Ćwiczenia praktyczne

Andrzej Kierzkowski, Marek Gawryszewski

Python

ĆWICZENIA PRAKTYCZNE

Poznaj programowanie z bliska! Naucz się Pythona!

- Dowiedz się, jak czytać i implementować algorytmy
- Naucz się analizować i rozwiązywać problemy
- Poznaj podstawy Pythona na praktycznych przykładach

Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Grafika na okładce została wykorzystana za zgodą Shutterstock.com

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/cwpyth
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-3513-4

Spis treści

1

Ćwiczenia z myślenia algorytmicznego

Pewnie oczekujesz wstępu do Pythona — wyjaśnienia, czym jest, programu-ćwiczenia pozwalającego wypisać coś na ekranie, opisu budowy programów albo informacji o obsłudze samego programu. Jednak w najbliższym czasie nie będziemy się zajmować Pythonem. Zajmiemy się czymś, co jest trzonem programowania, czyli *algorytmami*. Aby jednak nie zaczynać całkiem na sucho, pierwsze ćwiczenie niech będzie działającym programem. Nie będziemy się na razie wgłębiać w jego budowę. Spróbujemy go jedynie wpisać, uruchomić i zobaczyć efekt jego działania.

1.1. Na dobry początek — jednak prosty program

ĆWICZENIE

1.1 Pierwszy program

Napisz i uruchom program, który przywita Cię Twoim imieniem.

Uruchom program IDLE (z menu *Start* wybierz *Wszystkie programy*, następnie *Python 3.6* oraz *IDLE (Python 3.6 32-bit)*. Z menu *File* wybierz *New* (lub wciśnij kombinację klawiszy *Ctrl+N*). W otwarte okienko edycyjne wpisz poniższy program:

```
# Program wypisuje powitanie osoby, która
# właściwie wpisze swoje imię w odpowiednie miejsce.

imie = "Andrzej"  # Tu wpisz własne imię.

print("Witaj " + imie + "!")
```

Przepisz go dokładnie i bez błędów — każda pomyłka może spowodować kłopoty z uruchomieniem. Jedyna zmiana, jaką możesz wprowadzić, to zmiana imienia *Andrzej* na własne. Nie musisz też koniecznie wpisywać tekstów poprzedzonych znakiem `#`. Tak w Pythonie oznaczane są komentarze. Nie wpływają one na działanie programu, ale mają kolosalne znaczenie w przypadku, kiedy program trzeba poprawić albo wyjaśnić komuś jego strukturę. Mimo że nie musisz wpisywać komentarzy, zrób to, aby od początku nabrać dobrych przyzwyczajeń. I nie daj się zwieść myśli, że zrobisz to później. Wielokrotnie obiecywaliśmy sobie, że ponieważ jest mało czasu, skupimy się na tekście programu, a kiedyś, „w wolnej chwili", opiszemy go komentarzami. Jak się nietrudno domyślić, zaowocowało to tysiącami wierszy nieopisanego tekstu, który nigdy już nikomu się do niczego nie przyda. Zrozumienie, w jaki sposób program działa, może zająć więcej czasu niż napisanie go od nowa.

Nadszedł moment uruchomienia. Wciśnij klawisz *F5* (jest to odpowiednik wybrania z menu *Run* polecenia *Run*). Jeżeli przy wpisywaniu programu popełniłeś błędy, informacja o tym pojawi się w nowym okienku, a linia zawierająca błąd zostanie podświetlona. Nie próbuj na razie wgłębiać się w jej treść, tylko jeszcze raz dokładnie przejrzyj

program i popraw błąd. Jeżeli wpisałeś program poprawnie, powinieneś ujrzeć na ekranie wynik podobny do poniższego:

```
RESTART: D:/python/r_01/cw1_1.py
Witaj Andrzej!
>>>
```

Na koniec trzeba wyłączyć edytor oraz środowisko. W edytorze IDLE wciśnij kombinację *Ctrl+Q* (co odpowiada wybraniu z menu *File* polecenia *Exit*). Na pytanie, czy zapisać zmiany, odpowiedz przecząco. Następnie to samo zrób w oknie Python 3.6.0 Shell.

1.2. Wróćmy do metod

No właśnie. Przekonałeś się, że komputer do spółki z Pythonem potrafią zrozumieć to, co masz im do powiedzenia, pora więc… zająć się teorią. Tak powinieneś robić zawsze, kiedy przyjdzie Ci rozwiązać jakiś problem za pomocą komputera. Warto siąść z kartką papieru i zastanowić się nad istotą zagadnienia. Każda minuta poświęcona na analizę problemu może zaowocować oszczędnością godzin podczas pisania kodu… Najważniejsze jest dobrze zrozumieć problem i wymyślić *algorytm* jego rozwiązania. No właśnie. Co to słowo właściwie oznacza?

Najprościej rzecz ujmując, algorytm to po prostu *metoda* rozwiązania problemu albo — pisząc inaczej — *przepis na jego rozwiązanie*. Oczywiście nie jest to tylko pojęcie informatyczne — równie dobrze stosuje się je w wielu dziedzinach życia codziennego (jak choćby w gotowaniu). Tak samo jak można myśleć o przepisie na ugotowanie makaronu, można rozważać algorytm jego gotowania. Rozważając algorytm rozwiązania problemu informatycznego, należy mieć na uwadze:

- ***dane***, które mogą być pomocne do jego rozwiązania — wraz ze sposobem ich przechowania, czyli strukturą danych;
- ***wynik***, który chcemy uzyskać.

Gdzieś w tle rozważamy też *czas*, który mamy na uzyskanie wyniku z danych. Oczywiście tak naprawdę myślimy o dwóch czasach: jak szybko dany program trzeba napisać i jak szybko musi on działać. Łatwo jest szybko napisać program, który działa wolno, jeszcze łatwiej napisać powoli taki, który działa jak żółw. Prawdziwą sztuką jest szybko napisać coś, co pracuje sprawnie. Należy jednak mieć na uwadze, że

zwykle program (bądź też jego część) jest pisany raz, a wykorzystywany wiele razy, więc o ile nie grozi to zawaleniem terminów, warto poświęcić czas na udoskonalenie algorytmu.

A zatem rozważając dane, które masz do dyspozycji, oraz mając na uwadze czas, musisz określić, w jaki sposób uzyskać jak najlepszy wynik. Inaczej mówiąc, musisz określić *działania*, których podjęcie jest konieczne do uzyskania wyniku, oraz ich właściwą *kolejność*.

ĆWICZENIE

1.2 Algorytm gotowania makaronu

Zapisz sposób, czyli algorytm, gotowania makaronu.

Wróćmy do przykładu z makaronem. Przyjmijmy, że mamy makaron spaghetti jakości pozwalającej uzyskać zadowalający nas wynik, sól, wodę, garnek, cedzak, minutnik i kuchnię. Makaron możemy ugotować tak:

1. Zagotować w garnku wodę.
2. Do gotującej się wody włożyć makaron, tak aby był w niej zanurzony.
3. Posolić do smaku (w kuchni takie pojęcie jest łatwiej akceptowalne niż w informatyce — tu trzeba by dokładnie zdefiniować, co oznacza „do smaku”, a być może zaprojektować jakiś system doradzający, czy ilość soli jest wystarczająca; ponieważ chcemy jednak stworzyć algorytm prosty i dokładny, przyjmijmy normę — 3/4 łyżki soli kuchennej na 5 litrów wody).
4. Gotować około 8 minut, od czasu do czasu mieszając.
5. Zagotowany makaron odcedzić, używając cedzaka.
6. Również używając cedzaka, polać makaron dokładnie zimną wodą, aby się nie sklejał.
7. Przesypać makaron na talerz.

No i jedzenie gotowe. Można jeszcze pomyśleć nad przyprawieniem makaronu jakimś sosem, ale to już inny algorytm. Przedstawiona metoda nie musi być najlepsza czy jedyna. To po prostu algorytm gotowania makaronu.

Warto zwrócić uwagę, że oprócz samych składników oraz czynności niezwykle ważna jest *kolejność* wykonania opisanych czynności. Makaron posolony już na talerzu (po 7. punkcie algorytmu) smakowałby dużo

gorzej (choć chyba każdemu zdarzyła się taka wpadka). Polewanie zimną wodą makaronu przed włożeniem go do garnka (a więc przed punktem 1.) na pewno nie zapobiegnie jego sklejaniu.

Jakież to mało informatyczne! Czy to w ogóle ma związek z tworzeniem programów? Zdecydowanie TAK. W następnym ćwiczeniu rozważymy mniej kulinarny, a bardziej matematyczny problem (matematyka często przeplata się z informatyką i wiele problemów rozwiązywanych za pomocą komputerów to problemy matematyczne).

ĆWICZENIE

1.3. Algorytm znajdowania NWD

Znajdź największy wspólny dzielnik (NWD) liczb naturalnych *A* i *B*.

Zadanie ma wiele rozwiązań. Pierwsze nasuwające się, nazwiemy je *siłowe*, jest równie skuteczne, co czasochłonne i niezgrabne. Pomysł jest następujący. Począwszy od mniejszej liczby, a skończywszy na znalezionym rozwiązaniu, sprawdzamy, czy liczba dzieli *A* i *B* bez reszty. Jeżeli tak — to mamy wynik, jeżeli zaś nie — pomniejszamy liczbę o *1* i sprawdzamy dalej. Innymi słowy, sprawdzamy podzielność liczb *A* i *B* (załóżmy, że *B* jest mniejsze) przez *B*, potem przez *B* – 1, *B* – 2 i tak do skutku… Algorytm na pewno da pozytywny wynik (w najgorszym razie zatrzyma się na liczbie 1, która na pewno jest dzielnikiem *A* i *B*). W najgorszym razie będzie musiał wykonać *2B* operacji dzielenia i *B* operacji odejmowania. To zadanie na pewno jednak da się i należy rozwiązać lepiej.

Drugi algorytm nosi nazwę *algorytm Euklidesa*. Polega na powtarzaniu cyklu następujących operacji: podziału większej z liczb przez mniejszą (z resztą) i dalszych czynności prowadzących do wybrania dzielnika i znalezionej reszty. Operacja jest powtarzana tak długo, aż resztą będzie 0. Szukanym największym wspólnym dzielnikiem jest dzielnik ostatniej operacji dzielenia. Oto przykład (szukamy największego dzielnika liczb 12 i 32):

```
32 / 12 = 2 reszty 8
12 / 8  = 1 reszty 4
8 / 4   = 2 reszty 0
```

Szukanym największym wspólnym dzielnikiem 12 i 32 jest 4. Jak widać, zamiast sprawdzania 9 liczb (12, 11, 10, 9, 8, 7, 6, 5, 4) z wykonaniem dwóch dzieleń dla każdej z nich, jak miałoby to miejsce w przypadku rozwiązania „siłowego", wystarczyły nam tylko trzy dzielenia.

Bardzo ciekawy jest trzeci algorytm, będący modyfikacją *algorytmu Euklidesa*, ale niewymagający ani jednego dzielenia. Tak, to jest naprawdę możliwe. Jeżeli liczby są różne, szukamy ich różnicy (od większej odejmując mniejszą). Odrzucamy większą z liczb i robimy to samo dla mniejszej z nich i wyniku odejmowania. Na końcu, kiedy liczby będą sobie równe, będą jednocześnie wynikiem naszych poszukiwań. Nasz przykład z liczbami 32 i 12 będzie się przedstawiał następująco:

```
32 – 12 = 20
20 – 12 = 8
12 – 8 = 4
8 – 4 = 4
4 = 4
```

Znowu znaleziono poprawny wynik, czyli 4. Operacji jest co prawda więcej, ale warto zwrócić uwagę, że są to jedynie operacje odejmowania, a nie dzielenia. Koszt zaś operacji dodawania i odejmowania jest znacznie mniejszy niż dzielenia i mnożenia (pisząc „koszt", mamy tu na myśli czas pracy procesora, niezbędny do wykonania działania). Zastanówmy się jeszcze, na czym polega różnica między ostatnimi dwoma algorytmami. Po prostu szukanie reszty z dzielenia liczb A i B poprzez dzielenie zastąpiono wieloma operacjami odejmowania.

ĆWICZENIE

1.4 Algorytm znajdowania NWW

Znajdź najmniejszą wspólną wielokrotność (NWW) liczb naturalnych *A* i *B*.

Czasem najlepsze są rozwiązania najprostsze. Od razu możemy się domyślić, że „siłowe" rozwiązania (na przykład sprawdzanie podzielności przez A i B liczb od $A \cdot B$ w dół aż do większej z nich i przyjęcie jako wyniku najmniejszej, której obie są dzielnikami), choć istnieją, nie są tym, czego szukamy. A wystarczy przypomnieć sobie fakt z matematyki z zakresu szkoły podstawowej:

$$NWW(A,B) = \frac{A \cdot B}{NWD(A,B)}$$

i już wiadomo, jak problem rozwiązać, wykorzystując algorytm, który znamy. Usilne (i często uwieńczone sukcesem) próby rozwiązania problemu poprzez sprowadzenie go do takiego, który już został rozwiązany, to jedna z cech programistów. Jak zobaczysz w następnych ćwiczeniach, często stosując różne techniki, programiści są nawet w stanie sprowadzić rozwiązanie zadania do… rozwiązania tego samego zadania dla innych (łatwiejszych) danych w nadziei, że dane w końcu staną się tak proste, iż będzie można podać wynik „z głowy". I to działa!

Podstawą tak sprawnego znalezienia rozwiązania tego ćwiczenia okazała się elementarna znajomość matematyki. Jak już pisaliśmy, matematyka dość silnie splata się z programowaniem i dlatego dla własnego dobra przed przystąpieniem do „klepania" w klawiaturę warto przypomnieć sobie kilka podstawowych zależności i wzorów. Jako dowód na to zapraszamy do rozwiązania kolejnego ćwiczenia.

ĆWICZENIE

1.5 Algorytm potęgowania

Znajdź wynik działania A^B.

Z pomocą Pythona można bardzo łatwo obliczyć potęgę liczby. Operację taką zapisuje się przy użyciu specjalnego operatora: **. Zapis *A* ** *B* oznacza liczbę *A* podniesioną do potęgi *B*. Prawda, że proste? Wyobraźmy sobie jednak sytuację, w której operator potęgowania nie jest dla nas dostępny, i musimy poradzić sobie inaczej. Tylko jakim sposobem? Jako ułatwienie podpowiemy, że trzeba skorzystać z własności logarytmu i funkcji e^x.

Należy przeprowadzić następujące rozumowanie:

$$A^B = e^{\ln\left(A^B\right)} = e^{B\cdot\ln(A)}$$

ponieważ $x = e^{ln(x)}$ oraz $ln(x^y) = y \cdot ln(x)$. Obie funkcje (e^x i $ln(x)$) są w Pythonie dostępne, więc dzięki temu problem możemy uznać za rozwiązany. Nie było to trudne dla osób, które potrafią się posługiwać suwakiem logarytmicznym, ale pozostałych może przyprawić o ból głowy i wywołać konieczność przypomnienia sobie logarytmów. Warto

pamiętać, że rozwiązanie to będzie skuteczne jedynie dla dodatnich wartości podstawy potęgi i nie znajdziemy w ten sposób istniejącego wyniku działania $(-4)^4$.

ĆWICZENIE

1.6 Algorytm obliczania silni

Znajdź silnię danej liczby (*N!*).

Jak wiadomo z lekcji matematyki, silnia liczby jest iloczynem wszystkich liczb naturalnych mniejszych od niej lub jej równych, czyli:

$$N! = 1 \cdot 2 \cdot \ldots \cdot (N-1) \cdot N$$

Już bezpośrednio z tej definicji wynika jedno (całkiem poprawne) rozwiązanie tego problemu. Należy po prostu uzyskać wynik mnożenia przez siebie wszystkich liczb naturalnych mniejszych od lub równych danej liczbie. Ten algorytm nosi nazwę *iteracyjny* i zostanie dokładnie pokazany w ćwiczeniu 3.37.

Zastanów się jednak jeszcze nad drugim algorytmem. Silnia ma też drugą definicję (oczywiście równoważną poprzedniej):

$$N! = \begin{cases} 1 & \text{gdy} \quad N = 0 \\ N \cdot (N-1)! & \text{gdy} \quad N > 0 \end{cases}$$

W tej definicji jest coś dziwnego. Odwołuje się do… samej siebie. Na przykład przy liczeniu 5! każe policzyć 4! i pomnożyć przez 5. Jako pewnik daje nam tylko fakt, że 0! = 1. Jak się okazuje — to zupełnie wystarczy. Spróbuj na kartce, zgodnie z tą definicją, policzyć 5!. Powinieneś otrzymać taki ciąg obliczeń:

```
5! = 5 * 4!
5! = 5 * (4 * 3!)
5! = 5 * (4 * (3 * 2!))
5! = 5 * (4 * (3 * (2 * 1!)))
5! = 5 * (4 * (3 * (2 * (1 * 0!))))
5! = 5 * (4 * (3 * (2 * (1 * 1))))
5! = 5 * (4 * (3 * (2 * 1)))
5! = 5 * (4 * (3 * 2))
5! = 5 * (4 * 6)
5! = 5 * 24
5! = 120
```

Jak widać, otrzymaliśmy poprawny wynik. Prześledzenie tego przykładu pozwoli na zrozumienie takiego sposobu definiowania funkcji i przeprowadzania obliczeń. Metoda ta jest bardzo często wykorzystywana w programowaniu i nosi nazwę *rekurencja*. W skrócie mówiąc, polega ona na definiowaniu funkcji za pomocą niej samej, ale z mniejszymi (bądź w inny sposób łatwiejszymi) argumentami. A w przypadku programowania — na wykorzystaniu funkcji lub procedury przez nią samą.

ĆWICZENIE

1.7 Rekurencyjne mnożenie liczb

Spróbuj zdefiniować mnożenie dwóch liczb naturalnych *A* i *B* w sposób rekurencyjny.

To tylko ćwiczenie — do niczego się w przyszłości nie przyda (wszak komputery potrafią mnożyć), ale pozwoli Ci się bliżej zapoznać z rekurencją.

$$A \cdot B = \begin{cases} A & \text{gdy} \quad B = 1 \\ A + \left[A \cdot \left(B - 1 \right) \right] & \text{gdy} \quad B > 1 \end{cases}$$

Oczywiście można też:

$$A \cdot B = \begin{cases} B & \text{gdy} \quad A = 1 \\ \left[\left(A - 1 \right) \cdot B \right] + B & \text{gdy} \quad A > 1 \end{cases}$$

Wiele podejmowanych działań (zarówno matematycznych, jak i w życiu codziennym) podlega zasadzie rekurencji. Kilka ćwiczeń dodatkowych pod koniec rozdziału pozwoli jeszcze lepiej się z nią zapoznać.

ĆWICZENIE

1.8 Obliczanie ciągu Fibonacciego

Przemyśl sensowność rozwiązania rekurencyjnego problemu *N*-tego wyrazu ciągu Fibonacciego.

To ćwiczenie jest ilustracją swoistej „pułapki rekurencji", w którą łatwo może wpaść nieuważny programista. Wiele osób po poznaniu tej techniki stosuje ją, kiedy tylko się da. A już na pewno zawsze, gdy problem jest zdefiniowany w sposób rekurencyjny. Łatwo można stać się ofiarą tej pożytecznej techniki.

Rozważmy ciąg Fibonacciego, którego wyrazy opisane są definicją rekurencyjną:

$$F(N)=\begin{cases} 0 & \text{gdy } N=0 \\ 1 & \text{gdy } N=1 \\ F(N-1)+F(N-2) & \text{gdy } N>1 \end{cases}$$

Wydaje się, że nasz problem rozwiązuje już sama definicja. Wystarczy wykorzystać rekurencję. Spróbujmy więc na kartce, zgodnie z definicją, policzyć kilka pierwszych wyrazów ciągu:

$F(0)$	=	**0**
$F(1)$	=	**1**
$F(2)$	=	$F(1) + F(0) = 1 + 0 =$ **1**
$F(3)$	=	$F(2) + F(1) = F(1) + F(0) + F(1) = 1 + 0 + 1 =$ **2**
$F(4)$	=	$F(3) + F(2) = F(2) + F(1) + F(2) =$ $F(1) + F(0) + F(1) + F(1) + F(0) =$ $1 + 0 + 1 + 1 + 0 =$ **3**
$F(5)$	=	$F(4) + F(3) = F(3) + F(2) + F(3) =$ $F(2) + F(1) + F(2) + F(2) + F(1) =$ $F(1) + F(0) + F(1) + F(1) + F(0) + F(1) + F(0) +$ $F(1) = 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 =$ **5**
$F(8)$	=	$F(7) + F(6) = F(6) + F(5) + F(5) + F(4) =$ $F(5) + F(4) + F(4) + F(3) + F(4) + F(3) + F(3) +$ $F(2) =$ …to liczenie nie idzie chyba w dobrym kierunku…
$F(50)$	=	Czy są jacyś odważni?

Coś jest nie tak — algorytm liczący $F(8)$ każe nam w pewnym momencie liczyć aż trzy razy $F(4)$ i trzy razy $F(3)$. Oczywiście nie będzie tego liczył tylko raz i przyjmował wyniku dla wszystkich obliczeń, ponieważ występują one w różnych wywołaniach rekurencyjnych i wzajemnie nic nie wiedzą o swoich wynikach. Podobnie nie da się skorzystać z wyliczonych już poprzednich wartości, ponieważ nigdzie nie są przechowywane. To jest bardzo zły sposób rozwiązania tego problemu.

Mimo że funkcja posiada dobrą rekurencyjną definicję, jej zaprogramowanie za pomocą rekurencji nie jest dobre.

A jak zaprogramować obliczanie wartości takiej funkcji za pomocą komputera? Bardzo łatwo — iteracyjnie. Wystarczy liczyć jej kolejne wartości dla liczb od 2 aż do szukanej i pamiętać zawsze tylko ostatnie dwa wyniki. Ich suma stanie się za każdym razem nową wartością i do kolejnego przebiegu przyjmiemy właśnie ją i większą z poprzednich dwóch. Czas pracy rozwiązania iteracyjnego jest wprost proporcjonalny do wartości N. A od czego zależy ten czas w przypadku rozwiązania rekurencyjnego? Niestety od 2^N. Pamiętasz legendę o twórcy szachów? Jeżeli nie, koniecznie ją odszukaj. Jest ona piękną ilustracją wzrostu wartości funkcji potęgowej:

N	2^N
1	2
2	4
3	8
4	16
10	1024
11	2048
12	4096
20	1048576
30	1073741824
40	1099511627776
60	1152921504606846976
100	1267650600228229401496703205376
1000	ok. $1 \cdot 10^{301}$

Jak widać, wraz ze wzrostem wielkości danej czas rozwiązywania zadania będzie rósł w sposób niesamowity. W ćwiczeniu 3.57 będziesz miał możliwość sprawdzenia czasu działania algorytmu o *złożoności wykładniczej* (tak informatycy nazywają funkcję, która określa czas obliczeń w zależności od rozmiaru danych) dla różnych danych.

Algorytmów o złożoności wykładniczej nie należy stosować. Istnieje co prawda cała grupa problemów, dla których nie znaleziono metody

rozwiązania lepszej niż wykładnicza (i prawdopodobnie nigdy nie zostanie ona znaleziona), jednak przy ich rozwiązywaniu stosuje się inne, przybliżone, lecz szybciej działające algorytmy. W przeciwnym razie nawet dla problemu z bardzo małą liczbą danych trzeba by było czekać wieki na rozwiązanie.

Dużo lepsze są algorytmy o *złożoności wielomianowej* (takie, w których czas pracy zależy od potęgi rozmiaru problemu — na przykład od kwadratu problemu). Bardzo dobre — w klasie wielomianowych — są te o *złożoności liniowej* (i takie algorytmy udało się nam wymyślić!). Istnieje jednak jeszcze jedna klasa, którą informatycy lubią najbardziej. Poznasz ją w następnym ćwiczeniu.

A jako ostatnią informację z tego ćwiczenia zapamiętaj, że każdy algorytm rekurencyjny da się przekształcić do postaci iteracyjnej. Czasami tak łatwo jak silnię czy ciąg Fibonacciego, czasem trudniej lub bardzo trudno (bywa, że algorytm, który w postaci rekurencyjnej miał kilka wierszy, w postaci iteracyjnej ma ich wielokrotnie więcej). Prawie zawsze stracimy na czytelności — zwykle zyskamy jednak na czasie pracy i obciążeniu komputera. Jeżeli więc przekształcenie do postaci iteracyjnej jest proste i oczywiste, należy to zrobić — ale nie za wszelką cenę.

ĆWICZENIE

1.9 Algorytm podnoszenia 2 do potęgi naturalnej

Znajdź metodę obliczania wyrażenia 2^N, gdzie N jest liczbą naturalną.

Nasunął Ci się pierwszy pomysł: skorzystanie z naszego znakomitego algorytmu z ćwiczenia 1.5. Wszak $2^N = e^{N \cdot \ln(2)}$, więc z szybkim wyliczeniem nie będzie problemu. Pomysł nawet nam się podoba (świadczy o tym, że oswoiłeś się już z myślą, by rozwiązywać problemy przez ich sprowadzanie do już rozwiązanych). Ale kłopot polega na tym, że nasza metoda opiera się na funkcjach, które działają na liczbach rzeczywistych (e^x i $\ln(x)$). Ponieważ komputer reprezentuje liczby rzeczywiste z pewnym przybliżeniem, nie dostaniemy niestety dokładnego wyniku — liczby naturalnej. Dla odpowiednio dużego N wynik zacznie być obarczany błędem. A my tymczasem potrzebujemy wyniku będącego liczbą naturalną. Pomyślmy więc nad innym rozwiązaniem.

A gdyby tak po prostu N razy przemnożyć przez siebie liczbę 2 (a jeżeli $N = 0$, za wynik przyjąć 1)? Pomysł jest dobry. Jego złożoność jest *liniowa* (przed chwilą napisaliśmy, że dla liczby N należy pomnożyć

N razy — liniowość rozwiązania widać bardzo dobrze). Rozwiązanie jest poprawne.

Ale da się to zrobić lepiej — rekurencyjnie. Spróbujmy zdefiniować 2^N w następujący sposób:

$$2^N = \begin{cases} 1 & \text{gdy} \quad N = 0 \\ \left(2^{N/2}\right)^2 & \text{gdy} \quad N \text{ jest parzyste} \\ 2 \cdot \left(2^{N/2}\right)^2 & \text{gdy} \quad N \text{ jest nieparzyste} \end{cases}$$

(przez *N/2* rozumiemy całkowitą część dzielenia *N* przez 2). Jako drobne ćwiczenie matematyczne proponujemy sprawdzić (a może nawet udowodnić?), czy jest to prawda. Jeżeli ktoś chce się zmierzyć z dowodem, radzimy przypomnieć sobie dowody *indukcyjne*. Rekurencja w informatyce i indukcja w matematyce to rodzone siostry.

Powstaje pytanie (metodę — rekurencyjną — już mamy): czy to daje nam jakąś oszczędność? Przyjrzyjmy się jeszcze raz temu wzorowi. Za każdym razem wartość argumentu maleje nie o jeden czy dwa (jak było w przypadku silni czy ciągu Fibonacciego), ale… o połowę. Czyli gdy szukamy potęgi 32, za drugim razem będziemy już szukać 16, za trzecim — 8, potem — 4, 2, 1 i zerowej. To nie jest w żaden sposób liniowe. To jest o wiele lepsze! Jak nazwać złożoność tego algorytmu? Przyjęło się mówić, że jest to złożoność *logarytmiczna*. Oznacza to, że czas rozwiązania problemu jest zależny od logarytmu (w tym przypadku o podstawie 2) wielkości danych. To jest to, co informatycy lubią najbardziej.

Tabelka, którą pokazaliśmy wcześniej, byłaby niepełna bez danych o złożoności logarytmicznej. Powtórzmy ją zatem:

N	$\log_2(N)$	2^N
1	0	2
2	1,00	4
3	1,58	8
4	2,00	16
10	3,32	1024
11	3,46	2048
12	3,58	4096

N	$\log_2(N)$	2^N
20	4,32	1048576
30	4,91	1073741824
40	5,32	1099511627776
60	5,91	1152921504606846976
100	6,64	1267650600228229401496703205376
1000	9,97	ok. $1 \cdot 10^{301}$

Czy widzisz różnicę? Dla danej o wartości 1000 algorytm logarytmiczny musi wykonać tylko 10 operacji mnożenia, a liniowy — aż tysiąc. Gdybyśmy wymyślili algorytm wykładniczy, liczby operacji mnożenia nie dałoby się łatwo nazwać, a już na pewno nie dałoby się tych obliczeń przeprowadzić na komputerze.

Ten typ algorytmów, które sprowadzają problem nie tylko do problemów mniejszych tego samego typu, ale do mniejszych przynajmniej dwukrotnie, nazwano (naszym zdaniem słusznie) *dziel i zwyciężaj*. Zawsze, gdy uda Ci się podzielić w podobny sposób problem na mniejsze, masz szansę na uzyskanie dobrego, logarytmicznego algorytmu.

ĆWICZENIE

1.10 Algorytm określania liczb pierwszych

Sprawdź, czy liczba *N* jest liczbą pierwszą.

Dla przypomnienia: liczba pierwsza to taka, która ma tylko dwa różne, naturalne dzielniki: 1 i samą siebie.

Zadanie wbrew pozorom nie jest tylko sztuką dla sztuki. Funkcja sprawdzająca, czy zadana liczba jest pierwsza, czy nie (i znajdująca jej dzielniki), w szybki sposób (a więc o małej złożoności) miałaby ogromne znaczenie w kryptografii, i to takiej silnej, najwyższej jakości, a konkretnie — w łamaniu szyfrów. Warto więc poświęcić chwilkę na rozwiązanie tego zadania.

Pierwszy pomysł: dla każdej liczby od 2 do $N - 1$ sprawdzić, czy nie dzieli N. Jeżeli któraś z nich dzieli — N nie jest pierwsze. W przeciwnym razie jest. Pierwszy pomysł nie jest zły. Funkcja na pewno działa i ma złożoność liniową. Troszkę da się ją poprawić, ale czy bardzo?

Poczyńmy następującą obserwację. Jeżeli liczba N nie była podzielna przez 2, to na pewno nie jest podzielna przez żadną liczbę parzystą. Można więc śmiało wyeliminować sprawdzanie dla wszystkich liczb parzystych większych od 2. Czyli sprawdzać dla 2, 3, 5, 7 itd. Redukujemy w ten sposób problem o połowę, uzyskując złożoność, no właśnie — jaką? Tak, nadal liniową. Algorytm bez wątpienia jest szybszy, ale ciągle w tej samej klasie.

Pomyślmy dalej. Dla każdego „dużego" (większego od $\sqrt{N}$) dzielnika N musi istnieć dzielnik „mały" (mniejszy od $\sqrt{N}$) — będący ilorazem N i tego „dużego". Nie warto więc sprawdzać liczb większych od $\sqrt{N}$ — jeżeli przedtem nie znaleźliśmy dzielnika, dalej też go nie będzie. Czyli nie sprawdzamy liczb do $N - 1$, tylko do $\sqrt{N}$. Czy coś nam to dało? Oczywiście algorytm działa jeszcze szybciej. A jak z jego złożonością? Co prawda nie jest liniowa, ale wykładnicza (tylko z lepszym niż liniowa wykładnikiem). Proste pytanie: z jakim wykładnikiem złożoność wielomianowa jest liniowa, a z jakim jest taka, jaką uzyskaliśmy? Jeżeli podałeś wartości, odpowiednio, 1 i 1/2, to udzieliłeś poprawnej odpowiedzi.

1.3. Co powinieneś zapamiętać z tego cyklu ćwiczeń

- Co to jest algorytm?
- Co to jest złożoność algorytmu?
- Co to jest iteracja?
- Co to jest rekurencja?
- Dlaczego rekurencja nie zawsze jest dobra?
- Na czym polega metoda *dziel i zwyciężaj*?
- Jak wyglądają dobre algorytmy dla kilku prostych problemów: gotowania makaronu, szukania największego wspólnego dzielnika, najmniejszej wspólnej wielokrotności, silni, wyrazu ciągu Fibonacciego, potęgi liczby oraz sprawdzania, czy liczba jest pierwsza?

1.4. Ćwiczenia do samodzielnego rozwiązania

ĆWICZENIE

1.11 Gotowanie potraw

Napisz algorytm gotowania ulubionej potrawy.

Możesz posłużyć się książką kucharską. Zwróć szczególną uwagę na składniki (czyli „dane" algorytmu) oraz na kolejność działań.

ĆWICZENIE

1.12 Udzielanie pierwszej pomocy

Podaj algorytm udzielania pierwszej pomocy osobie poszkodowanej w wypadku samochodowym.

ĆWICZENIE

1.13 Obliczanie pierwiastków

Napisz algorytm liczenia pierwiastków równania kwadratowego.

Funkcja (dla przypomnienia) ma postać $f(x) = ax^2 + bx + c$. Przypomnij sobie szkolny sposób liczenia pierwiastków — on w zasadzie jest już bardzo dobrym algorytmem.

ĆWICZENIE

1.14 Obliczanie wartości wielomianu

Przeanalizuj problem obliczania wartości wielomianu.

Wielomian ma następującą postać: $w(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$. Porównaj metodę najbardziej oczywistą (mnożenie i dodawanie „po kolei") z algorytmem opartym na *schemacie Hornera*, który mówi, że wielomian można przekształcić do postaci: $w(x) = (\ldots (a_n x + a_{n-1})x + \ldots + a_1)x + a_0$. Aby to nieco rozjaśnić: wielomian trzeciego stopnia: $w_3(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ można przekształcić do postaci: $w(x) = ((a_3x + a_2)x + a_1)x + a_0$ — sprawdź, że to jest to samo. Porównaj liczbę

działań (mnożeń i dodawań) w obu przypadkach. Czy złożoność w którymś z nich jest lepsza? Jeżeli nie, to czy mimo wszystko warto stosować któryś z nich? Może jesteś w stanie zauważyć także jakieś inne jego zalety?

ĆWICZENIE

1.15 Zgadywanie liczb

Przeanalizuj grę w zgadywanie liczby.

Pamiętasz grę: zgadnij liczbę z zakresu 1 – 1000? Zgadujący podaje odpowiedź, a Ty mówisz „zgadłeś", „za dużo" albo „za mało". Gdyby zgadujący „strzelał", trafienie trwałoby długo. Można jednak wymyślić bardzo sprawny algorytm zgadnięcia liczby. Spróbuj go sformułować. Ile maksymalnie razy trzeba zgadywać, żeby mieć pewność uzyskania prawidłowego wyniku? Jaką złożoność ma algorytm? Czy przypomina Ci którąś z metod z poprzednich ćwiczeń? Zapamiętaj nazwę tej metody: *przeszukiwanie binarne*.

ĆWICZENIE

1.16 Położenie punktu względem trójkąta

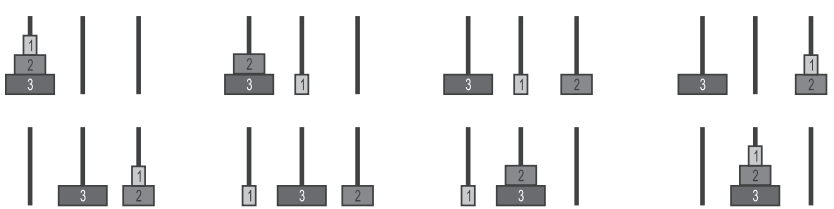
Sprawdź, czy punkt *X* leży wewnątrz, czy na zewnątrz trójkąta *ABC*.

Narysuj oba przypadki na kartce i rozważ pola trójkątów, które powstały poprzez połączenie wierzchołków trójkąta z punktem, oraz kombinacje ich sum. Podaj algorytm sprawdzania.

ĆWICZENIE

1.17 Wieże Hanoi

Napisz algorytm rozwiązania problemu wież Hanoi.



Wieże Hanoi to klasyka zadań informatycznych. Do dyspozycji masz trzy stosy, na których układasz kółka. Na początku kółka tworzą piramidę na jednym z nich. Należy przenieść ją całą na drugi stos zgodnie z zasadami: każdorazowo można przenieść tylko jedno kółko ze szczytu dowolnego stosu; nie można kłaść kółek większych na mniejsze. Przyjrzyj się ilustracji.

Podaj algorytm rozwiązania tego problemu. Zastanów się nad rozwiązaniem rekurencyjnym. Jaką złożoność może mieć wymyślony algorytm? Czy myślisz, że da się znaleźć rozwiązanie o lepszej złożoności?

ĆWICZENIE

1.18 Znajdowanie maksimum

Rozważ algorytmy przeszukiwania ciągu liczb w celu znalezienia maksimum.

Masz do dyspozycji nieuporządkowany skończony ciąg liczb i zadanie, aby znaleźć w nim największą liczbę. Przemyśl dwie metody:

1. Przesuwasz się po kolejnych wyrazach ciągu i sprawdzasz, czy bieżący nie jest większy od dotychczas znalezionego największego (który pamiętasz). Jeżeli tak, to przyjmujesz, że to on jest największy. Po dojściu do końca ciągu będziesz znał odpowiedź.
2. Działasz rekurencyjnie. Jeżeli ciąg jest jednoelementowy, uznajesz, że ten element jest największy. W przeciwnym razie dzielisz ciąg na dwie części i sprawdzasz, co jest większe — największy element lewego podciągu czy największy element prawego podciągu.

Drugi algorytm jest typu *dziel i zwyciężaj* i na pierwszy rzut oka wydaje się lepszy niż pierwszy (liniowy). Sprawdź, czy to prawda. Zrób to na kilku przykładach. Który algorytm jest lepszy? Dlaczego wynik jest taki zaskakujący?

ĆWICZENIE

1.19 Analizowanie funkcji Ackermanna

Przyjrzyj się funkcji Ackermanna.

$$A(m,n)=\begin{cases} n+1 & \text{gdy } m=0 \\ A(m-1,n) & \text{gdy } m>0, n=0 \\ A(m-1, A(m,n-1)) & \text{gdy } m,n>0 \end{cases}$$

Ta niewinnie wyglądająca funkcja zdefiniowana rekurencyjnie to prawdziwy koszmar. Spróbuj policzyć A (2, 3) bez pamiętania w czasie wyliczania wartości już policzonych. A co z A (3, 3)? Czy odważyłbyś się policzyć A (4, 3)? Czy algorytm rekurencyjny zdaje tu egzamin?

Spróbuj podejść do zadania w inny sposób. Zapisuj wyliczane wyniki w tabelce (na przykład w pionie dla wartości m, w poziomie dla n). Poniżej masz początek takiej tabelki:

***m*\n**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10	11
2	3	5	7	9						
3	5									
4										

Spróbuj policzyć kilka kolejnych wartości. Zastanów się, w jaki sposób można spróbować zabrać się do rozwiązania tego problemu iteracyjnie.

Python

ĆWICZENIA PRAKTYCZNE

Python to niezwykle wydajny i wszechstronny język programowania, który znajduje zastosowanie w różnych dziedzinach informatyki. Stanowi fundament rozwiązań wykorzystywanych przez YouTube'a czy aplikacje Google'a. Czytelność i zwięzłość kodu, szeroki wybór bibliotek standardowych oraz wsparcie ze strony producentów różnych systemów operacyjnych sprawiają, że język ten z roku na rok zyskuje coraz większą popularność wśród profesjonalnych programistów i osób amatorsko tworzących wtyczki czy skrypty uruchamiane w aplikacjach komercyjnych.

Czas do nich dołączyć! Sięgnij po książkę *Python. Ćwiczenia praktyczne*, która w prosty i przystępny sposób wprowadzi Cię w świat programowania komputerów przy użyciu jednego z najpopularniejszych języków! Na praktycznych przykładach naucz się czytać algorytmy i je implementować, poznaj podstawy Pythona, składnię oraz konstrukcje stosowane w tym języku — i zacznij myśleć jak zawodowy programista! Tylko krok dzieli Cię od rozpoczęcia przygody z programowaniem, więc nie trać czasu i już dziś zacznij ćwiczyć!

- **Wprowadzenie do algorytmiki**
- **Analiza i rozwiązywanie problemów**
- **Instalacja i korzystanie z IDE**
- **Struktura programu i instrukcje Pythona**
- **Zmienne, proste typy danych i operatory**
- **Korzystanie z bibliotek standardowych**
- **Złożone typy danych i programowanie obiektowe**
- **Praktyczne zadania do samodzielnego wykonania**

Dowiedz się, jak oswoić Pythona!

Helion

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

ISBN 978-83-283-3513-4

Informatyka w najlepszym wydaniu

cena: 24,90 zł