Pseudocode

enum dataState:

presorted,

reversesorted,

randomData

// Precondition: *A* is an array with the data,

// *start* is the index of the first element in the left division,

// *m* is the index of the first element in the right division,

// and *end* is the index of the last element in the right division.

// The left and right divisions must each be sorted.

// Postcondition: The elements from *start* to *end*, inclusive, are in

// increasing sorted order.

merge( A, start, m, end ):

size = end – start + 1

// Loop Invariant: at each execution of the guard, L[0…k-start] = A[start…k]

for k = start to k < m

L[k - start] = A[k]

k++

L[m – start] = ∞

i = j = 0 // we are indexing from zero

// Loop Invariant: at each execution of the guard, A[start…(start+k)]

// is sorted

for k = 0 to k < size

if L[i] < A[j + m] || j > end – m

A[k + start] = L[i]

i++

else

A[k + start] = A[j + m]

j++

k++

// Precondition: *A* is an array with the data that is to be

// sorted from *min* to *max* inclusive.

// Postcondition: The data in *A* from *min* to *max* are in increasing sorted

// order.

mergesort( A, min, max )

if max != min

center = floor( (max + min + 1) / 2 )

mergesort( A, min, center – 1 )

mergesort( A, center, max )

merge( A, min, center, max )

// Precondition: Left and right are objects to be swapped.

// Postcondition: The previous value of left is in right, and the previous value

// of right is in left.

my\_swap( left, right )

tmp = left

left = right

right = tmp

// Precondition: *A* is an array with the data that is to be

// sorted from *min* to *max* inclusive.

// Postcondition: The elements of A from position min to position max

// are in nondecreasing order.

insertionsort( A, min, max )

// Loop Invariant: at each execution of the guard, A[min...(i-1)] is sorted

for i = min + 1 to i <= max

j = i

// Loop Invariant: at each execution of the guard,

// A[j] is the minimum element of A[j...i]

while j > min && A[j] < A[j-1]

my\_swap( A[j-1], A[j] )

j--

i++

// Precondition: *A* is an array with the data that is to be

// sorted from *min* to *max* inclusive.

// Postcondition: The elements of A are in nondecreasing order.

bubblesort( A, min, max )

// Loop Invariant: at each execution of the guard, the last i elements are

// the largest i elements in sorted order

for i = 0 to i < max - min + 1

// Loop Invariant: at each execution of the guard, the largest element

// of A[min...j] is A[j]

for j = min to j < max - i

if A[j+1] < A[j]

my\_swap( A[j+1], A[j] )

j++

i++

// Precondition: *A* is an array with the data that is to be

// sorted from *min* to *max* inclusive.

// min is the index of the first element in the sort;

// max is the index of the last element in the sort.

// A must be at least of size max

// Postcondition: The elements of A from position min to position max are

// in nondecreasing order.

timsort( A, min, max )

if max - min < TIM\_SORT\_THRESHOLD

insertionsort( A, min, max )

else

center = (max + min + 1) / 2

timsort( A, min, center - 1 )

timsort( A, center, max )

merge( A, min, center, max )

// Precondition: The size of A is at least size.

// Postcondition: A is filled with integers that are either nondecreasing,

// nonincreasing, or random, depending on the value of state.

generateData( A, size, state = randomData )

// Loop Invariant: at each execution of the guard, A[0…k] contains

// randomized data

for k = 0 to k < size

A[k] = rand() % MAX\_DATUM\_VALUE

k++

if state == presorted

mergesort( A, 0, size - 1 )

else if( state == reversesorted )

mergesort( A, 0, size - 1 )

// Loop Invariant: at each execution of the guard, A[0…j]

// and A[(size-1-j)…(size-1)] are sorted

for j = 0 to j < size / 2

my\_swap( A[j], A[size - j - 1] )

j++

// Precondition: The sizes of both source and destination are at least size.

// Postcondition: The values of destination from position 0 to position (size-1)

// are the same as the values at the same indices in source.

datacopy( source, destination, size )

// Loop Invariant: at each execution of the guard,

// destination[0…k] = source[0…k]

for k = 0 to k < size

destination [k] = source[k]

k++

// Precondition: direction should not be randomData. Data must be at least

// length n.

// Postcondition: If the data satisfies the specified direction,

// then true is returned. Otherwise false is returned.

isSorted( data, n, direction = presorted )

// Loop Invariant: at each execution of the guard, data[0…(k-1)] is

// sorted in increasing order is direction = presorted or sorted in

// decreasing order if direction = reversesorted

for k = 1 to k < n

if ( direction == presorted && data[k] < data[k-1] ) || ( direction ==

reversesorted && data[k-1] < data[k] )

return false

k++

// Precondition: array1 and array2 must be of at least size n.

// Postcondition: The algorithm will return true if array1 is equal to its

// respective element from 0 to n; array1[0…n] = array2[0…n].

isEqual( array1, array2, n )

eq = true

// Loop Invariant: at each execution of the guard,

// array1[0…k] = array2[0…k]

k = 0

while eq && k < n

if array1[k] != array2[k]

eq = false

k++

return eq

main()

// Loop invariant: at each execution of the guard, each sorting algorithm

// will have outputted the time it took to sort arrays of size

// sqrt(2) ^ (1…exponent-1) for each of the previous sorting

// configurations in dataState

foreach ds in dataState

// Let’s test powers of sqrt(2),

// by doing so we will see how powers of 2 affect merge sort.

// Loop invariant: at each comparison of the guard,

// the amount of time it takes each of the sorting algorithms to

// sqrt(2) ^ (1…exponent–1) items will be outputted to the screen

for exponent = 1 to exponent < MAX\_EXPONENT

n = floor( (sqrt(2) ^ exponent) + 0.5 )

GenerateData( original\_data, n, ds )

if !isSorted( data, n, ds )

Error()

data = original\_data

t = currentTime()

timsort( data, 0, n-1 )

duration = currentTime() – t

if !isSorted( data, n, presorted )

Error()

output( time )

data = original\_data

t = currentTime()

mergesort( data, 0, n-1 )

duration = currentTime() – t

if !isSorted( data, n, presorted )

Error()

output( time )

data = original\_data

t = currentTime()

insertionsort( data, 0, n-1 )

duration = currentTime() – t

if !isSorted( data, n, presorted )

Error()

output( time )

data = original\_data

t = currentTime()

bubblesort( data, 0, n-1 )

duration = currentTime() – t

if !isSorted( data, n, presorted )

Error()

output( time )

output( exponent )

output( n )

exponent++