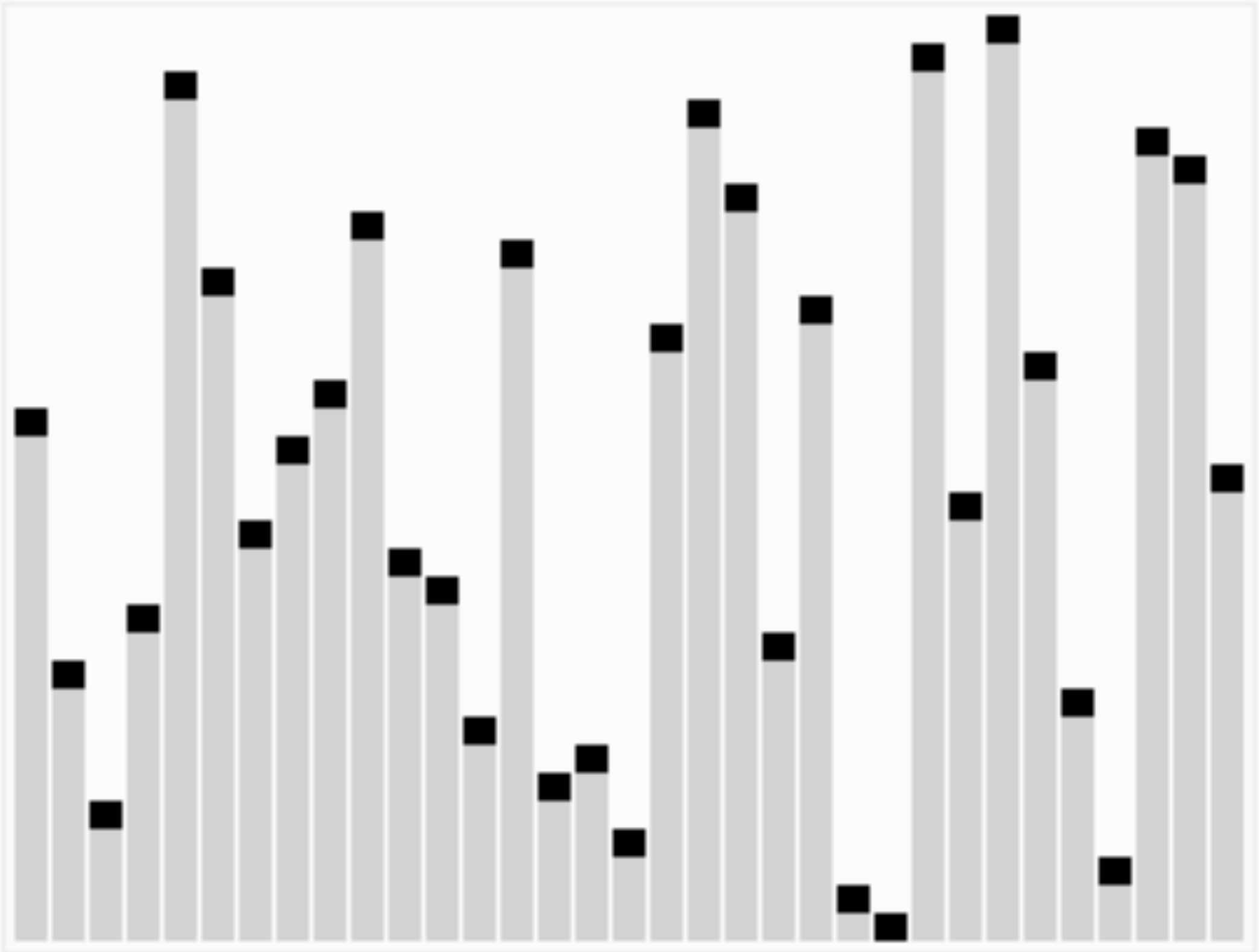
Deconstruction of a lazy, tail-recursive quicksort

From the *Joy of Clojure* Listing 6.3

Basic Quicksort

- Divide and conquer algorithm
- Sort against the pivot
- Split work on pivot
- O(n log n), typically



http://en.wikipedia.org/wiki/Quicksort

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

Let bindings

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

Destructuring

```
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

Tail recursion

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

Recursion

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

In-place functions

```
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

List API functions

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

Lazy sequences

Deconstruction

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

Let bindings

`let`

- Establish local lexical scope bindings
- Built from lambda
- Variants, `if-let`, `when-let`, `loop`

```clojure
;; Deriving let from lambda

((fn [y] (+ y 42)) 10)

(let [y 10] (+ 42 y))

;; Create your own let via macros.

(defmacro my-let [x body]
  (list (list `fn[(first x)]
              `~body)
        (last x)))

(my-let [z 42] (* z z))

;; if-let, when-let

(let [x false] (if x 'foo 'bar)) ;; bar

(if-let [x false] 'foo 'bar ) ;; bar

(when-let [x true] 'foo) ;; foo

;; often multiple bindings

(let [a 10 b 20] (+ a b)) ;; 30
```

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

Let bindings

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

Destructuring

Destructuring

- Mini language within the language
- Pull apart composite data structure to bind to locals.
- Works in let, loop, lambda, etc.

```clojure
;; Destructuring

(let [[a b c] (range 1 10)] (+ a b c))
;; 6

(let [[a b c & d] (range 1 10)] d)
;; (4 5 6 7 8 9)

(let [[a b c & d :as all] (range 1 10)] all)
;; (1 2 3 4 5 6 7 8 9)

(def my-name {:first "Rich" :last "Hickey"})

(let [{first :first last :last } my-name]
  (list first last))

(let [{:keys [first last]} my-name]
  (list first last))

;; ("Rich" "Hickey")
```

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

Destructuring

```
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

Tail recursion

`loop` / `recur` recursion

- `loop` also provide let bindings
- `loop` provides a recursion target
- recursion must be in tail position
- Same # of args passed to `recur` as `loop`

```clojure
(defn pow [num,n]
  (loop [p (dec n) result num]
    (if (zero? p) result
        (recur (dec p) (* result num)))))

(pow 2 3) ;; 8

(pow 2 1000) ;; Doesn't SO
```

```
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

Tail recursion

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

In-place functions

In-place functions

- Mini-macro for expressing lamdba
- Often found in `map` and `filter`
- Can take multiple arguments

```
(macroexpand '#(< % 42))

(fn* [p1__2026#] (< p1__2026# 42))

#(+ %1 %2 42)
```

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

In-place functions

```
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

List API functions

Lisps like lists

- *It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.* - Rich Hickey

```clojure
(seq (.split "1,2,3,4" ","))
;; ("1" "2" "3" "4")

(true? (seq [])) ;;false, nil-punning

(filter #(even? %) (range 0 10))
;; (0 2 4 6 8)

(remove #(even? %) (range 0 10))
;; (1 3 5 7 9)

(list 'Do 'Re 'Mi 'Fa 'Sol 'La 'Ti 'Do)
;; (Do Re Mi Fa Sol La Ti Do)

(list* 'a 'b  (list 'c))
;; (a b c)

(cons 'Do (list 'Re 'Mi))
;; (Do Re Mi)
```

```
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

List API functions

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

Lazy sequences

`lazy-seq` guidelines from JoC 6.3.2

- `lazy-seq` macro at the outermost level
- use `rest` instead of `next`
- Prefer higher-order functions
- Don’t hold onto your head

Losing your head

```
(let [r (range 1e9)] [(first r) (last r)]) ;=> [0 999999999]

(let [r (range 1e9)] [(last r) (first r)])
```

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts))))))

(defn qsort [xs] (sort-parts (list xs)))
```

Lazy sequences

```clojure
(defn sort-parts
  "Lazy, tail-recursive, incremental quicksort."
  [work]
  (lazy-seq
   (loop [[part & parts] work]
     (if-let [[pivot & xs] (seq part)]
       (let [smaller? #(< % pivot)]
         (recur (list* (filter smaller? xs)
                       pivot
                       (remove smaller? xs)
                       parts)))
       (when-let [[x & parts] parts]
         (cons x
               (sort-parts parts)))))))

(defn qsort [xs] (sort-parts (list xs)))
```