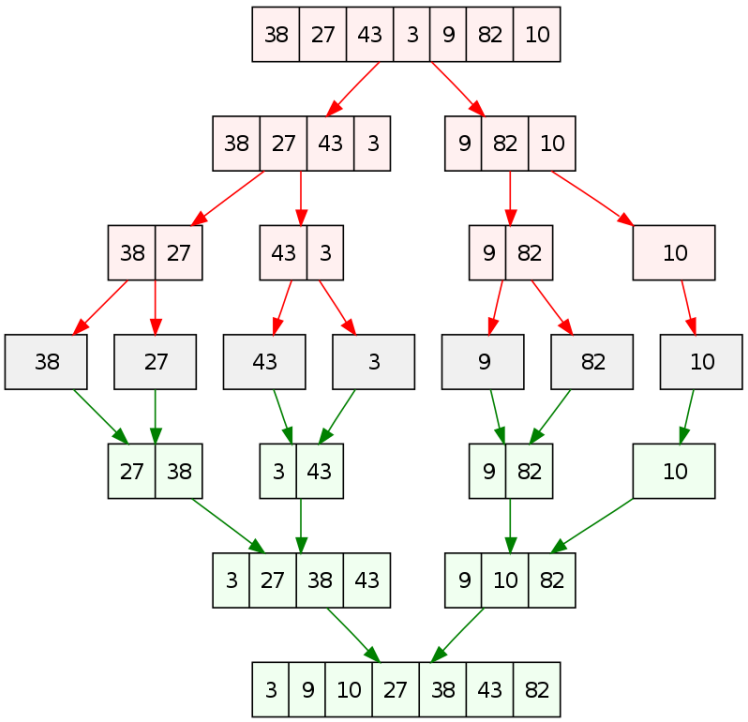
Sorting Algorithms

Ken Wu

7/13/2021

Mergesort Algorithm

```
# Import image
knitr::include_graphics("Mergesort.png")
```



Implementation 1: Rosetta code at Link

- Two functions are involved in this algorithm: a merge() function for comparing values and merging sub-vectors
- A mergesort() function that recursively calls itself to divide the vector into length-one sub-vectors.

```
mergesort <- function(x) {

  # A merge function to compare and append values to result container in order
  merge <- function(left, right) {
```

```
  # Create an empty container to hold the results
  result <- vector(mode = "double", length = 0)

  # This is the stop condition. While (as long as) left & right sub-vectors still have a value
  # Compare them and insert them into the container in order
  # That is, if the condition in while() evaluates to TRUE, run the action in {}
  while (length(left) > 0 && length(right) > 0) {
    # If the value in the left sub-vector is less than or equal to that in the right,
    # Add the left value to the container
    if (left[[1]] <= right[[1]]) {
      result <- c(result, left[[1]])
      # Remove that value from the left sub-vector
      left <- left[-1]
    } else {
      # If the value in the right sub-vector is less than or equal to that in the left,
      # add the right value to the container
      result <- c(result, right[[1]])
      # Remove that value from the right sub-vector
      right <- right[-1]
    }
  }

  # Keep comparing and inserting the values into the container
  if (length(left) > 0) result <- c(result, left)
  if (length(right) > 0) result <- c(result, right)
  # Output
  result
}


# This is the terminating condition for the mergesort function
# When the length of the vector is one, just return length-one vector itself
len <- length(x)
if (len <= 1) {
  x
} else {

  # Otherwise keep dividing the vector into two halves
  middle <- length(x) / 2
  # Add every element from the left of the middle to the left sub-vector
  # Use ceiling to handle cases where there is an odd number of elements in "x"
  left <- x[1:ceiling(middle)]
  # Add every element from the right of the middle to the right sub-vector
  right <- x[ceiling(middle + 1):len]
  # Recursively call mergesort() on the left and right sub-vectors
  left <- mergesort(left)
  right <- mergesort(right)

  # Order and combine the results
  # The condition below should evaluate to a single T/F
  if (left[length(left)] <= right[1]) {
    c(left, right)
  } else {
```

```
      merge(left, right)
    }
  }
}
```

Implementation 2: Geeksforgeeks at Link

- The && (AND) and || (OR) are non-vectorized operators; they consider only the first element of the vectors and give a vector of single element as output.

```
# A function to merge two sorted vectors
merge <- function(x, y) {

  # Pre-allocate container vector (all 0's) with length equaling the length of x and y combined
  container <- vector(mode = "double", length = length(x) + length(y))

  # Initialize i & j pointing to the first indices of the sorted vectors x & y
  # Initialize k which points to the first index of the container
  i <- 1
  j <- 1
  k <- 1
  for (k in 1:length(container)) {
    # If i is less than the length of x AND x[i] < y[j]
    # Or if j is larger than the length of y
    # Use [ instead of [[ since the latter returns an error when either i or j is out-of-bounds
    # Out-of-bounds with [ returns an NA and (NA || TRUE) evaluates to TRUE
    if ((i <= length(x) && x[i] < y[j]) || j > length(y)) {
      # Insert x[[i]] into the container and increment i to the next index
      # Index j remains unchanged when this action is executed
      container[[k]] <- x[[i]]
      i <- i + 1
    } else {
      # Otherwise, insert y[[j]] into the container and increment j to the next index
      # Index i remains unchanged when this action is executed
      container[[k]] <- y[[j]]
      j <- j + 1
    }
  }

  # Output
  container
}
```

Chart Flow (Partial)

1. Lines 109 thru 116: Pre-allocate output container and initialize indices.
2. line 117: Set k to 1; i is 1 and j is 1.

- If the condition in () evaluates to TRUE, the first element of the x vector is sub-assigned to the first element of container. Then, i is incremented by 1 while j is unchanged.

- If the condition in () evaluates to FALSE, the first element of the y vector is sub-assigned to the first element of container. Then, j is incremented by 1 while i is unchanged.

End of if statement. Return to line 117.

3. line 117: Set k to 2. Now, it must be the case that either i or j is 2 while the other is still 1. Compare those values and if the condition evaluates to true, execute 125-126. If FALSE, execute 130-131. Return to 117. So on and so forth until the condition (i <= length(x) && x[i] < y[j]) || j > length(y) evaluates to false.

```
# function to sort the array
mergeSort <- function(arr) {

  # if length of array is greater than 1,
  # then perform sorting
  if (length(arr) > 1) {

    # find mid point through which
    # array need to be divided
    mid <- ceiling(length(arr) / 2)

    # first part of array will be from 1 to mid
    a <- mergeSort(arr[1:mid])

    # second part of array will be
    # from (mid+1) to length(arr)
    b <- mergeSort(arr[(mid + 1):length(arr)])

    # merge above sorted arrays
    merge(a, b)
  }
  # else just return arr with single element
  else {
    arr
  }
}

# take sample input
arr <- sample(1:100, 10)

# call mergeSort function
result <- mergeSort(arr)

# print result
result
 [1]  1  9 34 41 44 53 54 84 96 99
```