Zadanie 1.

Przyklad na ktorym prezentuje dzialanie programu czerpie dane z pliku crc_input. W tym pliku uprzedno zostal roztal wygenerowany ciag zer I jedynek o dlugosci dokladnie 10021 znakow. Kazda ramka ma dlugosc 100 znakow czyli jak latwo obliczyc program powinien przeslac te dane w 101 ramkach. Procedura stworzenia ramki wyglada nastepujaco: najpierw okreslamy dzienik CRC w moim przykladzie przyjalem wartosc 1011, wartosc ta moze byc oczywiscie dowolna, oraz dlugosc moze byc rowniez dowolna z jednym zastrzezeniem – pierwszym znakiem dzielnika CRC musi byc 1.

```
CRC crc  = new CRC("1011");
```

Nastepnym krokiem jest pobranie danych z pliku tekstowego do pewnego stringa:

```
String inputContent = new String(Files.readAllBytes(Paths.get("./src/main/resources/crc_input")));
```

Zaraz potem czyscimy zawartosc pozostalych dwoch plikow tekstowych z ktorych bedziemy korzystac, po to aby dane nie mieszaly sie z poprzednio wykonanymi testami

```
new PrintWriter("./src/main/resources/crc_output").close();
new PrintWriter("./src/main/resources/crc_all_frames").close();
```

Plik crc_output przechowuje wynik naszego programu czyli polaczone dane wszystkich przeczytanych ramek, bedziemy pozniej porownywac zawartosc pliku crc_input z crc_output. Identycznosc tych dwoch plikow implikuje poprawnosc dzialania programu.
Plik crc_all_frames przechowuje zapisane wszystkie ramki ktore zostaly stworzone w celu wyslania informacji z pliku crc_input do crc_output. Plik ten jest szczegolnie przydatny w celu ustalenia poprawnosci generowania ramek. Ponizej przedstawione zostana przykladowe 3 ramki o dlugosci danych 100

```
01111110010001110000111010111010010111100000011110010001110011111001110110000111001101111000001111100110100101111011111110
01111110111000100101100011001011011011111000110010011111001100110110100110110111000110011111000001101111011110111001111110
01111110000001011011110101110111000001110111100001001001010000111010001101011010110001110011100111000100011000001111110
```

Na pierwszy rzut oka mozna zauwazyc ze kazda ramka zaczyna sie I konczy ciagiem znakow

`01111110` oraz pomiedzy tymi wartosciami nie da sie zauwazyc ciagow 5 jedynek pod rzad.
Serce programu znajduje sie w petli while ktora wykonuje sie tak dlugo az nie wyslemy wszystkich danych ktore chcemy wyslac. Jedoczesnie przy kazdej iteracji zwiekszamy liczbe ramek (wartosc ta bedzie nam potrzebna w raporcie wykonanym po zakonczeniu programu).

```
while (!inputContent.equals("")){
    framesNumber++;
```

Nastepnie mamy dwie mozliwosci: jesli dlugosc naszych danych jest wieksza niz pojedynczej ramki to wycinamy dlugosc danych maksymalych dla naszej ramki I dodajemy na koncu wartosc CRC, a reszte danych zostawiamy to przeslania w kolejnych iteracjach. Druga mozliwosc jest taka ze dane ktore mamy do przeslania zmieszcza sie w calosci w ramce, wtedy cala nasza wiadomosc zostanie przeslana ala uprzednio dodajemy do niej kod CRC.

```
if(inputContent.length() >= crc.getFrameDataSize()){
    message = crc.appendCRC(inputContent.substring(0, crc.getFrameDataSize()));
    inputContent = inputContent.substring(crc.getFrameDataSize(), inputContent.len
}
else{
    message = crc.appendCRC(inputContent);
    inputContent = "";
}
```

Natomiast procedura dodawania CRC do wiadomosci wyglada nastepujaco

```
public String appendCRC(String message) { return message + getCRC(message); }

private String getCRC(ArrayList<Character> message){

    while(message.get(0) == '0' && message.size() >= divisor.length()){
        message.remove(0);
    }

    if(message.size() + 1 == divisor.length()){
        return message.stream().map(Object::toString).collect(Collectors.joining());
    }

    for(int i = 0; i < divisor.length(); i++){
        if(message.get(i) == divisor.charAt(i)){
            message.set(i, '0');
        }
        else {
            message.set(i, '1');
        }
    }

    return getCRC(message);
}
```

Po obliczeniu CRC zwyczajnie doklejamy go do konca naszej wiadomosci. CRC obliczamy na zasadzie wykkonaniu doklejenia do naszej “czystej” wiadomosci n-1 bitow o wartoscu 0 (wartosc n to dlugosc wczesniej okreslonego dzielnika) oraz wykonaniu operacji XOR miedzy kazdym fragmentem naszej wiadomosci a dzielnikiem, jednoczesnie pomijamy bity wartosci 0 jesli takowe sie pojawia na poczatku wiadomosci. Procedure ta powtarzamy tak dlugo az rozmiar wiadomosci bedzie wynosil n -1.

Nastepnie wykonujemy 3 niezbedne operacje pierwsza z nich to rozepchanie bitow . Operacja ta polega na tym, ze po wystapieniu ciagu 5 jedynek stawiamy za nimi 0 nie zwazajac czy po tych pieciu jedynkach wystepuje 0 czy 1. Ma to na celu unikniecie przedwczesnego zakonczenia ramki podczas odczytu poniewaz wartosc 01111110 wskazuje koniec ramki. Dwie pozostale operacje to kolejno dodanie znacznika poczatku ramki oraz konca.

```
message = crc.stretchBites(message);
message = crc.addStartFrame(message);
message = crc.addEndFrame(message);
```

W tym momencie nasza ramka bylaby gotowa do wyslania. Jaka wspomnialem wczesniej zapisujemy gotowa ramke w pliku

```
try {
    Files.write(Paths.get("./src/main/resources/crc_all_frames"), (message + "\n").getBytes(), StandardOpenOption.APPEND)
}catch (IOException e) {;}
```

Nastepujace operacje beda mialy za cel wyluskac dane ktore chcielismy przeslac z ramki. Pierwsze dwie operacje to kolejno usuwanie ciegu znakow oznaczajacych poczatek I koniec ramki.

```
message = crc.removeStartFrame(message);
message = crc.removeEndFrame(message);
message = crc.shortBites(message);
```

Oczywiscie usuwajac ciag botow oznaczajacych koniec ramki robimy w ten sposob ze usuwamy pierwsze wystapienie ciagu 01111110 od poczatku pliku I to co byloby za tym ciagiem obcinamy (w praktyce jesli poprzednia procedure rozpychania bitow przeprowadzilismy prawidlowo to za tym ciagiem znakow nic nie powinno sie znajdowac.

```
public String removeEndFrame(String message){
    int i = 0;
    int j = 0;

    while(i < message.length()){
        if(message.charAt(i) == '1'){
            j++;
        }
        else {
            j = 0;
        }
        if(j == 6 && message.length() > i + 1 && message.charAt(i + 1) == '0') {
            return message.substring(0, i - 6);
        }
        i++;
    }
    return "";
}
```

Procedura shortBites to procedura odwrotna do procedury stretchBites. Dziala ona w ten sposob, ze jesli napotka ciag 11111 to usuwa zero ktore sie za tym ciagiem znajduje.

```
if(crc.checkCorrectnessData(message)){
    message = crc.removeCRC(message);
    try {
        Files.write(Paths.get("./src/main/resources/crc_output"), message.getBytes(), StandardOpenOption.APPEND);
    }catch (IOException e) {;}
}
```

Kolejny krok to sprawdzenie kodem crc czy wiadomosc jest taka sama jak ta ktora pakowalismy do ramki. Procedura jest podobna do tej w ktorej chcielismy uzyskac kod CRC, czyli polega na tym, ze na kazdej czesci naszej wiadomosci robimy operacje XOR z wczesniej ustalonym dzielnikiem. Jesli w wyniku tych operacji dostaniemy ciag tylko I wylacznie zer to oznacza, ze z duzym prawdopodobienstwem otrzymalismy ciag ktory wczesniej pakowalismy do ramki. Nie mamy tutaj absolytnie 100 % pewnosci, ze CRC wskaze nam jednoznacznie identycznosc danych wysylanych I odebranych poniewaz moze sie zdazyc, ze rozne ciagi znakow beda sie roznic a CRC wylkaze, ze sa one takie same. Prawdopodobienstwo takiej sytuacji jest jednak dosyc malo prawdopodobne I zalezy ono od dlugosci dzielnika (co jest rowniez dlugoscia kodu CRC pomniejszonym o jeden). Im dluzszy dzielnik tym teoretycznie wieksze szanse ze CRC daje poprawny wynik. Dlatego w praltyce zazwyczaj korzysta sie z CRC co najmniej 32 bitowych.

```
private boolean checkCorrectnessData(ArrayList<Character> message){

    while(message.get(0) == '0' && message.size() >= divisor.length()){
        message.remove(0);
    }

    if(message.size() + 1 == divisor.length()){
        for(int i = 0; i < message.size() - 1; i++){
            if(message.get(i) != '0'){
                return false;
            }
        }
        return true;
    }

    for(int i = 0; i < divisor.length(); i++){
        if(message.get(i) == divisor.charAt(i)){
            message.set(i, '0');
        }
        else {
            message.set(i, '1');
        }
    }

    return checkCorrectnessData(message);
}
```

Jesli procedura checkCorrectnessData wskaze na podstawie kodu CRC, ze ciag jest ten sam, to usuwamy CRC znajdujace sie na koncu naszego ciagu danych I wynik zapisujemy do pliku crc_output.

Ostatnim krokiem jest porownanie pliku crc_input oraz crc_output oraz wygenerowanie krotkiego raportu

```
outputContent = new String(Files.readAllBytes(Paths.get("./src/main/resources/crc_output")));
inputContent = new String(Files.readAllBytes(Paths.get("./src/main/resources/crc_input")));

System.out.println("---  REPORT  ---");
System.out.println("Frames sent: " + framesNumber);
System.out.println("Input file and output file have the same content: " + outputContent.equals(inputContent));
```

```
---  REPORT  ---
Frames sent: 101
Input file and output file have the same content: true
```

Zadanie 2.

To zadanie zrealizowalem w jezyku go ktory moim zdaniem lepiej realizuje wspolbieznosc niz inne jezyki programowania. Jak juz wyzej wspomnialem program jest napisany wspolbieznie symulujac przesylanie miedzy saba ramek enthernetowych.

Na potrzeby zadania korzystam z 6 zmiennych “globalnych” gdzie zmienna `var mutex = &sync.Mutex{}` blokuje dostep do danych spoldzielonych aby nie zachodzily kolizje podczas proby jednoczesnego zapisu danych w tym samym momencie przez dwa watki. Zmienna `var oneFieldTravelTime = 10.0` okresla czas w milisekundach potrzebny na przejscie z jednej komorki do drugiej, mozna to tlumaczyc w ten sposob ze jest to czas potrzebny na pokonanie pewnego odcinka na trasie kabla. Zmienna `var beginTimeConflict = 1` okresla ile czasu w sekundach dajemy na pierwsza probe rozwiazania konfliktu (wraz z podejmowanymi kolejnymi probami wartosc ta bedzie mnozona przez 2). Zmienna `var wg sync.WaitGroup` odpowiada za to aby program sie nie zakonczyl przed zakonczeniem pracy przez poszczegolne watki.

Program wykorzystuje rowniez 2 struktury gdzie pierwsza reprezentuje pole z ktorego tworzymy tablice I bedzie ona odwzorowywala nasz “kabel” a druga struktura bedzie reprezentowala hosta

```
type frame struct {
        sourceAddress string
        destinationAddress string
        data string
        lifetime float64
        startDate time.Time
        timeMargin float64
}

type host struct {
        ip string
        wireLocationIndex int
        receivedFrames []frame
        currentWireIndex int
        messageToSend string
        wire []frame
        framesToSend []string
        color string
}
```

Przyklad w ktorym demonstruje dzialanie programu sklada sie z 2 hostow oraz “kabla” reprezentowanego przez tablice majaca 20 elementow, gdzie kazdy host znajduje sie na jednym z koncow kabla.

Na ponizszym obrazku widac jak inicjuje tablice reprezentujaca “kabel” oraz dwa hosty

```
wire := make([]frame, wireLength)
for i := range wire{
        wire[i] = frame{ sourceAddress: "", destinationAddress: "", data: "", lifetime: 0,
                startDate: time.Date( year: 1970, month: 1, day: 1, hour: 1, min: 1, sec: 1, nsec: 1, time.UTC),
                timeMargin: -1}
}

host1  := host{ ip: "host1", wireLocationIndex: 0, receivedFrames: make([]frame, 0), currentWireIndex: 0,
        messageToSend: "", wire: wire, framesToSend: []string{"a", "b", "c", "d"}, color: "cyan"}
host2  := host{ ip: "host2", wireLocationIndex: wireLength - 1, receivedFrames: make([]frame, 0),
        currentWireIndex: wireLength -1, messageToSend: "", wire: wire, framesToSend: []string{"z", "y", "x", "w", "v", "u"},
        color: "green"}
```

Hosty te walczac o zasoby beda probowaly przeslac ciagi (kazda literka oznacza osobna ramke) odpowiednio pierwszy host ‘a’, ‘b’, ‘c’, ‘d’ oraz drugi host ‘z’, ‘y’, ‘x’, ‘w’, ‘v’, ‘u’.

Symulacja dziala na trzech watkach. Dwa pierwsze odpowiadaja za dwa hosty a trzeci za “kabel”. Hosty oczywiscie wrzucaja swoje ramki do “kabla” (w rzeczyswistosci prad o roznym napieciu) a kabel czyli watek ktory go reprezentuje odpowiada za czyszczenie komorek w tablicy po odpowiednim czasie.

Kazdy z hostow dziala z petli zawsze na poczatku sprawdzajac czy ma jakies do dane do wyslania, jesli takie ma to losuje liczbe z ustalonego przedzialu I czeka aby nie wysylac wszystkiego na raz,

natomiast jesli nie ma nic do wyslania to przechodzi w tryb nasluchiwania. Gdy host odczekal odpowiednia ilosc czasu I w kablu jest “cisza” to rozpoczyna nadawanie, jesli nie to czeka na odebranie wiadomosci ktora pozniej zapisuje.

```
func startHost(h* host){
	go func() {
		defer wg.Done()
		s1 := rand.NewSource(time.Now().UnixNano())
		r1 := rand.New(s1)
		for{
			mutex.Lock()
			toWait = beginTimeConflict
			mutex.Unlock()
			if len(h.framesToSend) == 0 {
				h.onlyListenToMessage()
			} else {
				h.listenToMessage(r1.Intn(hostRandRestTime) + 1)
			}
			for {
				h.messageToSend = h.framesToSend[0]
				if h.sendMessage() && h.simulateWaiting(){
					colorstring.Println(addColorToString(h.color, sumStrings(h.ip, "sent message: ",h.messageToSend, "successfully" )))
					h.framesToSend = append(h.framesToSend[:0], h.framesToSend[1:]...)
					break
				}else {
					colorstring.Println(addColorToString(h.color, sumStrings(h.ip, "detected conflict")))
					h.waitUntilWireBusy()
					h.handleConflict()
				}
			}
		}
	}()
}
```

```
func (h* host) listenToMessage(t int) {
	colorstring.Println(addColorToString(h.color, sumStrings(h.ip, "will wait for receive message ", strconv.Itoa(t), "seconds")))
	for j := 0; j < t; j++ {
		if h.wire[h.wireLocationIndex].data != "" && h.wire[h.wireLocationIndex].data != h.messageToSend{
			colorstring.Println(addColorToString(h.color, sumStrings(h.ip, "received successfully message: ", h.wire[h.wireLocationIndex].data)))
			h.receivedFrames = append(h.receivedFrames, h.wire[h.wireLocationIndex] )
			break
		}
		time.Sleep(time.Millisecond * time.Duration(oneFieldTravelTime))
	}
	for h.wire[h.wireLocationIndex].data != "" {
		time.Sleep(time.Millisecond * time.Duration(oneFieldTravelTime))
	}
}
```

Jesli w kablu jest cisza host moze zaczac nadawanie co jest realizowane w taki sposob ze najpierw stopniowo wypelniany jest kabel od hosta wysylajacego w kierunku hosta odbierajacego. Wypelnianie kabla jest realizowane na zasadzie dodawanie wartosci na w taki sposob ze jesli dwa hosty nadaja jednoczesnie wartosci musza sie spotkac I nalozyc na siebie. Host zawsze wysyla dane w czasie ktory jest potrzebny na przebycie drogi od hosta do najdalej oddalonego hosta. Czas ten zabezpiecza mozliwosc wykrycia kolizji. Jesli host wykryje kolizje, nie przestaje nadawac aby drugi host tez mogl ja wykryc.

```
func (h* host) sendMessage()bool {
	sentWithoutDisruption := true
	h.currentWireIndex = h.wireLocationIndex
	colorstring.Println(addColorToString(h.color, sumStrings(h.ip, "start to sending message")))
		for  range h.wire {
			mutex.Lock()
			h.wire[h.currentWireIndex] = h.getFrame()
			mutex.Unlock()
			if h.wire[h.wireLocationIndex].data != h.messageToSend {
				colorstring.Println(addColorToString(h.color, sumStrings(h.ip,
					"detected conflict but it stream all time to be sure that other hosts would detect the conflict")))
				sentWithoutDisruption = false
			}
			colorstring.Println(addColorToString(h.color, sumStrings( nums: "wire[", strconv.Itoa(h.currentWireIndex),
				"] = ", h.wire[h.currentWireIndex].data, " ")))
			h.incrementWireIndex()
			time.Sleep(time.Millisecond * time.Duration(oneFieldTravelTime))
		}
	return sentWithoutDisruption
}
```

Jesli kolizja zostala wykryta to host losuje liczbe z przedzialu 0 oraz 1. Gdy wylosuje 0 losuje kolejna liczbe z przedzialu 1 do n, w wypadku gdyby sytuacja sie powtorzyla to liczba bedzie losowana z przedzialu 1 do 2n itd. Natomiast jesli host wylosuje liczbe 1 to bezzwlocznie zaczyna nadawanie. Cala ta procedura moze maksymalnie zostac powtorzona 15 razy, po czym zaczynamy wszystko od nowa.

```
func (h* host) handleConflict(){
        s1 := rand.NewSource(time.Now().UnixNano())
        r1 := rand.New(s1)
        currentWait := 0
        if r1.Intn( n: 2) == 0 {
                currentWait = r1.Intn(toWait) + 1
                colorstring.Println(addColorToString(h.color, sumStrings(h.ip, "will wait",
                        strconv.Itoa(currentWait), "seconds before streaming to resolve conflict")))
                h.listenToMessage(currentWait)
                if toWait > int(math.Pow( x: 2,  y: 15)) {
                        mutex.Lock()
                        toWait = beginTimeConflict
                        mutex.Unlock()
                } else{
                        mutex.Lock()
                        toWait *= 2
                        mutex.Unlock()
                }
        }else {
                colorstring.Println(addColorToString(h.color, sumStrings(h.ip, "start streaming")))
        }
}
```

Testy przeprowadzam na “kablu” dlugosci 20 komorek, I zaczynam od czasu oczekiwania po wykryciu kofliktu na poziomie 1 sekundy.

```
host1 sent message:  d successfully
host1 dont have anything more to send, he will listen only
host1 finished work in time +2.338006e+001 seconds
```

```
host2 sent message:  u successfully
host2 dont have anything more to send, he will listen only
host2 finished work in time +2.609544e+001 seconds
```

Teraz ustawimy czas oczekiwania po wykryciu konfliktu na poziomie 10 sek

```
host1 dont have anything more to send, he will listen only
host1 finished work in time +2.298099e+001 seconds
```

```
host2 dont have anything more to send, he will listen only
host2 finished work in time +2.405206e+001 seconds
```

Teraz ustawmy na 20 sek

```
host2 finished work in time +2.293654e+001 seconds
host2 sent message:  u successfully
```

```
host1 finished work in time +2.353326e+001 seconds
host1 sent message:  d successfully
host1 dont have anything more to send, he will listen only
```

Teraz ustawmy na 2000 sek

```
host2 finished work in time +5.026903e+001 seconds
host2 sent message:  u successfully
host2 dont have anything more to send, he will listen only
```

```
host1 finished work in time +4.854045e+001 seconds
host1 sent message:  d successfully
host1 dont have anything more to send, he will listen only
```

I widzimy ze z ta wartoscia nie mozemy przesadzic bo jesli 2 hosty beda czekac to przeslanie danych trwa znacznie duzej