SPRAWOZDANIE - LISTA 3

Małgorzata Kowalczyk

Kamil Kowalski

16.11.2021

Zadanie 1

W pierwszym zadaniu, naszym celem było napisanie programu obliczającego prawdopodobieństwo co najwyżej k sukcesów, przy prawdopodobieństwie pojedyńczego sukcesu p. Wykonywana liczba mnożeń musiała być mniejsza niż $a \cdot k + b \cdot \log n + c$.

Przekształciliśmy dany wzór do postaci rekurencyjnej

$$P(n,k) = \sum_{i=0}^{k} \binom{n}{i} p^i (1-p)^{n-i}$$
$$a_0 = (1-p)^n$$
$$a_1 = np(1-p)^{n-1}$$
$$a_2 = \frac{n(n-1)}{2} p^2 (1-p)^{n-2}$$
$$a_i = \frac{n!}{i!(n-i)!} p^i (1-p)^{n-i}$$
$$a_{i+1} = \frac{n!}{(i+1)!(n-i-1)!} p^{i+1} (1-p)^{n-i-1}$$
$$\frac{a_{i+1}}{a_i} = \frac{p(n-i)}{(1-p)(i+1)}$$
$$a_{i+1} = \frac{p(n-i)}{(1-p)(i+1)} a_i.$$

Dzięki postaci rekurencyjnej, wyliczamy wartość a_0, korzystając z algorytmu quick_power(), a następnie każdą następną wartość $a_1, a_2, \ldots a_n$ obliczamy poprzez przemnożenie poprzedniego przez $\frac{a_{i+1}}{a_i}$.
Poszczególne wartości $a_1, a_2, \ldots a_n$ dodajemy do listy single_values i aby dostać końcowe prawdopobodobieństwo, sumujemy jej wszystkie elementy.

Dodatkowo umieściliśmy funkcję, która sprawdza, czy argumenty podane podczas wywołania funkcji są prawidłowe.
Wartości n oraz k muszą być dodatnie oraz $n > k$, dodatkowo wartość p musi należeć do przedziału $[0; 1]$.

Liczba mnożeń

Funkcja quick_power(p, n, count_mult = 0)

Algorytm quick_power() wykonuje od $\log_2 n$ (gdy wykładnik potęgi jest potęgą liczby 2) do $2 \log_2 n$ mnożeń (gdy po rozłożeniu wykładnika żaden z czynników nie będzie podzielny przez

2).
Gdy zliczamy dokładną liczbę wykonanych mnożeń w algorytmie, otrzymamy większą liczbę, ponieważ musimy uwzględnić wykonane dzielenia do obliczenia nowych wykładników potęg - takich obliczeń jest $\log_2 n + 2$.
Zatem łączna liczba wykonanych mnożeń to $2 \cdot \log_2 n + 2$.

Funkcja probability(n, k, p)

W funkcji probability() wykonujemy mnożenia by obliczyć wartość kolejnego składnika sumy. W każdej iteracji w pętli for wykonujemy 4 mnożenia, więc łącznie jest ich $4 \cdot k$.

Podsumowanie

Łączna liczba wykonywanych mnożeń przez nasz algorytm to $4 \cdot k + 2 \cdot \log n + 2$.

Przykładowe wywołania

In [2]:
```
probability(40, 6, 0.4)
```

Out[2]:
```
(0.0005948300671043771, 36)
```

In [3]:
```
probability(10, 5, 0.5)
```

Out[3]:
```
(0.623046875, 28)
```

In [4]:
```
probability(-40, 6, 0.4)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-d0e3fa965896> in <module>
----> 1 probability(-40, 6, 0.4)

<ipython-input-1-9968ebdbc936> in probability(n, k, p)
     19
     20 def probability(n, k, p):
---> 21     check_values(n, k, p)
     22     single_values = []
     23     a_0 = quick_power(1 - p, n)

<ipython-input-1-9968ebdbc936> in check_values(n, k, p)
      1 def check_values(n, k, p):
      2     if n < 0 or k < 0 or k > n:
----> 3         raise ValueError("n and k should be positive!")
      4     elif p < 0 or p > 1:
      5         raise Exception("Probability should be greater than 0 and less than
 1!")

ValueError: n and k should be positive!
```

Zadanie 2

Naszym zadaniem było napisanie algorytmu, dzięki któremu obliczymy wartość wielomianu stopnia n, w punkcie arg, o współczynnikach zawartych w liście coeff. Stworzyliśmy zatem dwie funkcje. W pierwszej wersji stosujemy podstawowy sposób wyliczania wartości wielomianu.

Aby lepiej zauważyć liczbę mnożeń i dodawań rozpiszmy poniższy przykład. Nasza funkcja wygląda następująco

$$f(x) = 3x^2 + 2x + 5$$

Obliczymy teraz wartość wielomianu w punkcie $x = 2$.

Wielomian rozpisujemy jako

$$f(x) = \sum_{k=0}^{n} a_k \cdot x^k,$$

gdzie a_k są liczbami rzeczywistymi reprezentującymi współczynniki wielomianu, a n to najwyższy wykładnik zmiennej.

$$f(2) = 5 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2 \cdot 2$$

$$f(2) = 5 \cdot 1 + 2 \cdot 2 + 3 \cdot 2 \cdot 2$$

$$f(2) = 5 + 2 \cdot 2 + 3 \cdot 2 \cdot 2$$

Zatem liczba mnożeń to 3, a liczba dodawań wynosi 2.

In [3]:
```
ordinary_polynomial_value_calc([5,2,3],2)
```

Out[3]:
```
(21, 3, 2)
```

Następnie zastosowaliśmy schemat Hornera, by ograniczyć liczbę mnożeń w naszym algortymie.

$$f(x_0) = a_o + a_1 x_0 + a_2 (x_0)^2 + \ldots + a_n (x_0)^n$$

A to może być zapisane w postaci

$$f(x_0) = a_0 + x_0(a_1 + x_0(a_2 + x_0(a_3 + \ldots + x_0(a_{(n-1)} + x_0 a_n)\ldots)))$$

Zatem naszą przykładową funkcję zapiszemy w postaci

$$f(2) = 5 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2$$

$$f(2) = 5 + 2 \cdot (2 + 2 \cdot 3)$$

Ograniczyliśmy w ten sposób liczbę mnożeń. Teraz wynosi ona 2, a liczba dodawań nie zmienia się. W naszym algorytmie pomijamy początkowe zliczanie sumy 0 z liczbą.

In [7]:
```
smart_polynomial_value_calc([5,2,3],2)
```

Out[7]:
```
(21, 2, 2)
```

Porównując ze sobą te dwie funkcje, zdecydowanie większą różnicę w ilości wykonywanych mnożeń, możemy zaobserwować dla wielomianów wyższych stopni.

In [9]:
```
ordinary_polynomial_value_calc([1,2,8,3,2,0,5],2)
```

Out[9]:
```
(413, 21, 6)
```

$$f(2) = 1 + 2 \cdot 2^1 + 8 \cdot 2^2 + 3 \cdot 2^3 + 2 \cdot 2^4 + 0 \cdot 2^5 + 5 \cdot 2^6 = 413$$

In [8]:
```
smart_polynomial_value_calc([1,2,8,3,2,0,5],2)
```

Out[8]:
```
(413, 6, 6)
```

$$f(2) = 1 + 2 \cdot (2 + 2 \cdot (8 + 2 \cdot (3 + 2 \cdot (2 + 2 \cdot (0 + 2 \cdot 5))))) = 413$$

In [10]:
```
ordinary_polynomial_value_calc([3,5,2,0,9,5,1,4,2,4],3)
```

Out[10]:
```
(103311, 45, 9)
```

In [11]:
```
smart_polynomial_value_calc([3,5,2,0,9,5,1,4,2,4],3)
```

Out[11]:
```
(103311, 9, 9)
```

Dodatkowo, gdy podamy złe dane, program zwróci wyjątek.

In [22]:
```
ordinary_polynomial_value_calc(5,'a')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-0c468ea21a75> in <module>
----> 1 ordinary_polynomial_value_calc(5,'a')

<ipython-input-13-4a4ddb5f278e> in ordinary_polynomial_value_calc(coeff, arg)
      1 def ordinary_polynomial_value_calc(coeff, arg):
      2     if not isinstance(coeff, list) or not isinstance(arg, (int, float)):
----> 3         raise TypeError("Wrong data given.")
      4 
      5     value = coeff[0]

TypeError: Wrong data given.
```

In [23]:
```
ordinary_polynomial_value_calc([5,4,3],'a')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-800c827b48aa> in <module>
----> 1 ordinary_polynomial_value_calc([5,4,3],'a')

<ipython-input-13-4a4ddb5f278e> in ordinary_polynomial_value_calc(coeff, arg)
      1 def ordinary_polynomial_value_calc(coeff, arg):
      2     if not isinstance(coeff, list) or not isinstance(arg, (int, float)):
----> 3         raise TypeError("Wrong data given.")
      4 
      5     value = coeff[0]

TypeError: Wrong data given.
```

Zadanie 3

W tym zadaniu musieliśmy napisać program zliczający ilość wystąpień każdego znaku w pliku tekstowym. Dodatkowo należało wykonać to, bez wyrażenia warunkowego if.
W algorytmie zaczynamy od usunięcia spacji w tekście oraz zamianie wszystkich liter na małe. Następnie tworzymy słownik char_count, w którym kluczem jest każda z występujących w tekście liter. Wszystkie wartości ustalamy na 0. Ostatni krok to zliczenie wystąpień każdej z liter. Robimy to w pętli for.
Poniżej przedstawiamy przykładowe wywołanie dla pliku z e-portalu.

In [19]:
```
counting_chars_without_ifs("L3_ZAD3_sample_text.txt")
```

Out[19]:
```
{'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}
```

Linki

https://github.com/github-kamilk/AiSD/tree/main/Lista_3