Laboratorium kryptograficzne dla licealistów 6

Projekt „Matematyka dla ciekawych świata”
Łukasz Mazurek

11.05.2017

1 Potęgowanie

W kryptografii często wykonuje się operację „potęgowania modulo”. Np. w algorytmie RSA, aby zaszyfrować wiadomość m kluczem e należy wykonać działanie

$$c = m^e \pmod{n}$$

W praktyce liczby e i n są bardzo duże (np. mają 1000 cyfr), dlatego ważne jest, aby operacja ta była wykonywana szybko. Następująca funkcja oblicza powyższe wyrażenie dla $n = 1\,000\,000\,009$, $m = 2$ i $e = 1\,000\,000$:

```
def potega(m, e, n):
  c = 1
  for i in range(e):
    c = c * m % n
  return c

print(potega(2, 1000000, 1000000009))
```

Przepisz powyższy kod i uruchom w interpreterze `repl.it`. Okaże się, że program rzeczywiście wylicza pewną wartość i po chwili wypisuje ją na ekran.

Czy ten kod działa *szybko*? Po naciśnięciu przycisku *run*, wynik działania pojawia się na ekranie *po chwili*. Co jeśli zwiększymy `e` dziesięciokrotnie? Uruchom program ponownie, zmieniając wartość zmiennej `e` na 10 000 000. Tym razem program powinien wykonywać się już *kilka sekund*. Dla `e` = 100 000 000 czas działania wynosiłby już **kilkadziesiąt sekund**. Tak jak wspominałem, w algorytmie RSA używa się wykładników które mają **setki cyfr**, a nasz program działa wolno już dla wykładnika, który ma raptem **9 cyfr**. Widzimy zatem, że w praktyce trzeba potrafić potęgować szybciej.

1.1 Pomiar czasu

Żeby wygodniej mówiło nam się o szybkości działania programu, przyda nam się narzędzie do pomiaru czasu. Do mierzenia czasu działania programu w Pythonie służy polecenie `timeit`. Przykład użycia:

```
from timeit import timeit

def potega(m, e, n):
  c = 1
  for i in range(e):
    c = c * m % n
  return c

print(timeit('potega(2, 1000000, 1000000009)', globals = globals(),
  number = 1))
```

Aby użyć polecenia `timeit`, należy najpierw zaimportować odpowiedni pakiet, a następnie wywołać polecenie `timeit` z odpowiednimi argumentami. Pierwszym argumentem jest string (otoczony apostrofami) zawierający polecenie do wykonania. Parametr `globals = globals()` jest niezbędny, aby używać zdefiniowanych wcześniej funkcji (w naszym przypadku, funkcji `potega`). Parametr `number` jest przydatny w przypadku bardzo szybkich fragmentów kodu — wówczas czas działania samego fragmentu jest porównywalny z błędem pomiaru wynikającym z czasem samego uruchomienia programu. Dlatego, aby uzyskać lepszą dokładność, możemy np. mierzyć czas tysiąckrotnego powtórzenia wykonania danego fragmentu kodu. My jednak na razie będziemy mierzyć pojedynczy przebieg programu, dlatego ustawimy `number = 1` (gdybyśmy nie napisali tego wprost, Python przyjąłby domyślną wartość `number = 1000000`).

Uruchom powyższy program (dla `e` $= 1\,000\,000$) na `repl.it` i sprawdź, jak szybko działa ten kod. Interpreter powinien wypisać liczbę ok. 0.1, co oznacza, że program działa ok. 0.1 sekundy. Możesz uruchomić program kilkukrotnie — okaże się, że czas działania za każdym razem będzie nieco inny, ale różnice nie będą znaczące.

Ile operacji wykonuje ten kod? Główną częścią programu jest pętla, w której program `e` razy wykonuje działanie mnożenia (`*`) i dzielenia modulo (`%`). Zatem dla `e` $= 1\,000\,000$ program wykonuje ok. 2 miliony operacji arytmetycznych i dzieje się to w czasie ok. 0.1 s. W ten sposób udało nam się oszacować szybkość Pythona! A dokładniej szybkość interpretera Pythona udostępnionego na stronie `repl.it`. Interpreter ten wykonuje ok. **20 mln operacji arytmetycznych na sekundę**.

1.2 Złożoność czasowa

Zadanie 1 *Porównaj czasy działania powyższego kodu dla następujących wartości parametru* `e`*:*
`[1000, 10000, 100000, 1000000, 10000000]`

Okaże się, że każde kolejne wykonanie trwa ok. 10 razy dłużej od poprzedniego. Oznacza to, że czas działania programu jest *wprost proporcjonalny* do `e`. W takim przypadku mówimy, że *złożoność czasowa* tego algorytmu jest *liniowa względem* `e` i oznaczamy ją przez $O(\texttt{e})$. W dalszej części zajęć skonstruujemy szybszy algorytm potęgowania, czyli algorytm o *mniejszej złożoności*.

2 Rekurencja

Funkcja rekurencyjna to funkcja, która wywołuje samą siebie. Najprostszym przykładem funkcji rekurencyjnej jest funkcja obliczającą *silnię*.

Silnią liczby n nazywamy iloczyn wszystkich liczb od 1 do n i oznaczamy $n! = 1{\cdot}2{\cdot}3{\cdot}\ldots{\cdot}n$. Rekurencyjna funkcja obliczająca silnię opiera się na rekurencyjnym wzorze na silnię, czyli wzorze, który do obliczenia silni liczby n korzysta z wartości silni obliczonych dla mniejszych liczb:

$$n! = n \cdot (n-1)!$$

Aby wywołanie funkcje rekurencyjnej miało szansę kiedyś się zakończyć, funkcja musi posiadać *warunek brzegowy*. W przypadku silni, warunkiem tym jest równość

$$0! = 1$$

Powyższe wzory prowadzą nas do następującej funkcji obliczającej silnię liczby n:

```
def silnia(n):
  if n == 0:
    return 1
  else:
    return n * silnia(n - 1)
```

Przepisz powyższy kod do interpretera `repl.it` i oblicz silnię liczby 10.

Zadanie 2 *Napisz funkcję rekurencyjną* `fib(n)`*, która obliczy n-tą liczbę Fibonnacci'ego zgodnie z następującymi wzorami:*

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Ile wynosi f_{10}*?*

2.1 Szybkie potęgowanie

Stosując standardowy algorytm potęgowania, aby np. obliczyć 2^{16} musimy wykonać ok. 16 mnożeń[1]. Da się jednak tę samą liczbę obliczyć szybciej:

1. Wykonajmy **pierwsze** mnożenie: $2 \cdot 2 = 4$. Po wykonaniu tego mnożenia wiemy, że $2^2 = 4$.
2. Wiemy skądinąd, że $2^2 \cdot 2^2 = 2^4$. Zatem wykonując **drugie** mnożenie otrzymujemy: $2^4 = 4 \cdot 4 = 16$.
3. Analogicznie, po **trzecim** mnożeniu uzyskamy $2^8 = 16 \cdot 16 = 256$.
4. Ostatecznie, po **czwartym** mnożeniu obliczymy szukaną wartość: $2^{16} = 256 \cdot 256 = 65536$.

Tym sposobem obliczyliśmy liczbę 2^{16} wykonując 4 zamiast 15 mnożeń. Czy to duża różnica? W tym przypadku niewielka. Ale gdybyśmy podnosili jakąś liczbę do potęgi $e = 2^{10} = 1024$, w pierwszym sposobie musielibyśmy wykonać 1023 operacji mnożenia, a w drugim tylko 10. Dla $e = 2^{20} = 1048576$ pierwszy sposób wymaga 1048575 operacji mnożenia, a drugi jedynie 20. Widzimy, że im większe e, tym różnica pomiędzy algorytmami robi się bardziej znacząca.

Oczywiście przedstawiony powyżej algorytm zadziała jedynie w przypadku, gdy wykładnik jest potęgą dwójki, np. przy obliczaniu 17^{64}, ale już nie przy obliczaniu 17^{59}. Ogólny algorytm szybkiego potęgowania działający dla dowolnej potęgi jest implementacją następujących wzorów rekurencyjnych, na potęgowanie:

$$m^0 = 1$$
$$m^e = m^{e/2} \cdot m^{e/2}$$
$$m^e = m \cdot m^{e-1}$$

Algorytm ten zaimplementowany w Pythonie wygląda tak:

```
def potega2(m, e, n):
  if e == 0:
    return 1
  elif e % 2 == 0:
    pom = potega2(m, e / 2, n)
    return (pom * pom) % n
  else:
    return (m * potega2(m, e - 1, n)) % n
```

Jest to algorytm rekurencyjny i jego działanie można odczytać następująco:

- Warunek brzegowy: jeśli doszedłeś do $e = 0$, to zwróć 1 (bo $m^0 = 1$).
- Jeśli e jest parzyste, to najpierw rekurencyjnie oblicz $m^{e/2}$, a następnie zwróć jako wynik $m^e = m^{e/2} \cdot m^{e/2}$.
- Jeśli e jest nieparzyste, to najpierw rekurencyjnie oblicz m^{e-1}, a następnie zwróć jako wynik $m^e = m \cdot m^{e-1}$.

[1] Dokładniej, wystarczy 15 mnożeń: $2^{16} = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$.

Dobrym zobrazowaniem działania algorytmu jest rozwinięcie jego kolejnych rekurencyjnych wywołań na przykładzie obliczania liczby 17^{59}:

$$\begin{aligned}
17^{59} &= 17 \cdot 17^{58} = 17 \cdot (17^{29} \cdot 17^{29}) \\
17^{29} &= 17 \cdot 17^{28} = 17 \cdot (17^{14} \cdot 17^{14}) \\
17^{14} &= 17^{7} \cdot 17^{7} \\
17^{7} &= 17 \cdot 17^{6} = 17 \cdot (17^{3} \cdot 17^{3}) \\
17^{3} &= 17 \cdot 17^{2} = 17 \cdot (17^{1} \cdot 17^{1}) \\
17^{1} &= 17 \cdot 17^{0} = 17 \cdot 1
\end{aligned}$$

Jeżeli powyższe mnożenia będziemy wykonywali „od końca", będziemy potrzebowali łącznie 10 operacji mnożenia, aby uzyskać wartość 17^{59}:

$$\begin{aligned}
17^{1} &= 17 \cdot 1 \\
17^{2} &= 17^{1} \cdot 17^{1} \\
17^{3} &= 17 \cdot 17^{2} \\
17^{6} &= 17^{3} \cdot 17^{3} \\
17^{7} &= 17 \cdot 17^{6} \\
17^{14} &= 17^{7} \cdot 17^{7} \\
17^{28} &= 17^{14} \cdot 17^{14} \\
17^{29} &= 17 \cdot 17^{28} \\
17^{58} &= 17^{29} \cdot 17^{29} \\
17^{59} &= 17 \cdot 17^{58}
\end{aligned}$$

Zadanie 3 *Zmierz czas działania powyższej procedury dla* `m = 2`, `n = 1000000009` *i następujących wartości* `e`*:*

```
[1000, 10000, 100000, 1000000, 10000000]
```

Porównaj czasy działania z czasami działania funkcji `potega`*, którą zaimplementowaliśmy na początku zajęć.*

2.2 Złożoność algorytmu

Jeżeli poprawnie rozwiązałeś poprzednie zadanie, powinieneś otrzymać czasy rzędu 10^{-5} sekundy[2]. To bardzo dobry wynik! Algorytm ten działa ponad tysiąckrotnie lepiej niż algorytm, który stworzyliśmy na początku zajęć.

Spróbujmy teraz dokładniej oszacować złożoność programu. Jeżeli przyjrzymy się otrzymanym wynikom czasowym, zauważymy, że wyniki dla poszczególnych wartości `e` są zbliżone. Może się nawet zdarzyć, że czas działania dla większego `e` będzie mniejszy niż analogiczny czas dla mniejszego `e`. Dzieje się tak, ponieważ obliczone czasy działania są porównywalne z błędem pomiarowym. Aby otrzymać dokładniejsze wyniki, będziemy mierzyć czasy tysiąckrotnego wykonania poszczególnych funkcji. W tym celu w poleceniu `timeit` zmień wartość parametru `number` na 1000 i jeszcze raz oblicz wszystkie czasy. Po tej zmianie powinieneś otrzymać czasy ok. 0.01 s.

Funkcja `potega`, którą napisaliśmy na początku zajęć, ma *złożoność liniową względem* `e`, tzn. jej czas wywołania jest wprost proporcjonalny do `e`. Czy tak samo jest w przypadku funkcji `potega2`? Nie! Największa wartość `e` jest 10 000 razy większa od najmniejszej, a czas działania dla największej wartości `e` jest tylko kilkukrotnie większy od czasu działania dla najmniejszej wartości `e`. Oznacza to, że algorytm `potega2` ma **złożoność mniejszą niż liniowa** względem `e`.

Jaką dokładnie złożoność ma nasz algorytm? Aby się o tym przekonać zmierz czasy działania funkcji dla następujących wartości `e`: (`**` w Pythonie oznacza podnoszenie do potęgi)

[2]Zapis `1.26e-05` oznacza $1.26 \cdot 10^{-5}$

```
[10**10, 10**20, 10**40, 10**80, 10**160]
```
Tym razem powinniśmy otrzymać ciąg wartości, z których każda kolejna jest ok. dwukrotnie większa od poprzedniej. Oznacza to, że czas działania programu jest proporcjonalny do wykładnika, do którego podnosimy liczbę 10, aby uzyskać liczbę `e`. Innymi słowy, **czas działania programu jest proporcjonalny do logarytmu z** `e`. W takim przypadku mówimy, że złożoność algorytmu wynosi $O(\log \texttt{e})$.

2.3 Dygresja o logarytmach

Logarytm jest funkcją odwrotną do potęgowania. Jeśli $a^b = c$, to $b = \log_a c$. Np. $\log_{10} 100 = 2$, $\log_2 32 = 5$, $\log_{10}(10^{160}) = 160$. Dlatego, żeby mówić o logarytmie, trzeba podać jego podstawę (liczbę w indeksie dolnym).

Powszechnie używana jest notacja $\log x$ (bez podstawy), jednak nie ma zgodności, co taki napis oznacza. Chemicy lubią używać takiej notacji do logarytmów o podstawie 10, fizycy do logarytmów o podstawie $e = 2,718281\ldots$ W informatyce najczęściej używamy logarytmów o podstawie 2. Na szczęście w przypadku badania złożoności i notacji $O(\ldots)$ nie ma znaczenia, którego logarytmu użyjemy, ponieważ każdy logarytm jest *proporcjonalny* do logarytmu o podstawie 2. Na przykład:

$$\log_{10} 1000 = \frac{\log_2 1000}{\log_2 10} \approx 0.301 \cdot \log_2 1000$$

Skoro notacja $O(x)$ oznacza, że czas działania jest proporcjonalny do x, to algorytm o złożoności $O(\log n)$ będzie miał czas działania proporcjonalny do logarytmu z n o dowolnej podstawie. To usprawiedliwia nas do pisania $O(\log \texttt{e})$ zamiast np. $O(\log_{10} \texttt{e})$.

2.4 Dygresja o potęgowaniu

Potęgowanie modulo jest na tyle przydatną funkcją, że zostało w efektywny sposób zaimplementowane w Pythonie. Oprócz znanej Wam już notacji `a**b`, do obliczania a^b w Pythonie istnieje również funkcja **pow**`(a,b)`. Funkcja ta może przyjąć trzeci, opcjonalny parametr: **pow**`(a, b, n)` i wówczas oblicza $a^b \pmod n$.

3 Największy wspólny dzielnik

Jak wiecie z wykładu, do obliczania największego wspólnego dzielnika dwóch liczb służy algorytm Euklidesa. Algorytm ten również opiera się na rekurencyjnej zależności:

$$NWD(a, b) = NWD(b, a \bmod b)$$

Warunkiem brzegowym jest w tym przypadku warunek

$$NWD(a, 0) = a$$

Zadanie 4 *Napisz funkcję* ***`NWD(a, b)`****, która obliczy największy wspólny dzielnik liczb* `a` *i* `b`*. Oblicz największy wspólny dzielnik liczb* 675675 *i* 272272*.*

4 Zadania dodatkowe

Zadanie 5 *Spróbuj oszacować złożoność algorytmu Euklidesa. Wiedząc, że interpreter Pythona ze strony* ***`repl.it`*** *wykonuje ok. 20 mln operacji na sekundy, jaki jest rząd wielkości liczb* `a` *i* `b`*, dla których wywołanie funkcji* ***`NWD(a, b)`*** *się wykona w czasie nieprzekraczającym kilku sekund?*

Zadanie 6 *Ciągi (a_n), (b_n) i (c_n) zdefiniowane są następującymi wzorami:*

$$\begin{aligned} a_1 &= 1 \\ b_1 &= 1 \\ a_n &= 2b_{n-1} \\ b_n &= a_{n-1} + 1 \\ c_n &= a_n + b_n \end{aligned}$$

Oblicz wartość c_{17}.

Wskazówka: do obliczania wyrazów każdego z ciągów stwórz oddzielną funkcję rekurencyjną.

Zadanie 7 *Napisz funkcję* `euklides(p, q)`*, która obliczy liczby całkowite a i b, takie że zachodzi równość*

$$a \cdot p + b \cdot q = 1$$

Skorzystaj z rozszerzonego algorytmu Euklidesa prezentowanego na wykładzie tydzień temu.

Zadanie 8 *Napisz funkcję* `odwroc(x, n)`*, która obliczy odwrotność liczby x modulo n, czyli taką liczbę y, że zachodzi kongruencja:*

$$x \cdot y \equiv 1 \pmod{n}$$

Wykorzystaj funkcję `euklides` *z poprzedniego zadania i metodę prezentowaną na wykładzie tydzień temu.*

5 Praca domowa nr 6

Rozwiązania zadań należy przesłać do czwartku 18 maja do godz. 16^{59} na adres `licealisci.pracownia@icm.edu.pl` wpisując jako temat wiadomości `Lx PD6`, gdzie `x` to numer grupy, np. `L3 PD6` dla grupy L3, itd.

Zadanie domowe 1 (1 pkt) *Napisz funkcje* `szyfruj_rsa(n, e, m)`*, która szyfruje liczbę* `m` *algorytmem RSA przy użyciu klucza publicznego* `e` *i danego* `n`*. Zaszyfruj liczbę 13 przy użyciu klucza* `e` $= 541$ *i* `n` $= 2501$*.*

Zadanie domowe 2 (2 pkt) *Napisz funkcję* `rozloz(n)`*, która rozkłada liczbę* `n` *na czynniki, tj. wypisuje liczbę* `n` *przedstawioną jako iloczyn dwóch liczb większych od 1 lub wypisuje stosowny komunikat, jeśli liczba jest pierwsza. Przykład wywołania funkcji:*

```
rozloz(245)
rozloz(541)
```

```
245 = 5 * 49
Liczba 541 jest pierwsza.
```

Zadanie domowe 3 (3 pkt) *Rozważmy algorytm RSA z $n = 1\,000\,009$ i kluczem publicznym $e = 37$. Korzystając z tych informacji, znajdź wartość klucza prywatnego d.*

Wskazówka: Przedstaw liczbę n jako iloczyn dwóch liczb pierwszych $p \cdot q$ i skorzystaj z faktu, że w algorytmie RSA liczby e i d spełniają zależność:

$$e \cdot d \equiv 1 \pmod{(p-1) \cdot (q-1)}$$

Różnych możliwych wartości d jest stosunkowo niewiele (możemy ograniczyć się do wartości mniejszych od n). Możemy więc sprawdzić wszystkie możliwe wartości d i wybrać tę, która spełnia powyższą kongruencję.

Sprawdź swoje rozwiązanie szyfrując liczbę 13 kluczem $e = 37$, a następnie rozszyfrowując obliczonym kluczem d.