Solution 1: Quicksort

The following is a simple implementation of the *quicksort* algorithm, as shown in the lecture. It notably departs from the lecture by choosing the pivot element as the middle element of the current subarray.

```r
quicksort <- function(v) {
  partition <- function(pivot, from, to) {
     repeat {
      while (v[from] < pivot) from <- from + 1
      while (v[to] > pivot) to <- to - 1
      if (from >= to) return(to)

      # swap v[from] and v[to]
      tmp <- v[from]
      # !! we are using <<- to write to the variable 'v' in the parent scope
      v[from] <<- v[to]
      v[to] <<- tmp

      from <- from + 1
      to <- to - 1
    }
  }
  qs.recursive <- function(from, to) {
     if (from >= to) return()
    pivot.pos <- (from + to) %/% 2
    pivot <- v[pivot.pos]
    pos <- partition(pivot, from, to)
    qs.recursive(from, pos)
    qs.recursive(pos + 1, to)
  }
  qs.recursive(1, length(v))
  v
}
```

(a) The following table lists the calls performed recursively by the `quicksort` function, when called with the input vector $[1, 4, 3]$.

call	**v** (input)	**return value**
qs.recursive(1,3)	[1,4,3]	
│ partition(4,1,3)	[1,4,3]	2
│ qs.recursive(1,2)	[1,3,4]	
│ │ partition(1,1,2)	[1,3,4]	1
│ │ qs.recursive(1,1)	[1,3,4]	
│ │ qs.recursive(2,2)	[1,3,4]	
│ qs.recursive(3,3)	[1,3,4]	

(Manually) write down this table for the input vectors $[2, 1]$, $[1, 2, 3, 4, 5]$, and $[2, 5, 1, 3, 4]$.

(b) Although fast in the average case, quicksort has worst-case performance $\Theta(n^2)$. What is the property of the input vector that causes this worst-case behavior? Write a function `quicksortAdversarial(n)` that generates an input vector of length n that causes quicksort to run in $\Theta(n^2)$ time.

(a) Students should write this manually, but the following code can be used to generate the table.

```r
recursion.table <- NULL
resetRecursionTable <- function() {
  recursion.table <<- data.frame(
    call = character(0), v = character(0), return.value = character(0)
  )
}

appendRecursionTable <- function(callname, args, v, depth, return.value = NA) {
  table.row <- data.frame(
    call = sprintf("%s%s(%s)",
      paste0(rep("| ", depth), collapse = ""),
      callname,
      paste(args, collapse = ", ")),
    v = sprintf("[%s]", paste0(v, collapse = " ")),
    return.value = ifelse(is.na(return.value), "", as.character(return.value))
  )
  recursion.table <<- rbind(recursion.table, table.row)
}

quicksort <- function(v) {
  partition <- function(pivot, from, to) {
     repeat {
      while (v[from] < pivot) from <- from + 1
      while (v[to] > pivot) to <- to - 1
      if (from >= to) return(to)

      # swap v[from] and v[to]
      tmp <- v[from]
      # !! we are using <<- to write to the variable 'v' in the parent scope
      v[from] <<- v[to]
      v[to] <<- tmp

      from <- from + 1
      to <- to - 1
    }
  }
  qs.recursive <- function(from, to, depth) {
    appendRecursionTable("qs.recursive", c(from, to), v, depth)
     if (from >= to) return()
    pivot.pos <- (from + to) %/% 2
    pivot <- v[pivot.pos]
    vin <- v
    pos <- partition(pivot, from, to)
    appendRecursionTable("partition", c(pivot, from, to), vin, depth + 1, pos)

    qs.recursive(from, pos, depth + 1)
    qs.recursive(pos + 1, to, depth + 1)
  }

  qs.recursive(1, length(v), 0)
  v
}

resetRecursionTable()
. <- quicksort(c(1, 4, 3))
print(recursion.table, right = FALSE)

#>   call                   v       return.value
#> 1 qs.recursive(1, 3)     [1 4 3]
#> 2 | partition(4, 1, 3)   [1 4 3] 2
#> 3 | qs.recursive(1, 2)   [1 3 4]
#> 4 | | partition(1, 1, 2) [1 3 4] 1
#> 5 | | qs.recursive(1, 1) [1 3 4]
#> 6 | | qs.recursive(2, 2) [1 3 4]
#> 7 | qs.recursive(3, 3)   [1 3 4]

resetRecursionTable()
. <- quicksort(c(2, 1))
print(recursion.table, right = FALSE)

#>   call                 v     return.value
#> 1 qs.recursive(1, 2)   [2 1]
#> 2 | partition(2, 1, 2) [2 1] 1
#> 3 | qs.recursive(1, 1) [1 2]
#> 4 | qs.recursive(2, 2) [1 2]

resetRecursionTable()
. <- quicksort(c(1, 2, 3, 4, 5))
print(recursion.table, right = FALSE)

#>    call                   v           return.value
#> 1  qs.recursive(1, 5)     [1 2 3 4 5]
#> 2  | partition(3, 1, 5)   [1 2 3 4 5] 3
#> 3  | qs.recursive(1, 3)   [1 2 3 4 5]
#> 4  | | partition(2, 1, 3) [1 2 3 4 5] 2
#> 5  | | qs.recursive(1, 2) [1 2 3 4 5]
#> 6  | | | partition(1, 1, 2) [1 2 3 4 5] 1
#> 7  | | | qs.recursive(1, 1) [1 2 3 4 5]
#> 8  | | | qs.recursive(2, 2) [1 2 3 4 5]
#> 9  | | qs.recursive(3, 3)   [1 2 3 4 5]
#> 10 | qs.recursive(4, 5)     [1 2 3 4 5]
#> 11 | | partition(4, 4, 5)   [1 2 3 4 5] 4
#> 12 | | qs.recursive(4, 4)   [1 2 3 4 5]
#> 13 | | qs.recursive(5, 5)   [1 2 3 4 5]

resetRecursionTable()
. <- quicksort(c(2, 5, 1, 3, 4))
print(recursion.table, right = FALSE)

#>    call                     v           return.value
#> 1  qs.recursive(1, 5)       [2 5 1 3 4]
#> 2  | partition(1, 1, 5)     [2 5 1 3 4] 1
#> 3  | qs.recursive(1, 1)     [1 5 2 3 4]
#> 4  | qs.recursive(2, 5)     [1 5 2 3 4]
#> 5  | | partition(2, 2, 5)   [1 5 2 3 4] 2
#> 6  | | qs.recursive(2, 2)   [1 2 5 3 4]
#> 7  | | qs.recursive(3, 5)   [1 2 5 3 4]
#> 8  | | | partition(3, 3, 5)   [1 2 5 3 4] 3
#> 9  | | | qs.recursive(3, 3)   [1 2 3 5 4]
#> 10 | | | qs.recursive(4, 5)   [1 2 3 5 4]
#> 11 | | | | partition(5, 4, 5) [1 2 3 5 4] 4
#> 12 | | | | qs.recursive(4, 4) [1 2 3 4 5]
#> 13 | | | | qs.recursive(5, 5) [1 2 3 4 5]
```

(b) The worst-case performance of $\Theta(n^2)$ for quicksort occurs when the partitioning step consistently produces highly unbalanced partitions. For an array of size n, if one partition has size close to 0 and the other has size close to $n-1$, the recurrence relation for the runtime becomes $T(n) = T(n-1) + \Theta(n)$, which solves to

$T(n) = \Theta(n^2)$.

This situation arises when the chosen pivot element is always the smallest or largest element in the current subarray. The provided implementation selects the pivot as the middle element of the subarray, `v[(from + to) %/% 2]`. Therefore, an adversarial input must be an array where, for every subarray that the algorithm considers, the middle element is one of the extreme values.

We can recursively construct the adversarial input for size n from the one for $n-1$. The reasoning is as follows: Given input of the form $[v_1, \ldots, v_{\lfloor n/2 \rfloor -1}, v_{\lfloor n/2 \rfloor}, v_{\lfloor n/2 \rfloor +1}, \ldots, v_n]$, the worst case happens when $v_{\lfloor n/2 \rfloor}$ is the smallest or largest element. Suppose we opt to use the smallest element. Then, `partition` will reorder the array to be $[v_{\lfloor n/2 \rfloor}, v_2, \ldots, v_{\lfloor n/2 \rfloor -1}, v_1, v_{\lfloor n/2 \rfloor +1}, \ldots, v_n]$ and call `qs.recursive` on the first part (which returns immediately) as well as the subarray $[v_2, \ldots, v_{\lfloor n/2 \rfloor -1}, v_1, v_{\lfloor n/2 \rfloor +1}, \ldots, v_n]$. By construction, we want $[v_2, \ldots, v_{\lfloor n/2 \rfloor -1}, v_1, v_{\lfloor n/2 \rfloor +1}, \ldots, v_n]$ to be the adversarial input of length $n-1$.

We can run the above reasoning in reverse order: Get the adversarial input of length $n-1$, add a new smallest element to the front, and swap it with the element in the middle position. The following solution does that a bit lazily, by calling itself and adding 1 to every element. It is not the most efficient solution possible, but it is easy to understand.

```r
quicksortAdversarial <- function(n) {
  if (n == 1) return(1)
  if (n == 2) return(c(2, 1))
  v <- quicksortAdversarial(n - 1)
  v <- v + 1  # we add 1 to the front and want it to be the new smallest element
  swap.position <- (n + 1) %/% 2 - 1
  prefix <- v[swap.position]
  v[swap.position] <- 1
  c(prefix, v)
}

# Example of a generated adversarial vector for n=10
quicksortAdversarial(3)

#> [1] 3 1 2

quicksortAdversarial(5)

#> [1] 2 5 1 3 4

quicksortAdversarial(10)

#>  [1] 10  3  7  5  1  2  4  6  8  9

## The call tale looks deceptively benign, but note that each partition()-call
## costs ‘to - from‘ ~ O(n) time.
resetRecursionTable()
. <- quicksort(quicksortAdversarial(10))
print(recursion.table, right = FALSE)

#>    call                                        v                        return.value
#> 1  qs.recursive(1, 10)                         [10 3 7 5 1 2 4 6 8 9]
#> 2  | partition(1, 1, 10)                       [10 3 7 5 1 2 4 6 8 9] 1
#> 3  | qs.recursive(1, 1)                        [1 3 7 5 10 2 4 6 8 9]
#> 4  | qs.recursive(2, 10)                       [1 3 7 5 10 2 4 6 8 9]
#> 5  | | partition(2, 2, 10)                     [1 3 7 5 10 2 4 6 8 9] 2
#> 6  | | qs.recursive(2, 2)                      [1 2 7 5 10 3 4 6 8 9]
#> 7  | | qs.recursive(3, 10)                     [1 2 7 5 10 3 4 6 8 9]
#> 8  | | | partition(3, 3, 10)                   [1 2 7 5 10 3 4 6 8 9] 3
#> 9  | | | qs.recursive(3, 3)                    [1 2 3 5 10 7 4 6 8 9]
#> 10 | | | qs.recursive(4, 10)                   [1 2 3 5 10 7 4 6 8 9]
#> 11 | | | | partition(4, 4, 10)                 [1 2 3 5 10 7 4 6 8 9] 4
#> 12 | | | | qs.recursive(4, 4)                  [1 2 3 4 10 7 5 6 8 9]
#> 13 | | | | qs.recursive(5, 10)          [1 2 3 4 10 7 5 6 8 9]
#> 14 | | | | | partition(5, 5, 10)        [1 2 3 4 10 7 5 6 8 9] 5
#> 15 | | | | | qs.recursive(5, 5)         [1 2 3 4 5 7 10 6 8 9]
#> 16 | | | | | qs.recursive(6, 10)        [1 2 3 4 5 7 10 6 8 9]
#> 17 | | | | | | partition(6, 6, 10)      [1 2 3 4 5 7 10 6 8 9] 6
#> 18 | | | | | | qs.recursive(6, 6)       [1 2 3 4 5 6 10 7 8 9]
#> 19 | | | | | | qs.recursive(7, 10)      [1 2 3 4 5 6 10 7 8 9]
#> 20 | | | | | | | partition(7, 7, 10)    [1 2 3 4 5 6 10 7 8 9] 7
#> 21 | | | | | | | qs.recursive(7, 7)     [1 2 3 4 5 6 7 10 8 9]
#> 22 | | | | | | | qs.recursive(8, 10)    [1 2 3 4 5 6 7 10 8 9]
#> 23 | | | | | | | | partition(8, 8, 10)  [1 2 3 4 5 6 7 10 8 9] 8
#> 24 | | | | | | | | qs.recursive(8, 8)   [1 2 3 4 5 6 7 8 10 9]
#> 25 | | | | | | | | qs.recursive(9, 10)  [1 2 3 4 5 6 7 8 10 9]
#> 26 | | | | | | | | | partition(10, 9, 10) [1 2 3 4 5 6 7 8 10 9] 9
#> 27 | | | | | | | | | qs.recursive(9, 9)   [1 2 3 4 5 6 7 8 9 10]
#> 28 | | | | | | | | | qs.recursive(10, 10) [1 2 3 4 5 6 7 8 9 10]
```

Solution 2: Radix Sort

In this exercise, we will gradually implement a radix sort algorithm. Radix sort repeatedly sorts a vector of numbers by the digits of these numbers. Here we will do "least significant digit" (LSD) radix sort, which sorts the vector by the digits of the numbers from least to most significant. We are using digits in base 10 for clarity, but the algorithm can be generalized to any base. In practice, one would typically use "digits" to base 256 by splitting a given set of numbers into bytes, since these can be handled efficiently by modern CPUs. It is even possible to sort floating point numbers like this, since their bit patterns preserve the order even when treating them as integers – one only needs to take extra care to handle the sign bit.

(a) Write a function `sortCount(v)` that implements a counting sort algorithm. Counting sort works by pre-allocating a counting vector with one entry for each distinct value one expects to see in the input vector, and then counting the number of times each value appears in the vector. The vector can then be reconstructed in sorted order from the counts. It should take an integer valued vector v with values between 0 and 9 (inclusive) and return it in sorted order. The algorithm should run in time $O(\text{length}(v))$.

Examples:

```r
sortCount(c(1, 9, 2))
#> [1] 1 2 9
sortCount(c(1, 2, 1, 2))
#> [1] 1 1 2 2
sortCount(c(1:9, 9:1))
#> [1] 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
```

(b) Write a function `orderCount(v)` that uses a counting sort algorithm to get the *order* of elements in the vector v. The order is a vector of the same length as v so that `v[orderCount(v)]` is sorted. Moreover, the order should be *stable*, meaning that elements with the same value should appear in the same order as they do in the original vector. The algorithm should work with vectors of integer values between 0 and 9 (inclusive) and run in time $O(\text{length}(v))$.

Hint: The trick here is to create a counting vector just as in (a) and then taking the cumulative sum (`cumsum`) of the counts. The resulting vector will contain the index of the last occurrence of each value in the input vector. You can iterate downward from the last index to the first, and use the cumsum of the counts to place each element in the correct position in the output vector. Then decrement the cumsum of the counts whenever you place an element, so that it still contains the index where the next occurrence of the value would go.

Examples:

```r
orderCount(c(1, 9, 2))
#> [1] 1 3 2
orderCount(c(1, 2, 1, 2))  # note stable order: 2 4 1 3 would be wrong!
#> [1] 1 3 2 4
```

```r
orderCount(c(0:4, 4:0))
#> [1]  1 10  2  9  3  8  4  7  5  6
orderCount(c(2, 2, 2, 1, 1, 1))  # note the stable order again.
#> [1] 4 5 6 1 2 3
```

(c) Write a function `sortRadix(v)` that iteratively uses the `orderCount` function on the digits of the elements of v to sort the vector. The algorithm should work with vectors of non-negative integer values and run in time $O(\text{length}(v) \cdot \text{maxDigits}(v))$. Begin by initializing a matrix of digits as follows:

```r
getDigitMatrix <- function(v) {
  maxval <- max(v)
  maxdigits <- ceiling(log10(maxval + 1))
  maxdigits <- max(maxdigits, 1)  # '0' is a single digit, but the above would give 0
  digit.strings <- sprintf("%0*d", maxdigits, v)
  matrix(
    as.integer(unlist(strsplit(digit.strings, ""))),
    nrow = length(v),
    byrow = TRUE
  )
}
# Example:
getDigitMatrix(c(123, 46, 0, 20))

#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    0    4    6
#> [3,]    0    0    0
#> [4,]    0    2    0
```

Then, for each digit position, starting with the least significant digit, use the `orderCount` function to sort the vector, as well as the rows of the digit matrix. The algorithm should return the sorted vector.

Examples:

```r
sortRadix(c(123, 46, 0, 20))
#> [1]   0  20  46 123
sortRadix(c(1, 2, 1, 2, 1, 2))
#> [1] 1 1 1 2 2 2
```

(a)
```r
sortCount <- function(v) {
  checkmate::assertIntegerish(v, lower = 0, upper = 9, any.missing = FALSE)
  counts <- integer(10)
  for (i in seq_along(v)) {
    counts[v[i] + 1] <- counts[v[i] + 1] + 1L
  }
  rep(0:9, counts)
  # # if you feel like 'rep()' is cheating because it is not 'low-level' enough,
  # # the following does it manually:
  # out <- integer(length(v))
  # ends <- cumsum(counts)
  # starts <- ends - counts + 1L
  # for (i in 0:9) {
  #   if (starts[i + 1L] <= ends[i + 1L]) {
  #     out[starts[i + 1L]:ends[i + 1L]] <- i
  #   }
  # }
  # out
}
```

```r
sortCount(c(1, 9, 2))
#> [1] 1 2 9

sortCount(c(1, 2, 1, 2))

#> [1] 1 1 2 2

sortCount(c(1:9, 9:1))

#>  [1] 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
```

(b)
```r
orderCount <- function(v) {
  checkmate::assertIntegerish(v, lower = 0, upper = 9, any.missing = FALSE)
  if (!length(v)) return(integer(0))  # the following only works for non-empty v
  counts <- integer(10)
  for (i in seq_along(v)) {
    counts[v[i] + 1] <- counts[v[i] + 1] + 1
  }
  ends <- cumsum(counts)
  out <- integer(length(v))
  for (i in length(v):1) {  # we made sure length(v) > 0, so this is safe
    out[ends[v[i] + 1]] <- i
    ends[v[i] + 1] <- ends[v[i] + 1] - 1
  }
  out
}

orderCount(c(1, 9, 2))
#> [1] 1 3 2

orderCount(c(1, 2, 1, 2))  # note stable order: 2 4 1 3 would be wrong!

#> [1] 1 3 2 4

orderCount(c(0:4, 4:0))

#>  [1]  1 10  2  9  3  8  4  7  5  6

orderCount(c(2, 2, 2, 1, 1, 1))  # note the stable order again.

#> [1] 4 5 6 1 2 3
```

(c)
```r
getDigitMatrix <- function(v) {
  maxval <- max(v)
  maxdigits <- ceiling(log10(maxval + 1))
  maxdigits <- max(maxdigits, 1)  # '0' is a single digit, but the above would give 0
  digit.strings <- sprintf("%0*d", maxdigits, v)
  matrix(
    as.integer(unlist(strsplit(digit.strings, ""))),
    nrow = length(v),
    byrow = TRUE
  )
}

sortRadix <- function(v) {
  checkmate::assertIntegerish(v, lower = 0, any.missing = FALSE)
  if (!length(v)) return(integer(0))  # the following only works for non-empty v
  digitmatrix <- getDigitMatrix(v)
  for (i in ncol(digitmatrix):1) {
    order <- orderCount(digitmatrix[, i])
    v <- v[order]
    digitmatrix <- digitmatrix[order, ]
  }
  v
}

sortRadix(c(123, 46, 0, 20))
#> [1]   0  20  46 123

sortRadix(c(1, 2, 1, 2, 1, 2))
#> [1] 1 1 1 2 2 2
```