Exercise

• Write a function `mergeSort` that accepts a list and uses the merge sort algorithm to produce a list with the same elements in non-decreasing order.
  
  - `mergeSort([5,2,8,4,9,6])` produces `[2,4,5,6,8,9]`

(a tricky recursive algorithm for us to practice...)
Merge sort

- **merge sort**: Repeatedly divides data in half, sorts each half, and combines the sorted halves into a sorted whole.

The algorithm:
- Divide the list into two roughly equal halves.
- Sort the left half.
- Sort the right half.
- Merge the two sorted halves into one sorted list.

- Often implemented recursively.
- An example of a "divide and conquer" algorithm.
  - Invented by John von Neumann in 1945
Merge sort example

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>22</td>
<td>18</td>
<td>12</td>
<td>-4</td>
<td>58</td>
<td>7</td>
<td>31</td>
<td>42</td>
</tr>
</tbody>
</table>

Diagram showing the merge sort process.
Merging sorted halves

<table>
<thead>
<tr>
<th>Subarrays</th>
<th>Next include</th>
<th>Merged array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>14 from left</td>
<td>14</td>
</tr>
<tr>
<td>i 1</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 from right</td>
<td>14 23</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>32 from left</td>
<td>14 23 32</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>41 from right</td>
<td>14 23 32 41</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>58 from right</td>
<td>14 23 32 41 58</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>67 from left</td>
<td>14 23 32 41 58 67</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>76 from left</td>
<td>14 23 32 41 58 67 76</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>85 from right</td>
<td>14 23 32 41 58 67 76 85</td>
</tr>
<tr>
<td>i2</td>
<td>i2</td>
<td>i</td>
</tr>
</tbody>
</table>
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted
public static void merge(int[] result, int[] left, int[] right) {
    int i1 = 0;  // index into left array
    int i2 = 0;  // index into right array

    for (int i = 0; i < result.length; i++) {
        if (i2 >= right.length ||
            (i1 < left.length && left[i1] <= right[i2])) {
            result[i] = left[i1];  // take from left
            i1++;
        } else {
            result[i] = right[i2];  // take from right
            i2++;
        }
    }
}
// Rearranges the elements of a into sorted order using
// the merge sort algorithm (recursive).
public static void mergeSort(int[] a) {
    if (a.length >= 2) {
        // split array into two halves
        int[] left = Arrays.copyOfRange(a, 0, a.length/2);
        int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

        // sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        merge(a, left, right);
    }
}
Suggested helpers

• Write a function `split` that accepts a list and produces a tuple of two lists representing its even and odd indexes.
  
  - `split([12, ~3, 0, 19, 1])` produces `([12, 0, 1], [~3, 19])`

• Write a function `merge` that accepts two sorted lists and produces a new merged sorted list.
  
  - `merge([4, 9, 11], [~3, 2, 10])` produces `[^3, 2, 4, 9, 10, 11]`
Helper solutions

(* Splits a list into 2 sublists of its even/odd indexes. *)
fun split([]) = ([], [])
| split([x]) = ([x], [])
| split(first :: second :: rest) = 
  let val (l1, l2) = split(rest)
  in (first :: l1, second :: l2)
  end;

(* Merges sorted L1 and L2 into a sorted whole. *)
fun merge([], L2) = L2
| merge(L1, []) = L1
| merge(L1 as first1 :: rest1, L2 as first2 :: rest2) = 
  if first1 < first2
  then first1 :: merge(rest1, L2)
  else first2 :: merge(L1, rest2);
Merge sort solution

(* Rearranges the elements of the given list to be in non-decreasing order using the merge sort algorithm. *)

fun mergeSort([]) = []
| mergeSort([[value]]) = [value]
| mergeSort(lst) =
  let
    val (left, right) = split(lst)
  in
    merge(mergeSort(left), mergeSort(right))
  end;
Efficiency exercise

• Write a function called reverse that accepts a list and produces the same elements in the opposite order.
  - reverse([6, 2, 9, 7]) produces [7,9,2,6]

• Write a function called range that accepts a maximum integer value \( n \) and produces the list \([1, 2, 3, \ldots, n-1, n]\). Produce an empty list for all numbers less than 1.
  - Example: range(5) produces [1,2,3,4,5]
Flawed solutions

• These solutions are correct; but they have a problem...

fun reverse([],) = []
|   reverse(first :: rest) = reverse(rest) @ [first];

fun range(0) = []
|   range(n) = range(n - 1) @ [n];
Efficiency of the @ operator

val x = [2, 4, 7];
val y = [5, 3];
val a = 9 :: x;
val z = x @ y;

• The :: operator is fast: O(1)
  ▪ simply creates a link from the first element to front of right

• The @ operator is slow: O(n)
  ▪ must walk/copy the left list and then append the right one
  ▪ using @ in a recursive function n times: function is O(n^2)
Fixing inefficient recursion

• How can we improve the inefficient range code?

```plaintext
fun range(0) = []
| range(n) = range(n - 1) :: [n];
```

• *Hint:* Replace `@` with `::` as much as possible.
fun range(n) =
let
  fun helper(min, max) =
    if min = max then [min]
    else min :: helper(min + 1, max)
  in
  helper(1, n)
end;
The `fibonacci` function we wrote previously is also inefficient, for a different reason.

- It makes an exponential number of recursive calls.

- Example: `fibonacci(5)`
  - `fibonacci(4)`
    - `fibonacci(3)`
      » `fibonacci(2)`
      » `fibonacci(1)`
    - `fibonacci(2)`
  - `fibonacci(3)`
    - `fibonacci(2)`
    - `fibonacci(1)`

- How can we fix it to make fewer (O(n)) calls?
/ Returns the nth Fibonacci number.  
// Precondition: n >= 1
public static int fibonacci(int n) {
    if (n == 1 || n == 2) {
        return 1;
    }
    int curr = 1; // the 2 most recent Fibonacci numbers
    int prev = 1;

    // k stores what fib number we are on now
    for (int k = 2; k < n; k++) {
        int next = curr + prev; // advance to next
        prev = curr; // Fibonacci number
        curr = next;
    }
    return curr;
}
Efficient Fibonacci in ML

(* Returns the nth Fibonacci number. 
   Precondition: n >= 1 *)
fun fib(1) = 1 |
   fib(2) = 1 |
   fib(n) = |
   let |
      fun helper(k, prev, curr) = |
         if k = n then curr |
         else helper(k + 1, curr, prev + curr) |
      in |
         helper(2, 1, 1) |
      end;