Problem Statement 3:-

1. ***Explaining how the highlighted constructs work?***

a. make(chan func(), 10)

This line creates a buffered channel cnp that can hold up to 10 elements. The type of the channel is chan func(), which means it can send and receive functions with no parameters and no return values.

b.
```
for i := 0; i < 4; i++ {
        go func() {
                for f := range cnp {
                        f()
                }
        }()
}
```

This loop runs 4 times, and in each iteration, it launches a new goroutine. Each goroutine contains a for-loop that continuously receives functions from the cnp channel and executes them **(f())**.

2. ***Giving use-cases of what these constructs could be used for.***

Make functions allow us to specify the type, length, and, optionally, the capacity.
Goroutines communicate using channels. Like slices and maps, channels are a built-in type created using the make function.

For loop allows you to iterate over a sequence of elements or repeat a set of statements until a specific condition is met.
In our case, we have a for loop that iterates 4 times and launches a new goroutine at every iteration.

3. ***What is the significance of the for loop with 4 iterations?***

Running the for loop with 4 iterations signifies that 4 goroutines could be run to achieve the concurrency in the program.

It allowed us to divide a task into multiple sub-tasks so that the execution could be faster.

4. ***What is the significance of make(chan func(), 10)?***

   Using the make function to declare a channel signifies that a channel of type function is created with some capacity. So, this channel holds the properties of a Buffered Channel.

   It limits our concurrence usage and the amount of work that is queued.

5. ***Why is "HERE1" not getting printed?***

   "HERE1" is not getting printed because the program terminates before the goroutine that processes the channel gets a chance to run. When **cnp <- func() { fmt.Println("HERE1") }** is executed, the main function does not wait for the goroutines to finish processing the channel. Instead, it immediately prints **"Hello"** and then exits, which stops all goroutines, including those that haven't had a chance to execute their tasks.

   So, to make this program run, we need to add some ***synchronization*** to wait for the goroutines to complete their work. We also have to close the channel so that no deadlock should be there.

```
package main

import (
	"fmt"
	"sync"
)

func main() {
	wg := sync.WaitGroup{}
	cnp := make(chan func(), 10)
	for i := 0; i < 4; i++ {
		wg.Add(1)
		go func() {
```

```
            defer wg.Done()
            for f := range cnp {
                f()
            }
        }()
    }
    cnp <- func() {
        fmt.Println("HERE1")
    }
    close(cnp)
    fmt.Println("Hello")
    wg.Wait()
}
```