Stable Sorting

If keys are unique, we generally consider sorted order to be increasing (sometimes also known as strictly increasing) order.

For example, given: 2 7 8 3 4 1 6 5, sorted order is: 1 2 3 4 5 6 7 8

If we allow keys to be repeated, the natural ordering also follows.
Given: 2 1 2 3 4 1 2 1, sorted order is: 1 1 1 2 2 2 3 4

Is the first 1 the same as the second 1 or third 1?

We typically focus on the problem of sorting keys but data is rarely only the keys. Generally, data is a key-value pair; i.e., (key, value).

While the keys may be the same (all 1s), their associated values are likely different.

Consider the following list of `(key, value)` pairs of type `(Num, Sym)`:

```
'((1 one) (2 one) (1 two) (1 three) (2 two) (1 four))
```

There are many orderings that we would consider sorted (by `key`):

- `'((1 one) (1 three) (1 two) (1 four) (2 two) (2 one))`
- `'((1 four) (1 three) (1 two) (1 one) (2 two) (2 one))`
- `'((1 three) (1 one) (1 two) (1 four) (2 one) (2 two))`
- `'((1 one) (1 two) (1 three) (1 four) (2 one) (2 two))`

In this example, there are 48 possible orderings that are sorted. Different sorting algorithms will produce different orderings.

A sort that is **stable** has the additional property that when `key`s are the same, the original order will be maintained; i.e., the only ordering that is stable is:

- `'((1 one) (1 two) (1 three) (1 four) (2 one) (2 two))`

A Stable Mergesort

Outline for a `mergesort` algorithm:

1. Split the given list into two lists of equal (or almost equal) length.
2. Recursively apply `mergesort` on each of the smaller lists.
3. Believe in recursion $\Rightarrow$ the recursion produces two sorted lists.
4. `Merge` the two smaller lists together into a single sorted list.

To achieve a **stable** mergesort, we need to think about how the *splitting* in step 1 and the *merging* in step 4 work.

A Stable Mergesort - splitting

In L14 S29, `keep-next` and `skip-next` were used to split a list into two:

```
(list 8 4 3 9 1 6 2 5 0 7)
                    ⇒ (list 8 3 1 2 0) (list 4 9 6 5 7)
```

- Items at even numbered indices are placed in one list, and odd numbered indices in the other.
- This has made some of the original ordering less obvious.
  For example: only using the two resulting lists, is it easy to determine if 2 appeared before or after 5 in the original list?
  It can be done but would take some work.

A more common method is to split the list into first half and second half.

```
(list 8 4 3 9 1 6 2 5 0 7)
                    ⇒ (list 8 4 3 9 1) (list 6 2 5 0 7)
```

- We know everything in the first list appeared before anything in the second list.

Splitting a list into a first half and second half

Recall from L10:

```
(define (first-n n lst)
  (cond [(or (empty? lst) (zero? n)) empty]
        [else (cons (first lst)
                    (first-n (sub1 n) (rest lst)))]))

(define (rest-n n lst)
  (cond [(or (empty? lst) (zero? n)) lst]
        [else (rest-n (sub1 n) (rest lst))]))

(check-expect (first-n 3 '(1 2 3 4 5 6 7))'(1 2 3))
(check-expect (rest-n 3 '(1 2 3 4 5 6 7)) '(4 5 6 7))
```

Splitting a list into a first half and second half

```
;; splits a list into nearly equal halves
;; split: (listof Any) -> (list (listof Any) (listof Any))
(define (split lst)
  (local [(define n (quotient (length lst) 2))]
         (list (first-n n lst) (rest-n n lst))))

(check-expect (split '(1 2 3 4 5 6 7)) '((1 2 3) (4 5 6 7)))
(check-expect (split '(a b c d e f)) '((a b c) (d e f)))

(check-expect (split '((1 one) (1 two) (1 three) (1 four)))
              '(((1 one) (1 two)) ((1 three) (1 four))))
```

Consider the behaviour `merge` from L10

```
;; merge: (listof Num) (listof Num) -> (listof Num)
;; Requires: lists must be sorted in increasing order
(define (merge lst1 lst2)
  (cond [(empty? lst1) lst2]
        [(empty? lst2) lst1]
        [(< (first lst1) (first lst2))
         (cons (first lst1) (merge (rest lst1) lst2))]
        [(> (first lst1) (first lst2))
         (cons (first lst2) (merge lst1 (rest lst2)))]
        [else (cons (first lst1
                    (cons (first lst2)
                          (merge (rest lst1) (rest lst2))))])))
```

Note: this version of merge only works with a list of keys.

```
(merge '(1 1) '(1 1)) => '(1 1 1 1) ; is it stable?
```

Consider the behaviour `merge` from L10

We need to modify `merge` to work with our `(key, value)` pairs.

```
;; Produce the key from a (list key value)
;; get-key: (list Num Sym) -> Num
(define (get-key kvp) (first kvp))
```

We can then modify the cases in `merge`

```
(define (merge lst1 lst2)
  (cond [(empty? lst1) lst2]
        ...
        [(< (get-key (first lst1)) (get-key (first lst2)))
         (cons (first lst1) (merge (rest lst1) lst2))]
        [...]))


(merge (list (1 'one) (1 'two)) (list (1 'three) (1 'four)))
=> (list (1 'one) (1 'three) (1 'two) (1 'four))
```

- This doesn't seem to help to achieve stability!

A simplified implementation of `merge`

```
(define (merge lst1 lst2)
  (cond [(empty? lst1) lst2]
        [(empty? lst2) lst1]
        [(< (first lst1) (first lst2)) ; (get-key (first lst))
          (cons (first lst1) (merge (rest lst1) lst2))]
        [else
          (cons (first lst2) (merge lst1 (rest lst2)))]))

(merge (list (1 'one) (1 'two)) (list (1 'three) (1 'four)))
=> (list (1 'three) (1 'four) (1 'one) (1 'two))
```

- This prioritizes items from `lst2` over `lst1` when keys are the same.
- To get the order we want, we could swap the arguments of `merge`.
- Alternatively, we could change < to <= to prioritize items from `lst1` over `lst2`.

Generalizing `merge`

What if we want to sort string? Or use <= instead of |<|?

- Define a higher-order function with a parameter <? that consumes a comparison operator for the type of items in the consumed lists.

```
(define (merge <? lst1 lst2)
  (cond [(empty? lst1) lst2]
        [(empty? lst2) lst1]
        [(<? (first lst1) (first lst2))
         (cons (first lst1) (merge <? (rest lst1) lst2))]
        [else
             (cons (first lst2) (merge <? lst1 (rest lst2)))]))

(check-expect (merge string<? '("Alice" "Carol") '("Bob"))
              '("Alice" "Bob" "Carol"))
(check-expect (merge > '(6 4 2) '(7 5 3 1)) '(7 6 5 4 3 2 1))
```

Generalizing `merge`

```
;; compares keys from (key, value) pairs using <
;; kvp<: (list Num Sym) (list Num Sym) -> Bool
(define (kvp< kvp1 kvp2)
  (< (get-key (first kvp1)) (get-key (first kvp2))))

;; compares keys from (key, value) pairs using <=
(define (kvp<= kvp1 kvp2)
  (<= (get-key (first kvp1)) (get-key (first kvp2))))
```

Note: `get-key` is now part of the comparison operator.

```
(check-expect (merge kvp< (list (1 'three) (1 'four))
                          (list (1 'one) (1 'two)))
              (list (1 'one) (1 'two) (1 'three) (1 'four)))

(check-expect (merge kvp<= (list (1 'one) (1 'two))
                           (list (1 'three) (1 'four)))
              (list (1 'one) (1 'two) (1 'three) (1 'four)))
```

Stable Mergesort - choices

```
;; Requires <? is a strict order operator
define (mergesort <? lst)
  (cond [(or (empty? lst) (empty? (rest lst))) lst]
        [else (local [(define s (split lst))]
                     (merge <? (mergesort (second s))
                               (mergesort (first s))))])]

;; Requires <=? is an ordering operator with equality
define (mergesort <=? lst)
  (cond [(or (empty? lst) (empty? (rest lst))) lst]
        [else (local [(define s (split lst))]
                     (merge <=? (mergesort (first s))
                                (mergesort (second s))))])]
```

Then there is this ...

A function that makes stable sorting functions

```
;; Produces a sort function from a predicate
;; Requires: the consumed comparison operator is a strict
    ordering
;; make-sort: (X X -> Bool) -> ((listof X) -> (listof X))
(define (make-sort <?)
  (local [(define (mergesort lst)
            (cond [(or (empty? lst) (empty? (rest lst))) lst]
                  [else
                    (local [(define s (split lst))]
                           (merge <? (mergesort (second s))
                           (mergesort (first s))))]))]
          mergesort))
```

See L18 Slides 31–35 for more.