# Lab 1: Single-node sort

TA in charge: Rohit Pai

## Updates

- April 1, 12:00PM: Added a Testing section to provide a brief overview of the test cases executed on your code.
- April 1, 4:35pm: We realized that the wrong version of the utilities were included in the repository attached to the GitHub Classroom invitation. We've fixed this. So if you have not yet accepted the invitation, you should get the correct version of the utilities. If you've already accepted the invitation and cloned the starter code, you should see a "pull request" in your github over the next hour or so you can accept to fix the problem. Alternatively, you can just directly get the correct utilities by cloning this public repo: https://github.com/ucsd-cse-x24-sp25/lab1
- April 2, 10:28am: clarified that pdf of matplotlib graph should be called "readme.pdf"
- April 3, 9:15am: combined "report" with deliverables in the "Submission" section
- April 3, 12:30pm: clarified the endianness of the length field
- April 4, 5:20 pm: changed "readme.pdf" to "report.pdf"

In this project, you are going to jump into using the Go language by implementing a simple sorting program. This course is a "learn by doing" course where you will need to write quite a bit of code. Although it does not have any specific programming language prerequisite, you will need to be able to quickly learn a new language (Go).

This project serves as a diagnostic for how prepared you are for the remainder of the course:

- **Green light**: If you are able to implement a working sort without any particular difficulty in a couple of hours, then you should be well-prepared for the remaining projects
- **Yellow light**: If you were able to implement a working sort, but took more than a couple hours or required lots of extra help, then you may find this course to be quite time intensive, especially the later projects, so plan your schedule accordingly
- **Red light**: If you were not able to implement a working sort program then it is not recommended that you take this course as the projects only get considerably more difficult over time.

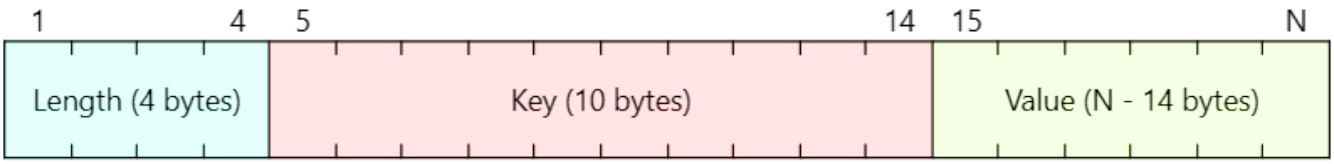
## Starter code

- To begin the project, please accept the GitHub invitation and clone your repository. Don't forget to commit and push your code and lab report when you're done.

# Sort specification

This project will read, write, and sort files consisting of zero or more records. A record has three fields:

1. A 4-byte length field in **Big Endian Unsigned Integer**, which contains the sum of the lengths of the key and value fields
2. A 10-byte key, then
3. A variable-length value. Thus, the minimum value of the length field is 10 bytes (which would mean that the value is 0 bytes long).




Note that although the length field could theoretically represent a record with a value that's up to (2^32 - 10) bytes in size, for this project, we will limit values to being at most 2^10 bytes long (1024 bytes).

You can read uint32 data from a buffer and write uint32 data to a buffer using the "encoding/binary" package. Detailed instructions can be found at https://piazza.com/class/m852xr25ufs14m/post/50

```go
package main
import (
        "encoding/binary"
        "fmt"
)

// Read a big-endian uint32 from a byte slice of length at least 4
func ReadBigEndianUint32(buffer []byte) uint32 {
        if len(buffer) < 4 {
        panic("buffer too short to read uint32")
        }
        return binary.BigEndian.Uint32(buffer[:])
}

// Write a big-endian uint32 to a byte slice of length at least 4
func WriteBigEndianUint32(buffer []byte, num uint32) {
        if len(buffer) < 4 {
        panic("buffer too short to write uint32")
        }
        binary.BigEndian.PutUint32(buffer, num)
}

func main() {
        // Reading a big-endian uint32 from a byte slice
        var data [4]byte = [4]byte{0x00, 0x00, 0x00, 0x01}
        num := ReadBigEndianUint32(data[:])
        fmt.Println(num) // Output: 1

        // Writing a big-endian uint32 to a byte slice
        var buffer [4]byte
        WriteBigEndianUint32(buffer[:], num)
        fmt.Println(buffer) // Output: [0 0 0 1]

        // Attempting to write a big-endian uint32 to a 2-byte buffer
        var shortBuffer [2]byte
        WriteBigEndianUint32(shortBuffer[:], num) // This will cause a panic
        fmt.Println(shortBuffer)                  // This line will not be reached due to the error
}
```

Your sort should be ascending based on the key, meaning that the output should have the record with the smallest key first, then the second-smallest, etc.

You are to write a sort program that reads an input file and produces a sorted version of the output file. Use src/sort.go as a starting point.  Your program should support the following interface:

Usage:

```
$ bin/sort inputfile outputfile
```

The tests will be conducted assuming that the following setup is maintained, and you must ensure that their implementation adheres to these conditions.

For this lab, it is OK to read the input file into memory, however you might consider how you could implement sort so that the memory usage of your program does not exceed a predefined value (e.g. a few 10s of megabytes).

Once the data is in memory, you can use Go's in-built sort function, but to do so you will need to implement a custom comparison function that understands this lab's record format.

To aid you in generating input files and validating the output of your program, we've provided you with a few utilities in binary format. Currently there are utilities for x86_64-based Linux, x86_64-based Windows, the Intel and Apple M1/M2/M3 macs.

# Utilities

## Gensort

Gensort generates random input. If the 'randseed' parameter is provided, the given seed is used to ensure deterministic output.

'size' can be provided as a non-negative integer to generate that many bytes of output. However human-readable strings can be used as well, such as "10 mb" for 10 megabytes, "1 gb" for one gigabyte", "256 kb" for 256 kilobytes, etc.  You can specify the input size in any format supported by the https://github.com/c2h5oh/datasize package.

The keys and values are randomly generated, and the length of each record is also randomly generated. Additionally, two optional arguments, *minlength* and *maxlength*, define the length of the values. These default to 10 bytes and 1034 bytes, respectively, which is the range used in the test cases.

Due to the randomly generated record sizes, the generated files are approximately the specified size but may vary by a few bytes.

Usage:

```
$ bin/gensort [-maxlength length] [-minlength length] [-randseed seed] outputsize
outputfile
```

The optional *seed* argument allows you to cause gensort to repeatedly create the same random sequence (useful for, e.g., testing in different locations without the need to copy a large test file).

### Showsort

Showsort shows the records in the provided file in a human-readable format, with the key followed by a space followed by the value.

Usage:

```
$ bin/showsort inputfile
```

Example output:

```
$ bin/showsort input.dat
EFB253283F9152A8C188    3AF1702130F6E3CA3D9FBFB4AC5D
F4D5396E86713903DEDB    E4E1D8ED9E9D7D8791842EEF6515FDC5F669B1331FE0A3F5DF6C1
B36C1A19C0D88502BE07    6E1A3D104C4FA51C9606C804E504399F
5EF7E9D00E85717BE084    4D876F
```

## Building

To build your sort program:

```
$ go build -o bin/sort src/sort.go
```

## Verifying your sort implementation

A simple way to verify the correctness of your implementation of sort is to run the standard unix sort command on the output of 'showsort'. For example, to generate, sort, and verify a 1 megabyte file:

```
$ bin/gensort "1 mb" example1.dat
Initializing the random number generator’s seed to a random value
Requested 1 mb (= 1048576) bytes

$ bin/sort example1.dat example1-sorted.dat

$ bin/showsort example1.dat | sort > example1-chk.txt
$ bin/showsort example1-sorted.dat > example1-sorted-chk.txt
```

```
$ diff example1-chk.txt example1-sorted-chk.txt
```

This last 'diff' should simply return to the command prompt. If it indicates any differences that means that there is an error in your sort routine.

You should verify your solution for different file sizes, including 0 bytes. Your program should support input sizes of at least 10s of megabytes.

## Testing

The test cases cover a range of file sizes from 14 bytes to 10 MB. They include a minimum-size file, a single-record test, and progressively larger inputs, from small (100 bytes, 1 KB) to medium (100 KB) and large (1 MB, 5 MB, 10 MB). The records are at most 1 KB each, resulting in up to 20,000 records in the largest cases.

## Submission

For your report, use matplotlib to generate a plot of your sort program's runtime for inputs between 1KB and 10MB in 20 linear steps. You can time how long your program takes to run using the time command in linux. In addition to the runtime, analyze your program's asymptomatic bound and plot the bound on the same graph. For example, if your sort runs in $O(n^2)$, plot $n^2$ alongside your runtime. Label the asymptotic bound clearly. Include this plot as "**report.pdf**" with your submission to expedite grading.

## Expected effort level

To set expectations for how much work is involved, we can say that our reference solution of sort.go was about 140 lines of code.

###