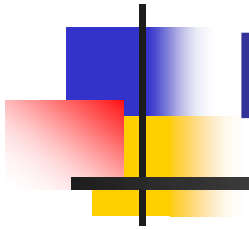


Practice for Scheme

Append a list to another list

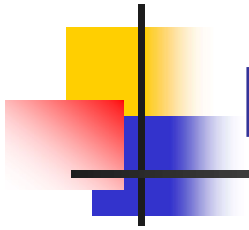
```
(define (append L R)
  (if (null? L) R
    (cons (car L) (append (cdr L) R))))
```

- (append '(1 2 3) '(4 5 6)) :
  (1 2 3 4 5 6)

Reverse a list

```
(define (reverse List)
  (if (null? List) List
    (append (reverse (cdr List))
      (cons (car List) '() ))))
```

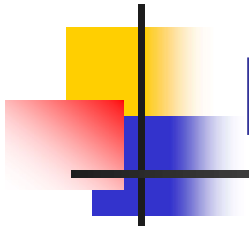
- (reverse '(2 4 6 1)) : (1 6 4 2)



Find the n-th Fibonacci number

```
(define (fibonacci n)
  (if (<= n 1) n
        (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))
```

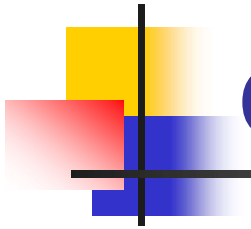
- (fibonacci 6) → 8



Merge two sorted lists

```
(define (merge x y)
  (cond ((null? x) y) ((null? y) x)
    ((<= (car x) (car y)) (cons (car x) (merge (cdr x) y)))
    (else (cons (car y) (merge x (cdr y))))))
```

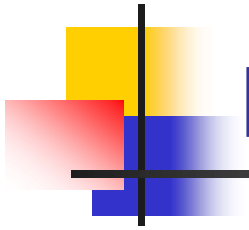
- (merge '(1 3 5 7) '(2 4 5)) → (1 2 3 4 5 5 7)



Count the number of k in a list

```
(define (count x k)
  (cond ((null? x) 0)
        ((= (car x) k) (+ 1 (count (cdr x) k)))
        (else (count (cdr x) k))))
```

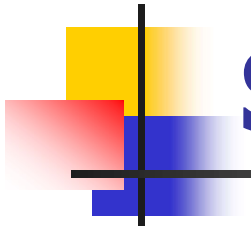
- (count '(1 2 2 2 3 5) 2) → 3



Filter list elements by a function

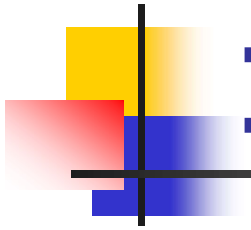
```
(define (filter func x)
    (cond ((null? x) x)
      ((func (car x)) (cons (car x) (filter func (cdr x))))
      (else (filter func (cdr x)))))
```

- (filter (lambda (e) (= (modulo e 2) 1))
  '(1 2 3 4 5 6 7)) → (1 3 5 7)



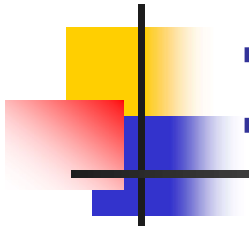
Sorting Problem

- Input:
  - A sequence of n numbers $<a_1, a_2,\ldots,a_n>$
- Output:
  - A reordering $<a'_1, a'_2,\ldots,a'_n>$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- There are many sorting algorithms
  - Insertion sort
  - Merge sort
  - Quick sort



Insertion Sort

- It uses an incremental approach!
- For a sequence of n numbers A[1..n], it consists of $n-1$ passes.
- For pass j = n-1 through 1
  - Use the fact that the elements in $A[j+1..n]$ are already known to be in sorted order.
  - Ensures that the elements in $A[j..n]$ are in sorted order.



Insertion Sort

```
INSERTION-SORT(A)
1   for j = A.length-1 downto 1
2       key = A[j]
3       // Insert A[j] into the sorted sequence A[j+1..n]
4       i = j + 1
5       while i<=n and A[i]<=key
6           A[i-1] = A[i]
7           i = i + 1
8       A[i-1] = key
```

Insertion Sort

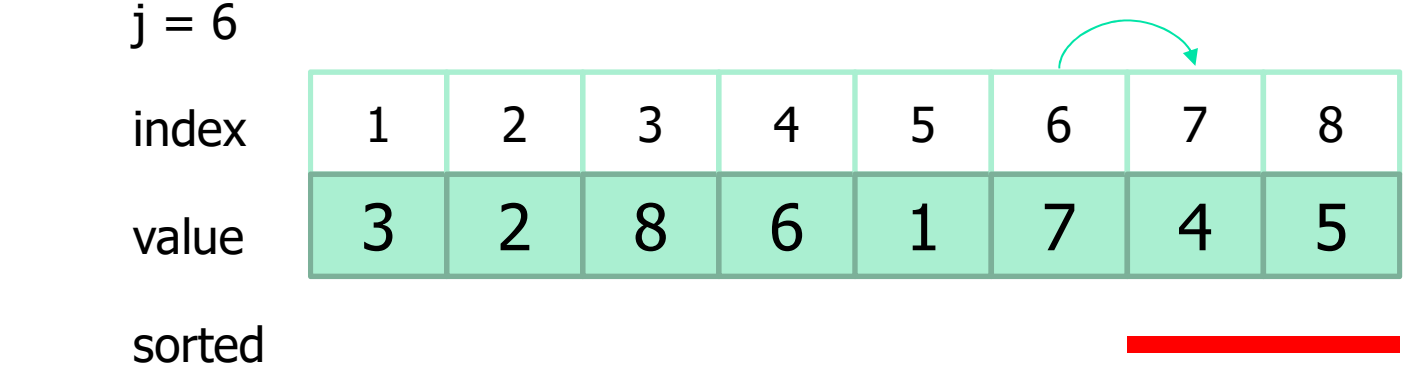
j = 7

index	1	2	3	4	5	6	7	8
value	3	2	8	6	1	7	4	5

sorted

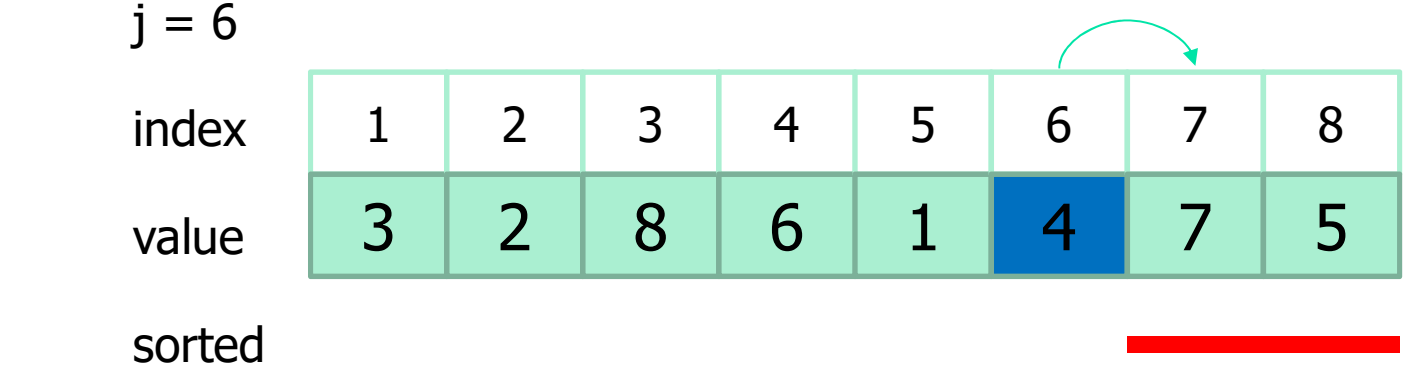
Insertion Sort

j = 6



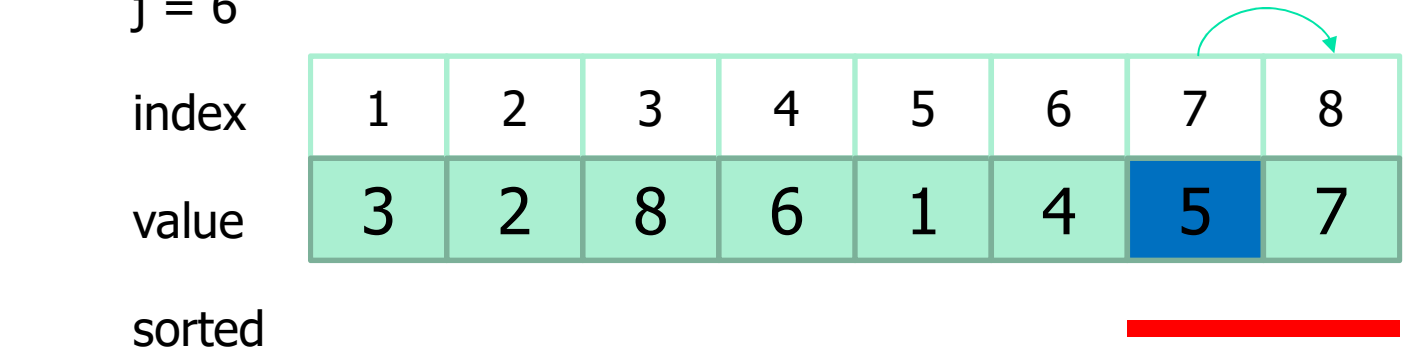
Insertion Sort

j = 6



Insertion Sort

j = 6



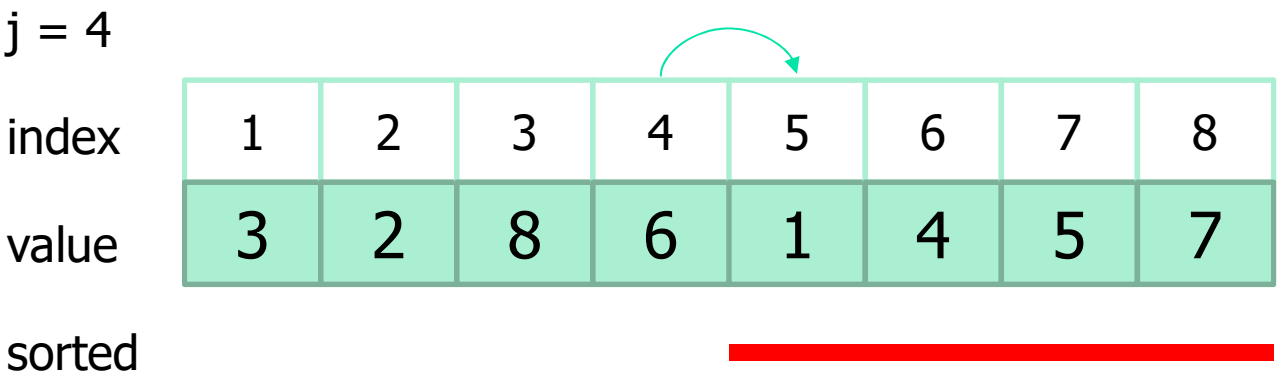
Insertion Sort

j = 5

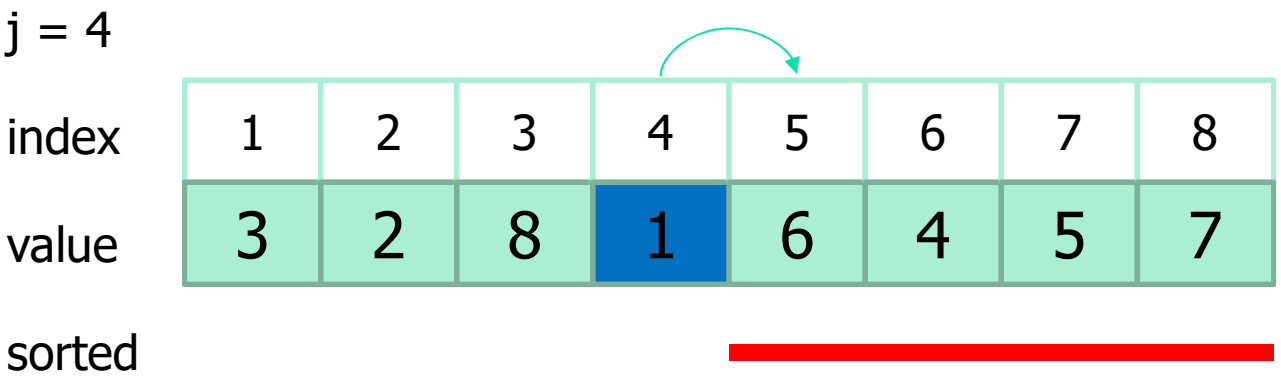
index	1	2	3	4	5	6	7	8
value	3	2	8	6	1	4	5	7

sorted

Insertion Sort

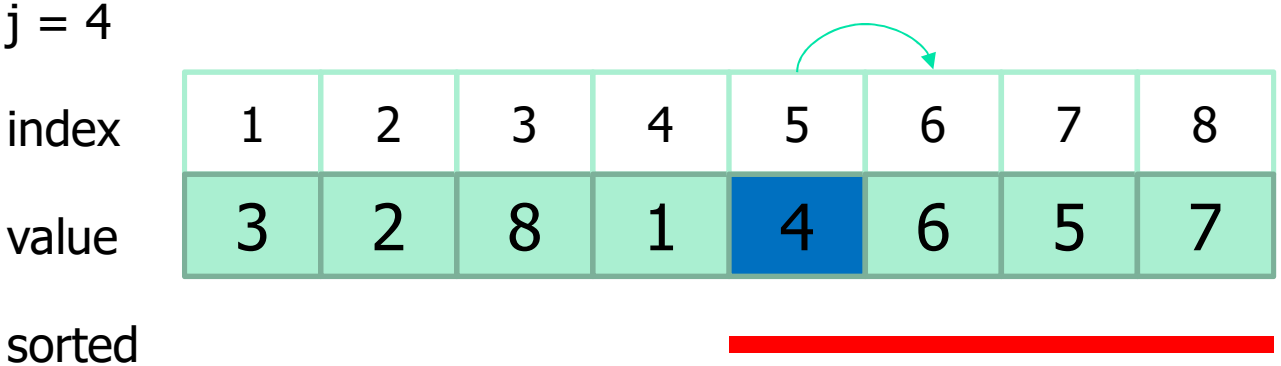


Insertion Sort



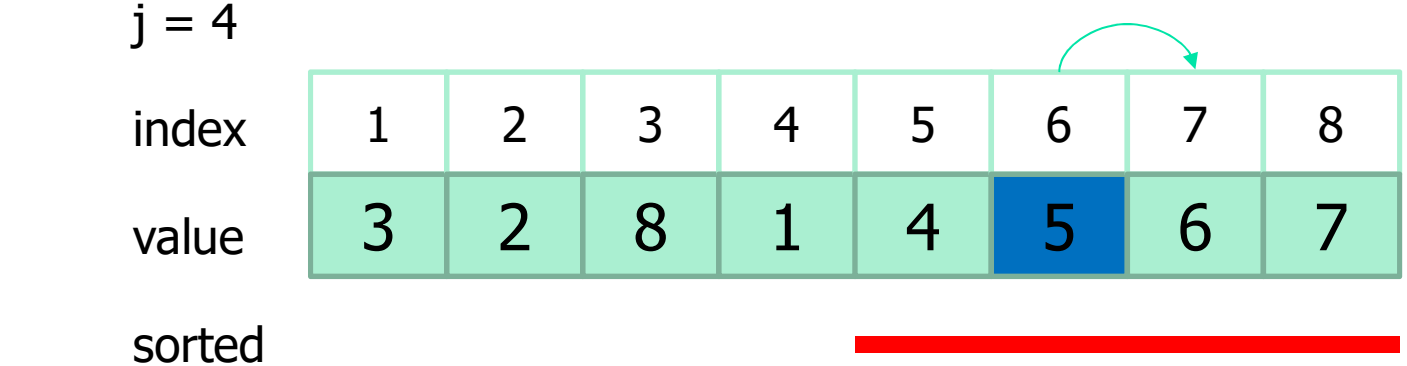
Insertion Sort

j = 4



Insertion Sort

j = 4



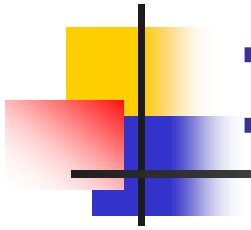
Insertion Sort

j = 4

index	1	2	3	4	5	6	7	8
value	3	2	8	1	4	5	6	7

sorted

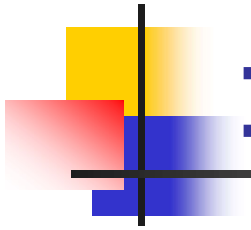
Insertion Sort



j = 1

index	1	2	3	4	5	6	7	8
value	1	2	3	4	5	6	7	8

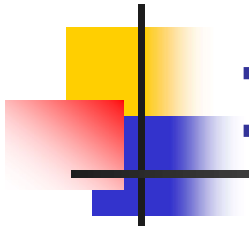
sorted



Insertion Sort

```
(define (insertion-sort L)
  (if (null? L) '()
    (insert (list (car L)) (insertion-sort (cdr L)))))
```

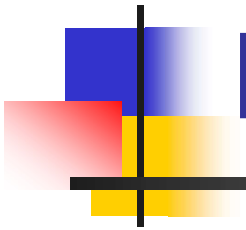
- (insertion-sort '(4 2 10 3 -1 5)) -> (-1 2 3 4 5 10)
- (insert '(4) '(-1 2 3 5 10)) -> (-1 2 3 4 5 10)



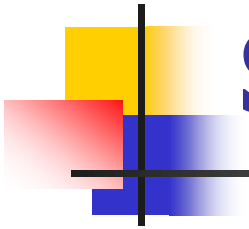
Insertion Sort

```
(define (insert L M)
  (if (null? L) M
      (if (null? M) L
          (if (< (car L) (car M)) (cons (car L) (insert (cdr L) M))
              (cons (car M) (insert L (cdr M)))))))
```

- (insert '(4) '(-1 2 3 5 10)) -> (-1 2 3 4 5 10)

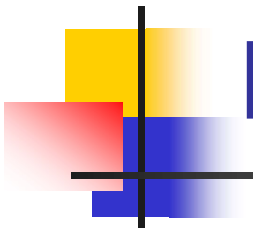


Practice for Scheme



Sorting Problem

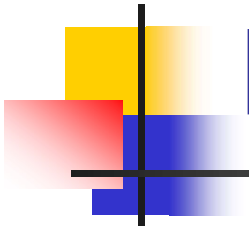
- Input:
  - A sequence of n numbers $<a_1, a_2,\ldots,a_n>$
- Output:
  - A reordering $<a'_1, a'_2,\ldots,a'_n>$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- There are many sorting algorithms
  - Insertion sort
  - **Merge sort**
  - Quick sort



Merge two sorted lists

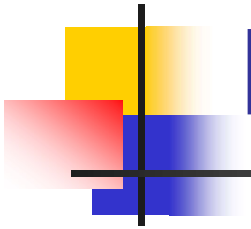
```
(define (merge x y)
  (cond ((null? x) y) ((null? y) x)
    ((<= (car x) (car y)) (cons (car x) (merge (cdr x) y)))
    (else (cons (car y) (merge x (cdr y))))))
```

- (merge '(1 3 5 7) '(2 4 5)) → (1 2 3 4 5 5 7)



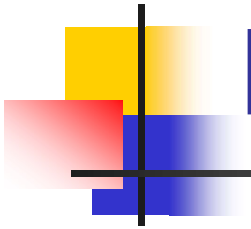
Divide and Conquer

- We solve a problem recursively by applying three steps at each level of the recursion:
  - Divide the problem into a number of subproblems that are smaller instances of the same problem.
  - Conquer the subproblem by solving them recursively.
    - If the problem sizes are small enough (i.e. we have gotten down to the base case), solve the subproblem in a straightforward manner
  - Combine the solutions to the subproblems into the solution for the original problem.



Merge Sort

- Merge Sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.
  - Divide: Divide the n-element sequence to be sorted into two subsequences of n=2 elements each.
  - Conquer: Sort the two subsequences recursively using merge sort.
  - Combine: Merge the two sorted subsequences to produce the sorted answer.
- The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order

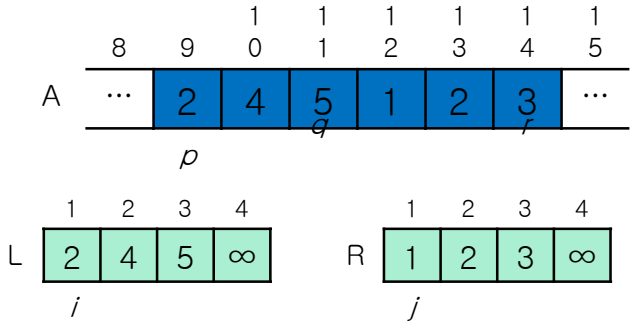


Merge Procedure

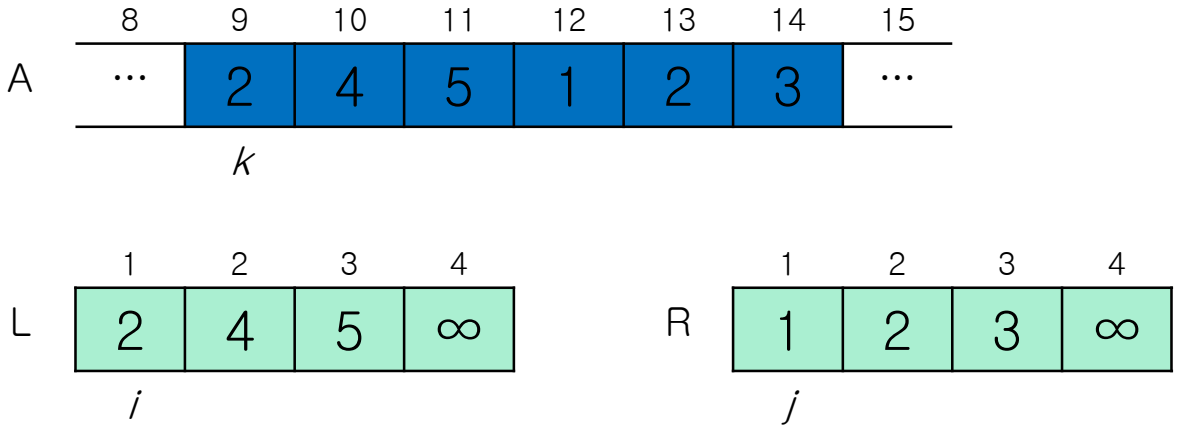
- The key operation of merge sort algorithm.
- The procedure assumes that that the subarrays A[p..q] and A[q+1..r] are in sorted order.
- It merges them to form a single sorted subarray that replaces the current subarray A[p..r].
- We merge by calling an auxiliary procedure MERGE(A, p, q, r) where A is an array and p, q, and r are indices into the array such that $p \leq q < r$.

Merge Procedure

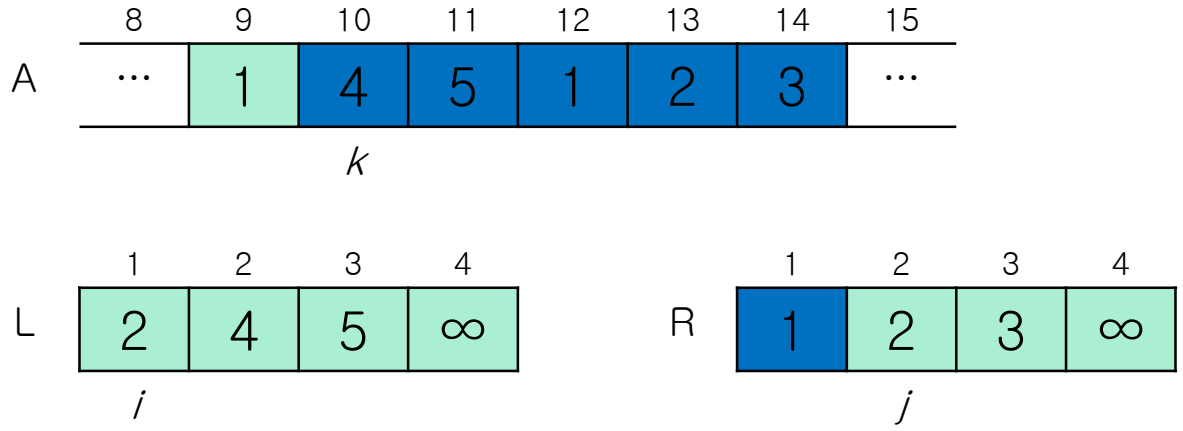
```
MERGE(A, p, q, r)
1   n1 = q – p + 1
2   n2 = r – q
3   let L[1 ... n1+1] and R[1 ... n2+1] be new arrays
4   for i = 1 to n1
5         L[i] = A[p+i-1]
6   for j = 1 to n2
7         R[j] = A[q+j]
8   L[n1+1] = ∞
9   R[n2+1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13        if L[i] ≤ R[j]
14              A[k] = L[i]
15              i = i + 1
16        else A[k] = R[j]
17              j = j + 1
```



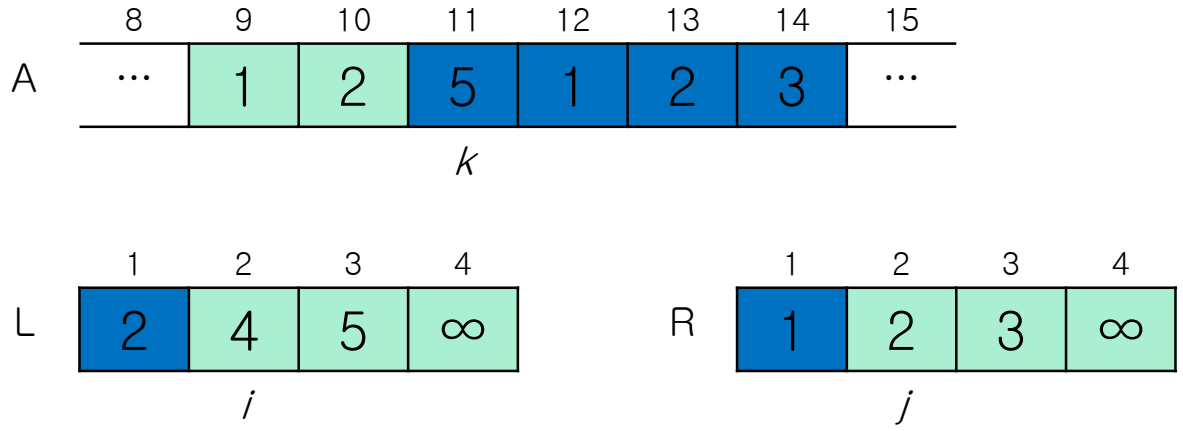
Operations of Merge(A, 9, 11, 14)



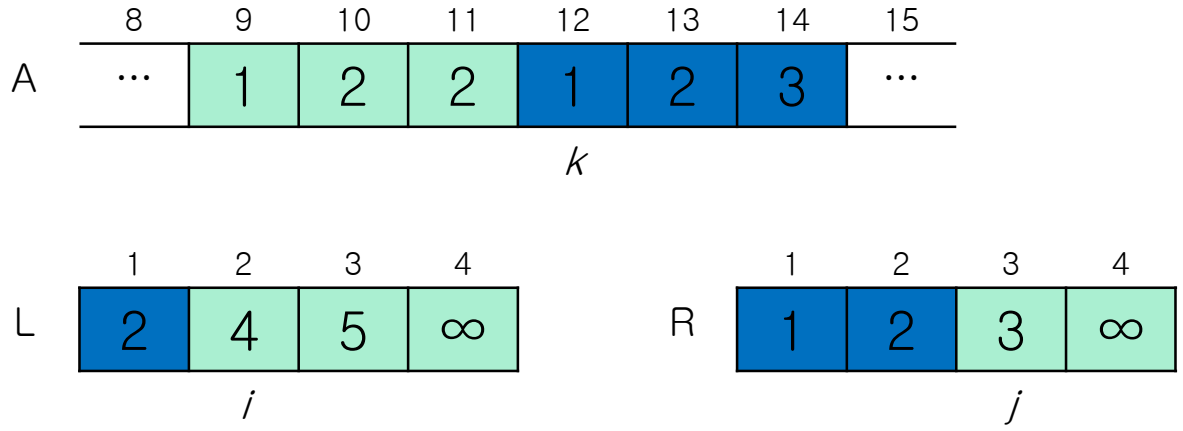
Operations of Merge(A, 9, 11, 14)



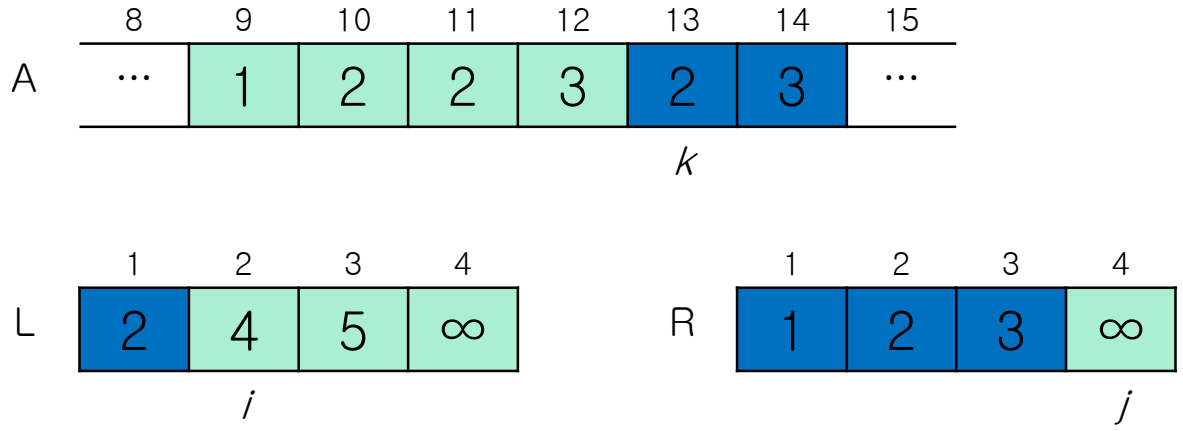
Operations of Merge(A, 9, 11, 14)



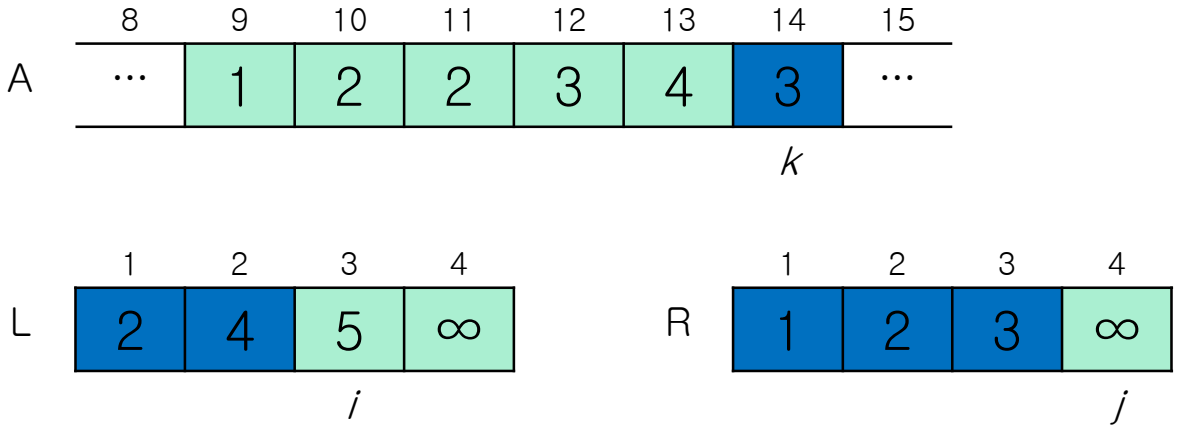
Operations of Merge(A, 9, 11, 14)



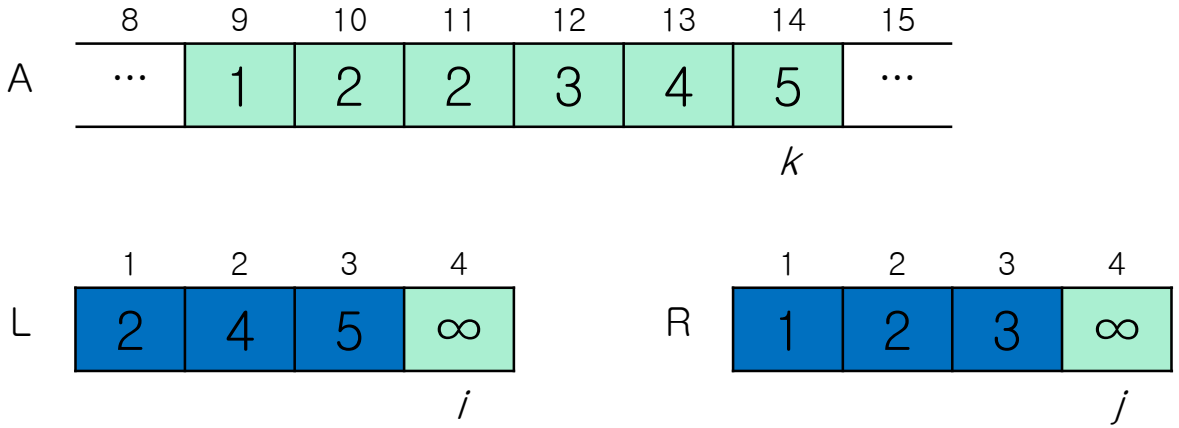
Operations of Merge(A, 9, 11, 14)

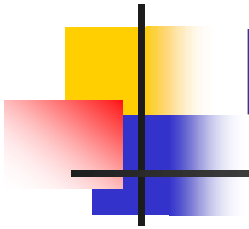


Operations of Merge(A, 9, 11, 14)



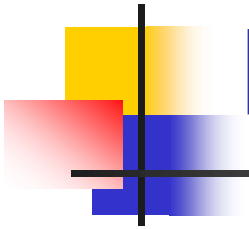
Operations of Merge(A, 9, 11, 14)





Merge Sort

- Merge Sort algorithm operates as follows
  - Divide: The divide step just computes the middle of the subarray, which takes Θ(1) time
  - Conquer: We recursively solve two subproblems, each of size n/2, which contributes 2T(n/2) to the running time.
  - Combine: We have already noted that the MERGE procedure on an n-element subarray takes Θ(n) time
- Thus, the recurrence for the worst-case running time T(n) of merge sort is
  - $T(1)=1$
    $T(n)=2T(n/2)+n$



Merge Sort

```
MERGE-SORT(A,p,r)
1       if p < r
2       q = ⌊(p+r)/2⌋
3           MERGE-SORT (A,p,q)
4           MERGE-SORT (A,q+1,r)
5           MERGE(A,p,q,r)
```

- Merge() is the procedure to merge two sorted lists.

Solving Recurrences

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n$$

Solving Recurrences

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n$$

$$= 2^2\, T\left(\frac{n}{2^2}\right) + 2n = 2^2\left(2T\left(\frac{n}{n^3}\right) + \frac{n}{2^2}\right) + 2n$$

Solving Recurrences

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n$$
$$= 2^2\, T\left(\frac{n}{2^2}\right) + 2n = 2^2\left(2T\left(\frac{n}{n^3}\right) + \frac{n}{2^2}\right) + 2n$$
$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

Solving Recurrences

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n$$

$$= 2^2\, T\left(\frac{n}{2^2}\right) + 2n = 2^2\left(2T\left(\frac{n}{n^3}\right) + \frac{n}{2^2}\right) + 2n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

...

$$= 2^k T\left(\frac{n}{2^k}\right) + kn$$

When $\frac{n}{2^k} = 1$,
we have n = 2^k and k = lg n

Solving Recurrences

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n \\
&= 2^2\, T\left(\frac{n}{2^2}\right) + 2n = 2^2\left(2T\left(\frac{n}{n^3}\right) + \frac{n}{2^2}\right) + 2n \\
&= 2^3 T\left(\frac{n}{2^3}\right) + 3n \\
&\ldots \\
&= 2^k T\left(\frac{n}{2^k}\right) + kn \qquad \text{When } \frac{n}{2^k} = 1, \\
&= nT(1) + n\lg n \qquad \text{we have n = } 2^k \text{ and k = lg n}
\end{aligned}
$$

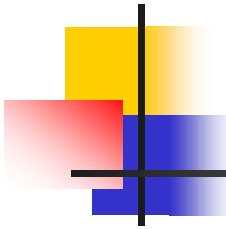
Solving Recurrences

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n \\
&= 2^2\, T\left(\frac{n}{2^2}\right) + 2n = 2^2\left(2T\left(\frac{n}{n^3}\right) + \frac{n}{2^2}\right) + 2n \\
&= 2^3 T\left(\frac{n}{2^3}\right) + 3n \\
&\ldots \\
&= 2^k T\left(\frac{n}{2^k}\right) + kn \qquad \text{When } \frac{n}{2^k} = 1, \\
&= nT(1) + n\lg n \qquad \text{we have n} = 2^k \text{ and k} = \lg \text{n} \\
&= n + n\lg n
\end{aligned}
$$

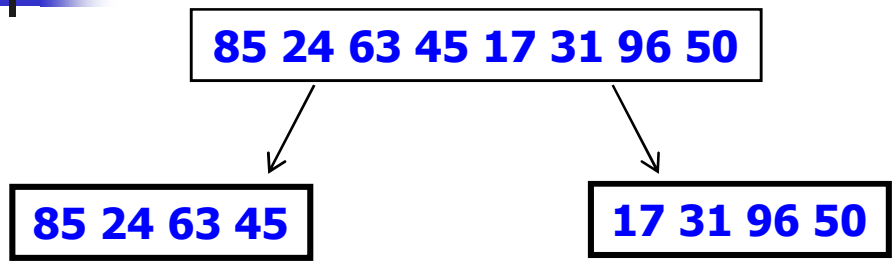


Operations of Merge Sort

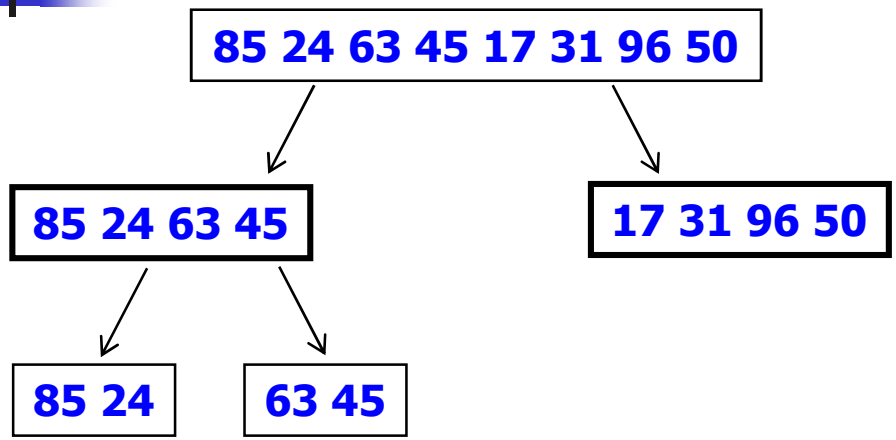
85 24 63 45 17 31 96 50

Initial sequence

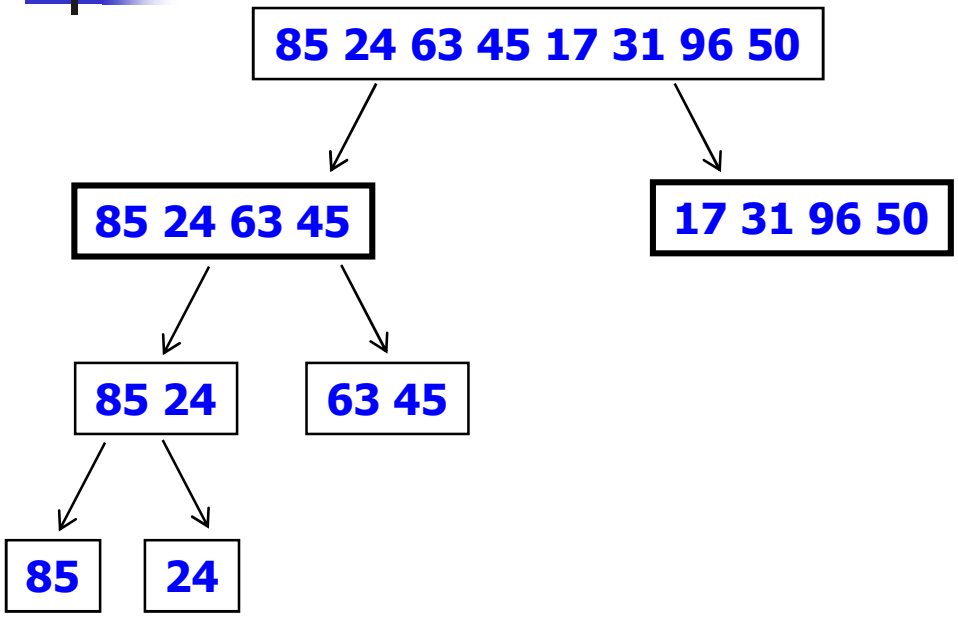
Operations of Merge Sort



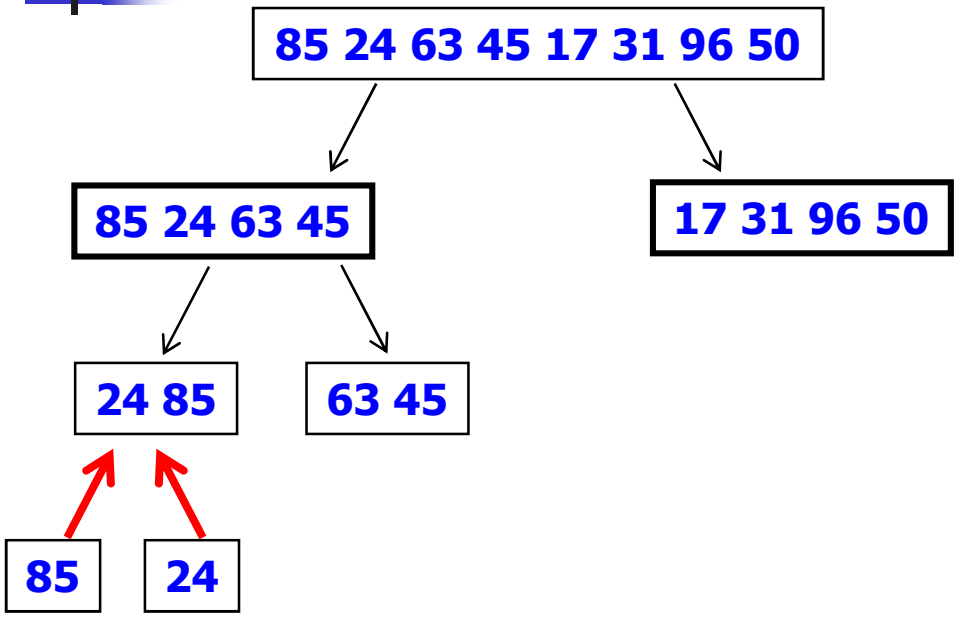
Operations of Merge Sort



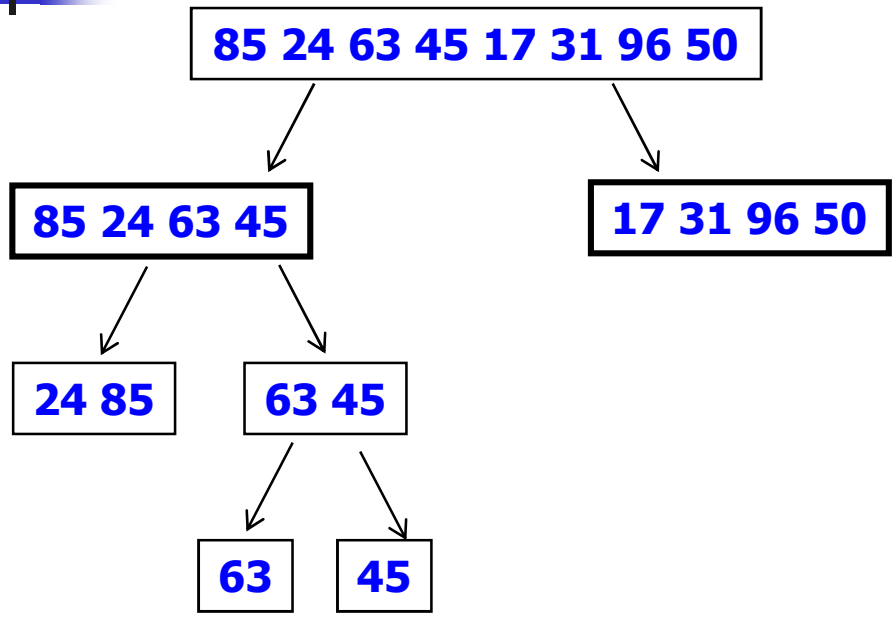
Operations of Merge Sort



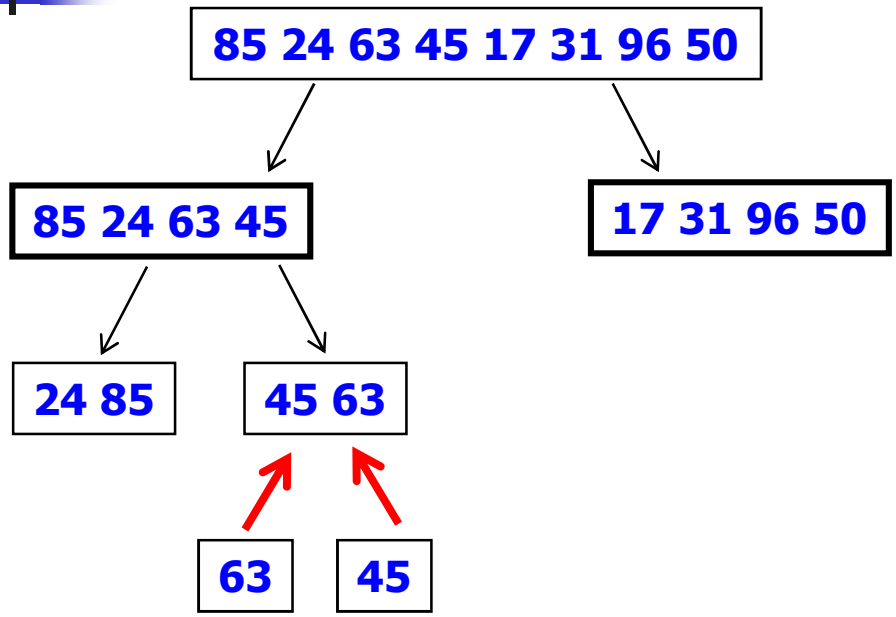
Operations of Merge Sort



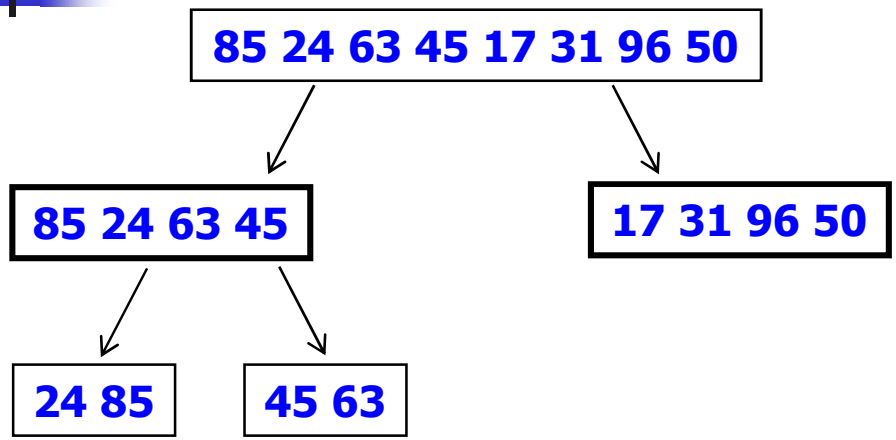
Operations of Merge Sort



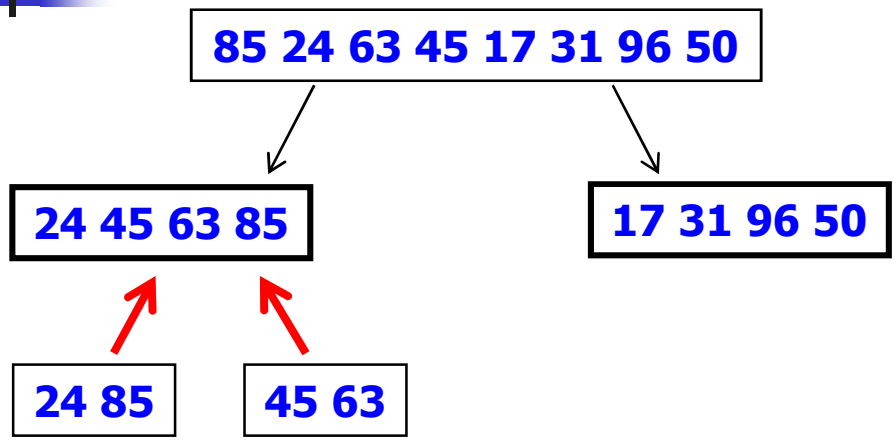
Operations of Merge Sort



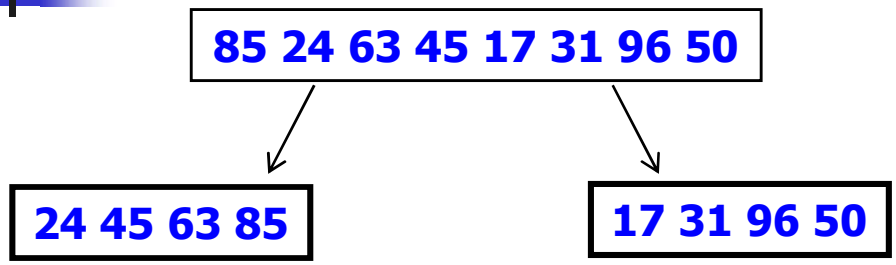
Operations of Merge Sort



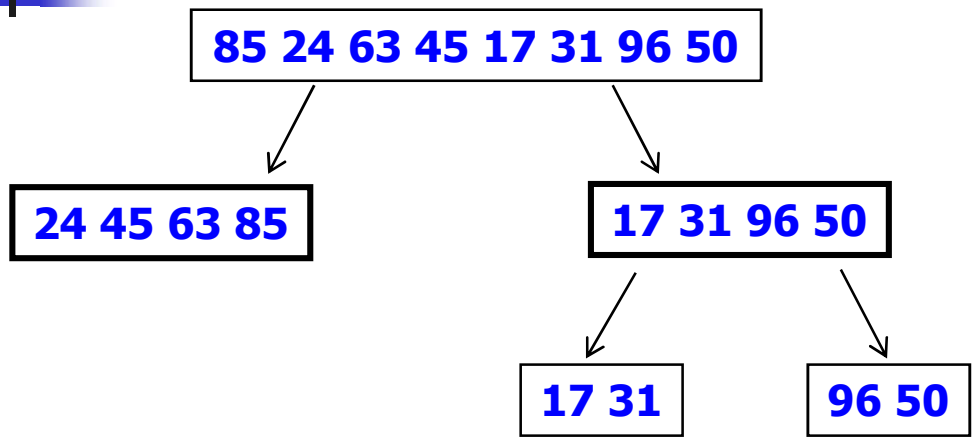
Operations of Merge Sort



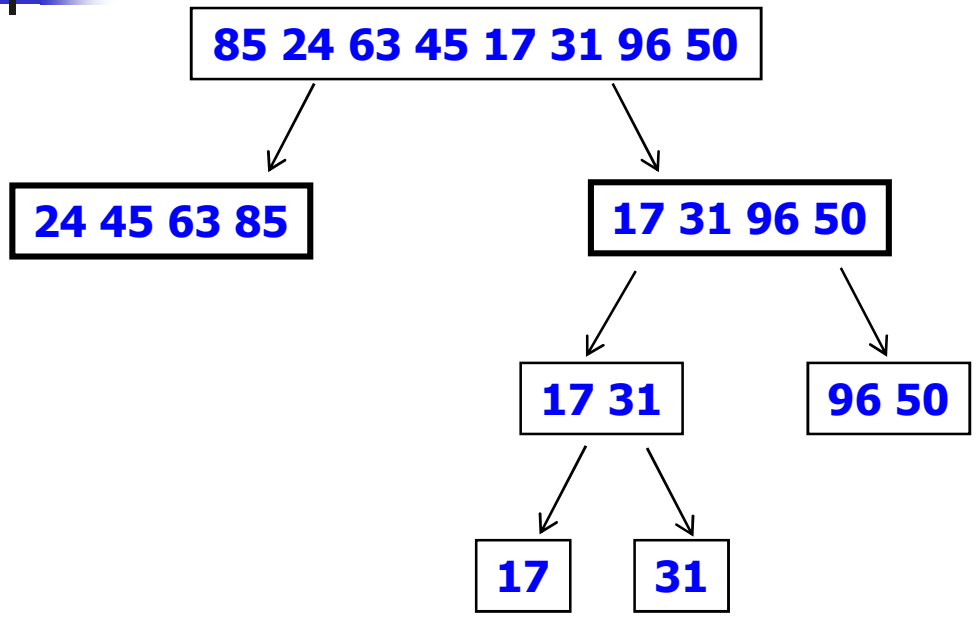
Operations of Merge Sort



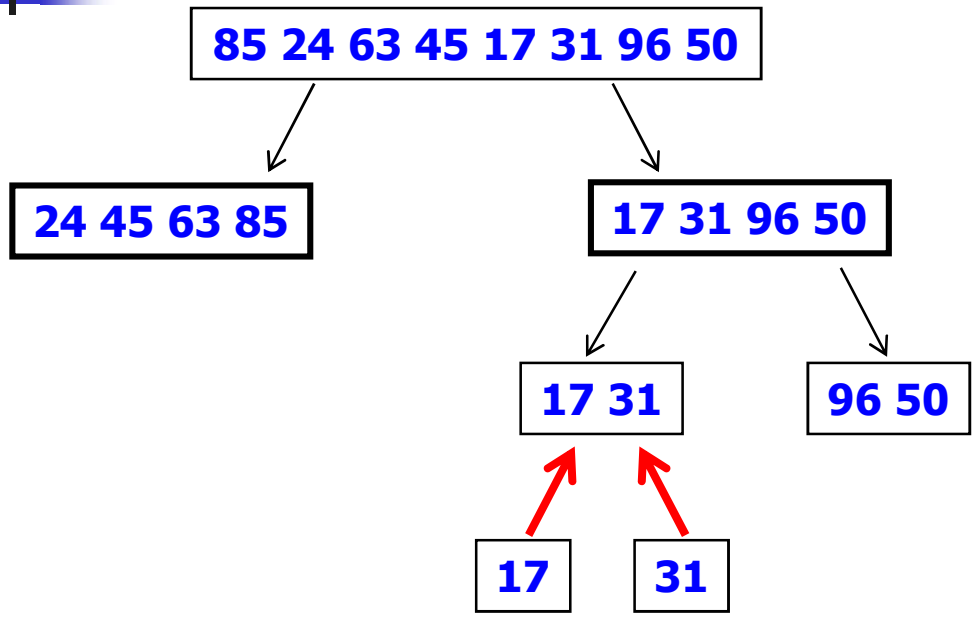
Operations of Merge Sort



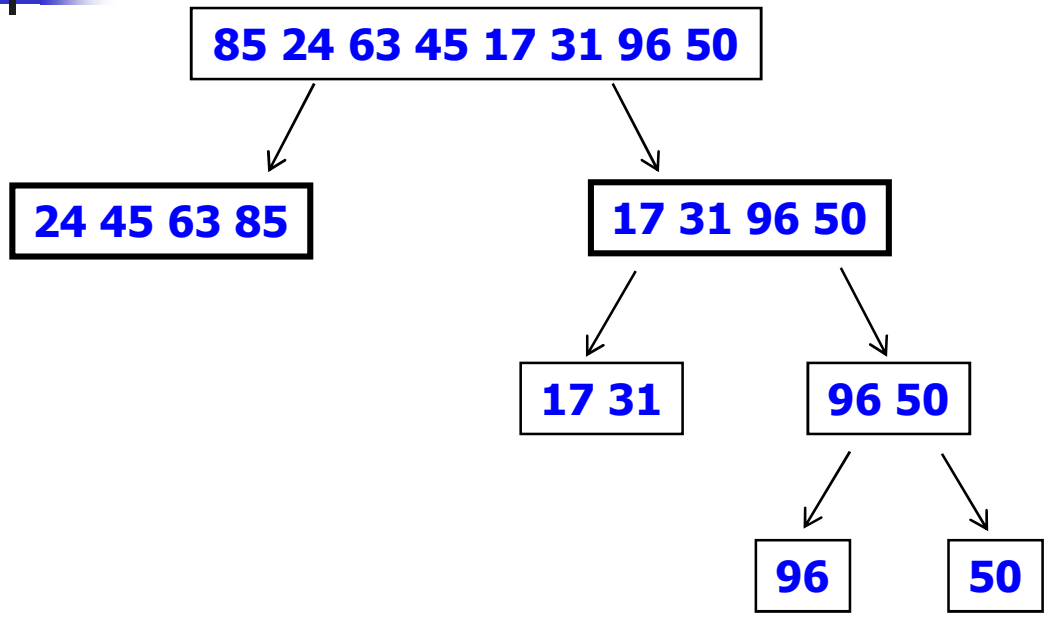
Operations of Merge Sort



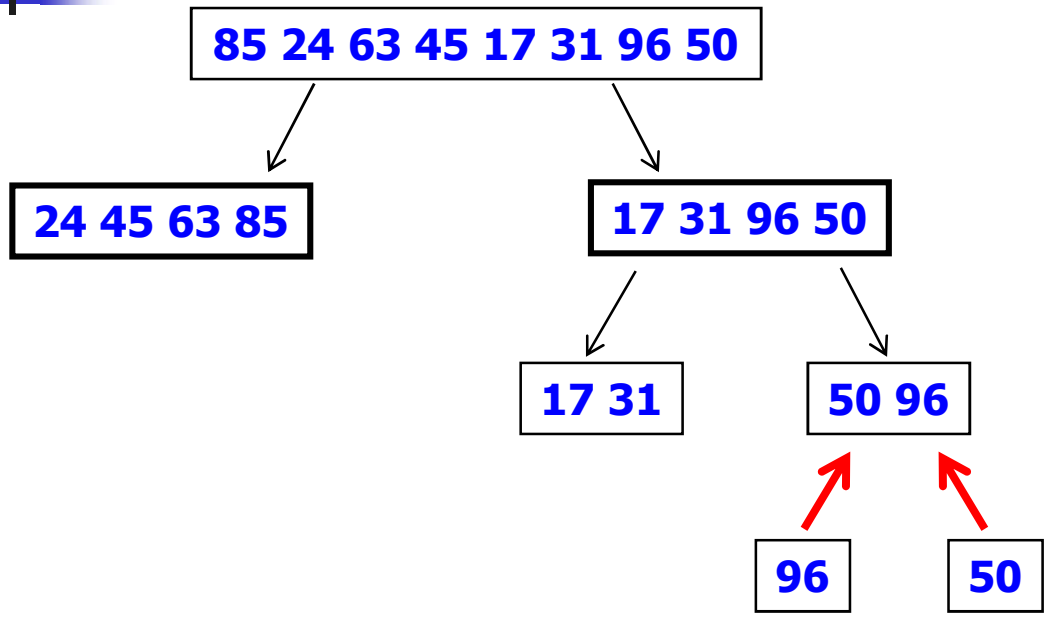
Operations of Merge Sort



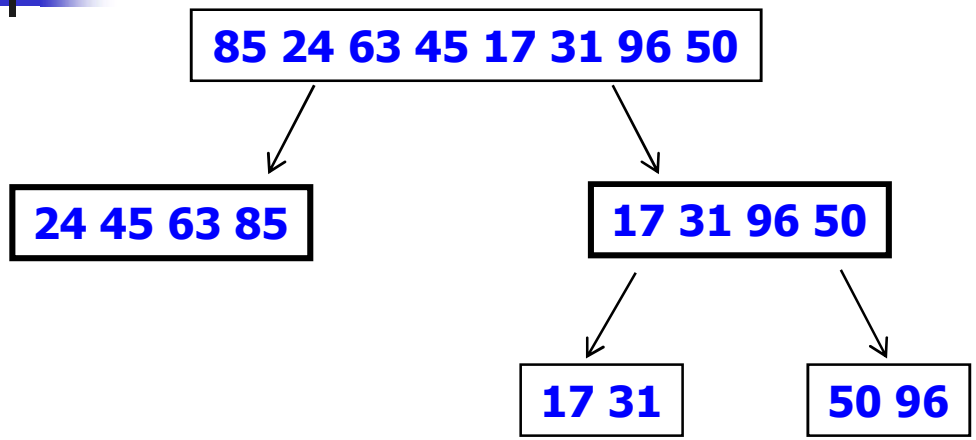
Operations of Merge Sort



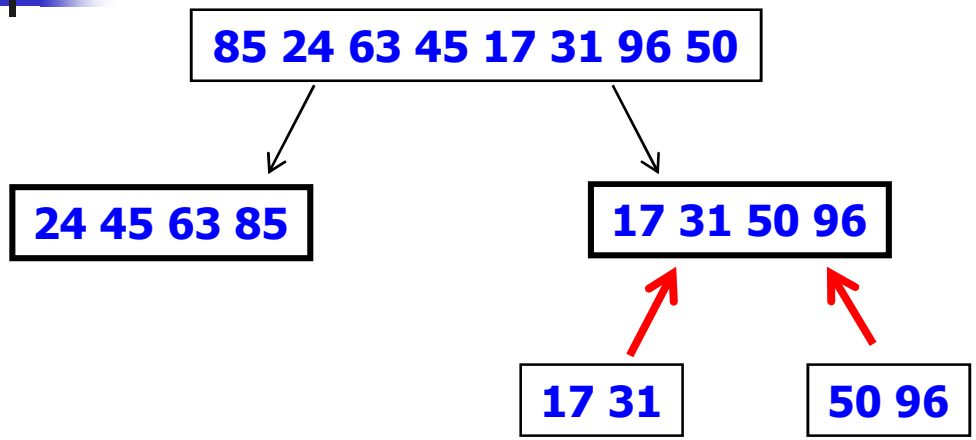
Operations of Merge Sort



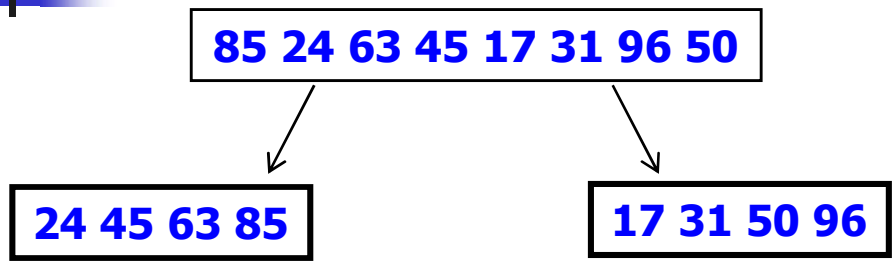
Operations of Merge Sort



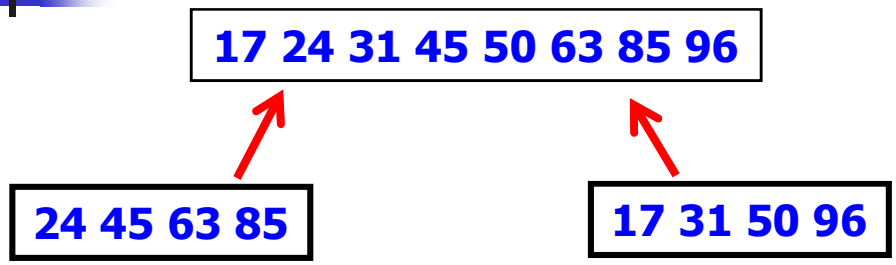
Operations of Merge Sort

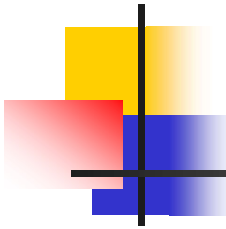


Operations of Merge Sort



Operations of Merge Sort



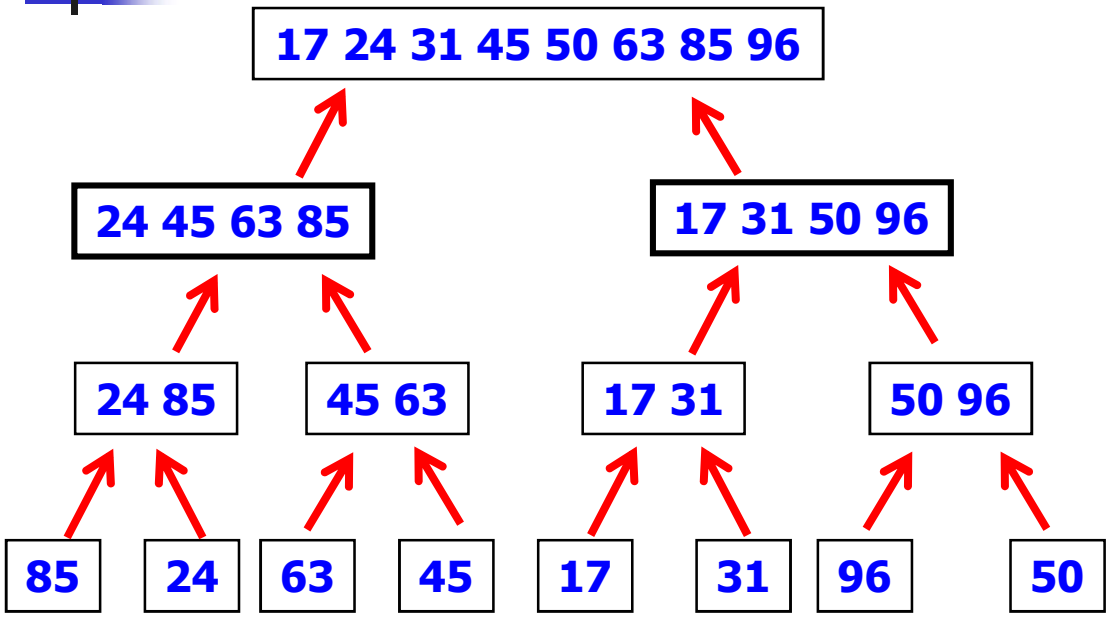


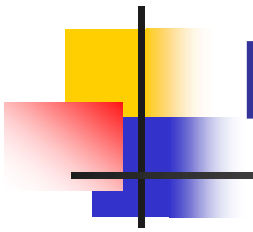
Operations of Merge Sort

17 24 31 45 50 63 85 96

Sorted sequence

Operations of Merge Sort

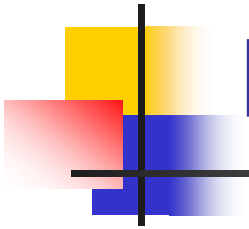




Merge two sorted lists

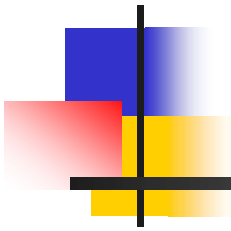
```
(define (merge x y)
  (cond ((null? x) y) ((null? y) x)
    ((<= (car x) (car y)) (cons (car x) (merge (cdr x) y)))
    (else (cons (car y) (merge x (cdr y))))))
```

- (merge '(1 3 5 7) '(2 4 5)) → (1 2 3 4 5 5 7)



Merge-sort

- (merge-sort ' (3 1 5 7 4)) → (1 3 4 5 7)



Other built-in Operators

Shorthand for nested car, cdr

- Scheme provides shorthands for expressions consisting of successive application of car and cdr
  - (car (cdr x)) -> (cadr x)
  - (cdr (car x)) -> (cdar x)
  - (car (cdr (car x))) -> (cadar x)

expression	shorthand	Value
x	x	((it seems that) you (like me)
(car x)	(car x)	(it seems that)
(car (car x))	(caar x)	it
(cdr (car x))	(cdar x)	(seems that)
(cdr x)	(cdr x)	(you (like) me)
(car (cdr x))	(cadr x)	you
(cdr (cdr x))	(cddr x)	((like) me)

The let construct

- (let ((x_1 E_1) (x_2 E_2) … (x_k E_k)) F)
  - E_1, E_2, … , E_k are all evaluated
  - Then F is evaluated with x_i
  - the result is the value of F
- The let constructor
  - Allows subexpressions to be named
  - Can be used to factor out common subexpression
    - ex) (let ((three-sq (square 3))) (+ three-sq three-sq))
      ; (+ (square 3) (square 3)) = 18

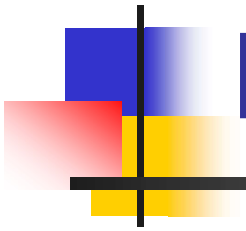
The let construct

- The sequential variant of the let

  (let* ((x_1 E_1) (x_2 E_2) … (x_k E_k)) F)

  - Binds x_i to the value of E_i before E_{i+1}
  - ex) (define x 0)

```
(let (( x 2 ) ( y x )) y )          ; 0
(let* (( x 2 ) ( y x )) y )         ; 2
```



List Manipulation

Getting the length of a list : `length`

- E.g. `(length '(1 2 3))` → `3`
- Length of an empty list) = 0
  - **`(length '()) ≡ 0`**
- Length of a nonempty list `(cons a y)` = length (y) + 1
  - `(length (cons a y))≡ (+ 1 length y))`
  - Let `(cons a y)` → `x`, `(cdr x)` → `y`

    `(length x) ≡ (+ 1 (length (cdr x)))`

```
(define (length x)
     (cond ((null? x) 0)
           (else (+ 1 (length (cdr x))))  ))
```

Appending two lists : append

- e.g.
  - `(append '() '(a b c d))` → `(a b c d )`
  - `(append '(a b c) '(d) )` → `(a b c d )`
  - `(cons 'a (append '(b c) '(d)))` → `(a b c d)`
- **`(append '() z)) ≡ z`**
- `(append (cons a y) z) ≡ (cons a (append y z))` → **`(append x z) ≡ (cons (car x) (append (cdr x) z)))))`**

```
(define (append x y)
      (cond ((null? x) y)
            (else (cons (car x) (append (cdr x) y)))))
```

Getting a flattened form of a list : `flatten`

- E.g.
  - `((a) ((b b)) (((c c c )))` → `(a b b c c c)`
- Use Scheme function `pair?`
  - Test whether its argument is a list
  - E.g. `(pair? 1)` → `#f`, `(pair? '(1))` → `#t`

```
(define (flatten x)
  (cond ((null? x) '())
        ((not (pair? x)) (list x))
        (else (append (flatten (car x) )
                      (flatten (cdr x)))))))
```

Mapping a function across list elements : `map`

- E.g.
  - `(map square '(1 2 3 4 5))` → `(1 4 9 16 25)`
    - `(define (square x) (* x x))`
  - `(map car '((a 1) (b 2) (c 3) (d 4)))` → `(a b c d)`

```
(define (map f x)
  (cond ((null? x) '())
        (else (cons (f (car x))
                    ( map f (cdr x))))))
```