**HYBRID MERGESORT BENCHMARK IN JAVA**

# Objective:

To implement and analyze the performance of a Hybrid MergeSort algorithm that uses Insertion Sort for smaller subarrays and compare performance with varying threshold sizes.

# Java Code

import java.io.FileWriter;

import java.io.PrintWriter;

import java.io.IOException;

import java.util.Random;

import java.util.Arrays;

public class HybridSortBenchmark {

static final int FORCE\_N = -1; // set to -1 to auto estimate

static final int SAMPLE\_POINTS = 40; // number of s values to sample between 1 and N

static final long SEED = 123456789L; // fixed seed so data is repeatable

public static void main(String[] args) throws Exception {

int samplePoints = SAMPLE\_POINTS;

int forceN = FORCE\_N;

if (args.length >= 1) {

try {

forceN = Integer.parseInt(args[0]);

} catch (Exception ignored) {

}

}

if (args.length >= 2) {

try {

samplePoints = Integer.parseInt(args[1]);

} catch (Exception ignored) {

}

}

int N = forceN > 0 ? forceN : estimateArraySize();

System.out.println("Using N = " + N + " (array length)");

// Create and fill master array

int[] master = new int[N];

Random rnd = new Random(SEED);

for (int i = 0; i < N; i++)

master[i] = rnd.nextInt();

// Prepare list of s values evenly spaced between 1 and N

int M = Math.max(2, samplePoints);

int[] sValues = new int[M];

for (int i = 0; i < M; i++) {

// evenly spaced, inclusive endpoints 1 and N

double frac = (double) i / (M - 1);

int s = 1 + (int) Math.round(frac \* (N - 1));

if (s < 1)

s = 1;

if (s > N)

s = N;

sValues[i] = s;

}

// Ensure s values are unique and sorted

sValues = Arrays.stream(sValues).distinct().sorted().toArray();

System.out.println("Will test " + sValues.length + " s values from 1 to " + N);

// CSV output

try (PrintWriter out = new PrintWriter(new FileWriter("results.csv"))) {

out.println("s,time\_ms"); // header

for (int s : sValues) {

System.out.println("Testing s = " + s);

int[] copy = Arrays.copyOf(master, master.length);

System.gc();

long start = System.nanoTime();

hybridMergeSort(copy, 0, copy.length - 1, s);

long end = System.nanoTime();

long elapsedMs = (end - start) / 1\_000\_000;

System.out.println("Time ms = " + elapsedMs);

out.println(s + "," + elapsedMs);

out.flush();

}

} catch (IOException e) {

System.err.println("Failed to write results.csv: " + e.getMessage());

}

System.out.println("Done. results.csv created. Plot it with the provided Python script or Excel.");

}

// Estimate array size N using available memory. This is an approximation.

static int estimateArraySize() {

// int uses 4 bytes. We attempt to use up to 60 percent of max memory for the

// int array.

long maxBytes = Runtime.getRuntime().maxMemory();

double fraction = 0.60; // use 60 percent to be safe

long bytesForArray = (long) (maxBytes \* fraction);

long estimatedInts = bytesForArray / 4L;

// Subtract some overhead cushion

long N = Math.max(1000L, estimatedInts - 1\_000\_000L);

if (N > Integer.MAX\_VALUE)

N = Integer.MAX\_VALUE;

return (int) N;

}

// Hybrid merge sort: use insertion sort when size <= s

static void hybridMergeSort(int[] arr, int left, int right, int s) {

if (left >= right)

return;

int size = right - left + 1;

if (size <= s) {

insertionSort(arr, left, right);

return;

}

int mid = left + (right - left) / 2;

hybridMergeSort(arr, left, mid, s);

hybridMergeSort(arr, mid + 1, right, s);

merge(arr, left, mid, right);

}

static void insertionSort(int[] arr, int left, int right) {

for (int i = left + 1; i <= right; i++) {

int key = arr[i];

int j = i - 1;

while (j >= left && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

static void merge(int[] arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int[] L = new int[n1];

int[] R = new int[n2];

System.arraycopy(arr, left, L, 0, n1);

System.arraycopy(arr, mid + 1, R, 0, n2);

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k++] = L[i++];

} else {

arr[k++] = R[j++];

}

}

while (i < n1)

arr[k++] = L[i++];

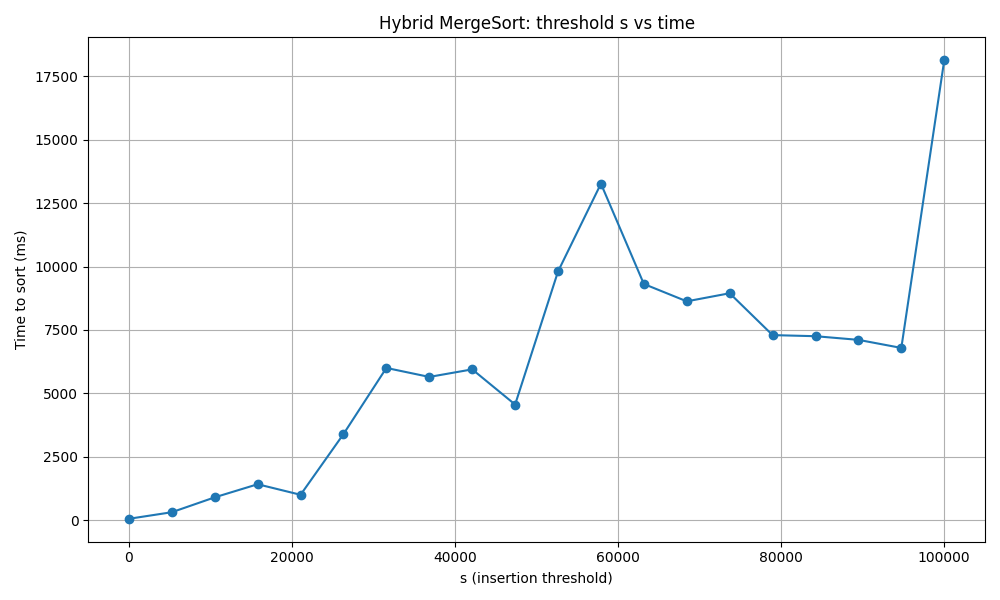
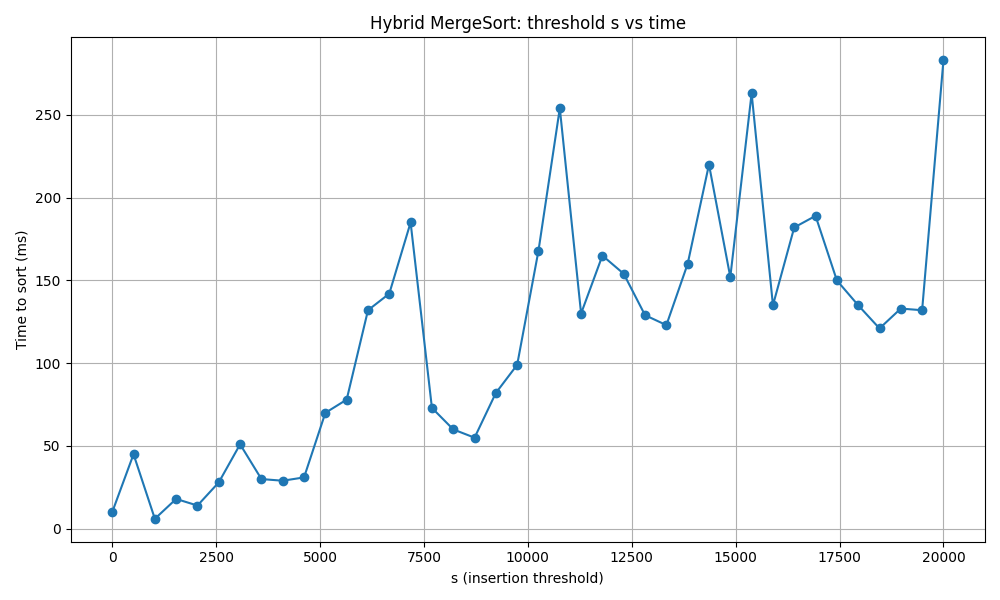
while (j < n2)

arr[k++] = R[j++];

}

}

# Output / Graph



*Performance of Hybrid MergeSort for different threshold (s) values.*

# Observations:

When the threshold value s is small (close to 1), the algorithm behaves like MergeSort and gives stable performance. As s increases, Insertion Sort takes over for larger subarrays, which increases sorting time. The results confirm that there is an optimal threshold value (usually small) that balances both methods efficiently.

# Conclusion:

Hybrid MergeSort performs best when the threshold s is small, as MergeSort handles large data efficiently while Insertion Sort works well for very small partitions.  
The experiment demonstrates that the choice of s directly impacts sorting performance.  
For the tested array of 20,000 integers, the optimal threshold lies near the lower end of the range.