Zad. 1

a) wzajemne wykluczanie:
   Załóżmy, że mamy dwa wątki, 0 i 1, wtedy:
   write(turn = 0) -> read(busy == false) -> write (busy == true) -> read(turn == 0)
   write(turn = 1) -> read(busy == false) -> write (busy == true) -> read(turn == 1)

   Wiedząc, że wkraczają do sekcji krytycznej:
   read(turn == 0) -> write(turn = 1)
   read(turn == 1) -> write(turn = 0)
   Zatem otrzymujemy pętlę, której nie da się wykonać.
b) niezagłodzenie:
   Algorytm może powodować zagłodzenie, ponieważ zmienna turn może być zmieniana przez inne wątki podczas czekania w pętli, co może prowadzić do sytuacji, w której dany wątek nigdy nie otrzyma swojej kolejki.
   write(turn = 0) -> read(busy == false) -> write(busy = true) -> read(turn == 0) -> CS
   write(turn = 1) -> read(busy == true)
   wątek(0) robi unlock() i od razu wchodzi do lock()
   write(busy = false)
   write(turn = 0) -> read(busy == false) -> write(busy = true) -> read(turn = 0) -> CS
   Dochodzi do zagłodzenia wątku(1)
c) niezakleszczenie:
   write(turn = 0) -> read(busy == false) -> write(busy = true)
   write(turn = 1) -> read(busy == true)
   read(turn == 1) -> write(turn = 1) -> read(busy == true)

Zad. 2

a) niezakleszczenia:
   Z wykładu wynika, że lock() nie może wywoływać zakleszczenia, więc w sytuacji gdy oba wątki są w nim, któryś wejdzie do sekcji krytycznej.
   Rozważmy teraz unlock(),
   1. Niech jeden wątek(0) będzie w unlock(), a drugi wątek(1) w lock() -> flag[0]=false, flag[1]=true. Brak zakleszczenia, ponieważ wątek(1) wejdzie do sekcji krytycznej.
   2. Niech obydwa wątki w unlock() -> flag[0]=false, flag[1]=false. Brak zakleszczenia.

   Czyli spełnia warunek.

b) niezagłodzenia:
   1. Niech wątek(0) wyjdzie z sekcji krytycznej do unlock() -> flag[0] = false, wątek(0) czeka w pętli.
   2. Wątek(1) wchodzi do lock() -> flag[1] = true i wchodzi do sekcji krytycznej.
   3. Wątek(1) wychodzi z sekcji krytycznej i wchodzi do unlock() -> flag[1] = false, ale nie wchodzi do pętli (flag[0] = false)
   4. Wątek(1) wraca do lock()

   Niestety, w tej wersji algorytmu pętla ‘while (flag[j] == true)’ może prowadzić do zagłodzenia, ponieważ drugi wątek może nigdy nie zmienić swojej flagi. W tym przypadku dochodzi do zapętlenia i wątek(0) zostaje zagłodzony. Czyli nie spełnia warunku.

Zad. 3

**Co to jest r-ograniczone czekanie?**
Algorytm ma własność r-ograniczonego czekania, jeśli istnieje liczba r, która gwarantuje, że każdy wątek czeka na swoją kolej nie dłużej niż r razy, gdy inne wątki wchodzą do sekcji krytycznej.

**Sekcja wejściowa i sekcja oczekiwania w algorytmie Petersona**

- **Sekcja wejściowa** (doorway section): Fragment kodu, w którym wątek oznacza swoje zamiary wejścia do sekcji krytycznej.
  flag[i] = true;
  victim = i;

- **Sekcja oczekiwania** (waiting section): Miejsce, w którym wątek faktycznie czeka na swoją kolej, tj. pętla, gdzie sprawdzane są zmienne flagowe.
  while (flag[j] && victim == i) {};

**FCFS:**

$D_A$ -> $D_B$ => $CS_A$ -> $CS_B$

while (flag[1] == false && victim== 0) => 0 wchodzi pierwsze do sekcji krytycznej.

while (flag[1] == true && victim==0) => 0 czeka

while (flag[1] == true && victim == 1) => 0 wchodzi pierwsze do sekcji krytycznej.

Zad. 4

# Bakery Algorithm

- Provides First-Come-First-Served for *n* threads
- How?
  - Take a “number”
  - Wait until lower numbers have been served
- Lexicographic order
  - (a,i) > (b,j)
    - If a > b, or a = b and i > j

```
class Bakery implements Lock {
   boolean[] flag;
   Label[] label;
  public Bakery (int n) {
     flag  = new boolean[n];
     label = new Label[n];
     for (int i = 0; i < n; i++) {
        flag[i] = false; label[i] = 0;
     }
  }
```

```
public void lock() {
 flag[i]  = true;
 label[i] = max(label[0], …,label[n-1])+1;
 while (∃k flag[k]
          && (label[i],i) > (label[k],k));
}
```

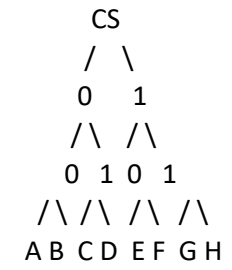
```
 public void unlock() {
    flag[i] = false;
 }
}
```

a) Wzajemne wykluczanie:
Niech wątki ‘0’ i ‘1’ dostaną się do sekcji krytycznej w tym samym momencie i label[0] < label[1]. Kiedy ‘1’ wchodziło to musiało widzieć flag[0]=false lub label[0] > label[1].
label(1) => read(flag(0)==false) => write(flag(0)) => label(0)
a to jest sprzeczne z założeniem, że ‘0’ ma mniejszy label, czyli dla dwóch wątków z tym samym labelem do sekcji krytycznej wejdzie ten, który ma mniejszy numer, czyli w sekcji krytycznej będzie max jeden wątek.

b) Brak zakleszczeń:
Zawsze istnieje wątek o najniższym label -> zawsze zostanie wpuszczony do sekcji krytycznej. Niemożliwe jest, żeby dwa wątki miały taki sam label, bo każdy wątek dostaje max(label) + 1

c) Niezagłodzenie:
Nie dojdzie do zagłodzenia, ponieważ zawsze nadejdzie moment, że dany wątek będzie miał najmniejszą nieobsłużoną etykietę.

Zad. 5




a) wzajemne wykluczanie:
   Jeśli wątek A założy blokadę w liściu, to zostanie "przeniesiony" na następny poziom. Wtedy jeśli wątek B będzie chciał założyć blokadę to zostanie na tym samym poziomie bo flag[B] == true, flag[A] == true, victim == B. Inne wątki mogą również chcieć założyć blokady na swoich liściach. Zauważmy, że tylko połowa z tych wątku przejdzie do następnego poziomu. Kontynuujemy ten proces dla kolejnych poziomów, aż zostanie nam tylko 1 wątek, które wejdzie do sekcji krytycznej. Skoro każdy wierzchołek spełnia warunek wzajemnego wykluczania, stąd całe drzewo spełnia ten warunek.
b) niezagłodzenie:
   Każdy lock() Petersona spełnia warunek niezagłodzenia, zatem każdy wierzchołek spełnia ten warunek, stąd na każdym poziomie drzewa nie dochodzi do zagłodzenia, czyli każdy wątek kiedyś przejdzie do następnego poziomu.
c) niezakleszczenie:
   Niech wątek A wejdzie do sekcji krytycznej. Po wyjściu A wiemy, że metoda unlock() zwalnia wszystkie zajęte wcześniej zamki przez A, zatem wszystkie wątki, które wcześniej "przegrały" z A wejdą na następny poziom.

Zad. 6

Zobaczyć innych!

Zad. 7

a) wzajemne wykluczenie:
   Jeśli wiele wątku dotrze jednocześnie do ‘y = i’ to ponieważ ‘x’ posiada już wartość, wszystkie wątki wejdą do sekcji krytycznej, czyli nie jest spełnione wzajemne wykluczenie.
b) niezagłodzenie:
   Jeśli wątek będzie wolniejszy od innych, może utknąć w pętli while i zagłodzić się.
c) niezakleszczenie:
   Załóżmy nie wprost, że dochodzi do zakleszczenia, mogło to się stać w pętli while.
   Wtedy jakiś wątek nadpisuje ‘y = -1’ na swoje ‘i’. Nadpisuje też ‘y’ i wchodzi do sekcji krytycznej. Później musi wyjść z sekcji krytycznej i ustawia ‘y = -1’. A to nie powinno być możliwe jeżeli wszystkie wątki czekają. Zatem nie dojdzie do zakleszczenia.

Zad. 8

a) Aby więcej niż jeden wątek zwróciło wartość ‘STOP’, trzeba spełnić wielokrotnie równanie ‘last == i’
   - jeden z wątków zwraca STOP, to kolejne będą wykonywać ‘visit’ z ‘goRight = true’, więc zwrócą ‘RIGHT’
   - co najmniej dwa wątki jednocześnie wykonują ‘visit’. Wtedy ‘last’ jest przypisane przez tylko jeden wątek, czyli tylko on skończy z ‘STOP’
b) Załóżmy nie wprost, że n wątków może otrzymać wartość zwracaną DOWN.
   Każdy wątek musi spełnić ‘last != i’. Natomiast aby dojść do tego warunku, każdy z wątków musi wykonać instrukcję ‘last = i’. N-ty z kolei wątek, który nadpisał tą zmienną spełni warunek ‘last == i’ zwróci wartość STOP. Sprzeczność.
c) Aby każdy wątek zwrócił wartość RIGHT, każdy wątek musiałby odczytać ‘goRight = true’, co jest niemożliwe, ponieważ zmienna jest zaczyna na ‘false’, a jedyne przypisanie znajduje się po odczytaniu wartości.

Zad. 9

Z danego wierzchołka wątki mają przynajmniej dwa różne "wyjścia". Jak pokazaliśmy w poprzednim zadaniu jedno "wyjście" może być odwiedzone razy.
Obserwacja: jeśli tylko jeden wątek dotyka określonego visit() to dostanie STOP.
W szczególności jakiś proces na pewno zostanie na pierwszym poziomie. W każdym wierzchołu przynajmniej jeden wątek, albo poprzez otrzymanie STOP, albo poprzez goRight, zostanie na tym samym poziomie. Jest tak ponieważ Down można dostać maksymalnie , a jeśli mamy tylko jeden proces w danym wierzchołku to dostanie on STOP.
Analogicznie na każdym poziomie jeden wątek będzie musiał na nim pozostać.
Zatem proces jest skończony.
Możemy dotrzeć do dowolnego wierzchołka w odległości manhattańskiej co najwyżej ‘n - 1’ od źródła.
A zatem mamy ‘n * (n + 1)/2’ odwiedzonych wierzchołków.
Realną liczbę odwiedzonych wierzchołków możemy oszacować przez zależność rekurencyjną F:
$F(0) = 0$
$F(1) = 1$ – jeśli tylko jeden wątek jest w wierzchołku to w nim zakończy.
$F(i) = \max_{a+b=i}(F(a) + F(b)) + 1$
Rozwiązaniem tej zależności rekurencyjnej jest , dowodem jest prosty dowód indukcyjny.