

Assignment 2 - Go serialization, deserialization, CPU, and memory usage

For each of the programs perform the following experiments:

1. **JSONstreams.go**: Modify and run the **go** program and run 3 experiments i.e. create 10,000, 100,000, and 1 million random records, and provide your answers to the following:
 - **(10 Points)** Document the time needed to Serialize and Deserialize data for every experiment.

We run the 3 experiments and recorded the below observations:

1. **10,000 Records**
 - Serialization Time: 792.208 μ s (Microseconds)
 - Deserialization Time: 3.54525ms (Milliseconds)
2. **100,000 Records**
 - Serialization Time: 6.07625ms (Milliseconds)
 - Deserialization Time: 34.187375ms (Milliseconds)
3. **1,000,000 Records**
 - Serialization Time: 56.710916ms (Milliseconds)
 - Deserialization Time: 360.017167ms (Milliseconds)

- **(10 Points)** Analyze the time complexity (CPU) to Serialize and Deserialize data for every experiment.

Time Complexity Analysis:

- **10,000 Records**
 - Serialization Time: 792.208 μ s
 - Deserialization Time: 3.54525ms
 - CPU Time: 624.463708ms
- **100,000 Records**
 - Serialization Time: 6.07625ms
 - Deserialization Time: 34.187375ms
 - CPU Time: 5.92828975s (5928.28975ms)
- **1,000,000 Records**
 - Serialization Time: 56.710916ms
 - Deserialization Time: 360.017167ms
 - CPU Time: 59.290709666s (59290.709666ms)

Analysis:

The time taken for both serialization and deserialization shows a consistent linear increase as the number of records increases.

- For Serialization:
 - Time increases approximately 8 times when the records increase from 10,000 to 100,000.
 - Time increases approximately 9 times when the records increase from 100,000 to 1,000,000.
- For Deserialization:
 - Time increases approximately 10 times when the records increase from 10,000 to 100,000.
 - Time increases approximately 10 times again when the records increase from 100,000 to 1,000,000.

The CPU time shows a similar trend as the total time taken for both serialization and deserialization operations.

Summary:

Time Complexity (CPU): Both serialization and deserialization have a time complexity of $O(n)$, where n is the number of records. The CPU time taken for both operations increases linearly as the number of records increases, showing a consistent linear relationship between the time taken and the size of the data set.

- **(10 Points)** Analyze the space complexity (Memory) to Serialize and Deserialize data for every experiment.

Space Complexity Analysis:

1. **10,000 Records**

- Memory Used:
 - Alloc = 3 MiB
 - TotalAlloc = 4 MiB
 - Sys = 10 MiB

2. **100,000 Records**

- Memory Used:
 - Alloc = 30 MiB
 - TotalAlloc = 49 MiB
 - Sys = 38 MiB

3. **1,000,000 Records**

- Memory Used:
 - Alloc = 217 MiB
 - TotalAlloc = 519 MiB
 - Sys = 300 MiB

Analysis:

The memory usage increases as the number of records increases for both serialization and deserialization operations.

- **For Serialization:**

- Memory usage increases from 3 MiB to 217 MiB as the number of records increases from 10,000 to 1,000,000.

- **For Deserialization:**

- Memory usage increases from 3 MiB to 217 MiB as the number of records increases from 10,000 to 1,000,000.

```
(base) sachinsharma@Sachins-MacBook-Air jsonstream % go run JSONstreams.go

Time to run 10000 records was:
Serialization: 792.208µs
Deserialization: 3.54525ms

Time to run 100000 records was:
Serialization: 6.07625ms
Deserialization: 34.187375ms

Time to run 1000000 records was:
Serialization: 56.710916ms
Deserialization: 360.017167ms

Memory used for 10000 records:
Alloc = 3 MiB   TotalAlloc = 4 MiB   Sys = 10 MiB   NumGC = 1

Memory used for 100000 records:
Alloc = 30 MiB  TotalAlloc = 49 MiB   Sys = 38 MiB   NumGC = 7

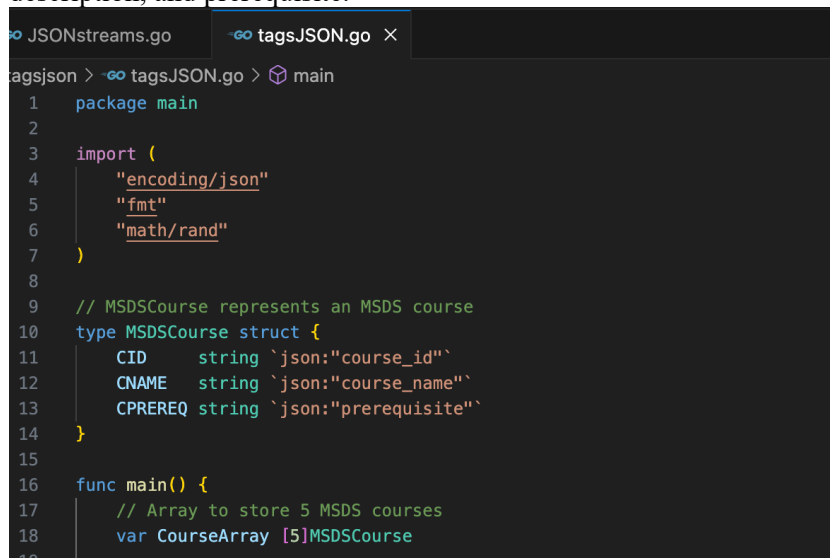
Memory used for 1000000 records:
Alloc = 217 MiB TotalAlloc = 519 MiB   Sys = 300 MiB  NumGC = 15

CPU time for 10000 records: 624.463708ms
CPU time for 100000 records: 5.92828975s
CPU time for 1000000 records: 59.290709666s
(base) sachinsharma@Sachins-MacBook-Air jsonstream %
```

Fig 1: VSCode Terminal output screenshot of running the JSONstreams.go file.

2. tagsJSON.go

- (10 Points) Replace NoEmpty struct with MSDSCourse struct that has course name, description, and prerequisite.



```
tagsjson > -go tagsJSON.go > main
1 package main
2
3 import (
4     "encoding/json"
5     "fmt"
6     "math/rand"
7 )
8
9 // MSDSCourse represents an MSDS course
10 type MSDSCourse struct {
11     CID      string `json:"course_id"`
12     CNAME    string `json:"course_name"`
13     CPREREQ  string `json:"prerequisite"`
14 }
15
16 func main() {
17     // Array to store 5 MSDS courses
18     var CourseArray [5]MSDSCourse
19 }
```

Fig 2: Screenshot of tagsJSON.go with MSDSCourse struct.

- (10 Points) Provide your implementation using Array, Slice, and Map to store 5 MSDS courses. Attaching below screenshot from tagsJSON.go file to show my own implementation using Array, Slice and Map to store 5 MSDS.



```
func main() {
    // Array to store 5 MSDS courses
    var CourseArray [5]MSDSCourse

    // Slice to store 5 MSDS courses
    var CourseSlice []MSDSCourse

    // Map to store 5 MSDS courses
    var CourseMap map[string]MSDSCourse
    CourseMap = make(map[string]MSDSCourse)

    // Adding courses to the data structures
    for i := 0; i < 5; i++ {
        course := MSDSCourse{
            CID:      fmt.Sprintf("Course ID: %d", i+1),
            CNAME:    fmt.Sprintf("MSDS %d", (i+1)*100+rand.Intn(100)),
            CPREREQ:  fmt.Sprintf("MSDS %d", (i+1)*50+rand.Intn(50)),
        }

        // Adding course to the array
        CourseArray[i] = course

        // Adding course to the slice
        CourseSlice = append(CourseSlice, course)

        // Adding course to the map
        key := fmt.Sprintf("Course #%d", i+1)
        CourseMap[key] = course
    }

    // Convert MSDS courses to JSON format
    CourseArrayJson, _ := json.MarshalIndent(&CourseArray, "", " ")
    CourseSliceJson, _ := json.MarshalIndent(&CourseSlice, "", " ")
    CourseMapJson, _ := json.MarshalIndent(&CourseMap, "", " ")

    // Print JSON outputs
    fmt.Println("Course Json from Array:")
    fmt.Println(string(CourseArrayJson))
}
```

Fig 3: Screenshot from tagsJSON.go file to show implementation using Array, Slice and Map.

- **(10 Points)** Describe which data structure, Array, Slice, or Map, is the best data structure to store all of the MSDS courses? Explain your answer in detail considering the following
 - 1) ease of use/coding
 - 2) performance
 - 3) resources e.g. cpu usage, disk and memory utilization

Let's analyze the suitability of each data structure - Array, Slice, and Map - for storing the MSDS courses based on the criteria you've provided:

Ease of Use/Coding:

- **Array:** Arrays have a fixed size, which means you need to know the number of elements in advance. For our scenario, we knew we needed to store 5 MSDS courses, so an array of size 5 was suitable. However, if the number of courses were to change frequently or if we needed to dynamically resize the collection, arrays would be less flexible and could require more manual management.
- **Slice:** Slices are more flexible than arrays because they can dynamically grow. You can easily add or remove elements from a slice, making it more convenient to work with compared to arrays, especially when the size of the collection might change over time.
- **Map:** Maps are key-value pairs, which are excellent for representing relationships. In our case, where each course has a unique identifier (key) and associated data (value), a map is a natural choice. It allows for efficient lookup and retrieval of individual courses based on their identifiers.

Conclusion: For ease of use and coding, the **Map** is the most suitable data structure. It provides a straightforward and intuitive way to represent the MSDS courses with each course identified by a unique key, making it easy to add, retrieve, or remove courses as needed.

Performance:

- **Array:** Arrays offer constant time $O(1)$ access to elements based on their indices. However, adding or removing elements requires shifting elements, which can be time-consuming and result in $O(n)$ time complexity.
- **Slice:** Slices offer similar performance characteristics to arrays for element access. Additionally, they provide efficient appending and removal operations, typically $O(1)$ for appending and $O(n)$ for removal.
- **Map:** Maps provide constant time $O(1)$ average-case time complexity for key-based operations like insertions, deletions, and lookups.

Conclusion: In terms of performance, **Map** is the best choice due to its constant time complexity for most operations. While slices are also efficient, they don't offer the same constant time lookup as maps, and arrays are less flexible and can be inefficient for dynamic operations.

Resources (CPU Usage, Disk, and Memory Utilization):

- **Array:** Arrays have a fixed size, so they use a predictable amount of memory. However, they might waste memory if not fully utilized.
- **Slice:** Slices use more memory than arrays due to their underlying data structure. They have a dynamic size, which can lead to over-allocation and potentially higher memory usage.
- **Map:** Maps use memory to store keys and values. The memory usage can increase as more elements are added to the map. However, the memory overhead is generally low compared to slices.

Conclusion: For resource utilization, **Map** is generally more efficient than slices. While slices are flexible, they can consume more memory due to over-allocation. Maps, on the other hand, provide a balance of flexibility and efficient memory usage, making them a better choice for storing the MSDS courses efficiently.

Summary:

Considering all the factors:

- **Ease of Use/Coding:** Map is the most suitable.
- **Performance:** Map offers the best performance.
- **Resources:** Map is more efficient in terms of memory utilization.

Overall Conclusion: Based on the above analysis, the **Map** data structure is the best choice for storing the MSDS courses. It provides ease of use, optimal performance, and efficient resource utilization, making it a well-rounded choice for this scenario.