Assignment 2

Instructions

To run any of the solutions to the problems in this assignment, use the following command in the terminal. X refrences to the perobles number:

```
go run pX.go
```

Replace the X with the number of the problem you want to run.

Problems

Problem 1

Basic Explanation of Implementation

1. **Goroutines**: Goroutines are lightweight threads managed by the Go runtime. By using 2 of these `go func() { ... }()`, the code spawns 2 goroutines concurrently to sort the left and right partitions of the array. And this continues to branch down until the array is sorted.

2. **Channels**: Channels are used for communication and synchronization between goroutines. In this implementation, `channelL` and `channelR` are used to receive the sorted left and right partitions respectively. This ensures that the main goroutine waits until both partitions are sorted before proceeding. And to not cause deadlocks or race conditions.

3. **WaitGroup**: `sync.WaitGroup` is used to wait for the completion of both goroutines. Before spawning the goroutines, the wait group is incremented by 2 (`wg.Add(2)`). Each goroutine defers `wg.Done()` to decrement the wait group when it completes. `wg.Wait()` blocks until the wait group counter goes to 0, indicating that both goroutines have finished execution.

4. **Partitioning and Sorting**: The array is partitioned based on a pivot element `p`, with elements less than or equal to the pivot placed in the left partition (`l`) and elements greater than the pivot placed in the right partition (`r`). Each partition is recursively sorted using quicksort.

5. **Merging Sorted Subarrays**: Once both left and right partitions are sorted, the sorted left partition is copied back to the original array. Then the pivot is placed at its correct position (after the sorted left partition). Finally, the sorted right partition is copied after the pivot.

Performance Comparison

Whether this concurrent implementation can beat a serial quicksort depends on various factors such as the size of the array, the efficiency of the partitioning scheme, and the overhead of managing goroutines and channels.

```
```

```
Serial version:
Original array: false, length: 10000000
Execution time: 5.140212767s
Sorted array: true, length: 10000000

Concurrent version:
Original array: false, length: 10000000
Execution time: 7.469341162s
Sorted array: true, length: 10000000
```

As demonstrated in the block above, the serial version outperforms the concurrent version in most cases. at this size and this seems to be a trend througout diffrent array sizes.

Problem 2

When the graphs grow larger, there can indeed be differences in performance between Dijkstra's algorithm and Floyd-Warshall's algorithm due to their respective time complexities and requirements.

1. **Dijkstra's Algorithm**:
   - Dijkstra's algorithm has a time complexity of `O(V^2)` for the basic implementation using an adjacency matrix, where `V` is the number of vertices. It iterates through all vertices in each iteration to find the minimum distance vertex, and then relaxes the edges connected to it.
   - As the number of vertices increases, the time taken by Dijkstra's algorithm can increase significantly. This is because it needs to consider every vertex for each iteration, resulting in a quadratic increase in time complexity.
2. **Floyd-Warshall's Algorithm**:
   - Floyd-Warshall's algorithm has a time complexity of `O(V^3)`, where `V` is the number of vertices. It computes the shortest paths between all pairs of vertices by considering all intermediate vertices.
   - While Floyd-Warshall's algorithm has a higher time complexity compared to Dijkstra's algorithm, it is more efficient for dense graphs or graphs with a large number of vertices, as it computes shortest paths between all pairs of vertices in a single execution.

Concurrency in Floyd-Warshall's Algorithm

To introduce concurrency to Floyd-Warshall's algorithm, the following steps were taken:

1. The function `floydWarshall()` was created to calculate the shortest paths between all pairs of vertices concurrently.
2. Inside `floydWarshall()`, a channel of type `chan bool` was initialized with a buffer size equal to the number of vertices `n`. This channel is used to signal the completion of each goroutine responsible for updating the distance matrix.
3. For each vertex `k`, a goroutine is launched to handle the updates of the distance matrix concurrently. Inside each goroutine, the algorithm proceeds as usual, calculating the shortest paths between all pairs of vertices, but in parallel for different values of `k`.

4. Upon completing the computation for a vertex `k`, the goroutine signals its completion by sending a boolean value `true` through the channel initialized in step 2.
5. The main goroutine waits for all `n` goroutines to complete by receiving `n` values from the channel.
6. Finally, the resulting distance matrix is returned.

Experiments and Result`

The experiments involved running both Dijkstra's algorithm and Floyd-Warshall's algorithm on graphs of varying sizes and analyzing their respective performances.

At small sizes like 5-50 Floyd-Warshall's algorithm is faster than Dijkstra's algorithm. At larger sizes like 500-5000, Dijkstra's algorithm is faster than Floyd-Warshall's algorithm.