CORY ALTHOFF

INFORMATYK *samouk*

Przewodnik
po strukturach danych
i algorytmach
dla początkujących

Helion

Tytuł oryginału: The Self-Taught Computer Scientist: The Beginner's Guide to Data Structures & Algorithms

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-9194-9

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

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Rozdział

1

Czym jest algorytm

Niezależnie od tego, czy chcesz odkrywać tajemnice wszechświata,
czy tylko rozwijać swoją karierę w 21. wieku,
podstawy programowania są kluczową umiejętnością, którą należy zdobyć.

Stephen Hawking

Algorytm jest sekwencją kroków pozwalających rozwiązać pewien problem. Oto przykładowo jeden z możliwych algorytmów robienia jajecznicy: trzy jajka należy rozbić i wlać do miseczki, następie trzeba je roztrzepać i wlać na patelnię, patelnię umieścić na kuchence, mieszać jajka i poczekać, aż się zetną; kiedy jajka nie będą już płynne, należy przełożyć je z patelni na talerz. Ta część książki jest w całości poświęcona algorytmom. Poznasz w niej algorytmy, których będziesz mógł używać do rozwiązywania takich problemów jak znajdowanie liczb pierwszych. Dowiesz się także, w jaki sposób można napisać nowy, elegancki algorytm oraz jak wyszukiwać i sortować dane.

W tym rozdziale wyjaśnię, jak można porównywać algorytmy, by je analizować. Dla programisty bardzo ważne jest zrozumienie, dlaczego jeden algorytm będzie lepszy od innych, gdyż programiści większość czasu poświęcają na pisanie algorytmów i określanie, jakich struktur danych należy w nich używać. Jeśli nie będziesz mieć pojęcia, dlaczego powinieneś wybrać ten konkretny algorytm, a nie inny, nie będziesz efektywnym programistą, zatem informacje zamieszczone w tym rozdziale mają kluczowe znaczenie.

Choć algorytmy są jednym z kluczowych pojęć w informatyce, to jednak informatycy nie są w stanie dojść do porozumienia i przedstawić formalnej definicji, czym jest algorytm. Istnieje wiele konkurujących definicji, a jedną z najbardziej znanych jest zaproponowana przez Donalda Knutha. Opisuje on algorytm jako określony, efektywny i skończony proces pobierający dane wejściowe i generujący na ich podstawie wyniki.

> *Określoność* (ang. *definiteness*) oznacza, że poszczególne kroki algorytmu są przejrzyste, spójne i jednoznaczne.
>
> *Efektywność* (ang. *effectiveness*) oznacza, że każdy krok algorytmu możemy wykonać precyzyjnie, by rozwiązać problem.
>
> *Skończoność* (ang. *finiteness*) oznacza, że algorytm zakończy się po określonej liczbie kroków.

Często spotykanym uzupełnieniem tej listy cech jest *poprawność* (ang. *correctness*). Algorytm zawsze powinien generować te same wyniki dla określonych danych wejściowych, przy czym wyniki te powinny być prawidłową odpowiedzią na rozwiązywany problem.

Większość algorytmów (choć nie wszystkie) spełnia te wymagania, a wyjątki od nich mają duże znaczenie. Jeśli na przykład tworzymy generator liczb losowych, naszym celem będzie uzyskanie losowości, tak by nikt nie mógł użyć danych wejściowych, by odgadnąć wyniki. Co więcej, nie wszystkie algorytmy rygorystycznie podchodzą do poprawności. Przykładowo akceptowalnym efektem pracy algorytmu może być oszacowanie wyniku, o ile tylko znana będzie niepewność tych szacunków. Jednak w przeważającej większości przypadków algorytmy powinny spełniać powyższe wymagania. Jeśli napiszemy algorytm do robienia jajecznicy, użytkownik nie będzie szczęśliwy, gdy od czasu do czasu zamiast jajecznicy wyjdą omlet lub jajko sadzone.

Analiza algorytmów

Czasami mamy więcej niż jeden algorytm służący do rozwiązania konkretnego problemu. Istnieje na przykład kilka różnych sposobów sortowania listy. Kiedy dany problem można rozwiązać przy użyciu kilku różnych algorytmów, jak określić, który z nich jest najlepszy? Czy będzie to najprostszy z algorytmów? A może najszybszy? Może najmniejszy? A może jeszcze jakiś inny?

Jednym ze sposobów oceniania algorytmów jest czas ich działania. **Czas działania** algorytmu to długość okresu czasu, jaki zajmuje komputerowi wykonanie algorytmu napisanego w określonym języku programowania, na przykład w Pythonie. Poniżej jako przykład przedstawiłem prosty algorytm napisany w Pythonie, który wyświetla kolejne liczby od 1 od 5:

```
for i in range(1, 6):
    print(i)
```

Czas działania tego algorytmu możemy ustalić, używając wbudowanego modułu Pythona o nazwie `time`, który pozwoli zmierzyć, ile czasu zajęło komputerowi wykonanie algorytmu:

```
import time

start = time.time()
for i in range(1, 6):
    print(i)
end = time.time()
print(end – start)

>> 1
>> 2
>> 3
>> 4
>> 5
>> 0.15141820907592773
```

Program wyświetla liczby od 1 do 5, a następnie prezentuje czas wykonania. W przedstawionym przykładzie widać, że wykonanie programu zajęło 0,15 sekundy.

A teraz spróbuj wykonać ten program jeszcze raz:

```
import time

start = time.time()
for i in range(1, 6):
    print(i)
end = time.time()
print(end – start)

>> 1
>> 2
>> 3
>> 4
>> 5
>> 0.14856505393981934
```

Po drugim uruchomieniu programu wyświetlony czas jego wykonania powinien być inny. Kiedy wykonasz ten program jeszcze raz, czas jego wykonania znowu będzie inny. Czas działania algorytmu za każdym razem jest inny, gdyż za każdym razem zmienia się moc obliczeniowa dostępna na komputerze, a to z kolei ma wpływ na czas wykonywania programu.

Co więcej, czas działania algorytmu będzie inny na różnych komputerach. Jeśli wykonamy go na komputerze o mniejszej mocy obliczeniowej, algorytm będzie działał wolniej, z kolei na bardziej potężnym komputerze algorytm będzie działał szybciej. Czas działania programu zależy także od języka programowania, w którym program został napisany. I tak program napisany w C będzie działał szybciej niż ten sam program napisany w Pythonie, gdyż język C zazwyczaj jest szybszy od Pythona.

Ze względu na to, że czas działania algorytmów zależy od wielu czynników, takich jak moc obliczeniowa komputera czy też zastosowany język programowania, nie jest najlepszym sposobem porównywania dwóch algorytmów. Zamiast tego informatycy porównują algorytmy na podstawie liczby kroków, jakie muszą zostać wykonane w celu rozwiązania problemu. Algorytmy można porównywać na podstawie liczby wykonywanych kroków bez zwracania uwagi na zastosowany język programowania

czy też używany komputer. Przeanalizujmy to na przykładzie. Poniżej jeszcze raz zamieściłem przedstawiony wcześniej prosty program:

```
for i in range(1, 6):
    print(i)
```

Wykonanie tego programu wymaga pięciu kroków (koniecznych jest pięć iteracji pętli, z których każda wyświetla wartość `i`). Liczbę kroków wykonywanych przez ten algorytm można zapisać przy użyciu poniższego równania:

```
f(n) = 5
```

Jeśli program będzie bardziej skomplikowany, równanie zapewne trzeba będzie zmienić. Możemy na przykład zliczać sumę wszystkich wyświetlanych liczb:

```
count = 0
for i in range(1, 6):
    print(i)
    count += i
```

Wykonanie tego algorytmu będzie wymagało jedenastu kroków. Pierwszym z nich jest określenie wartości początkowej 0 zmiennej `count`. Następnie algorytm wyświetla pięć liczb i pięć razy powiększa wartość zmiennej `count` (1 + 5 + 5 = 11).

Zatem nowe równanie określające liczbę kroków algorytmu będzie wyglądać następująco:

```
f(n) = 11
```

A co się stanie, jeśli zmienimy liczbę `6` umieszczoną w kodzie programu na jakąś zmienną?

```
count = 0
for i in range(1, n):
    print(i)
    count += i
```

Równanie zmieni postać na następującą:

```
f(n) = 1 + 2n
```

W tym przypadku liczba kroków wykonywanych przez algorytm zależy od wartości *n*. Cyfra 1 w powyższym równaniu reprezentuje pierwszy krok algorytmu, czyli instrukcję `count = 0`. Po tej pierwszej instrukcji algorytm wykonuje dwa razy po *n* kroków. A zatem, jeśli *n* wyniesie 5, to `f(n) = 1 +` 2×5. Zmienną *n* we wzorze na liczbę kroków wykonywanych przez algorytm informatycy nazywają **wielkością problemu**. W naszym przykładzie możemy powiedzieć, że czas wymagany do rozwiązania problemu o wielkości *n* wynosi `1 + 2n` albo w zapisie matematycznym $T(n) = 1 + 2n$.

Jednak równanie opisujące liczbę kroków algorytmu nie jest szczególnie pomocne, gdyż między innymi nie zawsze można dokładnie policzyć liczbę operacji wykonywanych przez algorytm. Jeśli na przykład algorytm będzie zawierał wiele instrukcji warunkowych, nie będziemy mogli z góry wiedzieć, które z nich zostaną wykonane. Na szczęście okazuje się, że wcale nie musimy zwracać

uwagi na dokładną liczbę kroków wykonywanych przez algorytm. Naprawdę interesuje nas to, jak będzie się zmieniać działanie algorytmu wraz ze wzrostem liczby *n*. W niewielkich zbiorach danych większość algorytmów działa dobrze. Nawet najmniej wydajny algorytm będzie działał świetnie dla *n* równego 1. Jednak w rzeczywistości wartość *n* zapewne będzie większa od 1 — może ona wynosić kilkaset tysięcy, milion lub nawet więcej.

Najważniejszą informacją na temat algorytmu nie jest dokładna liczba kroków, które zostaną wykonane, lecz raczej przybliżona liczba kroków wykonywanych, gdy wartość *n* będzie rosła. Wraz ze zwiększaniem się wartości *n* jedna z części równania zacznie dominować nad pozostałymi, aż do momentu gdy przestaną one mieć znaczenie. Przeanalizujmy kolejny program napisany w Pythonie:

```
def print_it(n):
    # pętla 1
    for i in range(n):
        print(i)
    # pętla 2
    for i in range(n):
        print(i)
        for j in range(n):
            print(j)
            for h in range(n):
                print(h)
```

Która z części tego programu jest najważniejsza z punktu widzenia określania liczby kroków koniecznych do wykonania w celu zakończenia algorytmu? Możesz uznać, że ważne są obie (zarówno pierwsza pętla, jak i druga — zawierająca zagnieżdżone pętle). W końcu, jeśli *n* przyjmie wartość 10000, komputer będzie musiał wyświetlić sporo liczb w obu pętlach.

Jednak okazuje się, że z punktu widzenia wydajności działania obu algorytmów poniższy fragment kodu nie ma znaczenia:

```
# pętla 1
for i in range(n)
    print(i)
```

Aby zrozumieć, dlaczego tak się dzieje, musimy przyjrzeć się, co się stanie, kiedy wartość *n* będzie rosła.

Poniżej przedstawiłem równanie na ilość kroków analizowanego algorytmu:

```
T(n) = n + n**3
```

Jeśli mamy dwie zagnieżdżone pętle, z których każda wykonuje *n* kroków, w sumie algorytm wykona ich *n***2 (*n* do potęgi drugiej), gdyż jeśli *n* przyjmie wartość 10, będziemy musieli wykonać dziesięć razy po dziesięć kroków, a to daje 10**2. Z tego samego powodu trzy zagnieżdżone pętle `for` zawsze będą wymagały wykonania *n***3 kroków. W ostatnim równaniu, jeśli *n* przyjmie wartość 10, pierwsza pętla programu wykona 10 kroków, a druga — 10^3, czyli 1000. Jeśli *n* wyniesie 1000, pierwsza pętla wykona 1000 kroków, a druga 1000^3, czyli miliard.

Rozumiesz, co się dzieje? Wraz ze wzrostem wartości *n* druga część algorytmu rośnie o tyle szybciej od pierwszej, że pierwsza przestaje mieć znaczenie. Gdybyśmy na przykład musieli wykonać nasz

program dla bazy danych zawierającej 100 000 000 rekordów, nie warto przejmować się tym, ile kroków będzie musiała wykonać pierwsza część programu, gdyż druga wykona ich wykładniczo więcej. W przypadku 100 000 000 rekordów druga część programu wymagałaby wykonania septyliona kroków (septylion to 1 i 24 zera). Zastosowanie takiego algorytmu nie jest rozsądne. W takim przypadku wykonanie dodatkowych 100 000 000 kroków nie ma znaczenia.

Ze względu na to, że najważniejszą częścią algorytmu jest ta, która wraz ze wzrostem wartości *n* rośnie najszybciej, informatycy do wyrażania efektywności algorytmów używają notacji dużego O, a nie równań T(*n*). **Notacja dużego O** to zapis matematyczny opisujący, jak czasowe lub pamięciowe (o tych wymaganiach dowiesz się dalej w tej książce) wymagania algorytmu zmieniają się wraz ze zmianami wartości *n*.

Informatycy używają notacji dużego O w celu przekształcenia równania T(*n*) na funkcję określającą rząd wielkości. **Rząd wielkości** to klasa w systemie klasyfikacji, przy czym każda z tych klas jest bardzo wiele razy większa lub mniejsza od kolejnej. W funkcji rzędu wielkości używamy dominującej części równania T(*n*), a wszystkie pozostałe pomijamy. Dominująca część równania T(*n*) jest rzędem wielkości algorytmu. Poniżej wymieniłem najczęściej używane klasy rzędu wielkości w notacji dużego O, posortowane od najlepszej (najbardziej wydajnej) do najgorszej (najmniej wydajnej):

czas stały,

czas logarytmiczny,

czas liniowy,

czas logarytmiczno-liniowy,

czas kwadratowy,

czas sześcienny,

czas wykładniczy.

Każdy z tych rzędów wielkości określa czasową złożoność algorytmu. **Złożoność czasowa** to maksymalna liczba kroków niezbędnych do wykonania algorytmu dla określonej wartości *n*.

Przyjrzyjmy się teraz nieco dokładniej każdemu z tych rzędów wielkości.

Czas stały

Najbardziej efektywny rząd wielkości jest nazywamy *złożonością o czasie stałym*. Algorytm jest wykonywany w **czasie stałym**, jeśli liczba kroków niezbędnych do jego wykonania nie zależy od wielkości problemu. W notacji dużego O złożoność o czasie stałym zapisujemy jako O(1).

Załóżmy, że prowadzimy księgarnię internetową i dajemy darmową książkę dla każdego pierwszego klienta bieżącego dnia. Klienci są zapisywani na liście o nazwie `customers`. Algorytm pobierający pierwszego klienta mógłby wyglądać następująco:

```
free_books = customers[0]
```

Równanie T(*n*) dla takiego algorytmu miałoby następującą postać:

```
T(n) = 1
```

Algorytm wymaga wykonania jednego kroku, niezależnie od liczby klientów księgarni. Jeśli tych klientów będzie 1000, algorytm będzie musiał wykonać jeden krok. Jeśli będzie ich 10 000, algorytm także będzie musiał wykonać jeden krok, podobnie jak w przypadku, gdyby tych klientów był trylion.

Jeśli spróbujemy przedstawić złożoność o czasie stałym na wykresie, który na osi X będzie miał liczbę danych wejściowych, a na osi Y liczbę wykonywanych kroków, będzie on mieć postać linii prostej, jak pokazałem na rysunku 1.1.

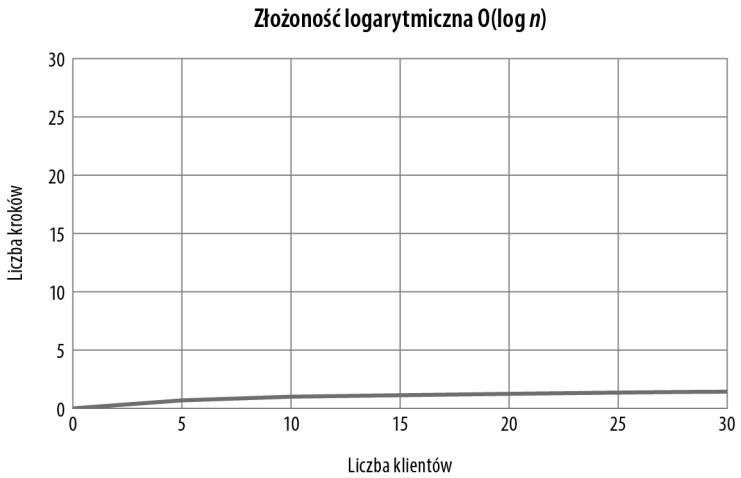
Rysunek 1.1. Złożoność stała

Jak widać, liczba kroków, które należy wykonać w celu zakończenia algorytmu, nie rośnie wraz z rozmiarem problemu. A zatem jest to najbardziej wydajny z algorytmów, gdyż czas jego wykonywania nie zmienia się wraz ze zwiększaniem zbioru danych.

Czas logarytmiczny

Czas logarytmiczny jest drugą najbardziej efektywną złożonością czasową. Algorytm działa w **czasie logarytmicznym**, jeśli czas jego działania jest proporcjonalny do logarytmu wielkości danych wejściowych. Taką złożonością czasową cechują się na przykład algorytmy wyszukiwania binarnego, które w każdej iteracji pomijają bardzo wiele danych. Jeśli jeszcze nie do końca rozumiesz, o co w tym chodzi, nie przejmuj się: wytłumaczę to szczegółowo dalej w tej książce. W notacji dużego O złożoność logarytmiczna jest zapisywana jako O(log *n*).

Przykład wykresu złożoności logarytmicznej przedstawiłem na rysunku 1.2.



Rysunek 1.2. Złożoność logarytmiczna

W algorytmie o złożoności logarytmicznej liczba kroków koniecznych do jego zakończenia rośnie znacznie wolniej od ilości danych wejściowych.

Czas liniowy

Kolejnym pod względem wydajności typem algorytmów są algorytmy o liniowym czasie działania. W ich przypadku czas działania rośnie w takim samym tempie co wielkość problemu. W notacji dużego O liniowy czas działania zapisujemy jako O(*n*).

Załóżmy, że musimy zmodyfikować zasady przyznawania darmowej książki w taki sposób, że zamiast przyznawać ją pierwszemu klientowi w danym dniu, chcemy przejrzeć listę klientów i przyznać książkę każdemu, którego imię zaczyna się od litery „B”.

```
free_book = False
customers = ["Leszek", "Beata", "Darek", "Balbina", "Krzysiek"]
for customer in customers:
    if customer[0] == 'B':
        print(customer)
```

Jeśli lista klientów będzie zawierać pięć elementów, zakończenie algorytmu będzie wymagać wykonania pięciu kroków. W przypadku listy zawierającej dziesięciu klientów tych kroków będzie dziesięć; w przypadku listy zawierającej dwudziestu klientów kroków będzie dwadzieścia i tak dalej.

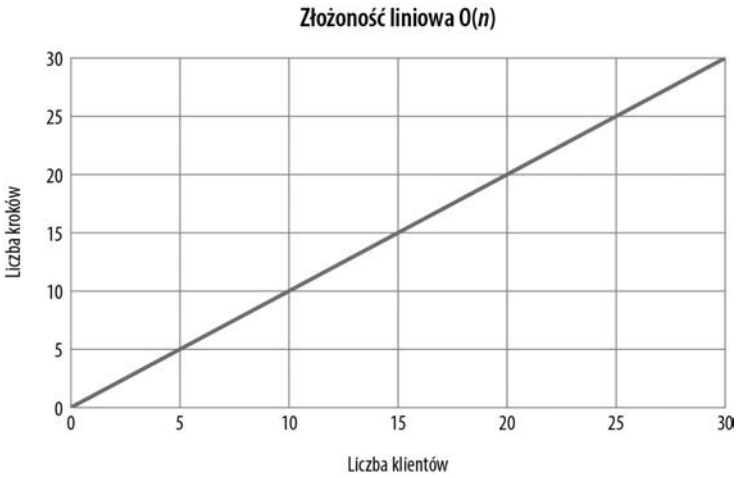
Poniżej przedstawiłem równanie złożoności czasowej tego programu:

```
f(n) = 1 + 1 + n
```

W notacji dużego O możemy pominąć wszystkie stałe i skoncentrować się na części dominującej:

```
O(n) = n
```

W algorytmach liniowych wraz ze wzrostem wartości *n* liczba kroków wykonywanych przez algorytm rośnie w takim samym tempie co *n* (patrz rysunek 1.3).



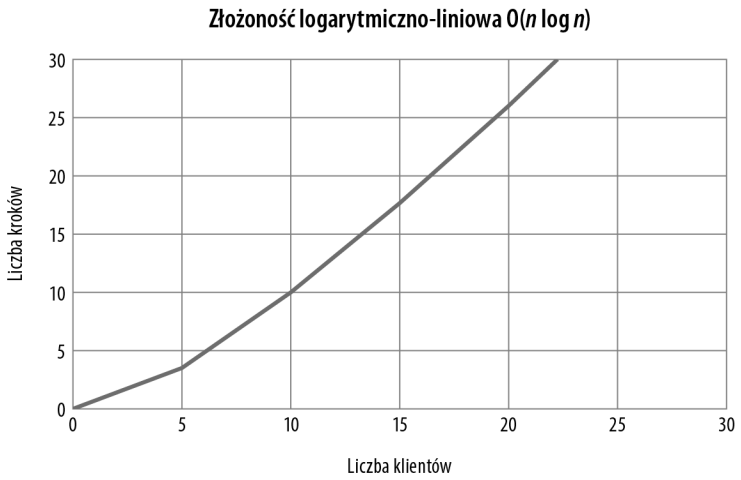
Rysunek 1.3. Złożoność liniowa

Czas logarytmiczno-liniowy

Złożoność algorytmu o **logarytmiczno-liniowym czasie** działania jest iloczynem złożoności logarytmicznej i liniowej. Przykładowo algorytm o takiej złożoności może *n* razy wykonywać operację o złożoności O(log *n*). W notacji dużego O złożoność logarytmiczno-liniową zapisujemy jako O(*n* log *n*). Algorytmy cechujące się tą złożonością często dzielą zbiór danych na mniejsze części i każdą z nich przetwarzają osobno. Wiele bardziej wydajnych algorytmów sortowania, które poznasz dalej w tej książce, takich jak sortowanie przez scalanie (ang. *merge sort*), ma właśnie tę złożoność.

Wykres złożoności logarytmiczno-liniowej przedstawiłem na rysunku 1.4.

Jak widać, złożoność logarytmiczno-liniowa nie jest równie wydajna jak złożoność liniowa. Z drugiej strony, ta złożoność rośnie znacznie wolniej od złożoności kwadratowej, którą przedstawiłem w następnym podrozdziale.



Rysunek 1.4. Złożoność logarytmiczno-liniowa

Czas kwadratowy

Po logarytmiczno-liniowym czasie działania kolejną pod względem efektywności klasą złożoności jest czas kwadratowy. Algorytm działa w **czasie kwadratowym**, jeśli jego wydajność jest wprost proporcjonalna do kwadratu wielkości problemu. W notacji dużego O tę złożoność zapisujemy jako O(*n***2).

Poniżej przedstawiłem przykład algorytmu o złożoności kwadratowej:

```
numbers = [1, 2, 3, 4, 5]
for i in numbers:
    for j in numbers:
        x = i * j
        print(x)
```

Powyższy algorytm mnoży każdą liczbę zapisaną na liście przez każdą z liczb z tej listy i zapisuje wynik w zmiennej, którą następnie wyświetla.

W tym przypadku wartość *n* odpowiada liczbie elementów listy `numbers`. Równanie na złożoność czasową tego algorytmu będzie mieć następującą postać:

```
f(n) = 1 + n * n * (1 + 1)
```

Czynnik `(1 + 1)` w tym wzorze reprezentuje dwie operacje, czyli mnożenie oraz wyświetlenie. Dwie zagnieżdżone pętle `for` sprawiają, że te dwie czynności (mnożenie i wyświetlenie) zostaną powtórzone *n* * *n* razy. Powyższe równanie możemy uprościć w następujący sposób:

```
f(n) = 1 + (1 + 1) * n**2
```

a następnie do postaci:

```
f(n) = 1 + 2 * n**2
```

Jak łatwo się domyślić, w tym równaniu dominuje czynnik *n***2, dlatego w notacji dużego O równanie przyjmie następującą postać:

```
O(n) = n**2
```

Złożoność kwadratowa przedstawiona na wykresie rośnie bardzo szybko wraz ze zwiększaniem się wielkości problemu, co pokazałem na rysunku 1.5.

Rysunek 1.5. Złożoność kwadratowa

Z reguły jeśli algorytm zawiera dwie zagnieżdżone pętle od 1 do *n* (lub od 0 do *n* – 1), będzie on miał złożoność czasową co najmniej rzędu O(*n***2). Wiele algorytmów sortowania, takich jak sortowanie przez wstawianie lub sortowanie bąbelkowe (które poznasz dalej w tej książce), ma właśnie taką złożoność.

Czas sześcienny

Po złożoności kwadratowej kolejną klasą jest złożoność sześcienna. Algorytm działa w **czasie sześciennym**, jeśli jego wydajność jest bezpośrednio proporcjonalna do wielkości problemu podniesionej do potęgi 3. W notacji dużego O tę złożoność zapisujemy jako O(*n***3). Algorytm o złożoności sześciennej jest podobny do algorytmu o złożoności kwadratowej, z tym że *n* rośnie do potęgi trzeciej, a nie drugiej.

Poniżej przedstawiłem przykład algorytmu o złożoności sześciennej:

```
numbers = [1, 2, 3, 4, 5]
for i in numbers:
    for j in numbers:
        for h in numbers:
            x = i + j + h
            print(x)
```

Równanie złożoności czasowej dla tego algorytmu ma następującą postać:

```
f(n) = 1 + n * n * n * (1 + 1)
```

którą można uprościć w poniższy sposób:

```
f(n) = 1 + 2 * n**3
```

Analogicznie jak w przypadku algorytmu o złożoności liniowej, w tym równaniu kluczowe znaczenie ma czynnik *n***3, który rośnie tak szybko, że wszystkie pozostałe, nawet *n***2, przestają mieć znaczenie. A zatem w notacji dużego O złożoność sześcienna jest zapisywana w następujący sposób:

```
O(n) = n**3
```

Jak już zaznaczyłem wcześniej, dwie zagnieżdżone pętle są oznaką złożoności kwadratowej, z kolei trzy zagnieżdżone pętle od 0 do *n* oznaczają, że algorytm ma złożoność sześcienną. Taką złożoność bardzo często mają algorytmy związane z nauką o danych oraz statystyką.

Zarówno złożoność kwadratowa, jak i sześcienna są specjalnymi przypadkami większej rodziny złożoności wielomianowych. Algorytm o **wielomianowym czasie działania** można ogólnie zapisać w notacji dużego O jako O(*n****a*); gdzie *a* o wartości 2 będzie oznaczać złożoność kwadratową, natomiast *a* o wartości 3 — złożoność sześcienną. Podczas projektowania algorytmów zazwyczaj będziemy chcieli unikać złożoności wielomianowej, o ile to tylko możliwe, gdyż dla dużych wartości *n* takie algorytmy mogą działać bardzo wolno. Czasami jednak nie ma możliwości uniknięcia takiej złożoności; w takich sytuacjach pewnym pocieszeniem może być świadomość, że wcale nie jest to najgorsza ze złożoności.

Czas wykładniczy

Niechlubne miano najgorszej możliwej złożoności czasowej przypada złożoności wykładniczej. Algorytm o **wykładniczym czasie działania** ma wydajność proporcjonalną do pewnej stałej podniesionej do potęgi równej wielkości problemu. Innymi słowy algorytm o czasowej złożoności wykładniczej wymaga wykonania *c* do potęgi *n* kroków. W notacji dużego O złożoność wykładniczą zapisujemy jako O(*c****n*), gdzie *c* jest pewną stałą. Konkretna wartość tej stałej nie ma znaczenia — kluczowe znaczenie ma *n* w wykładniku potęgi.

Na szczęście algorytmów o złożoności wykładniczej nie spotyka się zbyt często. Jednym z przykładów takiego algorytmu może być program starający się odgadnąć hasło o długości *n* składające się wyłącznie z cyfr za pomocą metody sprawdzania wszystkich możliwych kombinacji. Złożoność takiego algorytmu wynosi O(10**n*).

Poniżej przedstawiłem przykład algorytmu zgadującego hasło i mającego złożoność O(10**n*):

```
pin = 931
n = len(str(pin)) # Funkcja str zamienia liczbę na łańcuch, a str zwraca jego długość
for i in range(10**n):
    if i == pin:
        print(i)
```

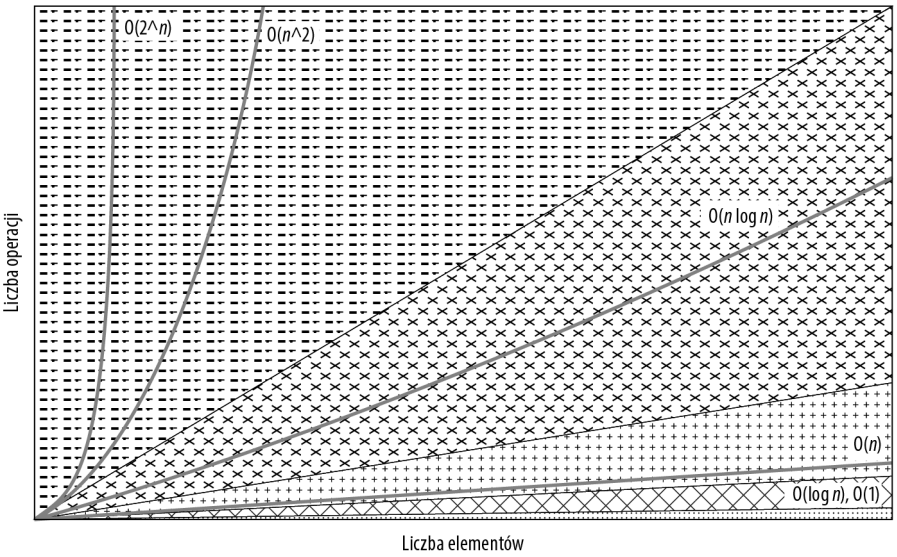
Liczba kroków, które należy wykonać w celu zakończenia tego algorytmu, rośnie niezwykle szybko wraz ze wzrostem wartości *n*. Dla *n* równego 1 algorytm wykonuje 10 kroków. Dla *n* równego 2 wykonywanych jest 200 kroków, a dla *n* równego 3 — 1000. Jak widać, na samym początku algorytmy o złożoności wykładniczej nie sprawiają wrażenia, jakby liczba wykonywanych przez nie operacji rosła błyskawicznie. Jednak w końcu szybkość tego wzrostu staje się ogromna. Odgadnięcie hasła składającego się z 8 cyfr wymaga wykonania 100 milionów kroków, a hasła składającego się z 10 cyfr — 10 miliardów kroków. To właśnie ze względu na tę złożoność wykładniczą tak ważne jest, by używane przez nas hasła były długie. Jeśli ktoś spróbuje odgadnąć Twoje hasło, używając programu takiego jak przedstawiony powyżej, znalezienie hasła o długości czterech cyfr nie będzie wielkim problemem. Jeśli jednak hasło będzie się składać z 20 cyfr, jego złamanie będzie niemożliwe, gdyż potrwałoby dłużej niż czas życia człowieka.

Przedstawiony sposób odgadywania hasła jest przykładem **algorytmu brutalnej siły**. Działanie algorytmów tego typu bazuje na sprawdzeniu każdej dostępnej możliwości. Algorytmy brutalnej siły zazwyczaj nie są efektywne i należy ich unikać.

Na rysunku 1.6 przedstawiłem porównanie wszystkich przedstawionych wcześniej złożoności algorytmów.

Porównanie złożoności optymistycznej i pesymistycznej

Wydajność algorytmów może się zmieniać w zależności od wielu różnych czynników, takich jak typ danych, na których algorytm operuje. Z tego względu, analizując wydajność algorytmów, musimy porównać złożoność uzyskiwaną przez algorytm w najlepszym przypadku, w przypadku najgorszym oraz średnim. **Złożoność optymistyczna** określa, jak algorytm działa, kiedy przekazujemy do niego optymalne dane wejściowe. Z kolei **złożoność pesymistyczna** definiuje działanie algorytmu w możliwie najgorszej sytuacji. I w końcu **złożoność średnia** precyzuje przeciętną wydajność algorytmu.



Rysunek 1.6. Wykres złożoności w notacji dużego O

Jeśli na przykład musimy kolejno przeszukiwać listę, możemy mieć szczęście i znaleźć to, czego szukamy, już po sprawdzeniu pierwszego elementu. To byłaby złożoność optymistyczna. Jeśli jednak poszukiwanego elementu nie będzie na liście, będziemy musieli sprawdzić wszystkie jej elementy. To byłaby złożoność pesymistyczna.

Kiedy musimy przeszukać całą listę element po elemencie sto razy, okaże się, że średnia złożoność wyniesie O(n/2), co w notacji dużego O odpowiada złożoności O(n). Porównując algorytmy, często zaczynamy od określenia ich złożoności średniej. Gdy chcemy wykonać dokładniejsze analizy, możemy także ustalić złożoności optymistyczną i pesymistyczną.

Złożoność pamięciowa

Komputery dysponują skończoną ilością zasobów, takich jak pamięć, dlatego oprócz analizowania złożoności czasowej algorytmów należy także uwzględnić wielkość zużywanych przez nie zasobów. **Złożonością pamięciową** algorytmu nazywamy wielkość pamięci niezbędnej do jego działania, wliczając do niej pewien stały obszar pamięci, obszar przeznaczony na struktury danych oraz obszar tymczasowy. Ten **stały obszar** pamięci to pamięć zajmowana przez program. **Obszar struktur danych** to wielkość pamięci, której program potrzebuje do przechowania używanego zbioru danych, na przykład przeszukiwanej listy. Wielkość pamięci używanej przez algorytm do przechowywania danych zależy od ilości danych wejściowych wymaganych przez rozwiązywany problem. **Obszar tymczasowy** to wielkość pamięci potrzebnej do bieżącego działania algorytmu, na przykład potrzebnej algorytmowi do tymczasowego skopiowania listy w celu przeniesienia danych.

Poznane wcześniej pojęcia związane ze złożonością czasową możemy zastosować także w odniesieniu do złożoności pamięciowej. Przykładowo silnię liczby *n* (iloczyn wszystkich dodatnich liczb całkowitych mniejszych lub równych *n*) można wyliczyć, używając algorytmu o stałej złożoności pamięciowej O(1):

```
x = 1
n = 5
for i in range(1, n + 1):
    x = x * i
```

Złożoność pamięciowa tego algorytmu jest stała, gdyż wielkość używanej przez niego pamięci nie zmienia się wraz ze zwiększaniem wartości *n*. Gdybyś zdecydował się zapisać wszystkie liczby używane do wyliczenia silni na liście, taki algorytm miałby liniową złożoność pamięciową O(*n*):

```
x = 1
n = 5
a_list = []
for i in range(1, n + 1):
    a_list.append(x)
    x = x * i
```

W tym przypadku złożoność pamięciowa wynosi O(*n*), gdyż ilość pamięci używanej przez algorytm rośnie w tym samym tempie co *n*.

Podobnie jak w przypadku złożoności czasowej, także dopuszczalny dla algorytmu poziom złożoności pamięciowej będzie zależeć od konkretnej sytuacji. Jednak ogólnie rzecz biorąc, im mniej pamięci używa algorytm, tym lepiej.

Dlaczego to ma znaczenie

Jako informatyk musisz znać i rozumieć kolejne rzędy wielkości złożoności, by optymalizować tworzone algorytmy. Podczas optymalizowania algorytmu należy skoncentrować się na zmniejszeniu rzędu wielkości jego złożoności, a nie na poprawianiu jego innych cech. W ramach przykładu załóżmy, że dysponujemy algorytmem o złożoności O(*n***2), w którego kodzie używamy dwóch pętli `for`. W takim przypadku od optymalizowania kodu wykonywanego wewnątrz pętli znacznie ważniejsze będzie ustalenie, czy można przepisać algorytm w taki sposób, by nie używał dwóch zagnieżdżonych pętli `for`, gdyż to pozwoliłoby zmniejszyć rząd wielkości jego złożoności.

Jeśli będziemy w stanie rozwiązać problem, używając algorytmu wykonującego dwie niepowiązane ze sobą pętle `for`, to taki algorytm będzie mieć złożoność O(*n*), czyli jego wydajność będzie znacznie lepsza. Taka zmiana będzie mieć znacznie większy wpływ na wydajność algorytmu niż jakiekolwiek usprawnienia wydajności, które wprowadzilibyśmy w algorytmie o złożoności O(*n***2). Niemniej jednak bardzo ważne jest, by uwzględniać scenariusze, w których złożoność algorytmu jest najlepsza i najgorsza. Może się zdarzyć, że będziemy używać algorytmu o złożoności

O(*n***2), lecz w najlepszym przypadku złożoność ta wynosi O(*n*), a dane, których używamy, odpowiadają temu najlepszemu przypadkowi. W takiej sytuacji wybór tego algorytmu może być dobrym rozwiązaniem.

Nasze decyzje dotyczące algorytmów mogą mieć ogromne konsekwencje w praktyce. Załóżmy na przykład, że pracujesz nad aplikacją internetową i jesteś odpowiedzialny za napisanie algorytmu służącego do obsługi żądań nadsyłanych przez klientów. W takim przypadku to, czy napiszemy algorytm o stałej złożoności, czy o złożoności kwadratowej, będzie się przekładać na różnicę pomiędzy wyświetlaniem strony w ciągu mniej niż sekundy (dzięki czemu użytkownicy będą zadowoleni) bądź wyświetlaniem jej przez ponad minutę (co pewnie doprowadzi do utraty klientów jeszcze zanim żądanie zostanie obsłużone).

Słownictwo

Algorytm: sekwencja kroków wykonywanych w celu rozwiązania problemu.

Czas działania: długość okresu czasu, jaki zajmuje komputerowi wykonanie algorytmu napisanego w jakimś języku programowania, na przykład w Pythonie.

Wielkość problemu: zmienna *n* w równaniu opisującym liczbę kroków niezbędnych do rozwiązania problemu.

Notacja dużego O: zapis matematyczny opisujący zmiany wymagań czasowych lub pamięciowych algorytmu wraz ze zmianami wartości *n*.

Rząd wielkości: jedna z klas systemu klasyfikacji; przy czym każda z tych klas jest wiele razy większa lub mniejsza od poprzedniej.

Złożoność czasowa: maksymalna liczba kroków wykonywanych przez algorytm dla zwiększającej się wartości *n*.

Czas stały: algorytm ma stały czas działania, jeśli liczba wykonywanych przez niego kroków jest taka sama niezależnie od wielkości problemu.

Czas logarytmiczny: algorytm ma logarytmiczny czas działania, jeśli czas jego wykonania jest proporcjonalny do logarytmu wielkości danych wejściowych.

Czas liniowy: algorytm ma liniowy czas działania, jeśli czas jego wykonania jest proporcjonalny do wielkości danych wejściowych.

Czas logarytmiczno-liniowy: algorytm ma logarytmiczno-liniowy czas działania, jeśli czas jego wykonania jest proporcjonalny do iloczynu złożoności liniowej i logarytmicznej.

Czas kwadratowy: algorytm ma kwadratowy czas działania, jeśli czas jego wykonania jest bezpośrednio proporcjonalny do kwadratu wielkości problemu.

Czas sześcienny: algorytm ma sześcienny czas działania, jeśli czas jego wykonania jest bezpośrednio proporcjonalny do sześcianu wielkości problemu.

Czas wielomianowy: algorytm ma wielomianowy czas działania, gdy jego złożoność czasową można zapisać w notacji dużego O jako O(*n****a*); gdy *a* = 2, mamy do czynienia ze złożonością kwadratową, a gdy *a* = 3 — ze złożonością sześcienną.

Czas wykładniczy: algorytm ma wykładniczy czas działania, jeśli czas jego wykonania odpowiada pewnej stałej podniesionej do potęgi, której wykładnikiem jest wielkość problemu.

Algorytm brutalnej siły: typ algorytmu, który rozwiązuje problem, sprawdzając wszystkie dostępne możliwości.

Złożoność optymistyczna: sposób działania algorytmu w przypadku dostarczenia optymalnych danych wejściowych.

Złożoność pesymistyczna: sposób działania algorytmu w przypadku dostarczania najgorszych możliwych danych.

Złożoność średnia: przeciętny sposób działania algorytmu.

Złożoność pamięciowa: wielkość obszaru pamięci używanej przez algorytm.

Obszar stały: wielkość obszaru pamięci wymaganego przez program.

Obszar struktur danych: wielkość obszaru pamięci używanego przez program do przechowywania struktur danych.

Obszar tymczasowy: wielkość obszaru pamięci potrzebnego do bieżącego działania algorytmu, na przykład gdy algorytm będzie musiał chwilowo skopiować listę, by przenieść dane.

Wyzwanie

1. Znajdź jakiś program, który napisałeś wcześniej. Przeanalizuj go i zapisz złożoności czasowe różnych zastosowanych w nim algorytmów.

Niektórzy twierdzą, że kiedyś po ukończeniu dobrej szkoły nie było potrzeby, by się dalej kształcić. Dziś każdy informatyk albo programista, który chce sobie zagwarantować dobrą posadę, musi przyjąć postawę *zawsze się uczę*. Jeśli nie masz za sobą studiów informatycznych, ale bardzo chcesz pracować jako inżynier oprogramowania, musisz poznać podstawy informatyki, jakimi są struktury danych i algorytmika. Znajomość tych zagadnień jest niezbędna każdemu programiście!

Ta książka jest kontynuacją bestsellera ***Programista samouk***. Dzięki niej zrozumiesz najważniejsze koncepcje związane z różnymi algorytmami i strukturami danych, a to z kolei pozwoli Ci na profesjonalne podejście do tworzenia kodu. Przystępnie opisano tu różne algorytmy, w tym wyszukiwania liniowego i binarnego, a także służące do pracy na ciągach znaków i do wykonywania obliczeń. Zaprezentowano również najważniejsze struktury danych, w tym tablice, listy połączone, tablice mieszające, drzewa i wiele innych. Poszczególne zagadnienia zostały pokazane od strony praktycznej, co sprawi, że bez trudu zastosujesz zdobytą wiedzę w codziennej pracy. W efekcie lektury poszerzysz swoje umiejętności, a jeśli zechcesz, przygotujesz się do kariery skutecznego programisty — nawet jeśli nie masz dyplomu inżyniera!

- algorytm i związane z nim pojęcia
- rekurencja i jej zastosowanie
- działanie najważniejszych algorytmów
- listy, stosy i kolejki
- drzewa binarne, kopce binarne i grafy

CORY ALTHOFF jest programistą samoukiem. Jego pierwsza książka, ***Programista samouk***, zdobyła ogromne uznanie i zainspirowała tysiące osób do samodzielnej nauki programowania. Pracował dla eBaya i innych firm w Dolinie Krzemowej. Biegle posługuje się Pythonem, Javą, JavaScriptem i kilkoma innymi językami programowania. Mieszka z rodziną w Kalifornii.

Możesz zostać profesjonalnym informatykiem!

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-9194-9
9 788328 391949
Cena: 59,00 zł