GENERATOR LICZB LOSOWYCH

Rachunek prawdopodobieństwa i statystyka

Technologia: **Java i Python**

Celem mojego projektu jest stworzenie siedmiu generatorów liczb pseudolosowych (**G**, **J**, **B**, **D**, **P**, **W**, **N**) – każdy z nich jest inny i spełnia odpowiednie wymagania.

1. Generator wyjściowy:
   - Generator **G** – generuje liczby całkowite
2. Generatory stworzone na podstawie **G**:
   - Generator **J** – generuje liczby z przedziału (0, 1)
3. Generatory stworzone na podstawie **J**:
   - Generator **B** – generuje liczby z rozkładu Bernoulliego
   - Generator **D** - generuje liczby z rozkładu dwumianowego
   - Generator **P** - generuje liczby z rozkładu Poissona
   - Generator **W** - generuje liczby z rozkładu wykładniczego
   - Generator **N** - generuje liczby z rozkładu normalnego

Zdecydowałem się na użycie prostego generatora multiplikatywnego zadanego wzorem $X_{i+1} = aX_{i-1} \; mod \; m$, gdzie:

X_0, X_1 … X_k – liczby, które są generowane (X_{i-1} jest zwane ziarnem),

a – pierwszy parametr generatora,

m – drugi parametr generatora.

Po prawej przykładowa tabela z wartościami liczb pseudolosowych w zależności od parametru *a* = i+1 oraz *m* = 11.

					X_i					
	a=1	2	3	4	5	6	7	8	9	10
i=0	1	**1**	1	1	1	**1**	**1**	**1**	1	1
1	1	**2**	3	4	5	**6**	**7**	**8**	9	10
2		**4**	9	5	3	**3**	**5**	**9**	4	1
3		**8**	5	9	4	**7**	**2**	**6**	3	
4		**5**	4	3	9	**9**	**3**	**4**	5	
5		**10**	1	1	1	**10**	**10**	**10**	1	
6		**9**				**5**	**4**	**3**		
7		**7**				**8**	**6**	**2**		
8		**3**				**4**	**9**	**5**		
9		**6**				**2**	**8**	**7**		
10		**1**				**1**	**1**	**1**		

Generator G

Generator G generuje liczby całkowite, o rozkładzie równomiernym. Jest to pierwszy generator, na podstawie którego będą działać wszystkie pozostałe generatory.

```
static ArrayList<Integer> G(int a, int mod, int seed, int amount){

    ArrayList<Integer> numbersG = new ArrayList<>( initialCapacity: amount+1);
    numbersG.add( index: 0, seed);

    for (int i = 1; i < amount+1; i++) {
        numbersG.add(i,  element: a*numbersG.get(i-1)%mod);
    }

    return numbersG;
}
```

Funkcja w implementacji przyjmuje parametry *a*, *mod*, *seed* oraz *amount*. Są to odpowiednio: parametr pierwszy, parametr drugi, ziarno oraz ilość liczb, które chcemy wygenerować.

Jako pierwszy element ciągu liczb wstawiamy ziarno, a kolejne elementy aż do *amount+1* zostają wyliczone z wzoru ogólnego.

Na sam koniec zwracana jest cała sekwencja liczb.

Dobór parametrów jest ważny, dlatego, że np. argument *mod* określa nasz górny zakres liczbowy. Przykładowo – dla *mod* = 11, będziemy dostawać liczby nie większe niż 10. Parametr *a* jest z góry ustalony i określa okresowość generowanego ciągu. Ziarno, czyli X_0 jest dowolne, ale determinuje liczby, które wygenerujemy.

Dla jak najlepszej „losowości” powinniśmy używać bardzo dużego *mod* oraz bardzo dużego *a* względnie pierwszego z *mod*. Dzięki temu zmniejszamy szansę na powtórzenie się jakichś liczb w naszym ciągu i zmniejszamy okresowość.

Generator J

Generator J generuje liczby z przedziału (0;1), o rozkładzie równomiernym. Jest to drugi generator, zbudowany na podstawie generatora G. Jest bardzo ważny, dlatego, że wszystkie pozostałe generatory z niego korzystają.

```
static ArrayList<Double> J(int a, int mod, int seed, int amount){

    ArrayList<Integer> numbersG;
    numbersG = G(a, mod, seed, amount);

    ArrayList<Double> numbersJ = new ArrayList<>(numbersG.size());

    for (int i = 0; i < numbersG.size(); i++) {
        numbersJ.add(i,  element: (double) numbersG.get(i) / (mod));
    }

    return numbersJ;
}
```

Argumenty generatora J są identyczne jak w G, ponieważ za pomocą tych argumentów generujemy najpierw ciąg liczb całkowitych z G. Następnie każdą z tych liczb dzielimy przez argument *mod*, dzięki czemu otrzymujemy ciąg liczb z przedziału (0;1).

Jest to poprawna metoda, dlatego, że *mod* górnie ogranicza nam zakres liczb, które możemy wygenerować w G. Przykładowo:

- Dla *mod* = 11, z G możemy otrzymać ciąg {3, 2, 1, 4, 5, 10}. W J dzielimy każdą z liczb przez *mod*, więc ciąg z J będzie wyglądał następująco: {3/11, 2/11, 1/11, 4/11, 5/11, 10/11}. Jak widać nigdy nie dostaniemy liczby większej niż 1 lub mniejszej od 0.

Generator B

Generator B generuje liczby z rozkładu Bernoulliego, czyli rozkładu dwupunktowego. Posiada on parametr *p*, którym zadajemy prawdopodobieństwo sukcesu.

```
static ArrayList<Integer> B(int a, int mod, int seed, int amount, double p){

    ArrayList<Double> numbersJ;
    numbersJ = J(a, mod, seed, amount);

    ArrayList<Integer> numbersB = new ArrayList<>(numbersJ.size());

    for (int i = 0; i < numbersJ.size(); i++) {
        if(numbersJ.get(i) <= p)
            numbersB.add(1);
        else
            numbersB.add(0);
    }

    return numbersB;
}
```

Najpierw generujemy ciąg z J za pomocą klasycznych parametrów. Następnie iterujemy się przez nasz ciąg J i sprawdzamy czy dany element jest mniejszy bądź równy *p*. Jeśli jest, to do wynikowego ciągu z B dodajemy 1. W przeciwnym przypadku dodajemy 0.

Przykładowo, dla *p* = 0.7, 70% liczb z przedziału (0;1) się wlicza, więc jeśli:

ciąg z J = {0.3, 0.1, 0.4, 0.9, 0.2}, to

ciąg z B = {1 , 1 , 1 , 0 , 1 }.

Generator D

Generator D generuje liczby z rozkładu dwumianowego, liczbę sukcesów w *n* próbach Bernoulliego. Posiada on parametr *p*, którym zadajemy prawdopodobieństwo sukcesu oraz *n* określający ilość prób.

```java
static ArrayList<Integer> D(int a, int mod, int seed, double p, int n){

    ArrayList<Double> numbersJ;
    ArrayList<Integer> numbersD = new ArrayList<>( initialCapacity: n + 1);

    int successCounter = 0;

    for (int i = 0; i < n+1; i++) {
        numbersJ = J(a, mod,  seed: seed + i, n);

        for (int j = 0; j < numbersJ.size(); j++) {
            if(numbersJ.get(j) <= p)
                successCounter++;
        }

        numbersD.add(successCounter);
        successCounter = 0;
    }

    return numbersD;
}
```

W generatorze D musimy *n* razy wygenerować ciąg z J oraz następnie sprawdzić, ile elementów w tym ciągu spełnia próbę Bernoulliego.

W praktyce, dla *n* = 2, *p* = 0.2 oraz wygenerowanych ciągów:

J_1 = {0.2, 0.3, 0.1}, J_2 = {0.5, 0.2, 0.7} i J_3 = {0.6, 0.4, 0.9}

otrzymamy D = {2, 1, 0}.

Generator P

Generator P generuje liczby z rozkładu Poissona. Posiada on parametr λ, który określa oczekiwaną liczbę zdarzeń w danym czasie. Algorytm użyty do generacji liczb to algorytm Knutha.

```java
static ArrayList<Integer> P(int a, int mod, int seed, int lambda, int amount){

    ArrayList<Double> numbersJ;
    numbersJ = J(a, mod, seed, amount);

    ArrayList<Integer> numbersP = new ArrayList<>( initialCapacity: amount + 1);

    double L = Math.exp(-lambda);
    int k = 0;
    double p = 1;
    int j = 0;

    for (int i = 0; i < amount + 1; i++) {
        while(p > L){
            k++;
            p = p * numbersJ.get(j);
            j++;

            if(j == numbersJ.size()-1)
                j = 0;
        }
        numbersP.add(k-1);
        k = 0;
        p = 1;
    }

    return numbersP;
}
```

Najpierw generujemy ciąg z J. Następnie, postępując zgodnie z algorytmem Knutha generujemy kolejne liczby rozkładu Poissona.

```
algorytm poisson random number (Knuth):
 init:
  Let L ← e^-λ, k ← 0 i p ← 1.
 do:
  k ← k + 1.
  Wygeneruj losową liczbę u z przedziału [0,1] i przypisz p ← p × u.
 while p > L.
 return k - 1.
```

Generator W

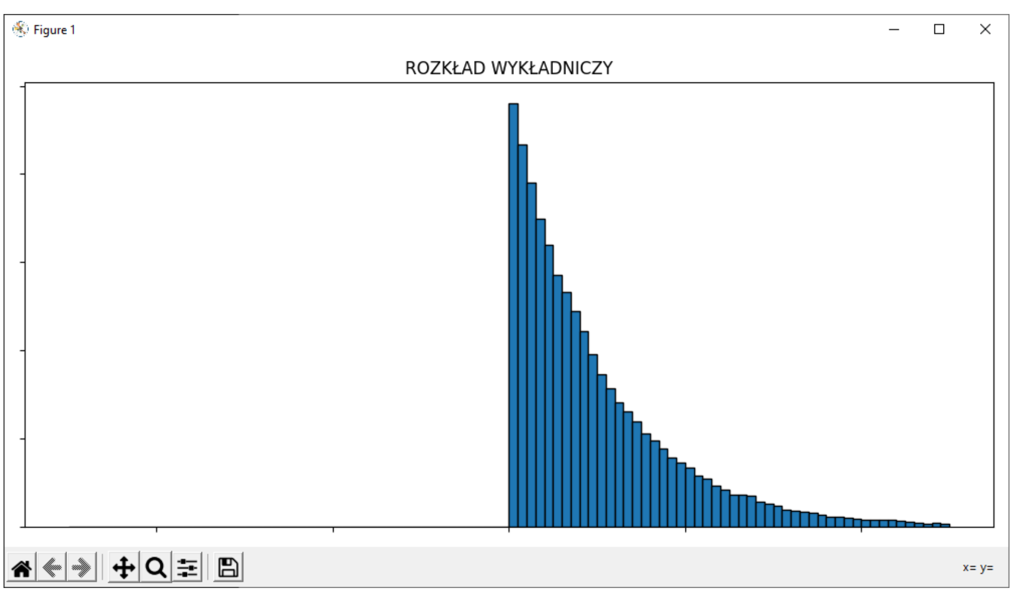
Generator W generuje liczby z rozkładu wykładniczego przy pomocy liczb pseudolosowych z generatora J.

```java
static ArrayList<Double> W(int a, int mod, int seed, int amount){

    ArrayList<Double> numbersJ;
    numbersJ = J(a, mod, seed, amount + 1);

    ArrayList<Double> numbersW = new ArrayList<>(amount + 1);

    for (int i = 0; i < amount + 1; i++) {
        numbersW.add(-Math.log(1 - numbersJ.get(i)));
    }

    return numbersW;
}
```

Najpierw generujemy ciąg z J, a następnie korzystając z wzoru:

$$x = -ln(1 - U)$$

generujemy ciąg liczb pseudolosowych na jego podstawie. Wygenerowany ciąg liczb z rozkładu wykładniczego przedstawiam na wykresie (ilość wygenerowanych liczb = 100000):



Generator N

Generator N generuje liczby z rozkładu normalnego. Generujemy pary liczb na podstawie losowych wartości ciągu z J.

```
static ArrayList<Double> N(int a, int mod, int seed, int amount){

    ArrayList<Double> numbersJ;
    numbersJ = J(a, mod, seed, amount);

    ArrayList<Double> numbersN = new ArrayList<>( initialCapacity: amount + 1);

    for (int i = 0; i < numbersJ.size() - 1; i = i + 2) {
        numbersN.add(Math.sqrt(-2 * Math.log(1 - numbersJ.get(i)))
                * Math.cos(2 * Math.PI*numbersJ.get(i+1)));
        numbersN.add(Math.sqrt(-2 * Math.log(1 - numbersJ.get(i)))
                * Math.sin(2 * Math.PI*numbersJ.get(i+1)));
    }

    return numbersN;
}
```

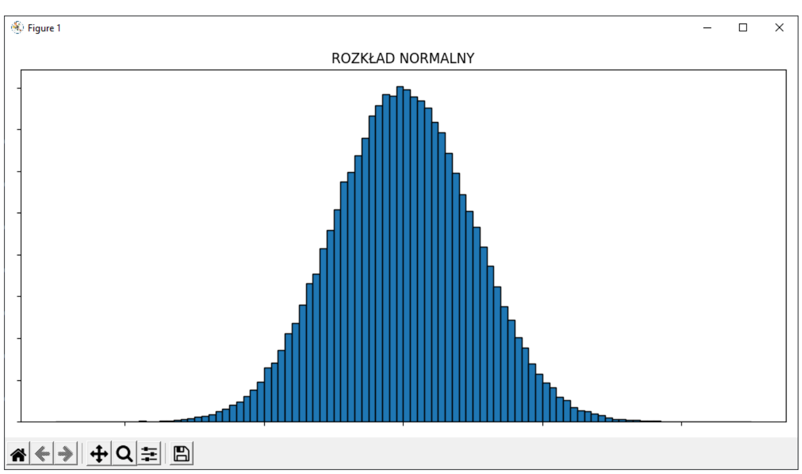
Na początku generujemy ciąg z J. Następnie, korzystając ze wzoru:

$$x = rcos(\theta) = \sqrt{-2ln(1-U_1)}cos(2\pi U_2)$$

$$y = rsin(\theta) = \sqrt{-2ln(1-U_1)}sin(2\pi U_2)$$

generujemy parę liczb x i y.

Na końcu do ciągu z N dodajemy kolejno liczbę x oraz y, po czym iterujemy się na kolejne dwa następne elementy J i robimy to samo.



Testowanie generatorów

Aby sprawdzić poprawność działania generatora, zastosowałem dwa sposoby testowania: test serii oraz test χ^2.

1. Test serii

Test serii, badający losowość próby wykonuje się poprzez postępowanie z listą kroków określoną poniżej:

- Ustalamy stały poziom istotności *alpha* i korzystamy z tablic wartości dla tego poziomu istotności.
- Obliczamy *medianę* z ciągu liczb pseudolosowych
- Na podstawie mediany oraz wejściowego ciągu *X* tworzymy nowy ciąg *Y*, który przyjmuje wartości:
  - *a* – gdy X_i > *mediana*
  - *b* – gdy X_i < *mediana*.
  - elementy X_i = *mediana* pomijamy.
- Obliczamy ilość *serii* = U_n, czyli podciągów *a* i *b* w ciągu *Y*.
- Obliczamy ilość wystąpień znaków *a* i *b* ($n_1 = \#_a$, $n_2 = \#_b$)
- Stawiamy hipotezy: H_0 oraz H_1, gdzie H_0 oznacza, że próby mają charakter losowy, a H_1, że nie mają charakteru losowego.
- Tworzymy zbiór krytyczny: $K = (-\infty; k_1\rangle \cup \langle k_2; \infty)$, gdzie
  - k_1 odczytujemy z tablic dla *alpha*/2 oraz n_1 i n_2
  - k_2 odczytujemy z tablic dla 1 - *alpha*/2 oraz n_1 i n_2
- Sprawdzamy czy $U_n \in K$.
  - Jeśli należy, to odrzucamy hipotezę H_0.
  - Jeśli nie należy, to nie odrzucamy hipotezy H_0.

Postępując zgodnie z tymi krokami możemy uzyskać informację na temat „losowości" naszego ciągu. Przyjmuję, że *alpha* = 0.05.

```python
def testSeriiShort(ciag):

    ciagSorted = ciag.copy()
    ciagSorted.sort()

    mediana = ciagSorted[int((len(ciagSorted))/2)] if len(ciagSorted)%2 != 0 else (1/2) * (ciagSorted[int(len(ciagSorted)/2)] + ciagSorted[(int((len(ciagSorted) - 1)/2))])

    ciagAiB = []

    for i in range(len(ciag)):
        if ciag[i] > mediana:
            ciagAiB.append('a')
        elif ciag[i] < mediana:
            ciagAiB.append('b')
        else:
            ciagAiB.append('-')

    liczbaSerii = 1
    aCounter = 0
    bCounter = 0

    for i in range(len(ciagAiB)-1):
        if ciagAiB[i] != '-':
            if ciagAiB[i] == 'a' and (ciagAiB[i+1] != 'a'):
                if i != len(ciagAiB)-2:
                    liczbaSerii += 1
            elif ciagAiB[i] == 'b' and (ciagAiB[i+1] != 'b'):
                if i != len(ciagAiB)-2:
                    liczbaSerii += 1

    for i in range(len(ciagAiB)):
        if ciagAiB[i] == 'a':
            aCounter += 1
        elif ciagAiB[i] == 'b':
            bCounter += 1

    k1 = rozkladSerii1[aCounter][bCounter]
    k2 = rozkladSerii2[aCounter][bCounter]

    print("------------------------------------------------------------------------------------------------------------------------")
    print("TEST SERII: KRÓTKI")
    print("------------------------------------------------------------------------------------------------------------------------")
    print("CZY", liczbaSerii, "∈ (-inf ;", k1, "]","∪", "[", k2, "; +inf]?       ", "NIE" if k1 < liczbaSerii < k2 else "TAK")
    print("------------------------------------------------------------------------------------------------------------------------")
    print("FINALNY WERDYKT:                          ", "CIĄG DOBRZE LOSOWY" if k1 < liczbaSerii < k2 else "CIAG SŁABO LOSOWY")
    print("------------------------------------------------------------------------------------------------------------------------")
    print("########################################################################################################################")
```

W powyższym kawałku kodu wykonuję dokładnie listę kroków testu serii. Najpierw wczytuję ciąg, sortuję go oraz obliczam medianę. Następnie tworzę ciąg *Y* z liter *a* i *b* w zależności od mediany. Później liczę ilość serii U_n i zliczam *a* i *b* (n_1, n_2). Dzięki n_1 oraz n_2 mogę dostać krańce zbioru krytycznego *K*, odczytać wartości z tablic oraz wydać finałowy werdykt na temat „losowości” ciągu wejściowego. Wynik przedstawiam w tabeli.

Przykład:

```
------------------------------------------------------------------------------------------------------------------------
TEST SERII: DŁUGI
------------------------------------------------------------------------------------------------------------------------
CIAG WEJSCIOWY:                            [16, 20, 25, 34, 22, 33, 47, 30, 28, 19, 22, 40, 36, 31, 38]
------------------------------------------------------------------------------------------------------------------------
CIAG WEJSCIOWY POSORTOWANY:                [16, 19, 20, 22, 22, 25, 28, 30, 31, 33, 34, 36, 38, 40, 47]
------------------------------------------------------------------------------------------------------------------------
MEDIANA CIAGU WEJSCIOWEGO:                 30
------------------------------------------------------------------------------------------------------------------------
CIAG SERIOWY a & b:                        ['b', 'b', 'b', 'a', 'b', 'a', 'a', '-', 'b', 'b', 'b', 'a', 'a', 'a', 'a']
------------------------------------------------------------------------------------------------------------------------
LICZBA SERII a & b:                        6
------------------------------------------------------------------------------------------------------------------------
LICZBA a:                                  7
------------------------------------------------------------------------------------------------------------------------
LICZBA b:                                  7
------------------------------------------------------------------------------------------------------------------------
ZBIOR KRYTYCZNY:                           (-inf ; 3 ] ∪ [ 12 ; +inf]
------------------------------------------------------------------------------------------------------------------------
CZY 6 ∈ (-inf ; 3 ] ∪ [ 12 ; +inf]?         NIE
------------------------------------------------------------------------------------------------------------------------
FINALNY WERDYKT:                           CIĄG DOBRZE LOSOWY
------------------------------------------------------------------------------------------------------------------------
########################################################################################################################
```

2. Test χ^2 (chi-kwadrat)

Test chi-kwadrat zamiast losowości danego ciągu sprawdza czy generowane dane są z na pewno generowane z dobrego rozkładu.

Test chi-kwadrat jest dany wzorem:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

W praktyce polega na podzieleniu danych wejściowych na odpowiednie podzbiory oraz zliczenie, ile wartości ciągu mieści się w danych przedziałach. Następnie porównujemy te dane z wartościami oczekiwanymi w tych przedziałach dla konkretnego rozkładu. Jeśli dane te są odpowiednio do siebie zbliżone oraz współczynnik podobieństwa jest wystarczająco duży, to możemy przyjąć, że liczby generowane są zgodne z rozkładem.

Test chi-kwadrat wykonuję w sposób następujący:

- Generuję ciągi losowe z generatorów P i B
- Dzielę ciąg na kubełki (przedziały) oraz zliczam ilość wystąpień elementów z danego ciągu w przedziale – są to tak zwane wartości *obserwowane*
- Liczę wartości *oczekiwane* poprzez wyliczenie prawdopodobieństwa na wystąpienie danego elementu ciągu w konkretnym przedziale
- Obliczam współczynnik chi-kwadrat z wzoru powyżej
- Porównuję współczynnik chi-kwadrat z wartością z tabeli rozkładu chi-kwadrat zależnej od *stopni swobody* (ilość kubełków, przedziałów)
- Sprawdzam współczynnik podobieństwa – jeśli jest wystarczająco duży, to znaczy, że dwa ciągi mają ten sam rozkład i tym samym generator generuje dobre liczby.

Implementacja testów chi-kwadrat dla B oraz P:

```python
def chiKwadratP(ciag, lambdaP):

    ciagSorted = ciag.copy()
    ciagSorted.sort()

    ciagNoDups = list(dict.fromkeys(ciagSorted.copy()))

    alpha = 0.05
    degFreedom = 5 # dzielimy na 6 kubełków

    observed = [0] * 6

    estimated = oczekiwanaP(lambdaP, ciagNoDups, ciag)

    j = 0

    for i in range(len(ciagSorted)):
        if 0 <= ciag[i] <= 2:
            observed[0] += 1
        elif 2 < ciag[i] <= 4:
            observed[1] += 1
        elif 4 < ciag[i] <= 6:
            observed[2] += 1
        elif 6 < ciag[i] <= 8:
            observed[3] += 1
        elif 8 < ciag[i] <= 10:
            observed[4] += 1
        else:
            observed[5] += 1

    chi = 0

    for i in range(len(observed)):
        chi += pow(observed[i] - estimated[i], 2) / estimated[i]

    if chi < rozkladChi[degFreedom]:
        print("CIAG POSIADA DOBRE LICZBY Z ROZKŁADU POISSONA")
        print("########################################################", end='\n\n')
    else:
        print("CIAG POSIADA ZŁE LICZBY Z ROZKŁADU POISSONA")
        print("########################################################", end='\n\n')
```

```python
def oczekiwanaP(lambdaP, ciagNoDups, ciag):

    result = [0, 0, 0, 0, 0, 0]
    amount = len(ciag)

    result[0] = (amount*pow(lambdaP, ciagNoDups[0])*math.exp(-lambdaP)/math.factorial(ciagNoDups[0]))
    result[0] += (amount*pow(lambdaP, ciagNoDups[1])*math.exp(-lambdaP)/math.factorial(ciagNoDups[1]))
    result[0] += (amount*pow(lambdaP, ciagNoDups[2])*math.exp(-lambdaP)/math.factorial(ciagNoDups[2]))
    result[1] = (amount*pow(lambdaP, ciagNoDups[3])*math.exp(-lambdaP)/math.factorial(ciagNoDups[3]))
    result[1] += (amount*pow(lambdaP, ciagNoDups[4])*math.exp(-lambdaP)/math.factorial(ciagNoDups[4]))
    result[2] = (amount*pow(lambdaP, ciagNoDups[5])*math.exp(-lambdaP)/math.factorial(ciagNoDups[5]))
    result[2] += (amount*pow(lambdaP, ciagNoDups[6])*math.exp(-lambdaP)/math.factorial(ciagNoDups[6]))
    result[3] = (amount*pow(lambdaP, ciagNoDups[7])*math.exp(-lambdaP)/math.factorial(ciagNoDups[7]))
    result[3] += (amount*pow(lambdaP, ciagNoDups[8])*math.exp(-lambdaP)/math.factorial(ciagNoDups[8]))
    result[4] = (amount*pow(lambdaP, ciagNoDups[9])*math.exp(-lambdaP)/math.factorial(ciagNoDups[9]))
    result[4] += (amount*pow(lambdaP, ciagNoDups[10])*math.exp(-lambdaP)/math.factorial(ciagNoDups[10]))
    result[5] += (amount*pow(lambdaP, 11)*math.exp(-lambdaP)/math.factorial(11))
    result[5] += (amount*pow(lambdaP, ciagNoDups[11])*math.exp(-lambdaP)/math.factorial(ciagNoDups[11]))

    return result
```

```python
def chiKwadratB(ciag, p):

    alpha = 0.05
    degFreedom = 1 # dzielimy na dwa kubełki

    observedZero = 0
    observedOne = 0

    for i in range(len(ciag)):
        if ciag[i] == 0:
            observedZero += 1
        else:
            observedOne += 1

    estimatedZero = (1-p)*len(ciag)
    estimatedOne = p*len(ciag)

    chi0 = pow(observedZero - estimatedZero, 2) / estimatedZero
    chi1 = pow(observedOne - estimatedOne, 2) / estimatedOne

    if chi0 + chi1 < rozkladChi[degFreedom]:
        print("CIAG POSIADA DOBRE LICZBY Z ROZKŁADU BERNOULLIEGO")
        print("########################################################", end='\n\n')
    else:
        print("CIAG POSIADA ZŁE LICZBY Z ROZKŁADU BERNOULLIEGO")
        print("########################################################", end='\n\n')
```

Przeprowadziłem testy chi-kwadrat dla ciągów B i P dla ilości danych odpowiednio: 10 000, 50 000 oraz 110 000.
Dla generatora P przyjąłem parametr *lambda* = 3.
W każdym przypadku test chi-kwadrat wykazał zgodność wygenerowanego ciągu z faktycznym rozkładem.

```
BERNOULLI CHI-KWADRAT TEST 10000
-------------------------------------------------------
CIAG POSIADA DOBRE LICZBY Z ROZKŁADU BERNOULLIEGO
#######################################################

POISSON CHI-KWADRAT TEST 10000
-------------------------------------------------------
CIAG POSIADA DOBRE LICZBY Z ROZKŁADU POISSONA
#######################################################


BERNOULLI CHI-KWADRAT TEST 50000
-------------------------------------------------------
CIAG POSIADA DOBRE LICZBY Z ROZKŁADU BERNOULLIEGO
#######################################################

POISSON CHI-KWADRAT TEST 50000
-------------------------------------------------------
CIAG POSIADA DOBRE LICZBY Z ROZKŁADU POISSONA
#######################################################


BERNOULLI CHI-KWADRAT TEST 110000
-------------------------------------------------------
CIAG POSIADA DOBRE LICZBY Z ROZKŁADU BERNOULLIEGO
#######################################################

POISSON CHI-KWADRAT TEST 110000
-------------------------------------------------------
CIAG POSIADA DOBRE LICZBY Z ROZKŁADU POISSONA
#######################################################
```

Projekt wykonał: Jan Skwarczek

Źródła:

- generatory 16.pdf (agh.edu.pl)
- Chi-squared test - Wikipedia
- Testy-los.pdf (rezolwenta.eu.org)
- Rozkład dwumianowy – Wikipedia, wolna encyklopedia
- Rozkład Poissona – Wikipedia, wolna encyklopedia
- Rozkład normalny – Wikipedia, wolna encyklopedia
- Rozkład wykładniczy – Wikipedia, wolna encyklopedia
- Wydział Matematyki. Testy zgodności. Wykład 03 - PDF Free Download (docplayer.pl)