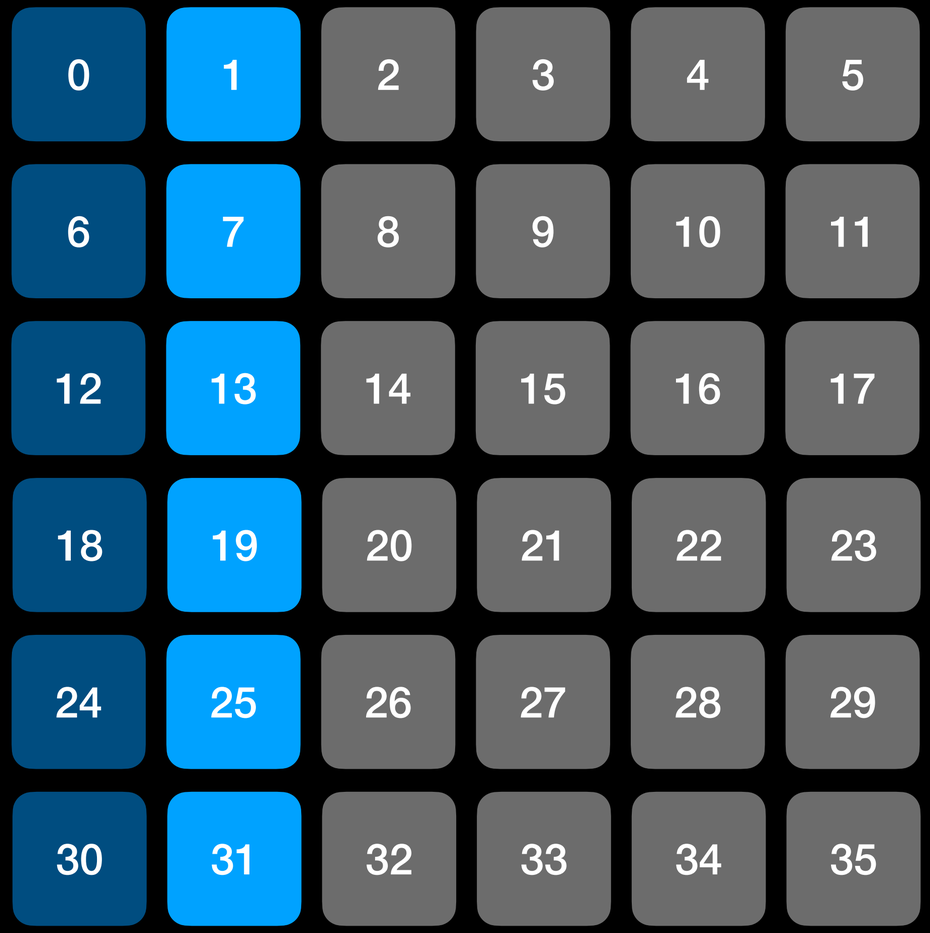



```
let grid = Grid<Int>(0...35, width: 6, height: 6)
grid.subgrid(origin: .zero, width:1, height: 6)
```

```
extension Grid {
    @inlinable
    func content(column: Int) -> GridSubGrid<Element> {
        subgrid(origin: .init(x: column, y: 0),
                width: 1,
                height: height)
    }
}
```

Grid (2D Array)

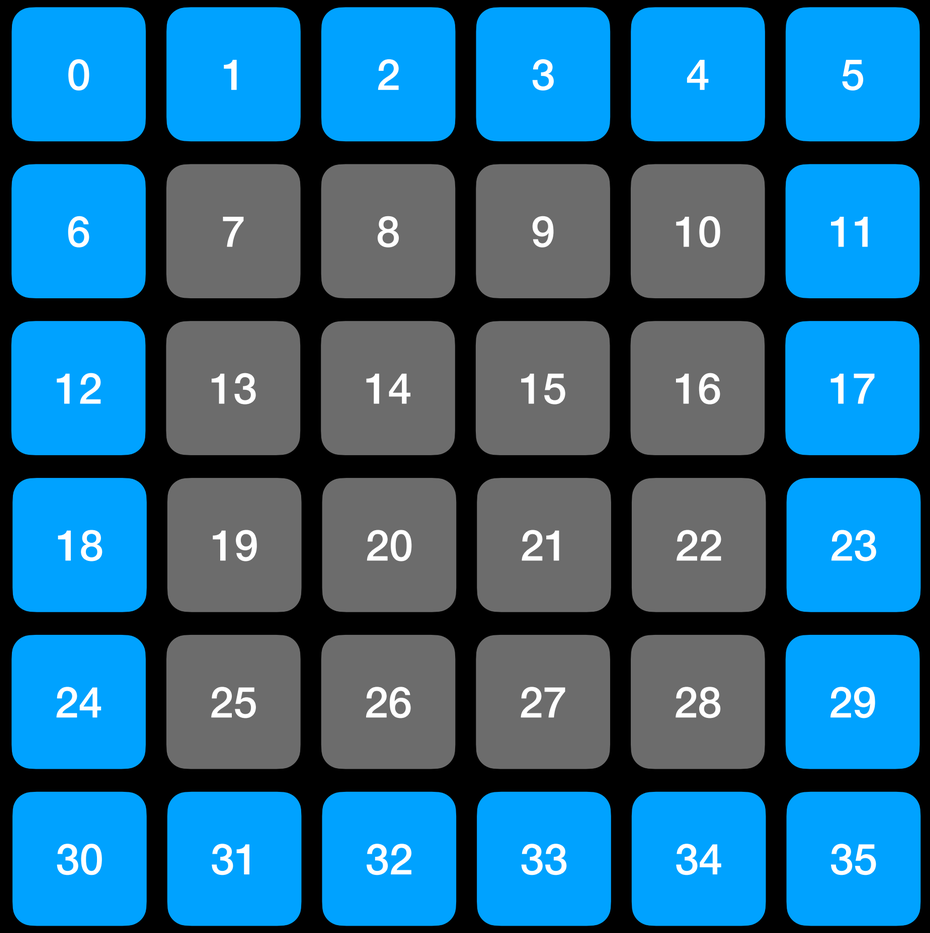
Windows




```
public struct GridWindowedSequence<T>: Sequence {
    internal let base: Grid<T>
    internal let width: Int
    internal let height: Int

    public struct Iterator: IteratorProtocol {
        internal var base: Grid<T>
        internal var width: Int
        internal var height: Int
        internal var x = 0
        internal var y = 0

        public mutating func next() -> GridSubGrid<T>? {
            if x > base.width - width { y += 1; x = 0 }
            guard y <= base.height - height else { return nil }
            defer { x += 1 }
            return base.subgrid(origin: .init(x: x, y: y), width: width, height: height)
        }
    }

    public func makeIterator() -> Iterator {
        Iterator(base: base, width: width, height: height)
    }
}

extension Grid {
    public func windows(width: Int, height: Int) -> GridWindowedSequence<Element> {
        GridWindowedSequence(base: self, width: width, height: height)
    }
}
```

Grid (2D Array)

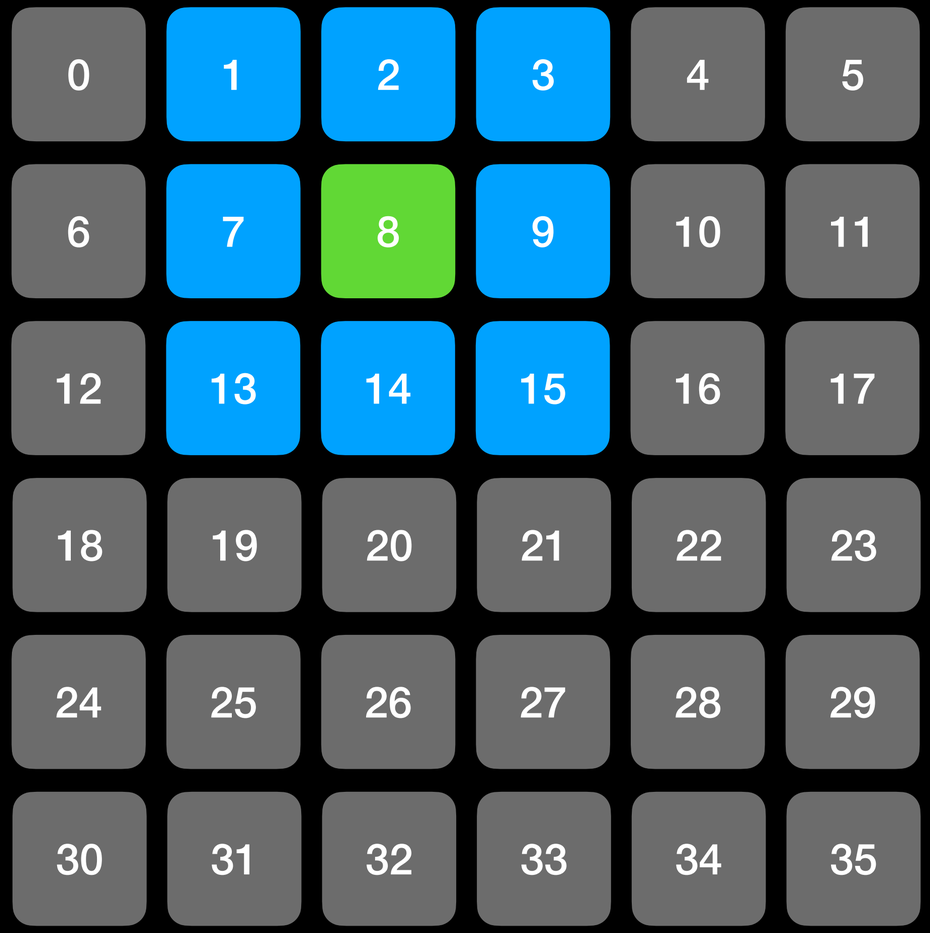
Border




```
public struct GridBorderSequence<Element>: Collection {
    // Implementation in materials
}

extension Grid {
    func border() -> GridBorderSequence<Element> {
        GridBorderSequence(grid: self, origin: .zero,
                           width: width, height: height,
                           includeDiagonals: true,
                           wrap: false)
    }

    func border(from origin: Point<Int>,
                width: Int, height: Int,
                includeDiagonals: Bool = true) -> GridBorderSequence<Element> {
        GridBorderSequence(grid: self,
                           origin: origin,
                           width: width, height: height,
                           includeDiagonals: includeDiagonals,
                           wrap: false)
    }
}
```

Grid (2D Array)

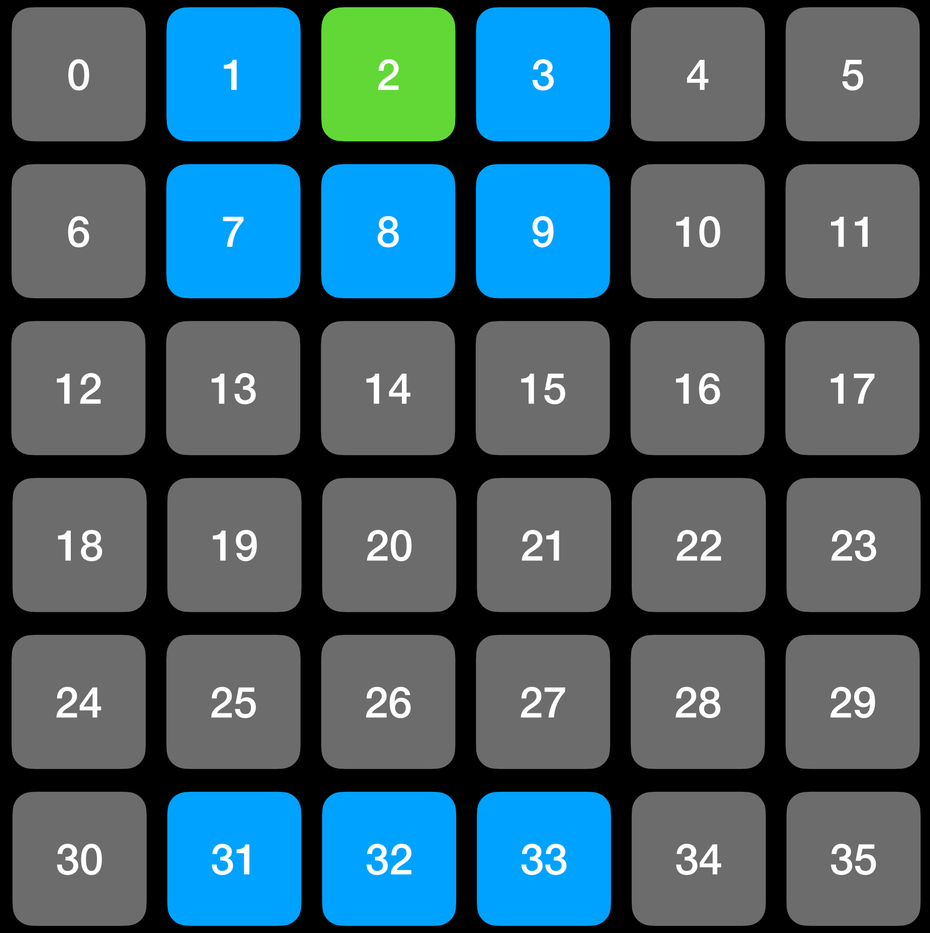
Border




```
public struct GridBorderSequence<Element>: Collection {
    // Implementation in materials
}

extension Grid {
    func border() -> GridBorderSequence<Element> {
        GridBorderSequence(grid: self, origin: .zero,
                           width: width, height: height,
                           includeDiagonals: true,
                           wrap: false)
    }

    func border(from origin: Point<Int>,
                width: Int, height: Int,
                includeDiagonals: Bool = true) -> GridBorderSequence<Element> {
        GridBorderSequence(grid: self,
                           origin: origin,
                           width: width, height: height,
                           includeDiagonals: includeDiagonals,
                           wrap: false)
    }
}
```

Grid (2D Array)

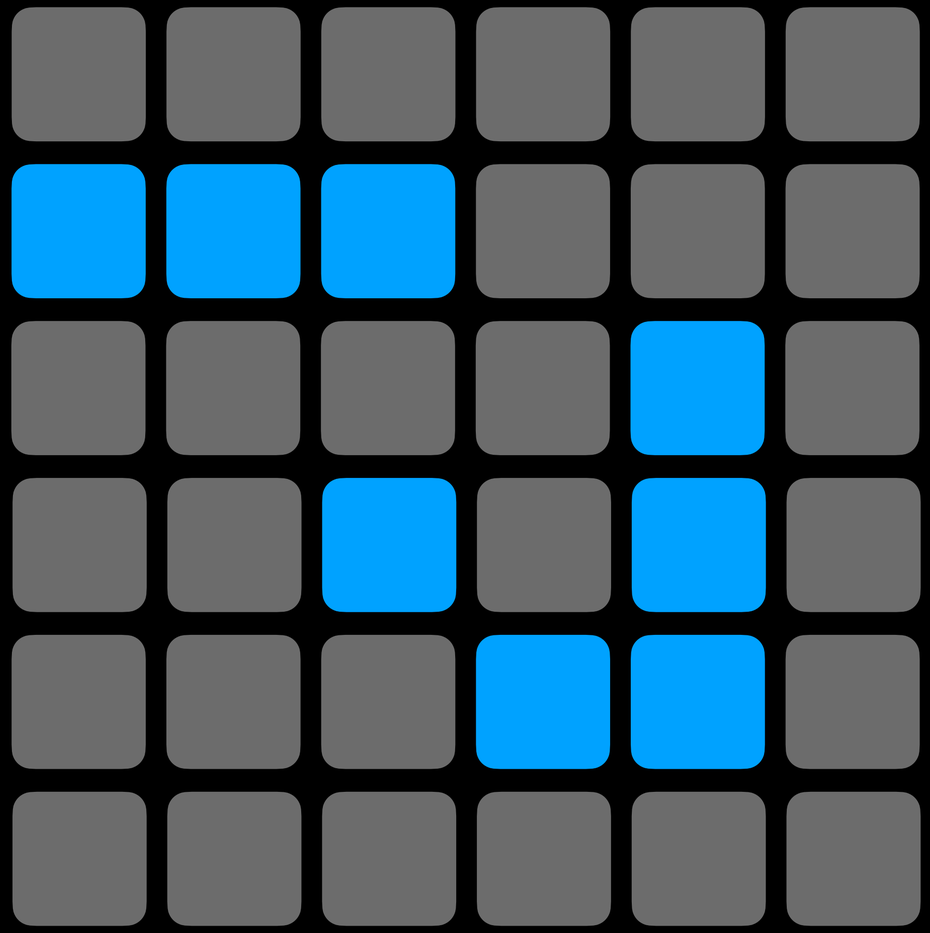
Border




```
extension Grid {
    func border(center: Point<Int>,
                distanceX: Int = 1, distanceY: Int = 1,
                includeDiagonals: Bool = true,
                wrapGrid: Bool = false) -> GridBorderSequence<Element> {
        GridBorderSequence(grid: self,
                           origin: center - Point(x: distanceX, y: distanceY),
                           width: 1 + distanceX + distanceX,
                           height: 1 + distanceY + distanceY,
                           includeDiagonals: includeDiagonals,
                           wrap: wrapGrid)
    }
}
```

Grid (2D Array)

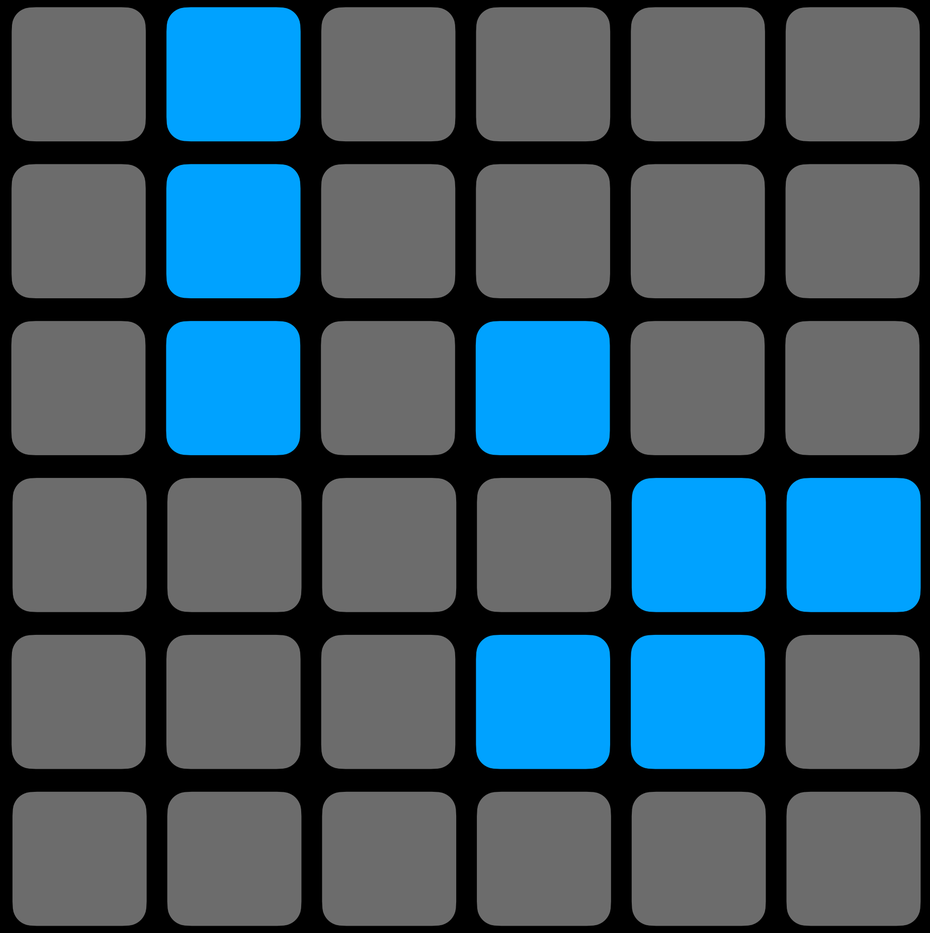
Conway's Game of Life



```
extension Grid where Element == Int {
    func gameOfLife() -> Self {
        indices.reduce(into: Grid(width: width,
                                  height: height,
                                  initial: 0)) {
            newGrid, point in
            let count = self.border(center: point, wrapGrid: true)
                            .sum()

            if count == 3 {
                newGrid[point] = 1
            } else
            if count == 2 && self[point] == 1 {
                newGrid[point] = 1
            }
        }
    }
}
```

Grid (2D Array)

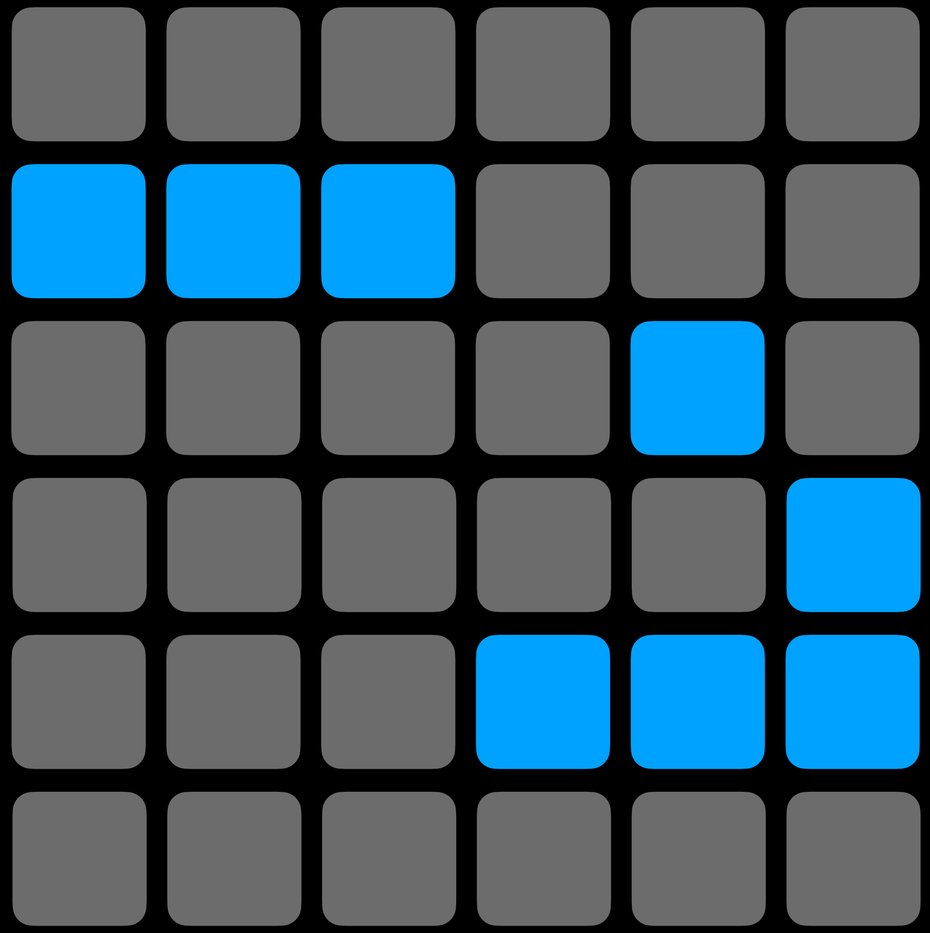
Conway's Game of Life



```
extension Grid where Element == Int {
    func gameOfLife() -> Self {
        indices.reduce(into: Grid(width: width,
                                  height: height,
                                  initial: 0)) {
            newGrid, point in
            let count = self.border(center: point, wrapGrid: true)
                            .sum()

            if count == 3 {
                newGrid[point] = 1
            } else
            if count == 2 && self[point] == 1 {
                newGrid[point] = 1
            }
        }
    }
}
```

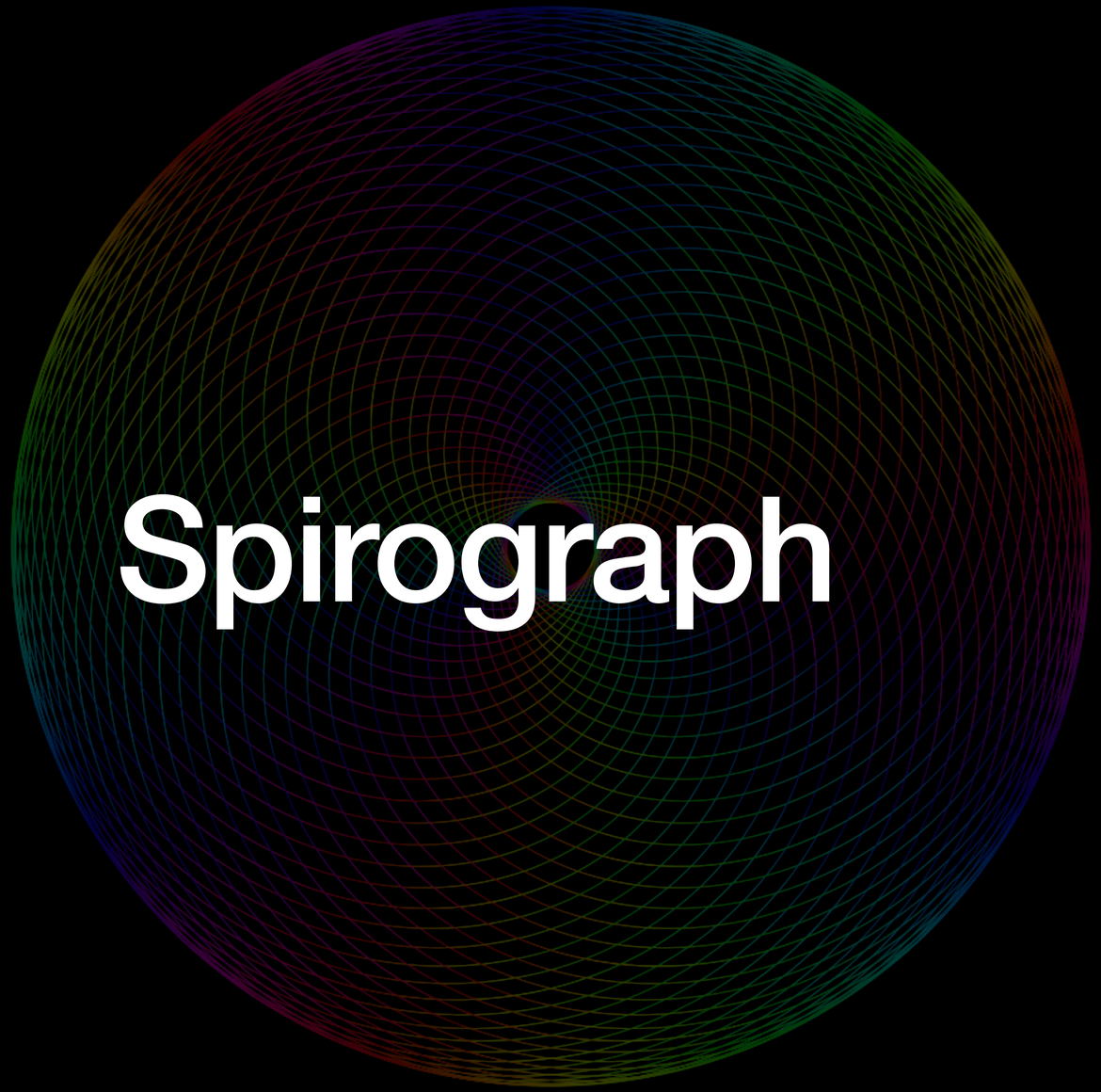
Grid (2D Array)

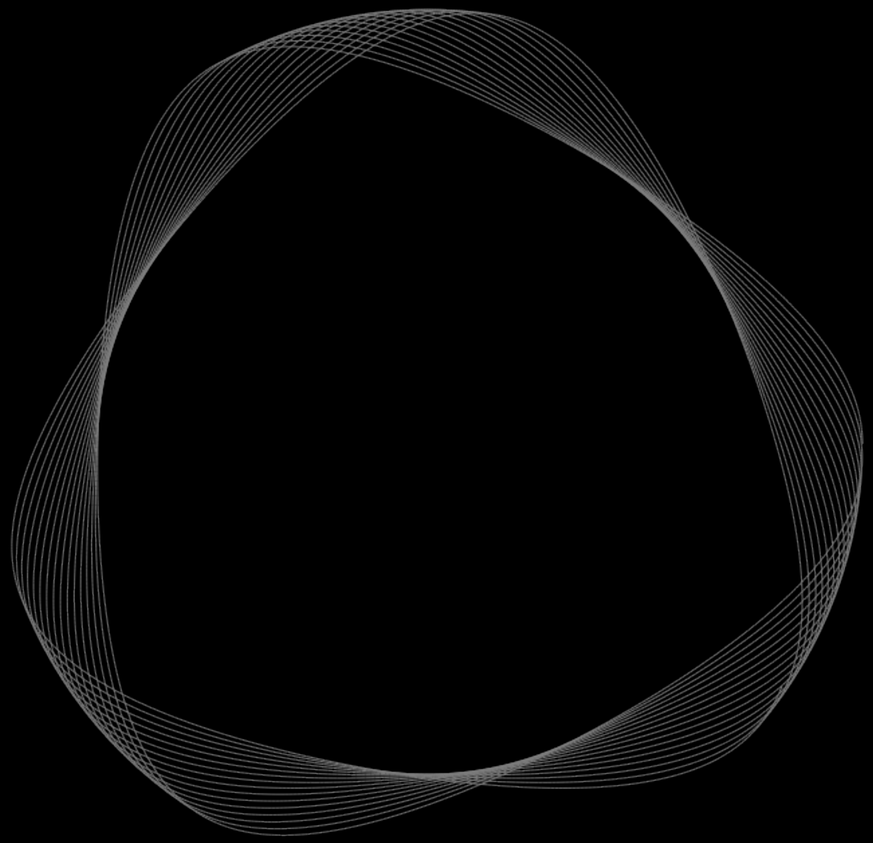
Conway's Game of Life



```
extension Grid where Element == Int {
    func gameOfLife() -> Self {
        indices.reduce(into: Grid(width: width,
                                  height: height,
                                  initial: 0)) {
            newGrid, point in
            let count = self.border(center: point, wrapGrid: true)
                            .sum()

            if count == 3 {
                newGrid[point] = 1
            } else
            if count == 2 && self[point] == 1 {
                newGrid[point] = 1
            }
        }
    }
}
```

Spirograph





Spirograph

```
func spirograph(innerRadius: Double,
                outerRadius: Double,
                distance: Double) -> some Sequence<CGPoint> {
    let Δradius = outerRadius - innerRadius
    let Δtheta = 0.01
    return sequence(state: 0.0) { theta in
        let x = Δradius * cos(theta) + distance * cos(Δradius * theta / innerRadius)
        let y = Δradius * sin(theta) + distance * sin(Δradius * theta / innerRadius)
        theta += Δtheta
        return CGPoint(x: x, y: y)
    }
}
```

Spirograph

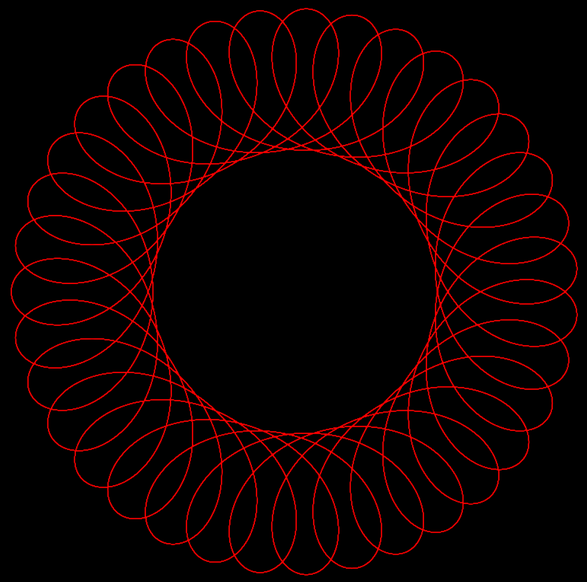
```
let path = spirograph(innerRadius: 105, outerRadius: 12, distance: 31)
    .lazy
    .prefix(100_000)
    .map { ($0 * scale) + patternOffset }
    .reduce(first: { point in
        let path = CGMutablePath()
        path.move(to: point)
        return path
    }, updateAccumulatingResult: { partialResult, point in
        partialResult.addLine(to: point)
    })
```

Spirograph
Custom reduce

```
extension Sequence {
    func reduce<Result>(first: (Iterator.Element) -> Result,
                        updateAccumulatingResult: (inout Result, Iterator.Element) throws -> ())
                        rethrows -> Result? {
        var iterator = self.makeIterator()
        guard let firstElement = iterator.next() else { return nil }
        var result = first(firstElement)
        while let n = iterator.next() {
            try updateAccumulatingResult(&result, n)
        }
        return result
    }
}
```

Spirograph

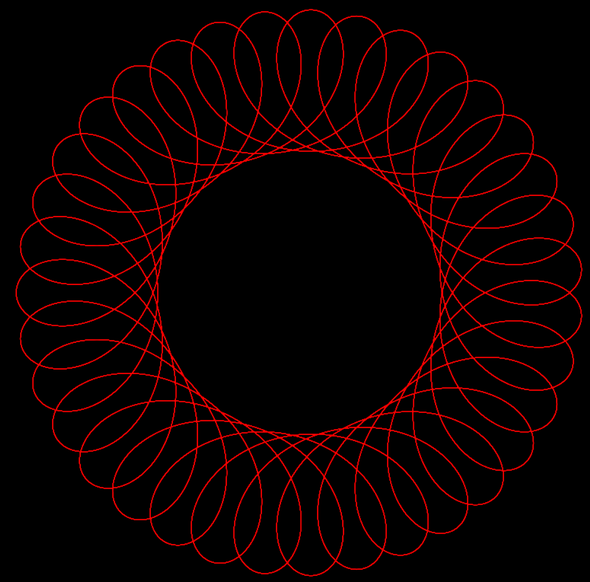
```
let path = spirograph(innerRadius: 105, outerRadius: 12, distance: 31)
    .lazy
    .prefix(100_000)
    .map { ($0 * scale) + patternOffset }
    .reduce(first: { point in
        let path = CGMutablePath()
        path.move(to: point)
        return path
    }, updateAccumulatingResult: { partialResult, point in
        partialResult.addLine(to: point)
    })
```



Spirograph

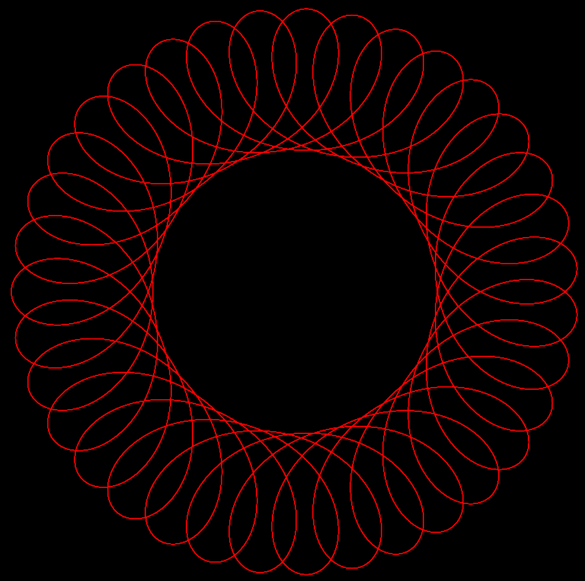
Extension to make paths

```
extension Sequence where Element == CGPoint {
    func path() -> CGPath? {
        let path = reduce(first: { point in
            let path = CGMutablePath()
            path.move(to: point)
            return path
        }, updateAccumulatingResult: { partialResult, point in
            partialResult.addLine(to: point)
        })
        return path?.copy()
    }
}
```



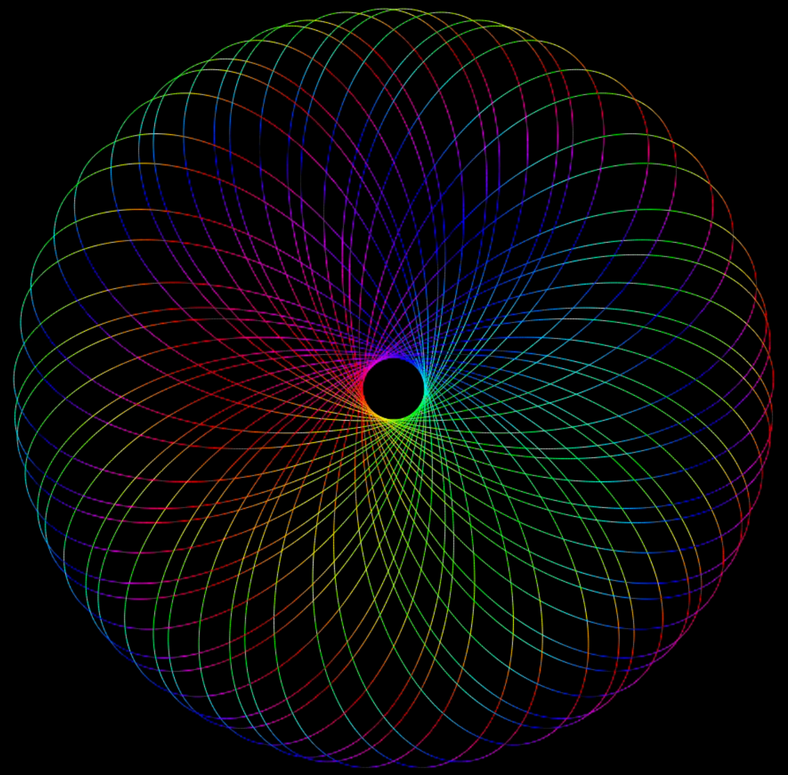
Spirograph

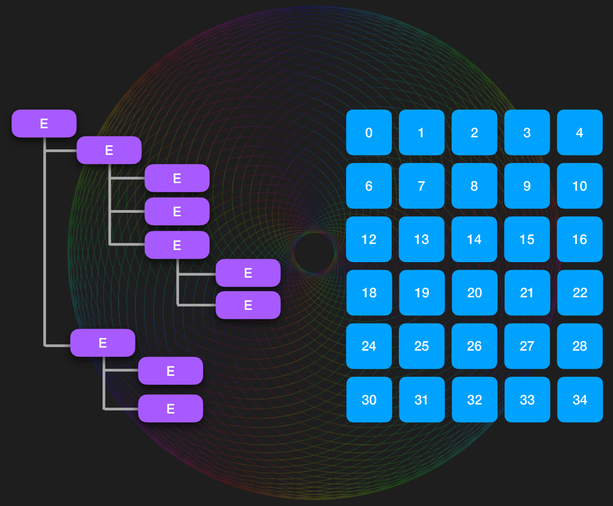
```
let path = spirograph(innerRadius: 105, outerRadius: 12, distance: 31)
    .lazy
    .prefix(100_000)
    .map { ($0 * scale) + patternOffset }
    .path()
```



Spirograph

```
let context = CGContext(...)
let image = spirograph(innerRadius: 71, outerRadius: 38, distance: 28)
    .lazy
    .prefix(100_000)
    .map { ($0 * scale) + patternOffset }
    .adjacentPairs() 📦
    .map { pointA, pointB in CGPath.line(from: pointA, to: pointB) }
    .enumerated()
    .lazy
    .map { index, path in
        let hue = (Double(index) / 255.0).truncatingRemainder(dividingBy: 1.0)
        return (hue, path)
    }
    .reduce(into: context) { context, content in
        let (hue, path) = content
        context.saveGState()
        context.addPath(path)
        context.setStrokeColor(.hue(hue))
        context.strokePath()
        context.restoreGState()
    }
    .makeImage()
```





Conform your custom containers to Sequence protocols

```
extension Sequence where
Element: AdditiveArithmetic {
    func sum() -> Element {
        self.reduce(.zero, +)
    }
}
```

Generalise your algorithms to protocol extensions

```
func run<T>(_ i: Array<T>)
```

```
func run<S: Sequence>(_ i: S)
```

Generalise parameters where possible

Sequencing Success

Profile your code

Use Lazy

```
EmptyCollection<Int>()
CollectionOfOne(13)
```

Take advantage of existing functionality

Any Questions ?