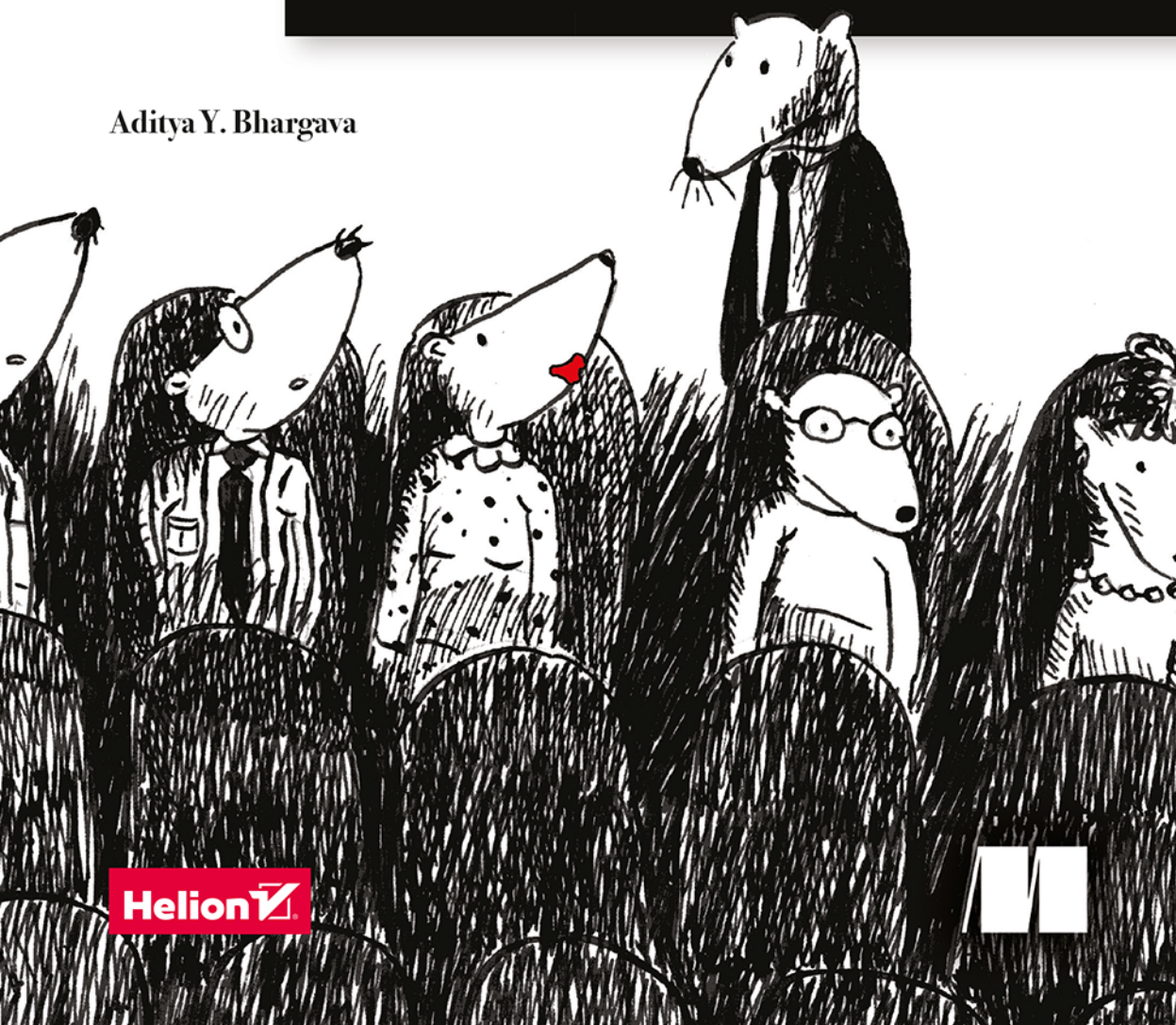
algorytmy

Ilustrowany przewodnik

Aditya Y. Bhargava



Helion

Tytuł oryginału: Grokking Algorithms: An illustrated guide
for programmers and other curious people

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-9874-0

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/algipv.zip

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

Printed in Poland.

Spis treści

Wprowadzenie do algorytmów 1

W tym rozdziale:

- zdobędziesz podstawowe wiadomości potrzebne do zrozumienia dalszej części książki,
- napiszesz swój pierwszy algorytm sortowania (wyszukiwanie binarne),
- nauczysz się opisywać czas działania algorytmów (notacja dużego O),
- poznasz powszechnie stosowaną technikę projektowania algorytmów (rekurencję).

Wprowadzenie

Algorytm to zestaw instrukcji opisujących, jak wykonać pewne zadanie. Algorytmem można wprawdzie nazwać każdy fragment kodu źródłowego, ale w tej książce przedstawiam nieco ciekawszą perspektywę. Algorytmy opisane tutaj wybierałem pod kątem ich szybkości lub rodzaju rozwiązanych problemów albo obu tych cech. Oto kilka ważnych informacji.

- W rozdziale 1. opisuję wyszukiwanie binarne i pokazuję, jak algorytm może przyspieszyć działanie programu. W jednym z przykładów liczba czynności, jakie należy wykonać, została zredukowana z czterech miliardów do zaledwie 32!

- Urządzenia GPS najkrótszą drogę do celu obliczają przy użyciu algorytmów grafów (opisanych w rozdziałach 6., 7. i 8.).
- Przy użyciu technik programowania dynamicznego (opisanych w rozdziale 9.) można napisać algorytm sztucznej inteligencji grający w szachy.

W każdym przypadku opisuję algorytm i podaję przykład jego zastosowania. Następnie omawiam czas działania, używając notacji wielkiego O. Na koniec podpowiadam, jakie jeszcze inne zadania można rozwiązać za pomocą danego algorytmu.

Czego nauczysz się o wydajności

Dobra wiadomość jest taka, że w Twoim ulubionym języku programowania prawdopodobnie dostępna jest implementacja każdego algorytmu opisanego w tej książce, zatem nie musisz wszystkiego pisać samodzielnie! Jednak cała ta pomoc jest bezużyteczna dla kogoś, kto nie rozumie, jakie są ich mocne i słabe strony. W książce tej nauczysz się porównywać różne algorytmy. Czy w danym przypadku lepiej wybrać sortowanie przez scalanie, czy sortowanie szybkie? Czy lepiej użyć tablicy, czy listy? Sam wybór struktury danych może być szalenie ważny dla wydajności.

Czego nauczysz się o rozwiązywaniu problemów

Poznasz techniki rozwiązywania zadań, które do tej pory mogły być poza Twoim zasięgiem. Oto przykłady.

- Jeśli lubisz tworzyć gry wideo, możesz napisać system SI (sztucznej inteligencji), który z wykorzystaniem algorytmów grafowych będzie śledził użytkownika na planszy.
- Dowiesz się, jak utworzyć system rekomendacji za pomocą algorytmu k najbliższych sąsiadów.
- Niektórych problemów nie da się rozwiązać w zadowalającym czasie! W części poświęconej problemom NP-zupełnym nauczę Cię rozpoznawać te problemy i wybierać algorytmy, które pozwolą uzyskać przybliżone rozwiązanie.

Mówiąc bardziej ogólnie, po przestudiowaniu tej książki będziesz znać niektóre z najszerzej wykorzystywanych algorytmów programistycznych. Następnie możesz bardziej szczegółowo zgłębić wybraną grupę algorytmów, np. SI, baz danych itd. Ewentualnie możesz w pracy zająć się poważniejszymi zadaniami.

Co musisz wiedzieć

Przystępując do lektury tej książki, musisz znać podstawy algebry. Weźmy np. funkcję *f(x) = 2x*. Wiesz, ile wynosi *f(5)*? Jeśli Twoja odpowiedź brzmiała *10*, tzn. że wiesz wystarczająco dużo.

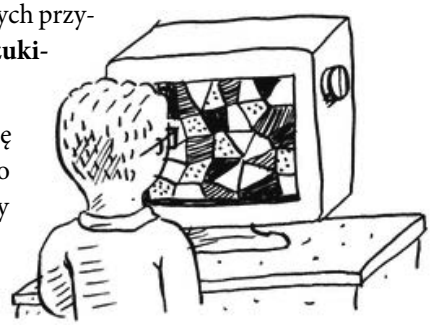
Ponadto studiowanie tego rozdziału (i całej książki) będzie łatwiejsze dla tych, którzy znają przynajmniej jeden język programowania. Wszystkie przykłady są napisane w Pythonie. Jeśli więc nie znasz jeszcze żadnego języka programowania i chcesz się jakiegoś nauczyć, wybierz właśnie Python, który jest świetnym wyborem dla początkujących. Jeżeli znasz inny język, np. Ruby, to też sobie poradzisz.

Wyszukiwanie binarne

Wyobraź sobie, że szukasz w książce telefonicznej (co za staroświecka metoda!) nazwiska na literę *K*. Choć możesz zacząć szukanie od początku i przewracać kartki, aż dojdziesz do litery *K*, prawdopodobnie tego nie zrobisz, tylko od razu otworzysz książkę na środkowej stronie, ponieważ wiesz, że litera *K* znajduje się mniej więcej w środku alfabetu.

Albo pomyśl sobie, że szukasz w słowniku słowa na literę O. Tym razem również od razu otworzysz książkę na dalszej stronie.

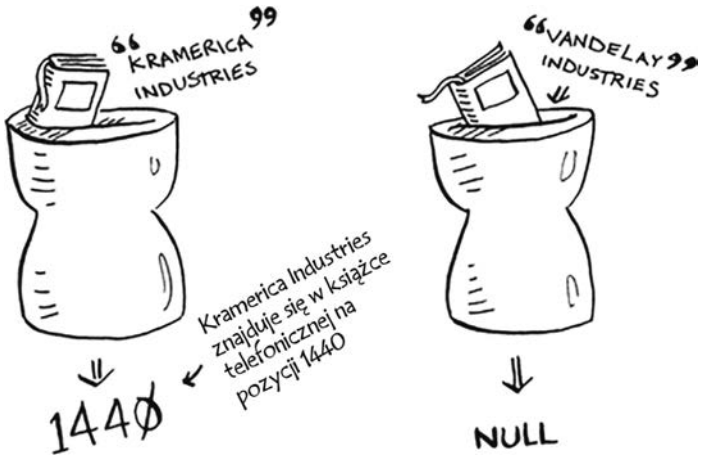
A teraz wyobraź sobie, że logujesz się na Facebooku. Gdy to robisz, Facebook musi sprawdzić, czy na pewno masz konto w tym portalu. W związku z tym musi poszukać Twojej nazwy użytkownika w swojej bazie danych. Powiedzmy, że Twoja nazwa użytkownika to *karlmageddon*. Serwis może zacząć szukanie tej nazwy od litery A, ale bardziej sensowne wydaje się rozpoczęcie gdzieś w środku.



To jest problem wyszukiwania i we wszystkich wymienionych przypadkach należy użyć tego samego algorytmu, czyli **wyszukiwania binarnego**.

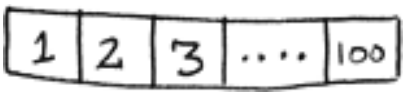
Wyszukiwanie binarne to algorytm. Na wejście podaje się posortowaną listę elementów (dalej wyjaśniam, dlaczego lista musi być posortowana). Jeśli lista ta zawiera szukany element, algorytm zwraca informację o jego położeniu. W przeciwnym przypadku zwracana jest wartość `null`.

Oto przykład.



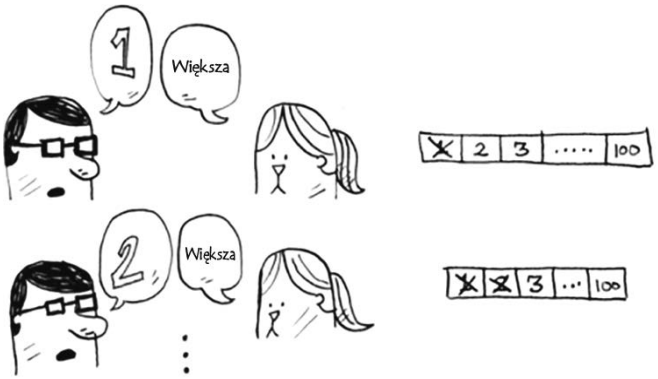
Szukanie firm w książce telefonicznej za pomocą wyszukiwania binarnego

Oto przykład działania algorytmu wyszukiwania binarnego. Myślę o liczbie z przedziału od 1 do 100.



Musisz zgadnąć, o jakiej liczbie myślę w jak najmniejszej liczbie prób. Za każdym razem, gdy wymienisz jakąś liczbę, powiem Ci, czy jest większa, mniejsza, czy taka sama jak moja.

Powiedzmy, że zaczynasz zgadywanie w taki sposób: 1, 2, 3, 4… Tak by to wyglądało.



Złe podejście do zgadywania liczb

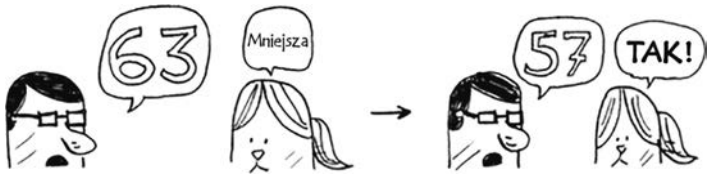
To jest **wyszukiwanie proste** (choć chyba lepsza byłaby nazwa **wyszukiwanie głupie**). Przy każdej próbie eliminuje się tylko jedną liczbę. Gdybym pomyślał o liczbie 99, do jej odgadnięcia trzeba by aż 99 prób!

Lepszy sposób wyszukiwania

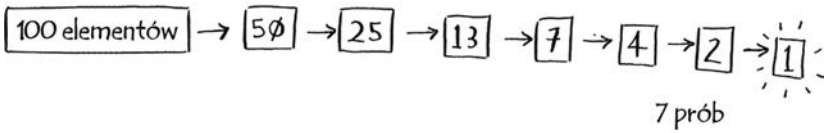
Oto lepsza metoda. Zaczniemy od 50.

Większa, ale właśnie wyeliminowałeś *połowę* liczb! Wiesz już, że moja liczba jest większa od wszystkich liczb z przedziału od 1 do 50. Następny strzał to 75.

Tym razem moja liczba jest mniejsza, ale znów pozbyłeś się połowy zbioru! *Wyszukiwanie binarne polega na zgadywaniu środkowej liczby w celu wyeliminowania w każdej próbie połowy pozostałych liczb.* Następny strzał to liczba 63 (w połowie drogi między 50 i 75).



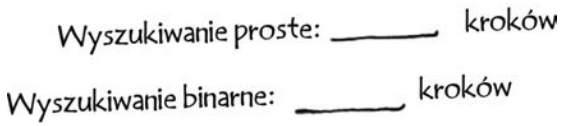
To jest właśnie wyszukiwanie binarne. Gratuluję poznania pierwszego algorytmu! Oto ile liczb można odrzucić w każdej próbie.



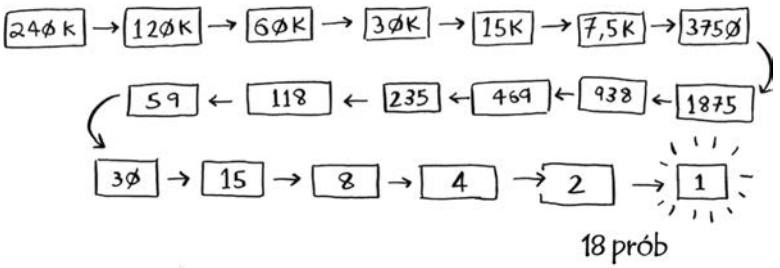
Eliminacja połowy liczb w każdej próbie wyszukiwania binarnego

Kiedy w każdej próbie będzie eliminowana tak duża część przeszukiwanego zbioru, to bez względu na to, o jakiej liczbie bym nie pomyślał, zawsze możesz ją odgadnąć najwyżej w siedmiu próbach!

Powiedzmy, że szukamy hasła w słowniku, który zawiera 240 000 słów. Jak myślisz, ile razy trzeba będzie zgadywać *w najgorszym przypadku* przy zastosowaniu każdego z opisanych algorytmów?



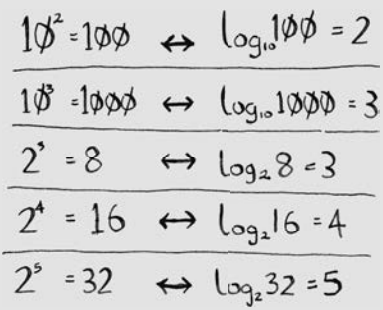
Gdyby szukane hasło było ostatnim słowem w słowniku, przy zastosowaniu algorytmu wyszukiwania prostego trzeba by wykonać 240 000 prób. Natomiast w wyszukiwaniu binarnym w każdej próbie eliminuje się połowę przeszukiwanego zbioru słów, aż pozostanie tylko jedno.



A zatem wyszukiwanie binarne wymaga wykonania 18 prób — spora różnica! Ogólnie rzecz biorąc, dla każdej listy *n* elementów wyszukiwanie binarne wymaga w najgorszym przypadku $\log_2 n$ prób, podczas gdy wyszukiwanie proste wymaga *n* prób.

Logarytmy

Możesz nie pamiętać, czym są logarytmy, ale na pewno wiesz, co to jest potęgowanie. Zapis $log_{10}100$ jest jak pytanie: „Ile dziesiątek trzeba przez siebie pomnożyć, aby otrzymać 100?". Odpowiedź wynosi *2*: $10 \cdot 10 = 100$. W związku z tym $log_{10}100$ wynosi *2*. Innymi słowy, logarytmy można interpretować jako odwrotność potęgowania.



Logarytmy są odwrotnością potęgowania

Kiedy w książce tej jest mowa o czasie wykonywania logarytmu wyrażonym w notacji dużego O (objaśnionej nieco dalej), *log* zawsze oznacza log_2. Przy szukaniu elementu za pomocą algorytmu szukania prostego w najgorszym przypadku może być konieczne sprawdzenie wszystkich elementów. Jeśli więc lista zawiera 8 elementów, maksymalnie trzeba by obejrzeć 8 elementów. W wyszukiwaniu binarnym w najgorszym przypadku należy sprawdzić *log n* elementów. Dla listy 8 elementów mielibyśmy zatem wynik $log\ 8 == 3$, ponieważ $2^3 == 8$. Innymi słowy, w przypadku listy 8 elementów należy sprawdzić maksymalnie 3 liczby. Gdyby lista zawierała 1024 elementy, to $log\ 1024 == 10$, ponieważ $2^{10} == 1024$. Aby więc znaleźć liczbę na liście 1024 liczb, w najgorszym przypadku należy wykonać 10 prób.

Uwaga

W książce tej bardzo często wyrażam czas działania przy użyciu logarytmów, więc koniecznie dokładnie zrozum to pojęcie. Jeśli sprawia Ci ono trudności, możesz obejrzeć świetny film na ten temat w Khan Academy (*khanacademy.org*).

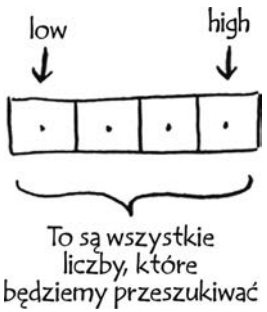
Uwaga

Wyszukiwanie binarne można stosować tylko wtedy, kiedy lista jest posortowana. Książka telefoniczna zawiera np. listę nazwisk posortowaną w kolejności alfabetycznej, więc można w niej szukać nazwisk za pomocą wyszukiwania binarnego. A co by się stało, gdyby nazwiska nie były posortowane?

Zobaczmy, jak napisać algorytm wyszukiwania binarnego w Pythonie. W przedstawionych tu przykładach używam tablic. Nie przejmuj się, jeśli nie wiesz, czym są tablice, ponieważ ich szczegółowy opis zamieściłem w następnym rozdziale. Na razie wystarczy tylko zapamiętać, że w pamięci komputera szereg elementów można zapisać w kolejnych komórkach i taki szereg nazywa się tablicą. Komórki mają numery zaczynające się od 0: pierwsza znajduje się na pozycji nr 0, druga na pozycji nr 1, trzecia na pozycji nr 2 itd.

Funkcja `binary_search` pobiera posortowaną tablicę i element. Jeśli element ten występuje w tablicy, funkcja zwraca numer jego pozycji. Będziesz śledzić, którą część tablicy trzeba przeszukać. Na początek zdefiniujemy naszą tablicę.

```
low = 0
high = len(list) - 1
```



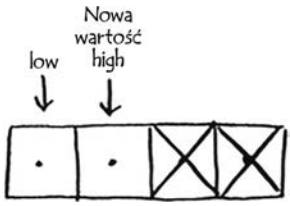
W każdej próbie sprawdzamy element środkowy.

```
mid = (low + high) / 2
guess = list[mid]
```

Python automatycznie zaokrągli wartość mid w dół, jeśli wynik low + high będzie nieparzysty.

Jeśli wartość `guess` jest za mała, odpowiednio zmieniamy `low`.

```
if guess < item:
low = mid + 1
```



A jeśli wartość `guess` jest za duża, to zmieniamy wartość `high`. Oto kompletny kod.

```
def binary_search(list, item):
  low = 0
  high = len(list)-1

  while low <= high:
    mid = (low + high)
    guess = list[mid]
    if guess == item:
      return mid
    if guess > item:
      high = mid - 1
    else:
      low = mid + 1
  return None

my_list = [1, 3, 5, 7, 9]

print binary_search(my_list, 3) # => 1
print binary_search(my_list, -1) # => None
```

Za pomocą low i high kontrolujemy, która część listy jest przeszukiwana.

Dopóki obszar poszukiwań nie został zawężony do jednego elementu…

…wybieramy środkowy element.

Znaleziono element.

Strzelono za dużą liczbę.

Strzelono za małą liczbę.

Nie ma takiego elementu.

Przetestujmy to!

Pamiętaj, że numerowanie w listach zaczyna się od 0. Druga komórka ma indeks 1.

Wartość None w Pythonie oznacza nic, czyli wskazuje, że elementu nie znaleziono.

ĆWICZENIA

1.1. Powiedzmy, że mamy posortowaną listę 128 nazwisk i chcemy ją przeszukać za pomocą algorytmu wyszukiwania binarnego. Ile maksymalnie prób zgadywania będzie trzeba wykonać?

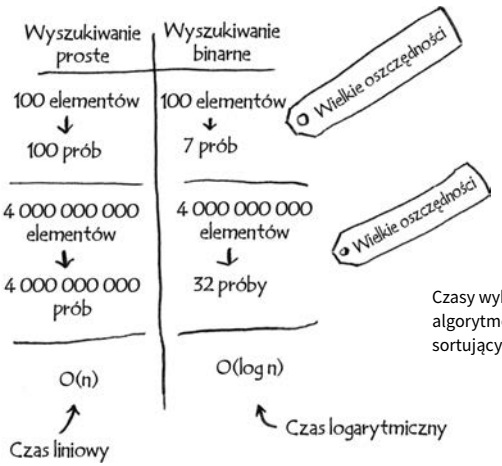
1.2. Ile maksymalnie prób zgadywania trzeba będzie wykonać, jeśli podwoi się liczbę elementów w liście?

Czas wykonywania

W opisie każdego algorytmu zamieszczam też informację na temat czasu wykonywania. Generalnie zawsze należy wybierać najefektywniejszy algorytm, zapewniający optymalne wykorzystanie czasu lub miejsca w pamięci.

Wracamy do wyszukiwania binarnego. Ile czasu można oszczędzić przy jego zastosowaniu? Pierwsza metoda polegała na sprawdzaniu każdej liczby po kolei. Jeśli lista zawiera 100 liczb, znalezienie na niej elementu może wymagać wykonania maksymalnie 100 operacji. Gdyby lista ta zawierała cztery miliardy elementów, trzeba by wykonać cztery miliardy prób. A zatem maksymalna liczba strzałów jest równa liczbie elementów w liście. Nazywa się to **czasem liniowym**.

Z wyszukiwaniem binarnym jest inaczej. Jeśli lista zawiera 100 elementów, to aby znaleźć w niej element, maksymalnie należy wykonać 7 prób. Gdyby lista zawierała cztery miliardy elementów, maksymalna liczba strzałów wynosiłaby 32. Nieźle, prawda? Wyszukiwanie binarne charakteryzuje się **logarytmicznym czasem wykonywania**. Tabela obok przedstawia zwięzłe podsumowanie naszych ustaleń.



Czasy wykonywania algorytmów sortujących

Notacja dużego O

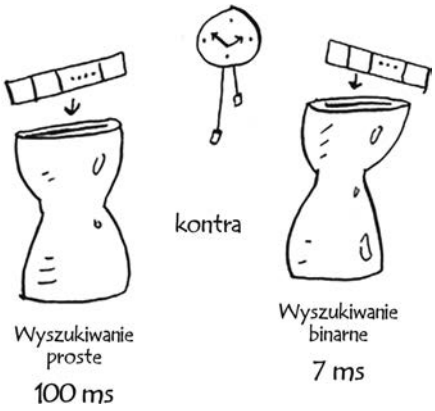
Notacja dużego O to specjalny sposób opisu szybkości działania algorytmów. A kogo to obchodzi? Otóż musisz wiedzieć, że bardzo często będziesz używać algorytmów opracowanych przez innych programistów i wówczas z przyjemnością zerkniesz na informację, jak szybko lub wolno one działają. W tym podrozdziale wyjaśniam zatem, czym jest notacja dużego O oraz przedstawiam przy jej użyciu niektóre najbardziej typowe czasy wykonania różnych algorytmów.

Czas wykonywania algorytmów rośnie w różnym tempie

Bartek pisze algorytm wyszukiwania dla agencji NASA. Algorytm ten będzie uruchamiany tuż przed lądowaniem rakiety na Księżycu, a jego zadaniem będzie pomoc w obliczeniu najlepszego miejsca do posadowienia statku.

Jest to doskonały przykład tego, jak czas wykonywania dwóch algorytmów może rosnąć w różnym tempie. Bartek zastanawia się nad wyborem algorytmu wyszukiwania prostego i binarnego. Jego rozwiązanie musi być i szybkie, i dawać poprawne wyniki. Z jednej strony, wyszukiwanie binarne jest szybsze, a Bartek ma tylko *10 sekund* na znalezienie miejsca do lądowania albo rakieta zejdzie z kursu. Z drugiej strony, proste wyszukiwanie jest łatwiejsze do zaimplementowania, więc występuje mniejsze ryzyko popełnienia błędu. A Bartek *bardzo* by nie chciał popełnić błędu w kodzie odpowiadającym za lądowanie rakiety! Aby mieć absolutną pewność o słuszności swojego wyboru, Bartek postanowił przetestować oba algorytmy na liście 100 elementów.

Powiedzmy, że sprawdzenie jednego elementu trwa milisekundę. Algorytm wyszukiwania prostego musi sprawdzić 100 elementów, więc wyszukiwanie zajmie nie więcej niż 100 ms. Algorytm wyszukiwania binarnego musi sprawdzić maksymalnie 7 elementów (ponieważ $\log_2 100$ z grubsza wynosi 7), a więc czas działania nie będzie dłuższy niż 7 ms. Jednak w rzeczywistości lista może mieć nawet miliard elementów. Ile wówczas czasu zajmie wyszukiwanie proste, a ile binarne? Zastanów się dokładnie nad odpowiedzią na to pytanie, zanim przeczytasz następne akapity.



Porównanie czasów wykonywania wyszukiwania prostego i binarnego w liście 100 elementów

Bartek wykonał wyszukiwanie binarne w liście zawierającej miliard elementów i stwierdził, że zajęło mu to 30 ms ($\log_2 1\,000\,000\,000$ wynosi mniej więcej 30). Pomyślał sobie: „Aha, 32 ms! Wyszukiwanie binarne jest około 15 razy szybsze od prostego, ponieważ wyszukiwanie proste w liście 100 elementów zajęło 100 ms, a wyszukiwanie binarne w tej samej liście trwało 7 ms. W związku z tym proste przeszukanie listy miliarda elementów powinno zająć $30 \cdot 15 = 450$ ms, prawda? To znacznie mniej niż graniczne 10 sekund". Na podstawie tych przemyśleń Bartek wybiera wyszukiwanie proste. Czy podjął słuszną decyzję?

Nie. Bartek grubo się myli. I to nawet bardzo grubo. Czas przeszukiwania listy miliarda elementów za pomocą algorytmu wyszukiwania prostego wynosi miliard milisekund, czyli 11 dni! Sęk w tym, że czasy wykonywania algorytmów wyszukiwania prostego i binarnego *nie rosną w tym samym tempie*!

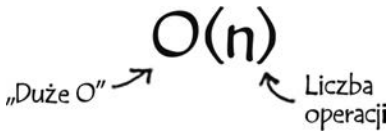
	Wyszukiwanie proste	Wyszukiwanie binarne
100 elementów	100 ms	7 ms
10 000 elementów	10 sekund	14 ms
1 000 000 000 elementów	11 dni	32 ms

Czasy wykonywania rosną w różnym tempie!

Jak widać, zwiększanie liczby elementów do przeszukania w przypadku wyszukiwania binarnego tylko nieznacznie wydłuża czas wykonywania algorytmu. Natomiast w przypadku wyszukiwania prostego czas ten wydłuża się *radykalnie*. Zatem w miarę zwiększania się liczby elementów wyszukiwanie binarne nagle staje się *znacznie* szybsze od wyszukiwania prostego. Bartek myślał, że wyszukiwanie proste jest tylko 15 razy wolniejsze od binarnego, ale był w błędzie. Kiedy lista zawiera miliard elementów, ten drugi algorytm jest aż 33 miliony razy szybszy. Dlatego sama informacja, ile czasu dany algorytm był wykonywany, jest niewystarczająca. Dodatkowo trzeba wiedzieć, jak wydłuża się czas wykonywania wraz ze zwiększaniem liczby elementów. I tu właśnie staje się przydatna notacja dużego O.

Notacja dużego O informuje o tym, jak szybki jest algorytm. Powiedzmy np., że mamy listę o rozmiarze n. Wyszukiwanie proste musi sprawdzić każdy element po kolei, więc wykona n operacji. W notacji dużego O czas wykonywania tego algorytmu wyrazimy tak: $O(n)$. Gdzie są sekundy? Nie ma sekund — ta notacja nie wyraża szybkości w sekundach. *Notacja dużego O pozwala porównać liczbę operacji do wykonania.* Informuje, jak szybko rośnie czas wykonywania algorytmu.

Oto inny przykład. Aby sprawdzić listę o rozmiarze *n*, wyszukiwanie binarne musi wykonać *log n* operacji. Jak zatem wyrazić czas wykonywania tego algorytmu w notacji dużego O? Ano tak: O(*log n*). Ogólnie poszczególne elementy tego zapisu mają następujące znaczenie.



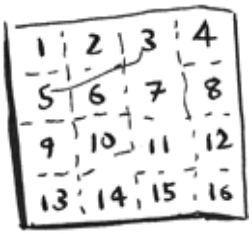
Poszczególne składniki zapisu w notacji dużego O

Z zapisu tego dowiadujemy się, ile operacji wykona dany algorytm. Nazwa notacja dużego O wzięła się z tego, że przed liczbą operacji stawia się literę „O ” (brzmi jak żart, ale to prawda!).

Przyjrzyjmy się kilku przykładom. Spróbuj określić czas wykonywania kilku następnych algorytmów.

Wizualizacja różnych czasów wykonywania

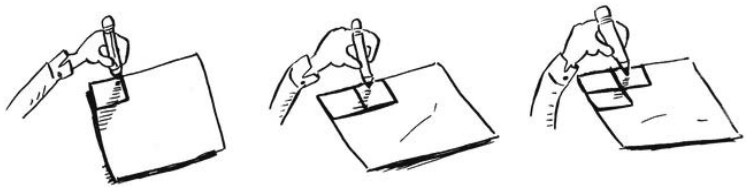
Obok przedstawiam praktyczny przykład metody, którą można zastosować w domu, jeśli ma się kawałek papieru i ołówek. Powiedzmy, że mamy narysować siatkę złożoną z 16 prostokątów.



Znasz dobry algorytm do narysowania takiej siatki?

Algorytm 1.

Jednym ze sposobów jest narysowanie 16 prostokątów, każdego osobno. Przypomnę, że notacja dużego O wyraża liczbę operacji. W tym przykładzie pojedynczą operacją jest narysowanie prostokąta. Do narysowania mamy 16 prostokątów. Ile operacji trzeba wykonać, aby narysować 16 takich prostokątów, po jednym na raz?



Rysowanie siatki po jednym prostokącie na raz

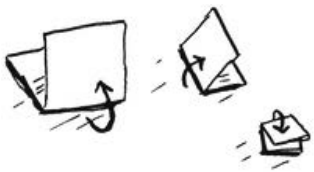
Narysowanie 16 prostokątów wymaga wykonania takiej samej liczby operacji. Jaki jest więc czas wykonywania tego algorytmu?

Algorytm 2.

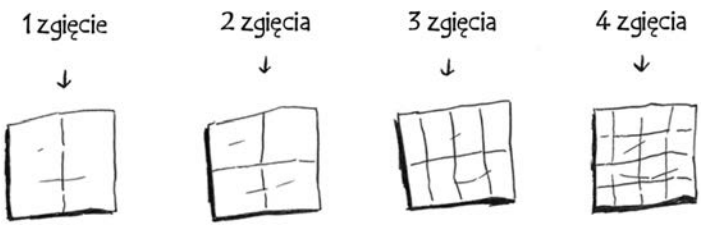
Teraz wypróbuj inny algorytm. Zegnij kartkę.

W tym przypadku zgięcie kartki na pół jest operacją, za pomocą której od razu utworzyliśmy dwa prostokąty!

Zegnij kartkę jeszcze raz, potem jeszcze raz, a potem jeszcze raz.



Teraz rozwiń kartkę i podziwiaj piękną siatkę prostokątów! Każde kolejne zgięcie podwaja liczbę kratek, dzięki czemu 16 prostokątów utworzyliśmy, wykonując tylko cztery operacje!



Rysowanie siatki w czterech krokach

Każde kolejne złożenie kartki powoduje podwojenie liczby kratek, dzięki czemu wystarczyły cztery operacje, aby narysować 16 kratek. Jaki zatem jest czas wykonywania tego algorytmu? Zanim przeczytasz następne akapity, zastanów się, jaki jest czas wykonywania obu opisanych do tej pory algorytmów.

Odpowiedzi: czas wykonywania pierwszego algorytmu to O(n), a drugiego — O($log\ n$).

Notacja dużego O określa czas działania w najgorszym przypadku

Powiedzmy, że za pomocą wyszukiwania prostego szukamy nazwiska w książce telefonicznej. Wiemy, że czas wykonywania tego algorytmu to O(n), co oznacza, że w najgorszym przypadku będziemy musieli przejrzeć wszystkie pozycje w książce. W tym przypadku jednak szukamy nazwiska Adit, które znajduje się na pierwszym miejscu w naszej książce. Nie musieliśmy więc przeszukiwać wszystkich pozycji, ponieważ szukany element znaleźliśmy już na pierwszej pozycji. Czy wykonywanie tego algorytmu zajęło O(n) czasu? Czy też może czas jego wykonywania wyniósł O(1), ponieważ szukany element trafiliśmy już w pierwszej próbie?

Algorytm wyszukiwania prostego nadal ma czas wykonywania O(n). W tym przypadku od razu znaleźliśmy to, czego było nam trzeba. To jest najlepszy przypadek. Jednak notacja dużego O zawsze opisuje *najgorszy przypadek*. Można więc powiedzieć, że w najgorszym przypadku będzie trzeba przejrzeć wszystkie pozycje w książce telefonicznej. To jest czas wykonywania O(n). Jest to rodzaj zapewnienia, że wyszukiwanie proste na pewno nie będzie trwać dłużej niż O(n).

> **Uwaga**
>
> Oprócz czasu wykonywania w najgorszym przypadku, ważnym parametrem jest średni czas działania algorytmu. Szerzej te dwa parametry porównuję w rozdziale 4.

Kilka typowych czasów wykonywania

Poniżej przedstawiam listę pięciu czasów wykonywania wyrażonych w notacji dużego O, które jeszcze nie raz spotkasz w swojej pracy. Pierwszy jest najszybszy.

- O($log\ n$) — zwany też **czasem logarytmicznym**. Przykład to wyszukiwanie binarne.
- O(n) — zwany też **czasem liniowym**. Przykład to wyszukiwanie proste.
- O($n \cdot log\ n$). Przykład to algorytm szybkiego sortowania (opisany w rozdziale 4.).
- O(n^2). Przykład to wolny algorytm sortowania, np. sortowanie przez wybieranie (opisany w rozdziale 2.).

- O(*n!*). Przykład to bardzo wolny algorytm sortowania, np. algorytm rozwiązujący problem komiwojażera (opisany w kolejnym punkcie!).

Powiedzmy, że jeszcze raz chcesz narysować siatkę składającą się z 16 kratek i do wyboru masz jeden z pięciu algorytmów. Jeśli wybierzesz pierwszy z nich, rysowanie siatki zajmie O(*log n*) czasu. Twój komputer jest w stanie wykonać 10 operacji na sekundę. Przy czasie wykonywania O(*log n*) narysowanie 16 kratek wymaga wykonania czterech operacji (log 16 wynosi 4). W związku z tym narysowanie siatki zajmie Ci 0,4 sekundy. A gdyby trzeba było narysować 1024 kratki? Liczba operacji wynosiłaby wówczas log 1024 czyli 10, a więc rysowanie siatki złożonej z 1024 prostokątów zajęłoby sekundę. Te obliczenia dotyczą pierwszego z wymienionych algorytmów.

Drugi algorytm jest wolniejszy, ponieważ czas jego wykonywania wynosi O(*n*). Zatem narysowanie 16 kratek wymaga wykonania 16 operacji, a narysowanie 1024 prostokątów wymaga wykonania 1024 operacji. Ile to czasu w sekundach?

Poniższe wykresy przedstawiają czas rysowania siatki przy użyciu wszystkich opisanych algorytmów, zaczynając od najszybszego.

Liczba kratek	O(log n)	O(n)	O(n log n)	O(n²)	O(n!)
16	0,4 s	1,6 s	6,4 s	25,6 s	66 301 lat
256	0,8 s	25,6 s	3,4 min	1,8 h	$8{,}6 \cdot 10^{505}$ lat
1024	1,0 s	1,7 min	17 min	1,2 d	$5{,}4 \cdot 10^{2638}$ lat

Spotyka się też inne czasy wykonywania, ale te wymienione wyżej występują najczęściej.

Oczywiście to wszystko jest uproszczone, ponieważ w rzeczywistości nie można wykonać takiej prostej konwersji czasu wykonywania wyrażonego w notacji dużego O na liczbę operacji, ale na razie to wystarczy. Do notacji dużego O wrócę jeszcze w rozdziale 4., gdy będziesz już znał trochę więcej algorytmów. Teraz przede wszystkim zapamiętaj następujące informacje.

- Szybkości wykonywania algorytmów nie wyraża się w sekundach, tylko w tempie wzrostu liczby operacji.
- Omawiając szybkość działania algorytmu, podaje się, jak szybko rośnie czas wykonywania wraz ze zwiększaniem rozmiaru zbioru wejściowego.

- Czas wykonywania algorytmów wyraża się za pomocą notacji dużego O.
- Algorytm o czasie wykonywania O(*log n*) jest szybszy niż O(*n*), a im większy zbiór danych do przeszukania, tym większa robi się różnica.

ĆWICZENIA

Przedstaw czas wykonywania w każdym z opisanych poniżej przypadków za pomocą notacji dużego O.

1.3. Dane jest nazwisko i trzeba znaleźć numer telefonu osoby o tym nazwisku w książce telefonicznej.

1.4. Dany jest numer telefonu i trzeba znaleźć nazwisko właściciela tego numeru w książce telefonicznej. (Podpowiedź: musisz przeszukać całą książkę!).

1.5. Chcesz przeczytać numery wszystkich osób w książce telefonicznej.

1.6. Chcesz przeczytać numery tylko osób o nazwiskach na A. (Tu jest pułapka! Trzeba posłużyć się wiedzą przedstawioną bardziej szczegółowo w rozdziale 4. Przeczytaj odpowiedź — może Cię zaskoczyć!).

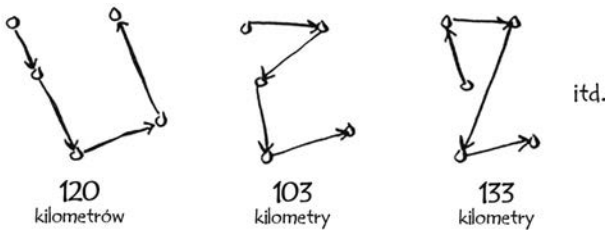
Problem komiwojażera

Czytając poprzedni punkt, niektórzy mogli sobie pomyśleć: „Nie ma szans, abym kiedykolwiek natrafił na algorytm o czasie wykonywania O(*n!*)". Jeśli należysz do tych osób, pozwól, że wyprowadzę Cię z błędu! Oto przykład algorytmu charakteryzującego się bardzo słabym czasem wykonywania. W informatyce problem ten jest powszechnie znany, ponieważ cechuje go właśnie wyjątkowe tempo wzrostu poziomu złożoności i wiele bardzo bystrych osób twierdzi, że nie da się z tym nic zrobić. Jest to tzw. **problem podróżującego komiwojażera** (ang. *traveling salesman problem*).

Mamy komiwojażera.

Komiwojażer ten musi zawitać do pięciu miast.

Nasz sprzedawca, nazwijmy go Opus, chce odwiedzić każde z pięciu miast, pokonując jak najmniejszy dystans. Oto jeden ze sposobów na rozwiązanie tego zadania: sprawdzić wszystkie możliwe kolejności odwiedzania miast.



Sprzedawca sumuje odległości, a następnie wybiera najkrótszą drogę. Dla pięciu miast istnieje 120 permutacji, a więc rozwiązanie zadania dla pięciu miast wymaga wykonania 120 operacji. Dla 6 miast liczba operacji wynosi już 720 (istnieje 720 permutacji), a dla 7 miast ta liczba sięga już 5040!

Miasta	Operacje
6	720
7	5040
8	40 320
...	...
15	1 307 674 368 000
...	...
30	265 252 859 812 191 058 636 308 480 000 000

Liczba operacji radykalnie rośnie

A to uogólnienie: dla *n* elementów obliczenie wyniku wymaga wykonania *n!* („*n* silnia") operacji, a więc jest to algorytm charakteryzujący się **czasem wykonywania** O(*n!*). Rozwiązanie problemu dla jakiejkolwiek liczby elementów, nie licząc paru najmniejszych, zajmuje bardzo dużo czasu. Gdyby miast było więcej niż 100, obliczenia trwałyby tak długo, że szybciej doszłoby do śmierci naszego Słońca.

Ten algorytm jest straszny! Opus powinien wybrać jakiś inny, prawda? Jednak nie może tego zrobić, ponieważ jest to właśnie jeden z nierozwiązanych problemów informatyki. Nikt nie zna szybkiego algorytmu rozwiązującego to zadanie, a bardzo mądrzy ludzie uważają, że opracowanie takiego algorytmu jest *niemożliwe*. Dlatego najlepszym wyjściem z tej sytuacji jest zadowolenie się wynikiem przybliżonym — więcej na ten temat piszę w rozdziale 10.

I jeszcze jedna uwaga: zaawansowani czytelnicy mogą zainteresować się binarnymi drzewami poszukiwań! Opisałem je krótko w ostatnim rozdziale.

Powtórzenie

- Wyszukiwanie binarne jest znacznie szybsze od wyszukiwania prostego.
- O(*log n*) oznacza szybszy algorytm niż O(*n*) i im większy zbiór do przeszukania, tym większa robi się różnica prędkości.
- Szybkości algorytmów nie mierzy się w sekundach.
- Szybkość algorytmów określa się, podając tempo *zwiększania się* ilości pracy.
- Szybkość algorytmów przedstawia się za pomocą notacji dużego O.

Skorowidz

Q

R

S

T

U

W

Z


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Poznaj algorytmy i przekonaj się, jak bardzo są przydatne!

Aby projektować aplikacje, nie wystarczy poznać kilka języków programowania i opanować zestaw narzędzi deweloperskich. Konieczna jest jeszcze wiedza o tym, w jaki sposób rozwiązać dany problem, innymi słowy, trzeba poznać algorytmy. Naturalnie w praktyce programistycznej stosuje się ograniczony zestaw algorytmów. Zostały one już dawno przeanalizowane i przetestowane. Opisano je w wielu trudnych książkach, najeżonych niezrozumiałymi schematami i dowodami z pogranicza matematyki, statystyki i jeszcze kilku innych nieprzystępnych dziedzin.

Jeśli chcesz po prostu zrozumieć działanie algorytmów, a nie masz ochoty na mozolne przedzieranie się przez setki trudnych stron, to trzymasz w ręku właściwą książkę! Dzięki temu interesującemu, przystępnemu podręcznikowi szybko przyswoisz sobie najważniejsze pojęcia i łatwo zrozumiesz, w jaki sposób algorytmy pomagają w rozwiązywaniu problemów programistycznych. W książce pokazano słabe i mocne strony najważniejszych algorytmów. Nie zabrakło przydatnych schematów i przykładowych fragmentów kodu napisanego w Pythonie. Docenią ją szczególnie programiści samoucy, inżynierowie i każdy, kto chce uzyskać wiedzę o algorytmach.

W tej książce przedstawiono między innymi:

- wyjaśnienie takich pojęć jak tablice skrótów, listy powiązane, rekurencja
- algorytmy sortowania, problem komiwojażera, algorytmy zachłanne
- analizę szybkości algorytmów metodą dużego O
- algorytmy grafów, w tym algorytm wyszukiwania wszerz i algorytm Dijkstry
- algorytm KNN służący do uczenia maszynowego

Aditya Y. Bhargava programuje od ponad dwudziestu lat. Jako nastolatek pisał gry wideo w językach Basic i ActionScript. Pracował w kilku startupach. Obecnie jest programistą w Etsy.com. Oprócz tego od kilku lat uczy programowania, z powodzeniem przedstawiając trudne koncepcje i idee w taki sposób, aby ich zrozumienie przychodziło bez trudu. Interesuje się sztuką, literaturą i oczywiście programowaniem.

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 978-83-283-9874-0
9 788328 398740
Cena: 67,00 zł