# Zadanie 1

$$P(n,k) = \sum_{i=0}^{k} \binom{n}{i} p^i (1-p)^{n-i}$$

Możemy przekształcić w następujący sposób

$$P(n,k) = \sum_{i=0}^{k} \binom{n}{i} p^i (1-p)^{n-i} = \sum_{i=0}^{k} \binom{n}{i} p^i (1-p)^{-i} (1-p)^{n} = \sum_{i=0}^{k} \binom{n}{i} \left(\frac{p}{1-p}\right)^i (1-p)^n.$$

Ponieważ czynnik $(1-p)^n$ nie zależy od n możemy wyciągnąć go przed sumę otrzymując wyrażenie

$$P(n,k) = (1-p)^n \sum_{i=0}^{k} \binom{n}{i} \left(\frac{p}{1-p}\right)^i.$$

By obliczyć czynnik przed sumą skorzystam z algorytmu *Exponentiation by squaring* o złożoności algorytmicznej $o(\log n)$.

By wyliczyć symbol newtona w kolejnych iteracjach przekształcam go w następujący sposób

$$\binom{n}{i} = \frac{n!}{(n-i)!i!} = n! \frac{i+1}{(i+1)!} \frac{1}{(n-i)(n-i-1)!} = \frac{n!}{(n-(i+1))!(i+1)!} \frac{i+1}{n-i} = \binom{n}{i+1} \frac{i+1}{n-i}.$$

Jeśli porównamy skrajne wyrazy i pomnożymy obustronnie przez $\frac{n-i}{i+1}$ otrzymamy

$$\binom{n}{i+1} = \binom{n}{i} \cdot \frac{n-1}{i+1}.$$

In [1]:

```
def probability(n, k, p):
    # w ciele funkcji umieść swój kod realizujący cel zadania;
    # argument 'n' niech będzie liczbą prób;
    # argument 'k' niech będzie maksymalną liczbą sukcesów;
    # argument 'p' niech będzie prawdpodobieństwem sukcesu w pojedynczej próbie;
    # w zmiennej 'prob' zwróć oczekiwane prawdopodobieństwo;
    # w zmiennej 'count_mult' zwróć liczbę mnożeń, jaką wykonał Twój program;
    # jeśli potrzebujesz, możesz dopisać również inne funkcje (pomocnicze),
    # jednak główny cel zadania musi być realizowany w tej funkcji;

    prob=1 #(1-p)^n
    count_mult=0 # liczba mnożeń
    rev=1-p # zdażenie przeciwne

    for i in bin(n)[:1:-1]:
        if i=="1":
            prob*=rev
            count_mult+=1
        rev**=2
        count_mult+=1

    prob_i=1 # prawdopodobieństwo w i-tym kroku (1 dla i==0)
    prob_frac=p/(1-p)
    count_mult+=1
    prob_sum=0 # wynik sumy
    newton=1 # n po 0
    for i in range(k+1):
        prob_sum += prob_i
        prob_i = prob_i*prob_frac*((n-i)/(i+1))
        count_mult+=3

    prob*=prob_sum


    return (prob, count_mult)
```

Łatwo można zauważyć, że funkcja wykonuje $3(k+1) + \log(n)$ mnożeń (k+1 bo wykonujemy dla i==k oraz dla i==0).

By zmniejszyć liczbę mnożeń możemy obliczać symbol newtona rekurencyjnie

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$

co daje nam $k$ mnożeń mniej, czyli $2(k+1) + log(n)$.

In [2]:

```
def probability_2k(n, k, p):
    # w ciele funkcji umieść swój kod realizujący cel zadania;
    # argument 'n' niech będzie liczbą prób;
    # argument 'k' niech będzie maksymalną liczbą sukcesów;
    # argument 'p' niech będzie prawdpodobieństwem sukcesu w pojedynczej próbie;
    # w zmiennej 'prob' zwróć oczekiwane prawdopodobieństwo;
    # w zmiennej 'count_mult' zwróć liczbę mnożeń, jaką wykonał Twój program;
    # jeśli potrzebujesz, możesz dopisać również inne funkcje (pomocnicze),
    # jednak główny cel zadania musi być realizowany w tej funkcji;

    prob=1
    count_mult=0
    rev=1-p

    for i in bin(n)[:1:-1]:
        if i=="1":
            prob*=rev
            count_mult+=1
#         print(i, prob)
        rev**=2
        count_mult+=1

    prob_frac_pow=p/(1-p)
    count_mult+=1
    prob_frac=1
    prob_sum=0
    for i in range(k+1):
        prob_sum+=newton(i,n)*prob_frac
        prob_frac*=prob_frac_pow
        count_mult+=2

    prob*=prob_sum




    return (prob, count_mult)

def newton(k,n):
    if k==0 or k==n:
        return 1
    return newton(k-1,n-1) + newton(k, n-1)
```

Ponieważ celem zadania było napisać program z jak najmniejszą liczbą mnożenie (nie zależnie od czasu wykonania) możemy drastycznie zmniejszyć ich ilość poprzez zamiane mnożenia na dodawanie (zamiana na postać binarną). Ponieważ mnożenie liczb nie całkowitych w postaci binarnej jest skomplikowane zwiększyłem każdą liczbę o $10^{\mathrm{acc}}$, gdzie acc oznacza dokładność z jaką chcemy wyliczyć prawdopodobieństwo. Uzyskałem w ten sposób jedynie $k+7$ mnożeń.

In [3]:

```
def probability_k(n, k, p, acc=16):
    # acc - dokładność do której liczby po przecinku (n jest odwrotnie proporcjonalne do acc)

    # w ciele funkcji umieść swój kod realizujący cel zadania;
    # argument 'n' niech będzie liczbą prób;
    # argument 'k' niech będzie maksymalną liczbą sukcesów;
    # argument 'p' niech będzie prawdpodobieństwem sukcesu w pojedynczej próbie;
    # w zmiennej 'prob' zwróć oczekiwane prawdopodobieństwo;
    # w zmiennej 'count_mult' zwróć liczbę mnożeń, jaką wykonał Twój program;
    # jeśli potrzebujesz, możesz dopisać również inne funkcje (pomocnicze),
    # jednak główny cel zadania musi być realizowany w tej funkcji;

    if not 0<=p<=1:
        return 0, 0

    prob=1
    count_mult=0
    prob_frac_pow=int((p/(1-p))*int(float("1e"+str(acc))))
    rev=int((1-p)*int(float("1e"+str(acc))))
    p = int(p*int(float("1e"+str(acc))))
    count_mult+=4

    for i in bin(n)[:1:-1]:
        if i=="1":
            prob=bin_mult(prob,rev)
        rev=bin_mult(rev,rev)

    prob = prob/float("1e"+str(bin_mult(acc,n)))
    count_mult+=1


    prob_frac=1
    prob_sum=0
    for i in range(k+1):
        prob_sum+=bin_mult(newton(i,n),prob_frac)/float("1e"+str(bin_mult(acc,i)))
        count_mult+=1
        prob_frac=bin_mult(prob_frac,prob_frac_pow)

    prob*=prob_sum
    count_mult+=1




    return (prob, count_mult)

def newton(k,n):
    if k==0 or k==n:
        return 1
    return newton(k-1,n-1) + newton(k, n-1)

def bin_mult(first_number,secound_number):
    secound=bin(secound_number)
    mult=0
    res=bin(0)

    for i in bin(first_number)[:1:-1]:
        if i=="1":
            res=bin(int(res,2)+int(secound,2))
        secound+="0"
    return int(res,2)
```

W podobny sposób możemy się pozbyć mnożeń z `for` uzyskując $6$ mnożeń nie zależnie od współczynników $n, k, p$.

Tutaj jedynie $6$ mnożeń niezależnie od $k$ oraz $n$

In [4]:

```
def probability_6_mult(n, k, p, acc=16):
    # acc - dokładność do której liczby po przecinku (n jest odwrotnie proporcjonalne do acc)

    # w ciele funkcji umieść swój kod realizujący cel zadania;
    # argument 'n' niech będzie liczbą prób;
    # argument 'k' niech będzie maksymalną liczbą sukcesów;
    # argument 'p' niech będzie prawdpodobieństwem sukcesu w pojedynczej próbie;
    # w zmiennej 'prob' zwróć oczekiwane prawdopodobieństwo;
    # w zmiennej 'count_mult' zwróć liczbę mnożeń, jaką wykonał Twój program;
    # jeśli potrzebujesz, możesz dopisać również inne funkcje (pomocnicze),
    # jednak główny cel zadania musi być realizowany w tej funkcji;

    if not 0<=p<=1:
        return 0, 0

    prob=1
    count_mult=0
    prob_frac_pow=int((p/(1-p))*int(float("1e"+str(acc))))
    rev=int((1-p)*int(float("1e"+str(acc))))
    p = int(p*int(float("1e"+str(acc))))
    count_mult+=4

    for i in bin(n)[:1:-1]:
        if i=="1":
            prob=bin_mult(prob,rev)
        rev=bin_mult(rev,rev)

    prob = prob/float("1e"+str(bin_mult(acc,n)))
    count_mult+=1


    prob_frac=1
    prob_sum=0
    for i in range(k+1):
        prob_sum+=bin_mult(bin_mult(newton(i,n),prob_frac),int(float("1e"+str(bin_mult(acc,k-i)))))
        #count_mult+=1
        prob_frac=bin_mult(prob_frac,prob_frac_pow)

    prob*=prob_sum/float("1e"+str(bin_mult(acc,k)))
    count_mult+=1
```

```
    return (prob, count_mult)

def newton(k,n):
    if k==0 or k==n:
        return 1
    return newton(k-1,n-1) + newton(k, n-1)

def bin_mult(first_number,secound_number):
    secound=bin(secound_number)
    mult=0
    res=bin(0)

    for i in bin(first_number)[:1:-1]:
        if i=="1":
            res=bin(int(res,2)+int(secound,2))
        secound+="0"
    return int(res,2)
```

Dla $k = 4$

In [5]:
```
probability(12,4,1/3)
```

Out[5]:
```
(0.6315207144349049, 22)
```

In [6]:
```
probability_2k(12,4,1/3)
```

Out[6]:
```
(0.6315207144349049, 17)
```

In [7]:
```
probability_k(12,4,1/3)
```

Out[7]:
```
(0.6315207144349043, 11)
```

In [8]:
```
probability_6_mult(12,4,1/3)
```

Out[8]:
```
(0.6315207144349043, 6)
```

Dla $k = 15$

In [9]:
```
probability(18,15,1/3)
```

Out[9]:
```
(0.9999983248175608, 56)
```

In [10]:
```
probability_2k(18,15,1/3)
```

Out[10]:
```
(0.9999983248175612, 40)
```

In [11]:
```
probability_k(18,15,1/3)
```

Out[11]:
```
(0.99999832481756, 22)
```

In [12]:
```
probability_6_mult(18,15,1/3)
```

Out[12]:
```
(0.9999983248175599, 6)
```

Każdy kolejny program wykonuje dużo mniej mnożeń, ale czas wykonania jest znacznie dłuższy od poprzednika. W dodatku dwa ostatnie pragramy mają duże ograniczenia, przy dużych wartościach n oraz acc nie jest możliwe wykonanie ich (przykładowo dla $n \geq 20 \text{ oraz } acc \geq 16$

# Zadanie 2

W zadaniu drugim korzystamy z wyliczania wartości wielomianu przy pomocy schematu Hornera

In [13]:
```
def ordinary_polynomial_value_calc(coeff, arg):
    # w ciele tej funkcji zawrzyj kod wyliczający wartość wielomianu w tradycyjny sposób;
    # argument 'coeff' niech będzie listą współczynników wielomianu w kolejności od stopnia zerowego (wyrazu w
    # argument 'arg' niech będzie punktem, w którym chcemy policzyć wartość wielomianu;
    # w zmiennej 'count_mult' zwróć liczbę mnożeń, jakie zostału wykonane do uzyskania tego wyniku;
    # w zmiennej 'count_add' zwróć liczbę dodawań, jakie zostału wykonane do uzyskania tego wyniku;

    value=0
    count_add=0
    count_mult=0

    for i in range(len(coeff)):
        value+=arg**i*coeff[i]
        count_mult+=i+1 # uznałem, że x**0 to nie jest mnożenie
        count_add+=1

    return value, count_mult, count_add

def smart_polynomial_value_calc(coeff, arg):
    # w ciele tej funkcji zawrzyj kod wyliczający wartość wielomianu w sposób maksymalnie ograniczający liczbę
    # argument 'coeff' niech będzie listą współczynników wielomianu w kolejności od stopnia zerowego (wyrazu w
    # argument 'arg' niech będzie punktem, w którym chcemy policzyć wartość wielomianu;
    # w zmiennej 'count_mult' zwróć liczbę mnożeń, jakie zostału wykonane do uzyskania tego wyniku;
    # w zmiennej 'count_add' zwróć liczbę dodawań, jakie zostału wykonane do uzyskania tego wyniku;

    value=0
    count_add=0
    count_mult=0
    value=coeff[-1]

    for i in coeff[-2::-1]:
        value=i+value*arg
        count_add+=1
        count_mult+=1


    return value, count_mult, count_add


# jeśli potrzebujesz, możesz dopisać również inne funkcje (pomocnicze),
# jednak główne cele zadania muszą być realizowane w powyższych dwóch funkcjach;
```

# Zadanie 3

In [14]:
```
def counting_chars_without_ifs(filename):
    file_ref = open(filename, 'r')
    text = file_ref.read().lower()
    # uzupełnij ciało tej funkcji kodem realizującym cel zadania;
    # w zmiennej 'char_count' zwróć słownik zawierający wszystkie znaki tekstu
    # jako klucze i ich liczebnoć jako wartości np. {'a': 6, 'b': 2 ...};
    # jeli potrzebujesz, możesz dopisać również inne funkcje (pomocnicze),
    # jednak główny cel zadania musi być realizowany w tej funkcji;

    char_count=dict()

    for i in text:
        while i not in char_count:
            char_count[i]=0
            break
        char_count[i]+=1
    for i in (' ','\n'):
        try:
            char_count.pop(i)
        except:
            pass

    return char_count
```

In [15]:
```
counting_chars_without_ifs(r"D:\GitHub\AiSD\List 3\Nowy folder\L3_ZAD3_sample_text.txt")
```

Out[15]:
```
{'h': 83,
 'a': 74,
 'p': 12,
 'y': 23,
 'f': 26,
 'm': 21,
 'i': 62,
 'l': 34,
 'e': 115,
 's': 51,
 'r': 48,
 'k': 9,
 ';': 4,
 'v': 12,
 'u': 25,
 'n': 79,
 't': 74,
 'o': 72,
 'w': 21,
 '.': 7,
 'g': 16,
 'c': 16,
 'b': 14,
 "'": 1,
 'd': 39,
 ',': 10,
 'q': 1,
 '-': 2,
 'j': 1}
```

In [16]:
```
def counting_chars_without_ifs_with_pandas(filename):
    file_ref = open(filename, 'r')
    text = file_ref.read().lower()
    import pandas as pd
    di = dict(pd.Series(list(text)).value_counts())
    for i in (' ','\n'):
        try:
            di.pop(i)
        except:
            pass
    return di
```

In [17]:
```
counting_chars_without_ifs_with_pandas(r"D:\GitHub\AiSD\List 3\Nowy folder\L3_ZAD3_sample_text.txt")
```

Out[17]:
```
{'e': 115,
 'h': 83,
 'n': 79,
 'a': 74,
 't': 74,
 'o': 72,
 'i': 62,
 's': 51,
 'r': 48,
 'd': 39,
 'l': 34,
 'f': 26,
 'u': 25,
 'y': 23,
 'm': 21,
 'w': 21,
 'g': 16,
 'c': 16,
 'b': 14,
 'v': 12,
 'p': 12,
 ',': 10,
 'k': 9,
 '.': 7,
 ';': 4,
 '-': 2,
 "'": 1,
 'q': 1,
 'j': 1}
```

In [18]:
```
def counting_chars_without_ifs_2(filename):
    file_ref = open(filename, 'r')
    text = file_ref.read().lower()
    # uzupełnij ciało tej funkcji kodem realizującym cel zadania;
    # w zmiennej 'char_count' zwróć słownik zawierający wszystkie znaki tekstu
    # jako klucze i ich liczebnoć jako wartości np. {'a': 6, 'b': 2 ...};
    # jeli potrzebujesz, możesz dopisać również inne funkcje (pomocnicze),
    # jednak główny cel zadania musi być realizowany w tej funkcji;

    char_count=dict()

    keys=set(list(text))
    for i in keys:
        char_count[i]=sum(char == i for char in text)

    for i in (' ','\n'):
        try:
            char_count.pop(i)
        except:
            pass

    return char_count
```

In [19]:
```
counting_chars_without_ifs_2(r"D:\GitHub\AiSD\List 3\Nowy folder\L3_ZAD3_sample_text.txt")
```

Out[19]:
```
{',': 10,
 ';': 4,
 'h': 83,
 'u': 25,
 'j': 1,
 'd': 39,
 't': 74,
 "'": 1,
 'f': 26,
 'q': 1,
 'r': 48,
 's': 51,
 'o': 72,
 'w': 21,
 'm': 21,
 'a': 74,
 'e': 115,
 'g': 16,
 'p': 12,
 '-': 2,
 'k': 9,
 'l': 34,
 '.': 7,
 'v': 12,
 'y': 23,
 'i': 62,
 'n': 79,
 'b': 14,
 'c': 16}
```