```
/*****************************************************************************
 *
 * James A. Avery
 * 50189091
 * CSCI 532 Semester Project
 *
 * Problem #3 (expanded)
 * Write a C/C++/Matlab/Java program to compute insertion sort and merge sort.
 * (You can use the code from the textbooks by Deitel and Deitel or from the
 *  web.)
 * Obtain the run time of both routines.
 * The input data should be an int array containing random element values
 *  (between, say, 0 and 1023).
 * Obtain run time T with 1D (input) array size of n=16, 256, 4096, 65536,
 *  1048576 (which equal 2^p, where p = 4, 8, 12, 16, 20).
 * The run time for each n should be average with about floor(512/(p*p)) runs.
 * Each run (for a given n) should use a different random input.
 * Plot (with Excel, Matlab, or other available tools) the run time for both
 *  routines on one plot, with the x axis in p values, and y axis in log(T).
 * Label on the plot which curve is for insertion sort and which is for merge
 *  sort.
 * Submit C/C++ programs and plot, with instructions in readme.txt on how to
 *  build and run the program. (Include the Dev-C++ or MS Studio or Java
 *  NetBeans project file.)
 *
 *****************************************************************************/

/**  References  ************************************************************
 *
 *  Tour of Go
 *    https://tour.golang.org/
 *  Merge Sort in Go
 *    https://austingwalters.com/merge-sort-in-go-golang/
 *
*****************************************************************************/

package main

import (
  "fmt"
  "math"
  "math/rand"
  "time"
  "os"
  "strings"
  "unicode"
  "encoding/csv"
  "strconv"
  "sync"
)


/*
By default, the program runs a decreasing number of trials, such that for an
    array of size n=2^p, the number of trials is floor(512/(p*p)), (per the
    project instructions).
I would like a more scientific approach (i.e. a more) uniform number of
    trials), so I have included a command line option to run the same number of
    trials for each vector size. On my computer, a slice of n = 2^20 = 1048567
  can be sorted with insertion sort in around four minutes, and the entire
```

```
  project, with 24-element slices, can run in under an hour.
*/
const StandardBatchSize int = 24

/*
If, instead, the program is being run as assigned, then calculate the batch
  size based on the formula floor(512/p^2)
This function is defined inline in the C++ version; my understanding is that Go
  lacks an "inline" keyword, but the Go compiler would likely build this as
  an inline function.
*/
func assignedBatchSize( p int ) int {
  return int( math.Floor( 512 / ( float64(p) * float64(p) ) ) )
}


/*
Arrays of size n = 20 take several minutes to run. Limiting the length of
  arrays to those that can be sorted quickly speeds testing of other issues.
*/
const MaxN int = 1000000

/*
A global WaitGroup to hold all the various tests
*/
var wg sync.WaitGroup



// Struct to allow passing a function along with its name
type NamedFunction struct {
  name string
  function func([]int) []int
}



func main() {
  // Put the command line arguments into an array for easier access
  args := os.Args

  // Flags to determine run-time behavior
  // Default values are the problem as assigned
  // Changing these through command line options results in shorter or more interesting tests
  runInsertionSort := true
  runMergeSort := true
  limitN := false
  demo := false
  time := true;

    // p determines both the array size and the number of tests per array
    // Arrays of size 2^p will be tested either floor(512/(p*p)) times or 32 times
  // Having a vector of ps will make things easier later
  ps := []int{ 4, 8, 12, 16, 20 }

  // Determine the sizes of the test runs
  // If false, then each vector of size 2^p will be tested floor(152/(p*p)) times
  // If true, each vector will be tested the same number of times (set above)
  equalBatchSizes := false

```

```
121
122   // Adjust run-time flags according to command line arguments
123   for _, flag := range args {
124     if flag == "merge-only" { runInsertionSort = false; }
125     if flag == "insertion-only" { runMergeSort = false; }
126     if flag == "equal-batches" { equalBatchSizes = true; }
127     if flag == "limit-n" { limitN = true; }
128     if flag == "demo-sorters" { demo = true; }
129     if flag == "no-time" { time = false; }
130   }
131
132   // Create a slice of functions to test, based on command line options
133   //  Default is both Insertion Sort and Merge Sort,
134   //  but these can be stopped by command line options above
135   var sorters []NamedFunction
136   if runInsertionSort {
137     sorters = append(sorters, NamedFunction{"Insertion Sort", InsertionSort})
138   }
139   if runMergeSort {
140     sorters = append(sorters, NamedFunction{"Merge Sort", MergeSort})
141   }
142
143   // Demonstrate that the sort functions sort correctly (disabled by default)
144   if demo {
145     for _, sorter := range sorters {
146       demoSorter(sorter)
147     }
148   }
149
150   // The heart of the project
151   // Time the run-time of each sort function as assigned (enabled by default)
152   if time {
153     for _, sorter := range sorters {
154       testSorter(sorter, ps, equalBatchSizes, limitN)
155     }
156   }
157
158   return
159 }
160
161
162
163 // Sort a slice in place using insertion sort and return the slice
164 func InsertionSort(v []int) []int {
165   // Loop through the elements of the array
166   for j := 1; j < len(v); j++ {
167     key := v[j]
168     i := j - 1;
169
170     for i >= 0 && v[i] > key {
171       v[i + 1] = v[i]
172       i -= 1;
173     }
174
175     v[i+1] = key
176   }
177
178   return v
179 }
180
```

```
181
182
183 // Sort a slice in place using Merge Sort, and return the slice
184 //  From https://austingwalters.com/merge-sort-in-go-golang/
185 func MergeSort(slice []int) []int {
186   if len(slice) < 2 {
187     return slice;
188   }
189
190   mid := (len(slice)) / 2
191   return Merge(MergeSort(slice[ : mid]), MergeSort(slice[mid : ]))
192 }
193
194
195
196 // Merge - merges left and right slices into newly created slice
197 func Merge(left, right []int) []int {
198   size := len(left) + len(right)
199   i, j := 0, 0
200
201   slice := make([]int, size, size)
202
203   for k := 0; k < size; k++ {
204     if i > len(left) - 1 && j <= len(right) - 1 {
205       slice[k] = right[j]
206       j++
207     } else if j > len(right) - 1 && i <= len(left) - 1 {
208       slice[k] = left[i]
209       i++
210     } else if left[i] < right[j] {
211       slice[k] = left[i]
212       i++
213     } else {
214       slice[k] = right[j]
215       j++
216     }
217   }
218
219   return slice
220 }
221
222
223
224 // Fill a vector in place with random integers
225 func RandomFill(v []int) {
226   for i := 0; i < len(v); i++ {
227     v[i] = rand.Intn(1024)
228   }
229 }
230
231
232
233 /* Given a NamedFunction, a slice of values for p, and booleans to determine
234   whether or not to process equal batch sizes and whether or not to limit
235   the size of the slice to slices that can be sorted quickly, calculate the
236   execution time for each of the sort algorithms.
237 */
238 func testSorter(f NamedFunction, ps []int, equalBatchSizes, limitN bool) {
239   // Create a file to output the results
240   fileName := strings.ToLower(strings.Map(
```

```
241     func(r rune) rune {
242       if unicode.IsSpace(r) {
243         return -1
244       }
245       return r
246     }, f.name)) + ".csv"
247
248   file, err := os.Create(fileName)
249   if err != nil {
250     panic(err)
251   }
252
253   defer file.Close()  // Close the file when done
254
255   writer := csv.NewWriter(file)
256   fmt.Println("\n", f.name)
257
258   // Iterate through each of the p values (p = 4, 8, 12, ...)
259   for _, p := range ps {
260     // Vector size
261     n := int(math.Pow(2, float64(p)))
262
263         // If n is being limited to certain fast-running small values,
264         //  then skip this iteration
265     if limitN && n > MaxN { continue }
266
267     // Calculate batch size
268     batchSize := StandardBatchSize
269     if !equalBatchSizes { batchSize = assignedBatchSize( p ) }
270
271     // First column of the output is the size of the slice
272     fmt.Printf("\n%v:\t", n)
273
274     var results []string
275     results = append(results, strconv.Itoa(n))
276
277     ch := make(chan uint64, batchSize)
278
279     for trial := 0; trial < batchSize; trial++ {
280       v := make([]int, n, n)
281       RandomFill(v)
282
283       wg.Add(1)
284       go TimeRun(f.function, ch, v)
285     }
286
287     wg.Wait()
288     close(ch)
289     for range ch {
290       duration := <-ch
291       results = append(results, strconv.FormatUint(duration, 10))
292     }
293
294     fmt.Println()
295     err = writer.Write(results)
296     if err != nil {
297       panic(err)
298     }
299
300     writer.Flush()
```

```
  }
}



// Find the length of time it takes to run a sort,
//  and push the time (in nanoseconds) to the channel, c
func TimeRun(f func([]int) []int, c chan uint64, v []int) {
  defer wg.Done()
  start := time.Now()
  f(v)
  duration := uint64(time.Now().Sub(start).Nanoseconds())
  fmt.Printf("%v ", duration)
  c <- duration
}



// Demonstrate that the sorter actually does sort a slice properly
func demoSorter(sorter NamedFunction) {
  fmt.Printf("\n=====  Demonstrating %v  =====\n", sorter.name)
  for i := 0; i < 10; i++ {
    v := make([]int, 10, 10)
    RandomFill(v)
    fmt.Printf("\nBefore: %v\n", v)
    v = sorter.function(v)
    fmt.Printf("After:  %v\n", v)
  }
}
```