
Helion
ALGORYTMY
BEZ TAJEMNIC
THOMAS H. CORMEN

Tytuł oryginału: Algorithms Unlocked

Tłumaczenie: Zdzisław Płoski

ISBN: 978-83-246-7485-5

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/algbet_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pamięci mojej ukochanej Matki, Renee Cormen.

Spis treści

Przedmowa

W jaki sposób komputery rozwiązują problemy? Jak to się dzieje, że Twój mały GPS odnajduje pośród bezliku możliwych tras najszybszą drogę do celu, i robi to w kilka sekund? Na czym polega ochrona numeru Twojej karty kredytowej przed przechwyceniem przez kogoś innego, kiedy robisz zakupy w Internecie? Odpowiedzią na te i mnóstwo innych pytań są **algorytmy**. Napisałem tę książkę, aby odkryć przed Tobą tajemnice algorytmów.

Jestem współautorem podręcznika *Wprowadzenie do algorytmów*[1]. To wspaniała książka (moja opinia jest oczywiście tendencyjna), lecz miejscami zdecydowanie zbyt fachowa.

Ta książka nie jest *Wprowadzeniem do algorytmów*. Nie jest nawet podręcznikiem. Nie opisuje szeroko dziedziny algorytmów komputerowych ani jej nie zgłębia. Nie sili się, by uczyć technik projektowania algorytmów komputerowych i nie ma w niej ani jednego problemu lub ćwiczenia do rozwiązania przez czytelników. Czym zatem jest ta książka? Jest miejscem, od którego możesz rozpocząć, jeżeli:

- interesuje Cię, jak komputery rozwiązują problemy;
- chcesz się dowiedzieć, jak ocenić jakość tych rozwiązań;
- ciekawi Cię, jak problemy obliczeniowe i metody ich rozwiązywania mają się do świata pozakomputerowego;
- umiesz nieco posługiwać się matematyką;
- i niekoniecznie musisz się legitymować napisaniem choćby jednego programu komputerowego (choć pewna umiejętność programowania z pewnością by nie zaszkodziła).

Niektóre książki o algorytmach komputerowych skupiają się na pojęciach, nie wchodząc zbytnio w szczegóły techniczne. Inne pełne są technicznych detali. Jeszcze inne są czymś pośrednim. Każdy rodzaj książki ma swoje miejsce. Tę książkę ulokowałbym gdzieś pośrodku. Owszem, jest w niej trochę matematyki, gdzieniegdzie staje się dość ścisła, lecz unikałem zagłębiania się w szczegóły (może z wyjątkiem samego końca, przy którym nie potrafiłem się już opanować).

Myślę o tej książce jak o czymś w rodzaju *antipasto*[2]. Powiedzmy, że wstępujesz do włoskiej restauracji i zamawiasz sobie coś na ząb, odkładając decyzję, czy zamówić resztę obiadu, na potem — po zjedzeniu przystawki. Dostajesz ją i jesz. Być może przystawka Ci nie zasmakuje, więc uznasz, że nie zamówisz niczego więcej. A może przypadnie Ci do gustu, lecz poczujesz, że już masz dość i nie musisz niczego więcej

[1] WNT, Warszawa 2004 — *przyp. tłum.*

[2] Z wł. przystawka, przekąska — *przyp. tłum.*

zamawiać. Niewykluczone również, że po skonsumowaniu przystawki poczujesz większy apetyt i zaczniesz się rozglądać za czymś posilniejszym. Traktując tę książkę jako przystawkę, liczę się z jedną z tych dwu reakcji: albo ją przeczytasz, odczujesz zadowolenie, lecz bez potrzeby głębszego wnikania w świat algorytmów, albo tak zagustujesz w tym, co tu przeczytasz, że zechcesz nauczyć się więcej. Każdy rozdział kończy się podrozdziałem zatytułowanym „Co czytać dalej", który powiedzie Cię do książek i artykułów głębiej drążących poszczególne zagadnienia.

Czego się nauczysz z tej książki?

Nie mogę Ci powiedzieć, czego się z tej książki nauczysz. *Chciałbym*, aby nauczyła Cię ona:

- Tego, czym są algorytmy komputerowe, a także pewnego sposobu ich opisywania i oceny.
- Prostych sposobów poszukiwania informacji w komputerze.
- Metod reorganizowania informacji w komputerze, porządkujących je według z góry określonych reguł. (Zadanie takie nazywamy „sortowaniem").
- Rozwiązywania podstawowych problemów, które umiemy zamodelować w komputerze za pomocą struktury matematycznej zwanej „grafem". Pośród wielu zastosowań grafy świetnie przydają się do modelowania sieci dróg (które skrzyżowania mają bezpośrednie połączenia z innymi skrzyżowaniami i jak długie są te drogi?), zależności między zadaniami (które zadanie powinno poprzedzać inne zadania?), zagadnień finansowych (jakie są przeliczniki walut krajów całego świata?) lub związków między ludźmi (kto kogo zna? kto kogo nie lubi? który aktor wystąpił w filmie z innym aktorem?).
- Rozwiązywania problemów, w których stawia się pytania dotyczące napisów, tj. ciągów znaków tworzących teksty. Niektóre z tych problemów mają zastosowania w takich dziedzinach jak biologia, gdzie znaki reprezentują podstawowe cząsteczki, a napisy — strukturę DNA.
- Podstawowych zasady kryptografii. Nawet jeśli nigdy nie przyszło Ci szyfrować żadnej wiadomości, Twój komputer zapewne to robi (kiedy np. kupujesz coś online).
- Podstawowych koncepcji dotyczących kompresji danych, wychodzących znacznie poza „f u cn rd ths u cn gt a gd jb n gd pay"[3].
- Tego, że pewne problemy są trudne do rozwiązania na komputerze w rozsądnym czasie, a przynajmniej nikt dotąd nie znalazł sposobu poradzenia sobie z nimi.

[3] Oczywiście jest to próbka „języka esemesowego" w wydaniu angielskim — *przyp. tłum.*

Co wypadałoby zawczasu wiedzieć, aby zrozumieć zamieszczony tu materiał?

Jak już powiedziałem, jest w tej książce trochę matematyki. Jeśli matematyka Cię przeraża, to możesz spróbować ją pomijać lub zajrzeć do książki mniej technicznej. Robiłem jednak, co w mojej mocy, aby uczynić tę matematykę przystępną.

Nie zakładam, że kiedykolwiek napisałeś jakiś program komputerowy (nie zakładam nawet, że kiedykolwiek zapoznałeś się z kodem takiego programu). Jeśli będziesz podążać za instrukcjami w wydzielonych polach, uda Ci się zrozumieć, w jaki sposób wyrażam kroki, które — razem wzięte — tworzą algorytm. Jeśli zaskoczysz z następującym dowcipem, część drogi będzie za Tobą:

— Słyszałaś o informatyku, który utknął pod prysznicem?

— ???

— Mył[4] włosy według przepisu na butelce z szamponem: namydlić, spłukać, powtórzyć.

Zastosowałem w książce styl dość nieformalny w nadziei, że lżejsza forma pomoże w uprzystępnieniu jej treści. Niektóre rozdziały wymagają znajomości materiału z poprzednich rozdziałów, jednak zależności takich jest niewiele. Początki niektórych rozdziałów są napisane językiem bardzo luźnym, przystępnym dla każdego, a zawarte w nich opisy stopniowo stają się bardziej techniczne. Jeśli nawet zauważysz, że treść któregoś rozdziału Cię przerasta, niewykluczone, że uda Ci się skorzystać przynajmniej z lektury początku następnego rozdziału.

Zgłaszanie błędów

Jeśli znajdziesz błąd w książce, wpisz erratę pod adresem *helion.pl/ksiazki/algbet.htm*.

Podziękowania

Wiele materiału zawartego w tej książce pochodzi z *Wprowadzenia do algorytmów*, zawdzięczam więc niemało moim współautorom tamtej książki: Charlesowi Leisersonowi, Ronowi Rivestowi i Cliffowi Steinowi. Zauważysz, że w książce bez skrupułów odwołuję się do (czytaj: robię reklamę) *Wprowadzenia do algorytmów* znanego daleko i szeroko pod nazwą *CLRS*, od inicjałów czterech jego autorów. Pisząc tę książkę samotnie, uświadomiłem sobie, jak bardzo brakuje mi współpracy z Charlesem, Ronem i Cliffem. Pośrednio dziękuję również wszystkim, którym wyraziliśmy wdzięczność w przedmowie do CLRS.

[4] Albo myła, biorąc jednak pod uwagę niefortunną proporcję płci w informatyce, musiał to być on.

Czerpałem również z wykładów, które prowadziłem w Dartmouth, zwłaszcza z Computer Science 1, 5 i 25. Dziękuję moim studentom za to, że wnikliwymi pytaniami umożliwili mi zorientowanie się, które z podejść pedagogicznych były skuteczne, a kamiennym milczeniem — które były chybione.

Podjęcie pracy nad tą książką zasugerowała Ada Brunstein, nasza redaktorka w MIT Press podczas przygotowywania trzeciego wydania CLRS. Zastąpił ją potem Jim DeWolf. Początkowo książka kandydowała do wydawanej przez MIT Press serii *Essential Knowledge*, lecz wydawcy z MIT Press uznali ją za zbyt techniczną jak na tę serię. (Wyobraźcie sobie, napisałem książkę zbyt techniczną dla MIT!). Jim sprawnie poradził sobie z tą potencjalnie niewygodną sytuacją, pozwalając mi pisać to, co chciałem, zamiast książki, o której początkowo przemyśliwano w MIT Press. Jestem również wdzięczny za pomoc Ellen Faran i Gitcie Devi Manaktali z MIT Press.

Julie Sussman, P.P.A., sprawowała pieczę nad opracowaniem technicznym drugiego i trzeciego wydania CLRS i miałem jeszcze raz prawdziwą przyjemność zawdzięczać jej adjustację tej książki. Najlepsza. Techniczna. Redaktorka. Wszech czasów. Nie zostawiła na mnie suchej nitki. Oto dowód w postaci fragmentu listu, który Julia wysłała mi w związku z wczesną wersją rozdziału 5:

> Szanowny Panie!
>
> Zbiegły rozdział, który ukrywał się w Pańskiej książce, został aresztowany przez organy ścigania. Nie jesteśmy w stanie określić, z której książki uciekł, przy czym nie wyobrażamy sobie, jak mógł przebywać w Pana książce tyle miesięcy bez Pańskiej wiedzy. Nie pozostaje nam więc nic innego, jak obciążyć Pana za to odpowiedzialnością. Wyrażamy nadzieję, że dołoży Pan starań, aby go zreformować i dać mu szansę, by stał się pożytecznym obywatelem Pańskiej książki. Raport oficera śledczego, Julii Sussman, w załączeniu.

Gdyby Cię interesowało, co znaczą litery „P.P.A.", wyjaśniam, że dwie pierwsze biorą się od słów „Professional Pain". Zapewne domyślasz się, co znaczy „A", lecz chciałbym podkreślić, że Julia jest dumna z tego tytułu — i słusznie! Stokrotne dzięki, Julio!

Żaden ze mnie kryptograf, toteż rozdział o podstawach kryptografii zyskał niezwykle dużo dzięki uwagom Rona Rivesta, Seana Smitha, Racheli Miller i Huijii Racheli Lin. W owym rozdziale jest przypis dotyczący znaków bejsbolowych i niech będą dzięki Bobowi Whalenowi, trenerowi bejsbolu w Dartmouth, za cierpliwe objaśnianie systemu oznaczeń w tej grze. Ilana Arbisser sprawdziła, czy biolodzy stosujący obliczenia komputerowe w sekwencjonowaniu DNA robią to w sposób wyjaśniony przeze mnie w rozdziale 7. Jim DeWolf przebił się wraz ze mną przez kilka kolejnych podejść do nadania książce tytułu, wszakże na pomysł nazwania jej *Algorithms Unlocked* wpadł student licencjat w Dartmouth, Chander Ramesh.

Instytut Informatyki Uniwersytetu Dartmouth jest fantastycznym miejscem do pracy. Moi kumple są świetni i koleżeńscy, a nasz zespół pracowniczy nie ma sobie równych. Jeśli rozglądasz się za studiami licencjackimi lub magisterskimi na kierunku informatyka lub chcesz się zaczepić na wydziale informatyki, spróbuj w Dartmouth.

Na koniec dziękuję mojej żonie Nicole Cormen, moim rodzicom, Renee i Perry'emu Cormenom, siostrze, Jane Maslin i rodzicom Nicole: Colette i Paulowi Sage'om, za ich miłość i życzliwą pomoc. Mój ojciec jest pewien, że rysunek na stronie 2 to 5, a nie S.

TOM CORMEN
Hanover, New Hampshire
Listopad 2012 roku.

1 Co to są algorytmy i dlaczego warto poświęcać im uwagę?

Zacznijmy od pytania, które mi często zadają: „Co to jest algorytm?”[1].

Ogólna odpowiedź mogłaby być taka: „zbiór kroków prowadzących do wykonania zadania”. Na co dzień stosujesz różne algorytmy. Masz algorytm mycia zębów: otwierasz tubkę z pastą, bierzesz szczoteczkę do ręki, wyciskasz na szczoteczkę tyle pasty, ile trzeba, zamykasz tubkę, wkładasz szczoteczkę do jednej ćwiartki paszczy, przesuwasz nią w górę i w dół (oraz w prawo i w lewo) przez *N* sekund itd. Jeśli musisz dojeżdżać do pracy, masz swój algorytm dojazdu do pracy. I tak dalej.

Jednak ta książka zajmuje się algorytmami, które działają na komputerach, a ogólnie rzecz ujmując — na urządzeniach obliczeniowych. Z algorytmami wykonywanymi na komputerach jest podobnie jak z tymi, które wykonujesz na co dzień. Korzystasz z GPS-u, żeby znaleźć drogę? Żeby ją odnaleźć, wykonuje on algorytm, który nazywamy „najkrótsza ścieżka”. Kupujesz coś w Internecie? Używasz wtedy (a przynajmniej należałoby!) bezpiecznej witryny sieciowej, która wykonuje algorytm szyfrowania. Gdy kupujesz jakieś towary w Sieci, kto Ci je dostarcza? Firma kurierska? Korzysta ona z algorytmów przydzielających paczki do furgonetek, a potem ustalających kolejność, w jakiej każdy kierowca powinien te paczki rozwozić. Algorytmy działają w komputerach na każdym kroku: w Twoim laptopie, smartfonie, na serwerach lub w systemach wbudowanych (np. w Twoim samochodzie czy w mikrofalówce, w systemach klimatyzacji) — wszędzie!

Co różni algorytmy wykonywane na komputerach od algorytmu wykonywanego przez Ciebie? Ty potrafisz tolerować algorytm, jeśli jest on niedokładnie opisany, a komputer nie. Na przykład, jeśli jedziesz do pracy, Twój algorytm jedź-do-pracy mógłby zawierać taką klauzulę: „jeśli jest tłoczno, wybierz inną trasę”. Ty możesz wiedzieć, co rozumiesz pod określeniem „jest tłoczno”, natomiast komputer tego nie pojmie.

Dlatego algorytm komputerowy jest zbiorem kroków prowadzących do wykonania zadania, opisanym na tyle precyzyjnie, że potrafi go wykonać komputer. Na czym polega ta precyzja, wiesz, jeśli masz choć trochę doświadczenia w programowaniu komputerów w Javie, C, C++, Pythonie, Fortranie, Mathlabie lub w czymś podobnym. Jeśli nie masz w dorobku ani jednego programu komputerowego, to niewykluczone, że poczujesz ten stopień dokładności, przyglądając się, jak opisuję algorytmy w tej książce.

Przejdźmy do następnego pytania: „Czego oczekujemy od algorytmu komputerowego?”.

---

[1] Lub, jak by rzekł mój kolega, z którym gram w hokeja: „What’s a nalgorithm?”.

Algorytmy komputerowe rozwiązują problemy obliczeniowe. Od algorytmu komputerowego oczekujemy dwóch rzeczy: mając dane do problemu, powinien zawsze wytwarzać poprawne rozwiązanie tego problemu, a robiąc to, powinien oszczędnie zużywać zasoby obliczeniowe. Przyjrzyjmy się obu tym postulatom.

Poprawność

Co oznacza poprawne rozwiązanie problemu? Na ogół potrafimy precyzyjnie określić, co powinno zawierać poprawne rozwiązanie. Na przykład, jeśli Twój GPS wytwarza poprawne rozwiązanie, gdy chodzi o znalezienie najlepszej trasy podróży, to mogłoby to polegać na wybraniu takiej trasy spośród wszystkich możliwych do wytyczenia między miejscem Twojego pobytu a miejscem docelowym, którą dotrzesz tam najszybciej. Albo trasy możliwie najkrótszej. Albo takiej, która nie tylko poprowadzi Cię do celu najszybciej, lecz również pozwoli uniknąć płacenia myta. Jasne, że informacje, których Twój GPS używa do wyznaczenia trasy, mogą nie odpowiadać rzeczywistości. O ile nie ma on dostępu w czasie rzeczywistym do danych o ruchu drogowym, mógłby przyjąć, że czas potrzebny do przebycia drogi równa się długości drogi podzielonej przez dopuszczalną na niej prędkość. Jeśli jednak droga jest zatłoczona, GPS mógłby Ci źle doradzić w poszukiwaniach najszybszej trasy. Mimo to nadal możemy uznać, że algorytm wytyczania trasy działa poprawnie — nawet jeśli nie można tego powiedzieć o jego danych wejściowych; dla zadanego wejścia algorytm trasujący określa trasę najszybszą.

Z kolei dla pewnych problemów ustalenie, czy dany algorytm wytwarza poprawne rozwiązanie, może się okazać trudne lub nawet niemożliwe. Jako przykład weźmy optyczne rozpoznawanie znaków. Czy ten obrazek o wymiarach 11 × 6 piksli jest cyfrą 5, czy literą S?

Niektórzy powiedzą, że to piątka, podczas gdy inni — że S, jak zatem moglibyśmy uznać poprawność lub niepoprawność decyzji komputerowej? Nie da się. W tej książce skoncentrujemy się na algorytmach komputerowych, których rozwiązania są znane.

Czasami jednak godzimy się z tym, że algorytm komputerowy produkuje niepoprawną odpowiedź, o ile tylko potrafimy panować nad tym, jak często mu się to zdarza. Dobry przykład stanowi szyfrowanie. Powszechnie używany kryptosystem RSA polega na określaniu, czy duże liczby — naprawdę duże, mające setki cyfr — są pierwsze. Jeśli zdarzyło Ci się pisać programy komputerowe, to jednym z nich był pewnie taki, który sprawdzał, czy liczba n jest pierwsza. Mógł on badać wszystkie potencjalne podzielniki od 2 do $n - 1$ i jeśli któryś z nich okazał się rzeczywiście podzielnikiem n, to n była liczbą złożoną. Jeśli żadna liczba między 2 a $n - 1$ nie jest podzielnikiem n, to n jest pierwsza. Jeśli jednak n ma setki cyfr, to potencjalnych podzielników jest mnóstwo — tak dużo, że nawet naprawdę szybki

komputer nie dałby rady tego sprawdzić w rozsądnym czasie. Oczywiście mógłbyś dokonać pewnych ulepszeń, na przykład wyeliminować wszystkie kandydatki parzyste po wykazaniu, że 2 nie jest podzielnikiem, lub poprzestać na dotarciu do $\sqrt{n}$ (bo jeśli d jest większe niż $\sqrt{n}$ i d jest podzielnikiem n, to n/d jest mniejsze niż $\sqrt{n}$ i również jest podzielnikiem n; jeśli więc n ma podzielnik, to znajdziesz go nim dojdziesz do $\sqrt{n}$). Jeśli n ma setki cyfr, to chociaż $\sqrt{n}$ ma około o połowę cyfr mniej niż n, nadal jest naprawdę wielką liczbą. I tu dobra wiadomość: znamy algorytm, który szybko sprawdza, czy liczba jest pierwsza. Złą wiadomością jest to, że może on popełniać błędy. W szczególności, jeśli oświadcza, że n jest złożona, to n jest na pewno złożona, lecz jeśli oświadcza, że n jest pierwsza, to istnieje pewna szansa, że n jest jednak złożona. Lecz owe złe wiadomości nie są aż tak złe: możemy utrzymywać wskaźnik błędu na naprawdę niskim poziomie, wynoszącym na przykład jeden błąd na każde 2^{50} razy. Jest to na tyle rzadko — jeden błąd raz na każde sto oktylionów — że większość z nas ze spokojem przyjmuje w RSA określanie tą metodą, czy liczba jest pierwsza.

Poprawność jest delikatnym zagadnieniem w innej klasie algorytmów, nazywanych aproksymacyjnymi. Algorytmy aproksymacyjne (przybliżania pewnej wartości — przyp. tłum.) są stosowane w problemach optymalizacyjnych, w których chcemy znaleźć najlepsze rozwiązanie względem pewnej miary ilościowej. Znajdowanie najszybszej trasy, wykonywane przez GPS, jest jednym z przykładów — tu miarą ilościową jest czas podróży. Dla niektórych problemów nie mamy dobrych algorytmów znajdujących optymalne rozwiązanie w rozsądnym czasie, znamy natomiast algorytm aproksymacyjny, który w zadowalającym czasie potrafi znaleźć rozwiązanie prawie optymalne. Przez „prawie optymalne" zazwyczaj rozumiemy, że ilościowa miara rozwiązania znajdowana przez algorytm aproksymacyjny różni się pewnym znanym czynnikiem od optymalnej miary ilościowej rozwiązania. Jeśli tylko określimy, ile ów pożądany czynnik wynosi, możemy mówić, że poprawnym rozwiązaniem algorytmu aproksymacyjnego jest każde rozwiązanie różniące się tym czynnikiem od rozwiązania optymalnego.

Użytkowanie zasobów

Co to znaczy w odniesieniu do algorytmu *efektywne użytkowanie zasobów obliczeniowych*? O jednej z miar efektywności wspomnieliśmy, omawiając algorytmy aproksymacyjne — był nią czas. Algorytm, który daje poprawne rozwiązanie, lecz zużywa dużo czasu na jego wytworzenie, może mieć znikomą wartość lub być bezwartościowy. Gdyby Twój GPS przez godzinę wyznaczał zalecaną trasę, chciałoby Ci się go w ogóle włączać? Rzeczywiście, czas jest podstawową miarą efektywności, której używamy do oceny algorytmu, kiedy już wykażemy, że daje poprawne rozwiązanie. Lecz nie jest to miara jedyna. Może nas interesować ilość pamięci komputerowej wymaganej przez algorytm („ślad" jaki odciska w pamięci), ponieważ algorytm musi działać w dostępnej pamięci. Inne zasoby, które mogą być używane przez algorytm, to: komunikacja sieciowa, losowe bity (ponieważ algorytmy, które dokonują

losowych wyborów, potrzebują źródła liczb losowych) lub operacje dyskowe (dla algorytmów przeznaczonych do pracy z danymi przechowywanymi na dysku).

W tej książce, tak jak w większości literatury o algorytmach, koncentrujemy się tylko na jednym zasobie — jest nim czas. Jak rozsądzamy o czasie wymaganym przez algorytm? W odróżnieniu od poprawności, niezależącej od konkretnego komputera, na którym działa algorytm, faktyczny czas algorytmów zależy od kilku czynników zewnętrznych względem samego algorytmu: szybkości komputera, języka programowania, w którym algorytm jest zrealizowany, kompilatora lub interpretera tłumaczącego program na kod wykonywany w komputerze, umiejętności osoby piszącej dany program i innych działań i zdarzeń zachodzących w komputerze równolegle z wykonywanym programem. Przy tym wszystkim zakłada się, że algorytm jest wykonywany tylko przez jeden komputer, mieszczący w pamięci wszystkie jego dane.

Gdybyśmy mieli oceniać szybkość algorytmu zrealizowanego w prawdziwym języku programowania, wykonując go na konkretnym komputerze dla określonych danych i mierząc jego czas działania, nie dowiedzielibyśmy się niczego o tym, jak szybko działałby ten algorytm z danymi innego rozmiaru, a może nawet z innymi danymi o tym samym rozmiarze. A gdybyśmy chcieli porównać względną szybkość danego algorytmu z innym, dla tego samego problemu, musielibyśmy zaimplementować oba i wykonać każdy z nich z różnymi danymi, o różnych rozmiarach. Jak zatem możemy ocenić szybkość algorytmu?

Otóż robimy to, łącząc dwa pomysły. Po pierwsze, określamy, jak długo działa algorytm w funkcji rozmiaru jego danych. W przykładzie ze znajdowaniem trasy dane wejściowe byłyby pewną reprezentacją mapy drogowej, a ich rozmiar zależałby od liczby skrzyżowań i liczby dróg łączących skrzyżowania na mapie. (Fizyczny rozmiar sieci dróg nie ma znaczenia, ponieważ wszystkie odległości możemy przedstawić za pomocą liczb, a wszystkie liczby zajmują tyle samo miejsca na wejściu; długość drogi nie wpływa na rozmiar danych wejściowych). W prostszym przykładzie — przeszukiwaniu zadanej listy elementów w celu sprawdzenia, czy określony element jest na niej obecny — rozmiarem danych byłaby liczba pozycji na liście.

Po drugie, skupiamy się na tym, jak szybko funkcja charakteryzująca czas działania rośnie z rozmiarem danych wejściowych, tzn. na *tempie wzrostu* czasu działania. W rozdziale 2 zapoznamy się z notacją, której używamy do scharakteryzowania czasu działania algorytmu, lecz to, co jest w naszym podejściu najbardziej interesujące, to uwzględnianie w czasie działania tylko dominującego składnika; nie zwracamy przy tym uwagi na współczynniki. Koncentrujemy się zatem na **rzędzie wzrostu** czasu działania. Załóżmy na przykład, że udało się nam ustalić, iż dana realizacja pewnego algorytmu przeszukiwania listy n elementów zajmuje $50n + 125$ cykli maszynowych. Składnik $50n$ zdominuje składnik 125, jeśli n stanie się dostatecznie duże, poczynając od $n \geq 3$ i zwiększając jeszcze tę przewagę dla list o większych rozmiarach. Wobec tego, opisując czas działania tego hipotetycznego algorytmu, nie bierzemy pod uwagę składnika 125 małego rzędu. Może się zdziwisz, ale odrzucamy też współczynnik 50. Prowadzi to do określenia czasu działania jako rosnącego liniowo

z rozmiarem danych n. Oto inny przykład: gdyby algorytm zużywał $20n^3 + 100n^2 + 300n + 200$ cykli maszynowych, to powiedzielibyśmy, że jego czas działania rośnie jak n^3. Również w tym przypadku składniki niższego rzędu: $100n^2$, $300n$ i 200, stają się coraz mniej istotne ze wzrostem rozmiaru n danych wejściowych.

W praktyce lekceważone przez nas współczynniki mają jednak znaczenie. Zależą one jednak w tak dużym stopniu od czynników zewnętrznych, że jest prawie pewne, iż przy porównywaniu dwóch algorytmów A i B, mających ten sam rząd wzrostu i działających na tych samych danych, A może działać szybciej niż B dla pewnej kombinacji maszyny, języka programowania, kompilatora (lub interpretera) i programisty, a B zadziała szybciej niż A dla innej kombinacji. Oczywiście, jeżeli oba algorytmy A i B produkują poprawne rozwiązania i A zawsze działa dwa razy szybciej niż B, to jeśli wszystko inne jest takie samo, będziemy zawsze preferowali wykonywanie A zamiast B. Jednakże z punktu widzenia abstrakcyjnego porównywania algorytmów zawsze koncentrujemy się na rzędzie wzrostu, nie przystrajając go współczynnikami lub składnikami niskiego rzędu.

I ostatnie pytanie, które stawiamy w tym rozdziale: „Dlaczego miałoby mi się opłacać zajmowanie algorytmami komputerowymi?". Odpowiedź na nie zależy od tego, kim jesteś.

Algorytmy komputerowe dla niekomputerowców

Nawet jeśli w sferze komputerów nie uważasz się za osobę dobrze poinformowaną, algorytmy komputerowe stanowią dla Ciebie dużą wartość. W końcu, wyjąwszy sytuację, w której podróżujesz po bezdrożach bez GPS-u, zapewne używasz go co dnia. Szukałeś czegoś dzisiaj w Internecie? W używanej przez Ciebie wyszukiwarce — Google, Bing lub jakiejkolwiek innej — są stosowane wyrafinowane algorytmy przeszukiwania Sieci i rozstrzygania, w jakiej kolejności należy przedstawiać ich wyniki. Prowadziłeś dziś samochód? O ile nie jeździsz klasycznym wehikułem, jego komputery pokładowe podjęły podczas Twej podróży miliony decyzji, a wszystkie oparte były na algorytmach. Mógłbym tak wymieniać bez końca.

Jako docelowy użytkownik algorytmów sobie zawdzięczasz chęć dowiedzenia się czegoś o tym, jak projektujemy, charakteryzujemy i oceniamy algorytmy. Zakładam, że przejawiasz przynajmniej umiarkowane zainteresowanie, skoro ta książka trafiła do Twoich rąk i czytasz ją do tego miejsca. Powodzenia! Zobaczmy, czy uda się nam rozkręcić Cię na tyle, że na następnym spotkaniu towarzyskim dotrzymasz tonu innym, gdy rozmowa zejdzie na algorytmy[2].

[2] No tak, przyznaję, jeśli nie mieszkasz w Dolinie Krzemowej, temat algorytmów rzadko wypłynie podczas koktajli, w których bierzesz udział, niemniej z pewnych względów my, profesorowie informatyki, uważamy, że jest ważne, aby nasi studenci nie konsternowali nas na spotkaniach towarzyskich brakiem wiedzy z poszczególnych dziedzin informatyki.

Algorytmy komputerowe dla komputerowców

Jeśli jesteś za pan brat z komputerami, to masz większe obowiązki wobec algorytmów! One nie tylko znajdują się w centrum wszystkiego, co się dzieje w Twoim komputerze — algorytmy są technologią, i to równie istotną jak wszystko, co współtworzy Twój komputer. Możesz zapłacić dodatkową cenę za komputer z najnowszym i największym procesorem, lecz aby te pieniądze okazały się trafionym wydatkiem, potrzebujesz, by w tym komputerze działały realizacje dobrych algorytmów.

Oto przykład ilustrujący, że algorytmy rzeczywiście stanowią technologię. W trzecim rozdziale przyjrzymy się trzem różnym algorytmom sortowania w porządku rosnącym listy n wartości. Niektóre z tych algorytmów będą osiągały czasy działania rosnące jak n^2, lecz inne będą działać w czasie rosnącym tylko jak $n \lg n$. Co to jest $\lg n$? Jest to logarytm przy podstawie 2 z n, czyli $\log_2 n$. Informatycy używają logarytmów z podstawą 2 tak często, że podobnie jak matematycy i naukowcy, którzy dla skrócenia stosują zapis $\ln n$ na oznaczenie logarytmu naturalnego — $\log_e n$, również oni używają własnego skrótu dla logarytmów przy podstawie 2. I tak, ponieważ funkcja $\lg n$ jest odwrotnością funkcji wykładniczej, rośnie ona bardzo powoli z n. Jeśli $n = 2^x$, to $x = \lg n$. Na przykład $2^{10} = 1024$, więc $\lg 1024$ wynosi tylko 10, podobnie $2^{20} = 1\,048\,576$, zatem $\lg 1\,048\,576$ wynosi zaledwie 20, a $2^{30} = 1\,073\,741\,824$, co oznacza, że $\lg 1\,073\,741\,824$ wynosi raptem 30. Tak więc we wzroście $n \lg n$ w porównaniu ze wzrostem n^2 zachodzi wymiana czynnika n na jedynie $\ln n$, a to już jest gra warta świeczki.

Skonkretyzujmy nieco bardziej ten przykład, wystawiając szybszy komputer (komputer A) wykonujący algorytm sortowania, którego czas działania dla n wartości rośnie jak n^2, przeciw wolniejszemu komputerowi B, wykonującemu algorytm sortowania w czasie rosnącym jak $n \lg n$. Każdy z komputerów musi posortować 10 milionów liczb. (Choć może się wydawać, że 10 milionów liczb to dużo, jeśli są to 8-bajtowe liczby całkowite, ich rozmiar na wejściu zajmie około 80 megabajtów, co zmieści się w pamięci nawet niedrogiego laptopa sprzed wielu lat). Przypuśćmy, że komputer A wykonuje 10 miliardów operacji na sekundę (jest szybszy od każdego sekwencyjnego komputera w chwili, gdy piszę te słowa), a komputer B wykonuje tylko 10 milionów operacji na sekundę, czyli komputer A jest 1000 razy szybszy od komputera B, biorąc pod uwagę samą moc obliczeniową. Aby powiększyć jeszcze różnicę, załóżmy, że światowej sławy programistka koduje komputer A w języku maszynowym, a kod wynikowy wymaga do posortowania n liczb $2n^2$ rozkazów. Załóżmy dalej, że program dla komputera B pisze programista zupełnie przeciętny, korzystając z języka wysokiego poziomu i mało efektywnego kompilatora, w wyniku czego powstaje kod złożony z $50n \lg n$ rozkazów. Aby posortować 10 milionów liczb, komputer A zużywa

$$\frac{2 \cdot (10^7)^2 \text{ rozkazów}}{10^{10} \text{ rozkazów / sekundę}} = 20\,000 \text{ sekund},$$

co stanowi więcej niż 5 i pół godziny, podczas gdy komputerowi B zabiera to

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ rozkazów}}{10^7 \text{ rozkazów / sekundę}} \approx 1163 \text{ sekundy}$$

— równowartość mniej niż 20 minut. Stosując algorytm, którego czas działania rośnie znacznie wolniej, nawet z marnym kompilatorem, komputer B działa 17 razy szybciej niż komputer A! Przewaga algorytmu $n \lg n$ byłaby jeszcze wyraźniejsza, gdybyśmy sortowali 100 milionów liczb. Komputerowi A zabrałoby to ponad 23 dni, algorytm $n \lg n$ na komputerze B uporałby się z robotą w cztery godziny. Uogólniając, wraz ze wzrostem rozmiaru problemu rośnie względna przewaga algorytmu $n \lg n$.

Nawet mimo imponującego postępu, który nieustannie obserwujemy w sprzęcie komputerowym, całościowa produktywność systemu zależy w równym stopniu od doboru efektywnych algorytmów, co od doboru szybkiego sprzętu lub sprawnie działających systemów operacyjnych. Błyskawiczny postęp, jaki dokonał się w innych technologiach komputerowych, dotyczy także algorytmów.

Co czytać dalej

W moim bardzo subiektywnym odczuciu najklarowniejszym i najbardziej użytecznym źródłem wiedzy o algorytmach komputerowych jest *Wprowadzenie do algorytmów* [CLRS09], napisane przez czterech diabelnie przystojnych facetów. Książka ta jest powszechnie określana jako „CLRS", od inicjałów autorów. Sięgałem do niej po większość materiału pomieszczonego w tej książce. Jest ona nieporównanie bardziej kompletna od tej książki, lecz zakłada się w niej, że masz choć trochę doświadczenia w programowaniu komputerów i jesteś za pan brat z matematyką. Jeśli uznasz, że odpowiada Ci poziom matematyczny niniejszej książki, i masz odwagę ruszyć głębiej w temat, nie możesz postąpić lepiej, niż sięgnąć po CLRS. (W mojej skromnej opinii, ma się rozumieć).

Książka Johna MacCormica *Nine Algorithms That Changed the Future* [Mac12] zawiera opis kilku algorytmów i związanych z nimi kwestii obliczeniowych, które oddziałują na nasze codzienne życie. Ujęcie zastosowane przez MacCormica jest mniej techniczne niż w tej książce. Jeśli dojdziesz do wniosku, że tutaj było „zbyt matematycznie", to polecam Ci skosztowanie lektury MacCormica. Zdołasz przyswoić z niej wiele, nawet jeśli masz wątłe przygotowanie matematyczne.

W mało prawdopodobnym przypadku, gdy uznasz, że CLRS jest zbyt rozwodnione, możesz spróbować zaczerpnąć z wielotomowego zbioru Donalda Knutha *The Art of Computer Programming* [Knu97, Knu98a, Knu98b, Knu11]. Chociaż tytuł serii sugeruje, że skupia się ona na pisaniu kodu, książki te zawierają wspaniałą, głęboką analizę algorytmów[3]. Ostrzegam jednak: materiał zawarty w *TAOCP* sięga głęboko. Przy okazji, jeśli zastanawiasz się, skąd się wzięło słowo „algorytm", to Knuth powiada, że pochodzi ono od imienia „al-Khowârizmî" perskiego matematyka z IX wieku.

[3] *Sztuka programowania*, WNT, Warszawa 2002 – 2007. Jak dotąd ukazały się 4 książki z jeszcze nie ukończonego cyklu, który Donald Knuth zaplanował na 7 tomów — *przyp. tłum.*

Oprócz CLRS przez lata ukazało się kilka innych świetnych tekstów o algorytmach komputerowych. Noty do rozdziału 1 w CLRS zawierają odwołania do wielu takich tekstów. Zamiast powielać je tutaj, kieruję Cię wprost do CLRS.

2 Jak opisywać i oceniać algorytmy komputerowe

W poprzednim rozdziale było Ci dane posmakować, jak formułujemy czas działania algorytmu komputerowego: przez skoncentrowanie się na czasie działania rozpatrywanym jako funkcja rozmiaru danych wejściowych, ze szczególnym uwzględnieniem tempa wzrostu czasu działania. W tym rozdziale cofniemy się nieco i przyjrzymy się sposobowi opisywania algorytmów komputerowych. Następnie poznamy notację, której używamy do charakteryzowania czasów działania algorytmów. Rozdział ten zamkniemy rzutem oka na kilka technik, których używamy do projektowania i studiowania algorytmów.

Jak opisywać algorytmy komputerowe

Algorytm komputerowy zawsze możemy opisać w postaci gotowego do wykonania programu w powszechnie stosowanych językach, takich jak Java, C, C++, Python czy Fortran. I rzeczywiście, w kilku podręcznikach tak właśnie postąpiono. Kłopot z używaniem prawdziwych języków programowania do określania algorytmów polega na tym, że musisz grzęznąć w szczegółach języka zaciemniających same koncepcje algorytmów. Inne podejście, przyjęte przez nas we *Wprowadzeniu do algorytmów*, polega na użyciu „pseudokodu", który wygląda niczym papka przyrządzona z różnych języków programowania z dodatkiem ludzkiej mowy. Jeśli zdarzyło Ci się korzystać z prawdziwego języka programowania, zrozumienie pseudokodu nie sprawi Ci trudności. Jeśli jednak nigdy nie programowałeś, pseudokod może wyglądać trochę tajemniczo.

Podejście, które przyjąłem w tej książce, polega na tym, że nie próbuję opisywać algorytmów ani dla software'u, ani dla hardware'u, lecz dla „wetware'u": szarej materii między Twoimi uszami. Zakładam również, że nigdy nie napisałeś programu komputerowego, nie będę więc wyrażał algorytmów w żadnym prawdziwym języku programowania, ani nawet w pseudokodzie. Zamiast tego będę opisywał je po angielsku[1], używając analogii do scenariuszy z realnego świata wszędzie, gdzie tylko zdołam. Żeby wskazać, co się dzieje (co w programowaniu nazywamy „przepływem sterowania"), będę używał list, w tym również list zawierających inne listy. Jeśli zechcesz napisać algorytm w prawdziwym języku programowania, to ufam, że poradzisz sobie z przetłumaczeniem mojego opisu na wykonywalny kod.

Choć będę próbował używać nietechnicznego języka na tyle, na ile się da, jest to książka o algorytmach dla komputerów, muszę więc posługiwać się terminologią

[1] A my będziemy to czynić po polsku (dodając od czasu do czasu angielskie brzmienia odpowiednich terminów), niekiedy pozostawimy jednak bez zmian angielskie nazwy (skróty) algorytmów — *przyp. tłum.*

komputerową. Na przykład, programy komputerowe zawierają **procedury** (w rzeczywistych językach programowania nazywane także funkcjami lub metodami), określające jak zrobić to czy tamto. Aby spowodować wykonanie przez procedurę tego, do czego jest przeznaczona, **wywołujemy** ją (ang. *call*). Wywołując procedurę, dostarczamy jej danych wejściowych[2] (zazwyczaj co najmniej jedną, choć niekiedy procedura nie wymaga żadnych danych). Dane wejściowe podajemy w nawiasach po nazwie procedury jako **parametry**. Na przykład, aby obliczyć pierwiastek kwadratowy z liczby, moglibyśmy zdefiniować procedurę PIERWIASTEK-KWADRATOWY(x); tutaj dana wejściowa procedury jest określona jako parametr x. Wywołanie procedury może, acz nie musi, dawać wynik (dane wyjściowe, ang. *output*), zależnie od tego, jak określimy procedurę. Jeśli procedura wytwarza wynik, zwykle traktujemy go jako coś, co jest przekazywane z powrotem do wywołującego (ang. *caller*; ten, kto wywołał procedurę). W żargonie komputerowym mówimy, że procedura **zwraca** (ang. *returns*) wartość.

Wiele programów i algorytmów pracuje na tablicach danych. **Tablica** (ang. *array*) gromadzi dane tego samego typu w jedną całość. Możesz myśleć o tablicy jak o tabeli, w której dla danego **indeksu** pozycji w tabeli, nazywanej inaczej **wpisem** (ang. *entry*), możemy mówić o **elemencie** tablicy o tym indeksie. Oto na przykład tabela pięciu pierwszych prezydentów USA:

Indeks	Prezydent
1.	George Washington
2.	John Adams
3.	Thomas Jefferson
4.	James Madison
5.	James Monroe

Na przykład elementem z indeksem 4 jest w tej tabeli James Madison. Nie myślimy o niej jako o pięciu osobnych jednostkach, lecz jako o jednej tabeli z pięcioma wpisami (pozycjami). Z tablicą jest podobnie. Indeksy tablicy są kolejnymi liczbami naturalnymi, przy czym mogą się zaczynać od którejkolwiek, zwykle jednak rozpoczynamy je od 1[3]. Mając nazwę tablicy i związany z nią indeks, łączymy je, stosując nawiasy kwadratowe, aby wskazać konkretny element tablicy. Na przykład i-ty element tablicy A oznaczamy jako $A[i]$.

Tablice w komputerach mają jeszcze inną ważną cechę — dostęp do każdego elementu tablicy zabiera tyle samo czasu. Komputer, któremu podasz indeks i do

[2] Ang. *inputs*. Dla skrócenia dane wejściowe będziemy nazywać też **wejściem**, a jeśli nie będzie to prowadziło do nieporozumień — po prostu **danymi** — *przyp. tłum.*

[3] Jeśli programujesz w Javie, C lub C++, to masz w nawyku zaczynanie tablic od 0. Zaczynanie tablic od 0 jest przyjemne dla komputerów, lecz dla wetware'u często bardziej intuicyjnie jest zaczynać od 1.

tablicy, może sięgnąć po *i*-ty element równie szybko jak po pierwszy, niezależnie od wartości *i*.

Spójrzmy zatem na nasz pierwszy algorytm: poszukiwanie elementu w tablicy. Mamy daną tablicę i chcemy się dowiedzieć, który jej element — jeśli w ogóle taki istnieje — ma zadaną wartość. Żeby zrozumieć, jak można przeszukać tablicę, pomyślmy o tablicy jak o długiej półce pełnej książek. Załóżmy, że chcesz sprawdzić, czy i gdzie na półce stoi książka Jonathana Swifta. Zwróćmy uwagę, że książki na półce mogą być ułożone w pewien sposób, być może alfabetycznie według autorów, alfabetycznie według tytułów lub, jak w bibliotece, według numerów katalogowych. Jest też możliwe, że na półce, tak jak u mnie w domu, książki nie stoją w żadnym konkretnym porządku.

Jeśli nie można założyć, że książki na półce są zorganizowane, to jak się zabrać do poszukiwania książki Jonathana Swifta? Oto algorytm, według którego bym postępował. Zacząłbym od lewego końca półki i spojrzałbym na książkę stojącą na samym skraju. Jeśli byłby to Swift, byłoby po robocie. W przeciwnym razie spojrzałbym w prawo, na następną książkę, i gdyby się okazało, że to jest książka Swifta, miałbym ją znalezioną. Jeżeli nie, szedłbym dalej w prawo, sprawdzając książkę za książką, aż znalazłbym książkę Swifta lub wyszedłbym poza półkę — wówczas mógłbym stwierdzić, że na półce nie ma żadnej książki Jonathana Swifta. (W rozdziale 3 zobaczymy, jak poszukiwać książki, kiedy *są* one na półce zorganizowane).

A oto jak opisujemy ten problem wyszukiwania w na sposób informatyczny. Pomyślmy o książkach na półce jak o tablicy książek. Pierwsza książka z lewego brzegu jest na pozycji 1, następna książka po jej prawej stronie jest na pozycji 2 itd. Jeśli na półce mamy n książek, to książka z prawego brzegu występuje na pozycji n. Chcemy znaleźć na półce numer pozycji z książką Jonathana Swifta.

Spoglądając na to jak na ogólny problem obliczeniowy, możemy powiedzieć, że mamy daną tablicę A (zapełniona książkami półka do przeszukania) mającą n elementów (poszczególne książki) i chcemy się dowiedzieć, czy wartość x (książka Jonathana Swifta) występuje w tablicy A. Jeżeli występuje, to chcemy określić taki indeks i, że $A[i] = x$ (i-ta pozycja na półce zawiera książkę Jonathana Swifta). Potrzebujemy również jakiegoś sposobu, żeby wskazać, że tablica A nie zawiera x (półka nie zawiera żadnych książek Jonathana Swifta). Nie zakładamy, że x pojawia się w tablicy tylko raz (możesz mieć kilka egzemplarzy pewnej książki), wobec tego, jeśli x występuje w tablicy A, to może występować wielokrotnie. Od algorytmu wyszukiwania żądamy tylko podania *dowolnego* indeksu, pod którym znajdujemy x w tablicy. Załóżmy, że indeksy tej tablicy zaczynają się od 1, zatem jej elementy ciągną się od $A[1]$ do $A[n]$.

Jeśli wyszukiwanie książki Jonathana Swifta rozpoczynamy od lewego końca, sprawdzając książkę po książce i przesuwając się w prawo, to sposób taki nazywamy **wyszukiwaniem liniowym** (ang. *linear search*). W rozumieniu tablicy w komputerze zaczynamy od początku tablicy, sprawdzamy kolejno każdy element ($A[1]$, potem $A[2]$, potem $A[3]$ i tak dalej, aż do $A[n]$) i odnotowujemy, gdzie znaleźliśmy x, jeśli je w ogóle znajdziemy.

Następująca procedura WYSZUKIWANIE-LINIOWE przyjmuje trzy parametry, które w jej specyfikacji oddzielamy przecinkami:

Procedura WYSZUKIWANIE-LINIOWE(*A*, *n*, *x*)

Dane wejściowe:

- *A*: tablica,
- *n*: liczba elementów w *A* do przeszukania,
- *x*: poszukiwana wartość.

Wynik: Indeks *i*, dla którego *A*[*i*] = *x*, albo specjalna wartość NIE-ZNALEZIONO, którą może być dowolny niedozwolony indeks tablicy, np. 0 lub ujemna liczba całkowita.

1. Ustaw *odpowiedź* na wartość NIE-ZNALEZIONO.
2. Dla każdego indeksu *i*, przechodząc kolejno od 1 do *n*:
   A. Jeśli *A*[*i*] = *x*, to ustaw *odpowiedź* na wartość *i*.
3. Zwróć jako wynik wartość *odpowiedź*.

Oprócz parametrów *A*, *n* i *x*, procedura WYSZUKIWANIE-LINIOWE używa **zmiennej** (ang. *variable*) o nazwie *odpowiedź*. W pierwszym kroku procedura **przypisuje** (ang. *assigns*) zmiennej *odpowiedź* wartość początkową NIE-ZNALEZIONO. W kroku drugim każdy wpis tablicy, od *A*[1] do *A*[*n*], jest sprawdzany, czy nie zawiera wartości *x*. Ilekroć wpis *A*[*i*] równa się *x*, w kroku 2A przypisuje się *odpowiedzi* bieżącą wartość *i*. Jeśli *x* występuje w tablicy, to wartością wyjściową w kroku 3 jest ostatni indeks, pod którym *x* wystąpiło. Jeśli *x* nie ma w tablicy, to równość sprawdzana w kroku 2A nie zachodzi i jako wartość wyjściową zwraca się NIE-ZNALEZIONO, przypisane odpowiedzi w kroku 1.

Nim przejdziemy do dalszego omawiania wyszukiwania liniowego, powiemy kilka słów o tym, jak się określa czynności powtarzane, takie jak w kroku 2. Powtarzanie pewnego działania dla zmiennej przyjmującej wartości z pewnego przedziału jest w algorytmach dość typowe. Gdy wykonujemy działania powtarzane, nazywamy taki zabieg **pętlą** (ang. *loop*), a każde powtórzenie w pętli określamy mianem jej **iteracji**. Pętlę w kroku 2 zapisałem: „Dla każdego indeksu *i*, przechodząc kolejno od 1 do *n*". Zamiast tego, poczynając od teraz, będę pisać: „Dla *i* = 1 do *n*", co jest krótsze, a wyraża tę samą strukturę. Zauważmy, że gdy zapisujemy pętlę w ten sposób, musimy nadać **zmiennej pętli** (ang. *loop variable*) — tutaj *i* — wartość początkową (tutaj: 1), a w każdej iteracji pętli musimy porównywać bieżącą wartość zmiennej pętli z jej granicą (tu: *n*). Jeśli bieżąca wartość zmiennej pętli jest mniejsza lub równa granicy, to wykonujemy wszystko, co jest w pętli zawarte (jej **treść**, ang. *body*) — w naszym przypadku krok 2A. Po powtórzeniu treści pętli **zwiększamy** (ang. *increment*) zmienną pętli, dodając do niej 1, i ponownie przechodzimy do porównania zmiennej pętli — teraz już mającej nową wartość — z granicą. Powta-

rzamy wielokrotnie porównanie zmiennej pętli z granicą, wykonanie treści pętli i zwiększanie zmiennej pętli — aż zmienna pętli przekroczy granicę. Wykonywanie jest wtedy kontynuowane od kroku następującego bezpośrednio po treści pętli (tutaj krok 3). Pętla postaci „Dla $i = 1$ do n" wykonuje n iteracji i $n + 1$ porównań z granicą (ponieważ zmienna pętli przekracza granicę w sprawdzeniu numer $n + 1$).

Mam nadzieję, że jest dla Ciebie oczywiste, iż procedura WYSZUKIWANIE-LINIOWE zawsze zwraca poprawną odpowiedź. Może jednak rzuciło Ci się w oczy, że jest to procedura nieekonomiczna: kontynuuje wyszukiwanie tablicy nawet po znalezieniu indeksu i, dla którego $A[i] = x$. Zwykle nie ma powodu, aby kontynuować poszukiwania egzemplarza książki, gdy się już znalazło jeden na półce — prawda? Wobec tego możemy zaprojektować procedurę wyszukiwania liniowego tak, aby kończyła poszukiwania z chwilą znalezienia wartości x w tablicy. Zakładamy, że jeśli mówimy: zwróć wartość, to procedura natychmiast zwraca ją wywołującemu, który przejmuje dalszą kontrolę.

Procedura LEPSZE-WYSZUKIWANIE-LINIOWE(A, n, x)

Dane wejściowe i wynik: takie same jak w Wyszukiwaniu-Liniowym.

1. Dla $i = 1$ do n:
   A. Jeśli $A[i] = x$, to zwróć wartość i jako wynik.
2. Zwróć jako wynik NIE-ZNALEZIONO.

Możesz wierzyć lub nie, lecz potrafimy wyszukiwanie liniowe usprawnić jeszcze bardziej. Zauważmy, że w każdym przejściu przez pętlę w kroku 1 procedura LEPSZE-WYSZUKIWANIE-LINIOWE wykonuje dwa sprawdzenia: test w kroku 1, czy $i \leq n$ (i jeśli tak, to wykonaj następną iterację pętli), i sprawdzian równości w kroku 1A. W kategoriach przeszukiwania półki te testy odpowiadają konieczności wykonania dla każdej książki dwóch sprawdzeń: czy wyszliśmy poza koniec półki, a jeśli nie, to czy następna książka jest autorstwa Jonathana Swifta. Oczywiście nie zapłacisz zbyt wielkiej ceny za wyjście poza koniec półki (chyba że trzymasz nos naprawdę blisko grzbietów książek podczas ich przeglądania i walniesz głową w ścianę), lecz w programie komputerowym przy sięganiu po elementy próba wyjścia poza koniec tablicy na ogół kończy się bardzo źle. Twój program mógłby ulec załamaniu lub uszkodzić dane.

Można zrobić tak, aby w odniesieniu do każdej książki było potrzebne tylko jedno sprawdzenie. Gdyby tak wiedzieć na pewno, że na Twojej półce stoi książka Jonathana Swifta? Miałoby się wtedy pewność, że się ją znajdzie, nie trzeba by więc sprawdzać, czy półka się skończyła. Można by po prostu sprawdzać kolejno każdą książkę, czy jest napisana przez Swifta.

Mogłoby się jednak zdarzyć, że wypożyczyłeś wszystkie książki Jonathana Swifta lub myślałeś, że masz coś jego pióra, ale tego nie sprawdziłeś, więc nie możesz mieć pewności, że stoją na półce. Oto, co możesz zrobić w takiej sytuacji. Weź puste pudełko o rozmiarach książki i napisz na jego wąskim boku (tam, gdzie powinien

być grzbiet książki): *Podróże Guliwera* Jonathana Swifta. Zastąp tym pudełkiem książkę stojącą z prawego brzegu. Odtąd, przeszukując półkę od lewej do prawej, musisz sprawdzać tylko, czy patrzysz na coś napisanego przez Swifta; możesz zapomnieć o sprawdzaniu, czy wychodzisz poza koniec półki, ponieważ *wiesz*, że znajdziesz coś Swifta. Pozostaje tylko pytanie, czy znalazłeś naprawdę książkę Swifta, czy tylko jej atrapę, czyli puste pudełko opatrzone jego nazwiskiem. Jeśli znalazłeś puste pudełko, to nie masz książki Swifta naprawdę. Łatwo to jednak sprawdzić i musisz to zrobić tylko raz, na końcu poszukiwania, a nie przy każdej książce na półce.

Jest jeszcze jeden szczegół, na który trzeba zwrócić uwagę: co się stanie, gdy na półce jedyną książką Swifta była ta, która stała z brzegu po prawej stronie? Jeśli zamienisz ją na puste pudełko, to Twoje poszukiwania zakończą się na pustym pudełku i mógłbyś dojść do wniosku, że nie masz takiej książki. Musisz więc wykonać jeszcze jedno sprawdzenie, tej właśnie okoliczności, lecz jest to tylko jedno sprawdzenie, a nie jedno sprawdzenie dla każdej książki na półce.

Używając terminów algorytmu komputerowego, umieścimy poszukiwaną wartość x na ostatniej pozycji, $A[n]$, przechowawszy uprzednio zawartość $A[n]$ w innej zmiennej. Po znalezieniu x sprawdzamy, czy *rzeczywiście* ją znaleźliśmy. Tę umieszczaną w tablicy wartość nazywamy **wartownikiem** (ang. *sentinel*), lecz możesz o niej myśleć, jak o pustym pudełku.

Procedura WYSZUKIWANIE-LINIOWE-Z-WARTOWNIKIEM(A, n, x)

Wejście i wyjście: takie same jak w WYSZUKIWANIU-LINIOWYM.

1. Przechowaj $A[n]$ w zmiennej *ostatni*, po czym podstaw x do $A[n]$.
2. Ustaw i na 1.
3. Dopóki $A[i] \neq x$, wykonuj, co następuje:
   A. Zwiększ i.
4. Odtwórz $A[n]$ z *ostatniego*.
5. Jeśli $i < n$ lub $A[n] = x$, to zwróć wartość i jako wynik.
6. W przeciwnym razie zwróć jako wynik NIE-ZNALEZIONO.

Krok 3 jest pętlą, lecz nie chodzi w niej o założenie licznika w zmiennej pętli, o nie! Pętla ta jest powtarzana tak długo, jak długo jest spełniony warunek, w tym wypadku warunek $A[i] \neq x$. Pętlę taką interpretujemy jako wykonującą sprawdzenie (tu: czy $A[i] \neq x$) i jeśli sprawdzany warunek jest spełniony, to wykonujemy wszystko, co jest zawarte w jej treści (tu: krok 3A, którym jest zwiększenie i). Następnie wracamy, wykonujemy sprawdzenie i jeśli test jest pomyślny, wykonujemy treść. Kontynuujemy to postępowanie, wykonując testy i powtarzając treść pętli, aż test ujawni wartość „fałsz". Wtedy kontynuujemy od następnego kroku po treści pętli (tutaj: przechodzimy do kroku 4).

Procedura WYSZUKIWANIE-LINIOWE-Z-WARTOWNIKIEM jest nieco bardziej skomplikowana niż dwie pierwsze procedury wyszukiwania liniowego. Dzięki temu, że w pierwszym kroku umieszcza x w $A[n]$, mamy zagwarantowane, iż $A[i]$ będzie równe x w którymś ze sprawdzeń w kroku 3. Kiedy to nastąpi, opuszczamy pętlę z kroku 3, więc indeks i już się nie zmieni. Zanim cokolwiek zrobimy, w kroku 4 następuje odtworzenie oryginalnej wartości w $A[n]$. (Moja matka uczyła mnie, żeby odkładać rzeczy na swoje miejsce po użyciu). Następnie musimy ustalić, czy naprawdę znaleźliśmy x w tablicy. Ponieważ podstawiliśmy x pod ostatni element $A[n]$, wiemy, że jeśli znaleźliśmy x w $A[i]$, przy czym $i < n$, to naprawdę znaleźliśmy x i należy zwrócić indeks i. A co, jeśli znaleźliśmy x w $A[n]$? Oznacza to, że nie znaleźliśmy x przed dojściem do $A[n]$, musimy więc ustalić, czy w $A[n]$ rzeczywiście było x. Jeśli tak, to zwracamy indeks n równy i w tym momencie, lecz jeśli nie, to musimy zwrócić NIE-ZNALEZIONO. Te sprawdzenia są wykonywane w kroku 5, w którym zwraca się również poprawny indeks, jeśli x było pierwotnie w tablicy. Jeśli x znaleziono tylko dlatego, że w kroku 1 włożono je do tablicy, to krok 6 zwraca NIE-ZNALEZIONO. Chociaż w WYSZUKIWANIU-LINIOWYM-Z-WARTOWNIKIEM trzeba wykonać dwa sprawdzenia po zakończeniu pętli, wykonuje ono tylko po jednym teście w każdej iteracji pętli, co czyni je sprawniejszym niż WYSZUKIWANIE-LINIOWE i LEPSZE-WYSZUKIWANIE-LINIOWE.

Jak charakteryzować czasy działania

Wróćmy do procedury WYSZUKIWANIE-LINIOWE i zastanówmy się nad jej czasem działania. Przypomnijmy, że określamy czas działania jako funkcję rozmiaru danych wejściowych. Naszymi danymi są tutaj tablica n-elementowa wraz z liczbą n i wartość x, której poszukujemy. Rozmiary n i x są bez znaczenia, gdy tablica staje się duża — w końcu n jest tylko jedną liczbą całkowitą, a wielkość x to tyle, co jeden element n-elementowej tablicy — powiemy więc, że rozmiar wejścia wynosi n, czyli jest równy liczbie elementów w A.

Musimy poczynić kilka prostych założeń co do długości. Załóżmy, że każda pojedyncza operacja — czy to operacja arytmetyczna (jak dodawanie, odejmowanie, mnożenie lub dzielenie), czy porównanie, podstawienie zmiennej, indeksowanie tablicy albo wywołanie lub powrót z procedury — zabiera pewną stałą porcję czasu, która nie zależy od rozmiaru danych wejściowych[4]. Czas może się zmieniać od operacji do operacji, tak więc dzielenie może trwać dłużej niż dodawanie, lecz jeśli krok zawiera tylko pojedyncze operacje, to każde poszczególne wykonanie tego kroku zajmuje pewną stałą ilość czasu. Ponieważ wykonywane operacje różnią się

[4] Jeśli orientujesz się trochę w rzeczywistej architekturze komputerów, to pewnie wiesz, że czas dostępu do określonej zmiennej lub elementu tablicy niekoniecznie jest stały, gdyż może zależeć od tego, czy zmienna lub element tablicy znajdują się w pamięci podręcznej, w pamięci operacyjnej, czy na dysku, w systemie pamięci wirtualnej. W niektórych wyrafinowanych modelach komputerów jest to brane pod uwagę, lecz często wystarczy po prostu przyjąć, że wszystkie zmienne i wpisy tablic są w pamięci operacyjnej i dostęp do nich zajmuje tyle samo czasu.

w poszczególnych krokach, a także ze względu na czynniki zewnętrzne wymienione wcześniej, czas działania poszczególnych kroków może się zmieniać. Powiedzmy, że każde wykonanie kroku *i* zajmuje t_i czasu, gdzie t_i jest pewną stałą, która nie zależy od *n*.

Oczywiście musimy uwzględnić fakt, że niektóre kroki są wykonywane wielokrotnie. Kroki 1 i 3 są wykonywane tylko raz, lecz jak to jest w przypadku kroku 2? Musimy porównywać *i* z *n* łącznie *n* + 1 razy: *n* razy, kiedy $i \le n$, i raz, gdy *i* równa się *n* + 1, po czym opuszczamy pętlę. Krok 2A jest wykonywany dokładnie *n* razy — po razie dla każdej wartości *i* od 1 do *n*. Nie wiemy z góry, ile razy podstawimy do zmiennej *odpowiedź* wartość *i*; może to być dowolna krotność, poczynając od 0 (jeśli *x* nie występuje w tablicy), a kończąc na *n* (gdy każda wartość w tablicy równa się *x*). Jeśli chcemy być precyzyjni w naszych obliczeniach — a zazwyczaj nie jesteśmy tak dokładni — to musimy zauważyć, że w kroku 2 są wykonywane dwie różne rzeczy, powtarzane różną liczbę razy: porównywanie *i* z *n* ma miejsce *n* + 1 razy, lecz zwiększanie *i* odbywa się tylko *n* razy. Rozbijmy czas dotyczący wiersza 2 na czas porównywania t_2' i czas zwiększania t_2''. Podobnie rozdzielmy czas w kroku 2A na t_{2A}' (sprawdzanie, czy $A[i] = x$) i t_{2A}'' (nadawanie zmiennej *odpowiedź* wartości *i*). Wobec tego czas działania WYSZUKIWANIA-LINIOWEGO będzie się zawierał gdzieś między

$$t_1 + t_2' \cdot (n+1) + t_2'' \cdot n + t_{2A}' \cdot n + t_{2A}'' \cdot 0 + t_3$$

a

$$t_1 + t_2' \cdot (n+1) + t_2'' \cdot n + t_{2A}' \cdot n + t_{2A}'' \cdot n + t_3 .$$

Przepiszemy teraz te ograniczenia, grupując składniki mnożone przez *n* i zbierając resztę składników, co uwidoczni, że czas działania mieści się gdzieś między

$$(t_2' + t_2'' + t_{2A}') \cdot n + (t_1 + t_2' + t_3)$$

a

$$(t_2' + t_2'' + t_{2A}' + t_{2A}'') \cdot n + (t_1 + t_2' + t_3) .$$

Zauważmy, że każde z tych ograniczeń ma postać $c \cdot n + d$, gdzie *c* i *d* są stałymi niezależnymi od *n*. To znaczy oba są *funkcjami liniowymi* zmiennej *n*. Czas działania WYSZUKIWANIA-LINIOWEGO jest ograniczony z dołu przez funkcję liniową zmiennej *n* i ograniczony z góry przez funkcję liniową zmiennej *n*.

Do wskazania, że czas działania jest ograniczony z góry przez pewną funkcję liniową zmiennej *n* i z dołu przez pewną (być może inną) funkcję liniową zmiennej *n*, stosujemy specjalną notację. Zapisujemy, że czas działania wynosi $\Theta(n)$. To znaczy używamy greckiej litery Θ i mówimy „teta od *n*" lub „teta *n*". Zgodnie z obietnicą złożoną w rozdziale 1, w notacji tej zaniedbuje się składniki niskiego rzędu $(t_1 + t_2' + t_3)$ i współczynniki przy *n* ($t_2' + t_2'' + t_{2A}'$ dla dolnego ograniczenia i $t_2' + t_2'' + t_{2A}' + t_{2A}''$ dla ograniczenia górnego). Mimo że tracimy na dokładności, cha-

rakteryzując czas działania jako $\Theta(n)$, zyskujemy zalety polegające na zwiększeniu wyrazistości rzędu rośnięcia czasu działania i pominięciu uciążliwych szczegółów.

Ową notację Θ stosuje się do funkcji w sposób ogólny, nie tylko do tych, które opisują czasy działania algorytmów, i nie tylko do funkcji liniowych. Pomysł polega na tym, że jeśli mamy dwie funkcje, $f(n)$ i $g(n)$, to mówimy, że $f(n)$ jest $\Theta(g(n))$, jeżeli dla dostatecznie dużych n funkcja $f(n)$ różni się od $g(n)$ stałym czynnikiem. Możemy więc powiedzieć, że czas działania WYSZUKIWANIA-LINIOWEGO różni się od n stałym czynnikiem, gdy n staje się dostatecznie duże.

Istnieje przytłaczająca, fachowa definicja notacji Θ, lecz na szczęście rzadko musimy się do niej odwoływać, aby używać notacji Θ. Skupiamy się po prostu na dominującym składniku, odrzucając składniki niskiego rzędu i stałe czynniki. Na przykład funkcja $n^2/4 + 100n + 50$ jest $\Theta(n^2)$; odrzucamy w niej składniki niskiego rzędu $100n$ i 50 i pozbywamy się stałego czynnika 1/4. Chociaż składniki niskiego rzędu dominują $n^2/4$ dla małych wartości n, z chwilą gdy n przekroczy 400, składnik $n^2/4$ wynosi 250 000, natomiast składniki niskiego rzędu $100n + 50$ wynoszą tylko 100 050. Dla $n = 2000$ różnica przedstawia się jak 1 000 000 wobec 200 050. W świecie algorytmów nieco nadużywamy notacji i piszemy $f(n) = \Theta(g(n))$, tak że możemy napisać $n^2/4 + 100n + 50 = \Theta(n^2)$.

Spójrzmy teraz na czas działania LEPSZEGO-WYSZUKIWANIA-LINIOWEGO. Ten jest trochę bardziej skomplikowany niż WYSZUKIWANIE-LINIOWE, gdyż nie wiemy z góry, ile razy zostanie powtórzona treść pętli. Jeśli $A[1]$ równa się x, to zostanie ona wykonana tylko raz. Jeśli x nie występuje w tablicy, to pętla będzie iterowana pełne n razy, czyli maksymalną możliwą liczbę. Każde powtórzenie treści pętli zabiera pewną stałą ilość czasu, możemy zatem powiedzieć, że *w przypadku najgorszym* LEPSZE-WYSZUKIWANIE-LINIOWE zajmuje $\Theta(n)$ czasu na przeszukanie tablicy n elementów. Dlaczego „w najgorszym przypadku"? Skoro dążymy do tego, żeby algorytmy miały krótkie czasy działania, przypadek najgorszy to takie dane wejściowe spośród wszystkich możliwych, dla którym algorytm zużywa najwięcej czasu.

W przypadku najlepszym, kiedy $A[1]$ równa się x, LEPSZE-WYSZUKIWANIE-LINIOWE zajmuje tylko stałą ilość czasu: nadaje i wartość 1, sprawdza, że $i \leq n$, test $A[i] = x$ okazuje się prawdziwy i procedura zwraca wartość i, czyli 1. Ta ilość czasu nie zależy od n. Piszemy, że *czas działania w przypadku najlepszym* algorytmu LEPSZE-WYSZUKIWANIE-LINIOWE wynosi $\Theta(1)$, ponieważ w najlepszym przypadku czas działania różni się stałym czynnikiem od 1. Inaczej mówiąc, czas działania w przypadku najlepszym jest stałą, która nie zależy od n.

Widzimy przeto, że notacji Θ nie możemy stosować w całościowym twierdzeniu, obejmującym wszystkie przypadki czasu działania LEPSZEGO-WYSZUKIWANIA-LINIOWEGO. Nie możemy powiedzieć, że jego czas działania zawsze wynosi $\Theta(n)$, ponieważ w najlepszym przypadku wynosi $\Theta(1)$. Nie możemy też powiedzieć, że jest zawsze równy $\Theta(1)$, ponieważ w przypadku najgorszym wynosi $\Theta(n)$. Możemy natomiast powiedzieć, że funkcja liniowa zmiennej n jest *górnym ograniczeniem* we wszystkich przypadkach, i mamy na to sposób zapisu: $O(n)$. Chcąc wysłowić tę notację, mówimy: „O duże od n" lub po prostu „O od n". Funkcja $f(n)$ jest $O(g(n))$, jeśli

dla dostatecznie dużego n jest ograniczona z góry przez pewną stałą przemnożoną przez $g(n)$. Znowu trochę nadużywamy notacji i piszemy $f(n) = O(g(n))$. Dla LEPSZEGO-WYSZUKIWANIA-LINIOWEGO możemy wypowiedzieć całościowe twierdzenie, że jego czas działania jest we wszystkich przypadkach $O(n)$; aczkolwiek czas ten może być lepszy niż liniowa funkcja zmiennej n, gorszym nie jest nigdy.

Notacji O używamy żeby pokazać, iż czas działania nigdy nie jest *gorszy* od stałej przemnożonej przez pewną funkcję zmiennej n. Jak jednak wskazać, że czas działania nigdy nie jest *lepszy* od stałej przemnożonej przez pewną funkcję zmiennej n? Jest to dolne ograniczenie i używamy tu notacji Ω, zwierciadlanego odbicia notacji O: funkcja $f(n)$ jest $\Omega(g(n))$, jeśli z chwilą gdy n stanie się dostatecznie duże, $f(n)$ jest ograniczone z dołu przez pewną stałą mnożoną przez $g(n)$. Mówimy, że „$f(n)$ jest omega duże od $g(n)$" lub po prostu „$f(n)$ jest omega od $g(n)$" i możemy zapisać $f(n) = \Omega(g(n))$. Ponieważ notacja O daje górne ograniczenie, notacja Ω daje ograniczenie dolne, a notacja Θ daje zarówno ograniczenie górne, jak i dolne, możemy podsumować, że funkcja $f(n)$ jest $\Theta(g(n))$ wtedy i tylko wtedy, kiedy $f(n)$ jest zarówno $O(g(n))$, jak i $\Omega(g(n))$.

Potrafimy sformułować całościowe twierdzenie o dolnym ograniczeniu czasu działania LEPSZEGO-WYSZUKIWANIA-LINIOWEGO: we wszystkich przypadkach wynosi ono $\Omega(1)$. Oczywiście jest to żałośnie słabe stwierdzenie, gdyż spodziewamy się, że dowolny algorytm dla dowolnych danych wejściowych zużyje przynajmniej stałą ilość czasu. Notacji Ω nie używamy zbyt często, choć niekiedy jest przydatna.

Wspólnym terminem na określenie notacji Θ, O i Ω jest określenie **notacja asymptotyczna**. Określenie jest trafne, ponieważ notacje te ujmują wzrost funkcji, gdy jej argument asymptotycznie dąży do nieskończoności. Wszystkie te asymptotyczne notacje dają nam luksus zaniedbywania składników niskiego rzędu i stałych czynników, możemy więc pomijać żmudne detale i skupiać się na tym, co ważne: jak funkcja rośnie z n.

Powróćmy obecnie do WYSZUKIWANIA-LINIOWEGO-Z-WARTOWNIKIEM. Podobnie jak LEPSZE-WYSZUKIWANIE-LINIOWE, każda iteracja jego pętli zajmuje stałą ilość czasu, a iteracji tych może być od 1 do n. Zasadnicza różnica między WYSZUKIWANIEM-LINIOWYM-Z-WARTOWNIKIEM a LEPSZYM-WYSZUKIWANIEM-LINIOWYM polega na tym, że czas przypadający na jedną iterację w WYSZUKIWANIU-LINIOWYM-Z-WARTOWNIKIEM jest mniejszy niż czas jednej iteracji w LEPSZYM-WYSZUKIWANIU-LINIOWYM. Oba zużywają liniową ilość czasu w przypadku najgorszym, lecz czynnik stały w WYSZUKIWANIU-LINIOWYM-Z-WARTOWNIKIEM jest lepszy. Choć możemy się spodziewać, że WYSZUKIWANIE-LINIOWE-Z-WARTOWNIKIEM będzie szybsze w praktyce, będzie to tylko kwestia stałego czynnika. Gdy wyrażamy czasy działania LEPSZEGO-WYSZUKIWANIA-LINIOWEGO i WYSZUKIWANIA-LINIOWEGO-Z-WARTOWNIKIEM, używając notacji asymptotycznej, są one równoważne: $\Theta(n)$ w najgorszym przypadku, $\Theta(1)$ w najlepszym i $O(n)$ we wszystkich przypadkach.

Niezmienniki pętli

We wszystkich trzech odmianach naszego wyszukiwania liniowego łatwo było zauważyć, że każda z nich daje poprawną odpowiedź. Niekiedy bywa trochę trudniej. Istnieje w tej mierze dużo różnych sposobów, więcej niż zdołam tu przedstawić. W jednej z typowych metod wykazywania poprawności stosuje się **niezmiennik pętli** (ang. *loop invariant*), asercję, w przypadku której udowadniamy, że jest prawdziwa za każdym razem, gdy rozpoczynamy iterowanie pętli. Aby niezmiennik pętli był pomocny w dowodzeniu poprawności, musimy wykazać, że ma trzy cechy:

- **Inicjowanie:** musi być prawdziwy przed pierwszą iteracją pętli.
- **Utrzymanie:** jeśli jest prawdziwy przed pewną iteracją pętli, pozostaje prawdziwy przed następną iteracją.
- **Zakończenie:** pętla się kończy i gdy to nastąpi, niezmiennik pętli, wraz z przyczyną zakończenia pętli, niesie jakąś użyteczną cechę.

  Za przykład niech tu posłuży niezmiennik LEPSZEGO-WYSZUKIWANIA-LINIOWEGO:

  Na początku każdego powtórzenia kroku 1, jeśli x występuje w tablicy A, to występuje w **podtablicy** (ciągłym fragmencie tablicy) od $A[i]$ do $A[n]$.

Nie żądamy nawet, by ten niezmiennik pętli wykazywał, że jeśli procedura zwraca indeks inny niż NIE-ZNALEZIONO, to zwracany indeks jest poprawny — procedura może zwrócić w kroku 1A indeks i tylko wówczas, gdy x równa się $A[i]$. Użyjemy natomiast tego niezmiennika pętli do pokazania, że jeśli procedura zwraca NIE-ZNALEZIONO w kroku 2, to x nie ma w tablicy:

- **Inicjowanie**: na początku $i = 1$, więc podtablica w niezmienniku pętli rozciąga się od $A[1]$ do $A[n]$, czyli jest całą tablicą.
- **Utrzymanie**: załóżmy, że jeśli na początku iteracji dla wartości i element x występuje w tablicy A, to jest obecny w podtablicy ciągnącej się od $A[i]$ do $A[n]$. Jeśli przejdziemy tę iterację bez zwracania, to wiemy, że $A[i] \neq x$, dlatego z całą pewnością możemy stwierdzić, że jeśli x występuje w A, to występuje w podtablicy od $A[i+1]$ do $A[n]$. Ponieważ i jest zwiększane przed następną iteracją, niezmiennik pętli będzie przed tą następną iteracją spełniony.
- **Zakończenie**: ta pętla musi się zakończyć — albo z powodu powrotu z procedury w kroku 1A, albo dlatego, że $i > n$. Rozpatrzyliśmy już przypadek, w którym pętla kończy się z powodu powrotu z procedury w kroku 1A.

  Aby rozpatrzyć przypadek, w którym pętla kończy się z powodu $i > n$, uciekamy się do antytezy niezmiennika pętli. Antytezą stwierdzenia „jeśli A, to B" jest „jeśli nie B, to nie A". Antyteza stwierdzenia jest prawdziwa wtedy i tylko wtedy, gdy stwierdzenie jest prawdziwe. Antytezą naszego niezmiennika pętli jest: „jeśli x nie występuje w podtablicy rozciągającej się od $A[i]$ do $A[n]$, to nie ma go w tablicy A".

  Rozważmy sytuację, w której $i > n$. Podtablica od $A[i]$ do $A[n]$ jest wtedy pusta, nie może więc zawierać x. Z antytezy niezmiennika pętli wynika zatem, że x nie

występuje na żadnej pozycji w A, a to jest stosowna okoliczność, aby zwrócić NIE-ZNALEZIONO w kroku 2.

No proszę, ile wnioskowania dla takiej prościutkiej pętli! Czy musimy się tak przebijać za każdym razem, gdy piszemy pętlę? Ja nie, lecz jest paru informatyków, którzy domagają się tak ścisłego wnioskowania dla każdej pętli z osobna. Kiedy piszę prawdziwy kod, mam poczucie, że podczas pisania pętli prawie cały czas kołacze mi w głowie myśl o niezmienniku pętli. Może ona tkwić tak głęboko, że nawet sobie jej nie uświadamiam, lecz gdyby było trzeba, potrafiłbym ją wysłowić. Choć większość z nas zgodziłaby się, że niezmiennik pętli jest zabójczy dla zrozumienia prostej pętli w LEPSZYM-WYSZUKIWANIU-LINIOWYM, niezmienniki pętli są dość poręczne, gdy chcemy zrozumieć, dlaczego bardziej skomplikowane pętle działają należycie.

Rekursja

Za pomocą techniki **rekursji** (ang. *recursion*) rozwiązujemy problem przez rozwiązanie mniejszych egzemplarzy (ukonkretnień) tego samego problemu. Oto mój ulubiony przykład rekursji: obliczanie $n!$ („n silnia"), funkcji określonej dla nieujemnych wartości n następująco: $n! = 1$, jeśli $n = 0$, i

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1,$$

jeśli $n \geq 1$. Na przykład $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Zauważmy, że

$$(n-1)! = (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1,$$

więc

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

dla $n \geq 1$. Zdefiniowaliśmy $n!$ przez odwołanie do „mniejszego" problemu, mianowicie $(n-1)!$. Rekurencyjną procedurę obliczania $n!$ moglibyśmy zapisać tak:

Procedura SILNIA(n)

Dane wejściowe: liczba całkowita $n \geq 0$.

Wynik: wartość $n!$.

1. Jeśli $n = 0$, to zwróć 1 jako wynik.
2. W przeciwnym razie zwróć n mnożone przez wartość zwróconą przez rekurencyjne wywołanie SILNIA($n - 1$).

Sposób, w jaki zapisałem krok 2, jest trochę zawiły. Mógłbym po prostu napisać „W przeciwnym razie zwróć $n \cdot$ SILNIA($n - 1$)", używając rekurencyjnego wywołania, zwracającego wartość w większym wyrażeniu arytmetycznym.

Aby rekursja działała, muszą być spełnione dwa warunki. Po pierwsze, musi istnieć jeden lub więcej **przypadków bazowych** (ang. *base cases*), w których rozwiązanie obliczamy bezpośrednio, bez użycia rekursji. Po drugie, każde rekurencyjne wywołanie procedury musi odnosić się do *mniejszego egzemplarza tego samego problemu*, co ostatecznie doprowadzi do przypadku bazowego. W procedurze SILNIA przypadek bazowy pojawia się, gdy $n = 0$, a każde rekurencyjne wywołanie dotyczy egzemplarza, w którym wartość n jest zmniejszona o 1. Jeśli tylko wartość n jest nieujemna, wywołania rekurencyjne będą schodziły do przypadku bazowego.

Uzasadnienie, że algorytm rekurencyjny działa poprawnie, może wydać się na pierwszy rzut oka banalne. Jego sednem jest przyjęcie, że każde rekurencyjne wywołanie wytwarza poprawny wynik. Jeśli tylko jesteśmy gotowi wierzyć, że wywołania rekurencyjne robią, co należy, uzasadnienie poprawności jest często łatwe. Oto jak możemy uzasadnić, że procedura SILNIA zwraca poprawną odpowiedź. Jest oczywiste, że dla $n = 0$ zwracana wartość, czyli 1, równa się $n!$. Zakładamy więc, że dla $n \geq 1$ wywołanie rekurencyjne SILNIA($n - 1$) działa poprawnie, tzn. zwraca wartość $(n - 1)!$. Procedura mnoży wtedy tę wartość przez n, wyliczając wartość $n!$, którą następnie zwraca.

A oto przykład, w którym wywołania rekurencyjne nie są mniejszymi egzemplarzami tego samego problemu, mimo że matematycznie wszystko jest w porządku. Jest niewątpliwie prawdą, że jeśli $n \geq 0$, to $n! = (n + 1)! / (n + 1)$. Jednak następująca procedura rekurencyjna, robiąca użytek z tego wzoru, zawiedzie przy próbie podania odpowiedzi dla $n \geq 1$:

Procedura ZŁA-SILNIA(n)

Wejście i wyjście: jak w procedurze SILNIA.

1. Jeśli $n = 0$, to zwróć 1 jako wynik.
2. W przeciwnym razie: ZŁA-SILNIA($n + 1$) / ($n + 1$).

Gdybyśmy wywołali procedurę ZŁA-SILNIA(1), wygenerowałaby ona rekurencyjne wywołanie ZŁA-SILNIA(2), które wygenerowałoby rekurencyjne wywołanie ZŁA-SILNIA(3) i tak dalej, nigdy więc nie nastąpiłoby zejście do przypadku bazowego, w którym $n = 0$. Gdybyśmy napisali tę procedurę w prawdziwym języku programowania i naprawdę wykonali ją na komputerze, szybko otrzymalibyśmy sygnał w rodzaju „stack overflow error”[5].

Niejednokrotnie algorytmy używające pętli potrafimy przepisać na postać rekurencyjną. Oto wyszukiwanie liniowe bez wartownika, zapisane rekurencyjnie:

[5] Przekroczenie rozmiaru stosu programowego, na którym odkłada się ślady kolejnych, rozpoczętych wywołań — *przyp. tłum.*

Procedura REKURENCYJNE-WYSZUKIWANIE-LINIOWE(A, n, i, x)

Dane wejściowe: jak w WYSZUKIWANIU-LINIOWYM, lecz z dodanym parametrem i.

Wynik: indeks elementu równego x w podtablicy od $A[i]$ do $A[n]$ lub NIE-ZNALEZIONO, jeśli x nie występuje w tej podtablicy.

1. Jeśli $i > n$, to zwróć NIE-ZNALEZIONO.
2. W przeciwnym razie ($i \leq n$), jeśli $A[i] = x$, to zwróć i.
3. W przeciwnym razie ($i \leq n$ i $A[i] \neq x$), zwróć REKURENCYJNE-WYSZUKIWANIE-LINIOWE(A, n, $i + 1$, x).

Podproblemem jest tutaj wyszukiwanie x w podtablicy rozciągającej się od $A[i]$ do $A[n]$. Przypadek bazowy występuje w kroku 1, kiedy podtablica jest pusta, to znaczy gdy $i > n$. Wartość i zwiększa się w każdym z rekurencyjnych wywołań w kroku 3, więc jeśli żadne rekurencyjne wywołanie nie zwróci wartości i w kroku 2, to w końcu i stanie się większe od n i osiągniemy przypadek bazowy.

Co czytać dalej

Rozdziały 2 i 3 w CLRS [CLRS09] ujmują większość materiału zawartego w tym rozdziale. Autorzy wczesnego podręcznika algorytmów: Aho, Hopcroft i Ullman [AHU73] przetarli szlak w stosowaniu notacji asymptotycznej w analizie algorytmów. Istnieje sporo prac poświęconych dowodzeniu poprawności programów; jeśli chcesz zgłębić ten obszar, zajrzyj do książek Griesa [Gri81] i Mitchella [Mit96].

3 Algorytmy sortowania i wyszukiwania

W rozdziale 2 zobaczyliśmy trzy odmiany liniowego przeszukiwania tablicy. Czy potrafilibyśmy zrobić to lepiej? Odpowiedź brzmi: to zależy. Jeśli nie wiemy niczego o uporządkowaniu elementów w tablicy, to nie — nie uda się zrobić tego lepiej. W najgorszym przypadku musimy przejrzeć wszystkie n elementów, bo jeśli nie znajdziemy poszukiwanej wartości pośród pierwszych $n - 1$ elementów, to nie można wykluczyć, że jest nią element n-ty, ostatni. Dlatego nie zdołamy osiągnąć w przypadku najgorszym czasu działania lepszego niż $\Theta(n)$, jeżeli nie wiemy nic o uporządkowaniu elementów.

Przypuśćmy jednak, że tablica jest posortowana w porządku niemalejącym: każdy element tablicy jest mniejszy lub równy następującemu po nim, stosownie do pewnej definicji relacji „jest mniejsze". W tym rozdziale przekonamy się, że jeśli tablica jest posortowana, to możemy skorzystać z prostego sposobu, zwanego wyszukiwaniem binarnym, aby przeszukać n-elementową tablicę w czasie zaledwie $O(\lg n)$. Jak widzieliśmy w rozdziale 1, wartość $\ln n$ rośnie bardzo wolno w porównaniu z n, toteż wyszukiwanie binarne bije na głowę wyszukiwanie liniowe w przypadku najgorszym[1].

Co to znaczy, że jeden element jest mniejszy od drugiego? Jeśli elementy są liczbami, sprawa jest oczywista. Jeśli elementy są napisami złożonymi ze znaków drukarskich, to możemy rozważyć **uporządkowanie leksykograficzne** (ang. *lexicographic ordering*): element jest mniejszy od drugiego elementu, jeśli występuje w słowniku przed tym drugim elementem. Jeżeli elementy są danymi jakiejś innej postaci, to musimy zdefiniować, co to znaczy „mniejszy niż". Kiedy już wyraźnie określimy pojęcie „mniejszy niż", możemy ustalić, czy tablica jest posortowana.

Powracając do przykładu z książkami na półce z rozdziału 2, moglibyśmy posortować książki alfabetycznie według nazwisk autorów, alfabetycznie według tytułów lub — gdyby chodziło o bibliotekę — według numerów katalogowych. W tym rozdziale powiemy, że książki na półce są posortowane, jeśli występują w porządku alfabetycznym według autorów, czytając od lewej do prawej. Półka może jednak zawierać więcej niż jedną książkę tego samego autora; możliwe, że masz kilka dzieł Williama Szekspira. Jeśli chcemy odnaleźć nie tylko dowolną książkę Szekspira, lecz konkretną książkę tego autora, to możemy powiedzieć, że jeśli dwie książki mają tego samego autora, to jedna z nich — ta, której tytuł jest alfabetycznie pierwszy — powinna stać z lewej. Moglibyśmy też powiedzieć, że interesuje nas tylko nazwisko autora, więc podczas wyszukiwania zadowoli nas cokolwiek, co zostało napisane przez Szekspira. Informacje, które dopasowujemy, nazywamy **kluczem** (ang. *key*). W naszym przykładzie z półką kluczem jest samo nazwisko autora, a nie kombinacja

[1] Jeśli jesteś nieobeznany z komputerami i ominąłeś podrozdział „Algorytmy komputerowe dla niekomputerowców" w rozdziale 1, to powinieneś przeczytać chociaż materiał o logarytmach z tamtego podrozdziału.

uwzględniająca najpierw autora, a potem tytuł, jeśli dwa dzieła mają tego samego autora.

Co zatem trzeba zrobić, aby najpierw posortować tablicę? W tym rozdziale poznamy cztery algorytmy: sortowanie przez wybieranie, sortowanie przez wstawianie, sortowanie przez scalanie i sortowanie szybkie, stosując każdy z nich w przykładzie z półką z książkami. Każdy algorytm sortowania ma zalety i wady, więc pod koniec rozdziału dokonamy przeglądu i porównania tych algorytmów. Wszystkie algorytmy sortowania, które obejrzymy w tym rozdziale, zajmują w najgorszym przypadku albo $\Theta(n^2)$, albo $\Theta(n \lg n)$ czasu. Dlatego gdybyś musiał wykonać tylko kilka przeszukań, lepiej daj sobie spokój i użyj wyszukiwania liniowego. Jeśli jednak trzeba będzie wyszukiwać wiele razy, postąpisz lepiej, wykonując najpierw sortowanie, a potem stosując wyszukiwanie binarne.

Problem sortowania jest ważny sam w sobie, nie tylko jako wstępny zabieg poprzedzający wyszukiwanie binarne. Pomyśl o tych wszystkich danych, które trzeba sortować: wpisy w książce telefonicznej — według nazwisk, comiesięczne wyciągi bankowe — według numerów i (lub) dat rachunków, czy choćby wyniki z wyszukiwarki sieciowej — według trafności dopasowania do zapytania. Co więcej, sortowanie jest często krokiem w innym algorytmie. Na przykład w grafice komputerowej obiekty są często uwarstwione, ułożone jeden nad drugim. Program obrazujący obiekty na ekranie („renderujący") może potrzebować sortowania obiektów według relacji „ponad", aby rysować je od dołu do góry.

Nim przejdziemy dalej, kilka słów o tym, czym jest to, co sortujemy. Oprócz klucza (który na użytek sortowania nazywamy **kluczem sortowania** (ang. *sort key*), sortowane elementy zawierają na ogół coś, co nazywamy **danymi towarzyszącymi** (satelickimi, ang. *satellite data*)[2]. Dane towarzyszące mogłyby co prawda pochodzić z satelity, zazwyczaj jednak tak nie jest. Dane towarzyszące to informacje, które są skojarzone z kluczem sortowania i powinny podróżować wraz z nim, gdy dochodzi do przemieszczania elementów. W przykładzie z książkami na półce kluczem sortowania jest nazwisko autora, a danymi towarzyszącymi — sama książka.

Moim studentom wyjaśniam, co to są dane towarzyszące, w sposób, który z pewnością do nich trafia. Trzymam arkusz z ocenami studenckimi, na którym wiersze są uporządkowane alfabetycznie według nazwisk studentów. Aby ustalić ostateczne wyniki na koniec semestru, sortuję wiersze według klucza, który stanowi kolumna z wyrażoną procentowo punktacją ich dorobku. Wtedy pozostałe kolumny, z nazwiskami studentów, stają się danymi towarzyszącymi. Sortuję punkty procentowe w porządku malejącym, tak więc góra wykazu odpowiada ocenom A (bardzo dobry), a na dole są D i E (trójki i dwójki)[3]. Rozważmy, co by się stało, gdybym przeorganizował tylko kolumnę z punktami procentowymi, nie ruszając całych wierszy z nimi. Nazwiska studentów pozostałyby wówczas w porządku alfabetycznym, bez związku

---

[2] Nazywane też po prostu **danymi dodatkowymi** — *przyp. tłum.*

[3] Na oznaczenie złych stopni w Dartmouth używa się liter E i F. Dlaczego? Dokładnie nie wiem, zgaduję jednak, że uprościło to program komputerowy, który zamienia oceny literowe na numeryczne.

z punktami procentowymi. Studenci o nazwiskach występujących wcześniej w alfabecie byliby uszczęśliwieni, natomiast studenci o nazwiskach z końca alfabetu — nie za bardzo.

Oto garść innych przykładów kluczy sortowania i danych towarzyszących. W książce telefonicznej kluczem sortowania mogłoby być nazwisko. Danymi towarzyszącymi byłyby wówczas adres i numer telefonu. W wyciągu bankowym kluczem sortowania mógłby być numer rachunku, a dane towarzyszące mogłyby zawierać kwotę rachunku i datę przelewu. W wyszukiwarce kluczem sortowania mogłaby być miara trafności dopasowania do zapytania, a danymi towarzyszącymi lokalizator URL strony w Sieci i inne dane, które wyszukiwarka zapamiętuje o stronie.

Pracując w tym rozdziale na tablicach, będziemy postępować tak, jakby każdy element zawierał tylko klucz sortowania. Gdybyś miał urzeczywistnić któryś z podanych tu algorytmów sortowania, musiałbyś się upewnić, że wraz z przenoszeniem klucza zawsze przenosisz dane stowarzyszone z każdym elementem, lub przynajmniej wskaźnik do danych towarzyszących.

Aby analogia z półką książek nadawała się do zastosowania w komputerze, musimy założyć, że półka i książki mają dwie dodatkowe właściwości, które — przyznaję — nie są zanadto realne. Po pierwsze, wszystkie książki na półce są tego samego formatu, ponieważ w tablicy komputerowej wszystkie wpisy tablicy mają taki sam rozmiar. Po drugie, możemy ponumerować pozycje książek na półce od 1 do n i każdą pozycję będziemy nazywać **przegródką** (ang. *slot*). Przegródka 1 występuje jako pierwsza z lewej, a przegródka n — jako ostatnia z prawej. Jak zapewne zgadujesz, każda przegródka na półce odpowiada wpisowi w tablicy.

Chciałbym również odnieść się do słowa „sortowanie". W mowie potocznej „sortowanie" może oznaczać coś innego niż znaczenie, w którym używamy go w informatyce. Słownik online w moim macu[4] definiuje słowo „sort" następująco: „grupować systematycznie, według typu, klasy itd." — w ten sposób możesz „sortować" na przykład odzież: koszule tu, spodnie tam i tak dalej. W świecie algorytmów komputerowych sortowanie oznacza: ułożyć w pewnym dobrze określonym porządku, a „grupowanie systematyczne, według typu, klasy itd." zwie się „szufladkowaniem" lub „kubełkowaniem"[5].

Wyszukiwanie binarne

Zanim poznamy pewne algorytmy sortowania, przyjrzyjmy się wyszukiwaniu binarnemu, które wymaga, aby przeszukiwana tablica była już posortowana. Zaletą wyszukiwania binarnego jest to, że przeszukanie n-elementowej tablicy zajmuje w nim tylko $O(\lg n)$ czasu.

W przykładzie z półką nasze książki są od początku posortowane według nazwisk autorów, poustawiane na półce od lewej do prawej. Jako klucza użyjemy nazwiska autora. Powiedzmy, że poszukujemy dowolnej książki Jonathana Swifta. Zauważ, że skoro nazwisko autora zaczyna się od „S", dziewiętnastej litery alfabetu

[4] Por. *http://oxforddictionaries.com/us/definition/american_english/sort — przyp. tłum.*

[5] W oryginale: „bucketing, bucketizing or binning" — *przyp. tłum.*

[angielskiego], możesz pominąć około trzy czwarte półki (gdyż 19/26 niewiele odbiega od 3/4) i dopiero tam poszukiwać przegródki. Jeżeli jednak masz wszystkie dzieła Szekspira, to masz kilkanaście książek autora, którego nazwisko wypada przed Swiftem[6], wskutek czego książki Swifta mogłyby wypaść bardziej w prawo, niż oczekiwałeś.

Spójrzmy więc lepiej, jak mógłbyś zastosować wyszukiwanie binarne do odnalezienia książki Jonathana Swifta. Przejdź do przegródki położonej dokładnie w połowie półki i sprawdź autora znajdującej się tam książki. Powiedzmy, że znalazłeś książkę Jacka Londona. Ponieważ książki są posortowane według nazwisk autorów, wiesz teraz nie tylko, że nie jest to książka, której szukasz, lecz także i to, że pośród wszystkich książek na lewo od książki Londona nie ma tej, której poszukujesz. Patrząc tylko na jedną książkę, wyeliminowałeś z grona kandydatek połowę książek na półce! Wszelkie książki Swifta muszą się znajdować po prawej stronie półki. Znajdujesz więc teraz przegródkę pośrodku prawej połowy półki i spoglądasz na stojącą tam książkę. Powiedzmy, że jest to coś Lwa Tołstoja. Znów nie jest to książka, której szukasz, lecz wiesz również, że możesz wyeliminować wszystkie książki na prawo od niej — połowę z tych, które pozostały jako kandydujące. W tym momencie wiesz, że jeśli Twoja półka zawiera jakieś książki Swifta, to są one wśród jednej czwartej książek stojących na prawo od książki Londona i na lewo od książki Tołstoja. Następnie znajdujesz książkę w przegródce na środku tej rozpatrywanej ćwiartki. Jeśli jest napisana przez Swifta, skończyłeś wyszukiwanie. W przeciwnym razie możesz znów wyeliminować połowę z pozostałych książek. Ostatecznie albo znajdziesz książkę Swifta, albo dotrzesz do punktu, w którym nie ma już żadnych kandydujących do sprawdzenia przegródek. W drugim przypadku stwierdzasz, że na półce nie ma żadnych książek Jonathana Swifta.

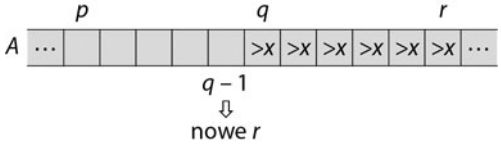
W komputerze wyszukiwanie binarne wykonujemy na tablicy. W dowolnej chwili rozpatrujemy tylko podtablicę, tj. fragment tablicy między dwoma indeksami, łącznie z elementami przez nie wskazywanymi. Nazwijmy te indeksy p i r. Na początku $p = 1$, a $r = n$, tzn. podtablicą jest cała tablica. Za każdym razem rozmiar rozpatrywanej podtablicy zmniejszamy o połowę, aż wystąpi jedno z dwu zdarzeń: znajdziemy poszukiwaną wartość albo podtablica stanie się pusta (tzn. p będzie większe niż r). Właśnie to powtarzane połowienie rozmiaru podtablicy sprawia, że czas działania wynosi $O(\lg n)$.

Omówimy teraz działanie wyszukiwania binarnego nieco bardziej szczegółowo. Powiedzmy, że w tablicy A poszukujemy wartości x. W każdym kroku rozpatrujemy tylko podtablicę zaczynającą się od $A[p]$ i kończącą na $A[r]$. Ponieważ popracujemy trochę na podtablicach, oznaczmy tę podtablicę przez $A[p..r]$. W każdym kroku obliczamy punkt środkowy q rozważanej podtablicy, biorąc średnią z p i r i odrzucając ewentualną część ułamkową: $q = \lfloor (p + r) / 2 \rfloor$. (Do odrzucenia części ułamkowej używamy tutaj operacji „podłoga”, $\lfloor\ \rfloor$. Gdybyś miał zrealizować tę operację w językach takich jak Java, C lub C++, do odrzucenia części ułamkowej mógłbyś użyć dzielenia

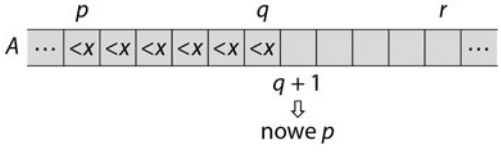
[6] W oryginale nazwisko Shakespeare poprzedza leksykograficznie nazwisko Swift — *przyp. tłum.*

całkowitego). Sprawdzamy, czy $A[q]$ równa się x. Jeśli tak, to gotowe, ponieważ możemy po prostu zwrócić q jako indeks miejsca, w którym tablica A zawiera x.

Gdy jednak okaże się, że $A[q] \neq x$, to robimy użytek z założenia, że tablica A jest już posortowana. Skoro $A[q] \neq x$, są dwie możliwości: $A[q] > x$ albo $A[q] < x$. Najpierw zajmujemy się przypadkiem $A[q] > x$. Z tego, że tablica jest posortowana, wiemy, iż nie tylko $A[q]$ jest większe od x, lecz również — myśląc o tablicy jako mającej elementy ułożone od lewej do prawej — każdy element na prawo od $A[q]$ jest większy niż x. Możemy więc wyeliminować z rozważań prócz $A[q]$ wszystkie elementy leżące na prawo od niego. Następny krok rozpoczniemy z niezmienionym p, natomiast r określimy jako równe $q - 1$:



Jeśli natomiast okaże się, że $A[q] < x$, to wiemy, że oprócz $A[q]$ każdy element tablicy położony na lewo od $A[q]$ jest mniejszy niż x, możemy więc poniechać rozważania tych elementów. Następny krok zaczynamy z niezmienionym r, a wartość p ustalamy na $q + 1$:



Oto dokładna procedura wyszukiwania binarnego:

Procedura WYSZUKIWANIE-BINARNE(A, n, x)

Dane wejściowe i wynik: jak w WYSZUKIWANIU-LINIOWYM.

1. Ustaw p na 1 i r na n.
2. Dopóki $p \leq r$, wykonuj, co następuje:
   A. Nadaj q wartość $\lfloor (p + r) / 2 \rfloor$.
   B. Jeśli $A[q] = x$, to zwróć q.
   C. W przeciwnym razie ($A[q] \neq x$), jeśli $A[q] > x$, to ustaw r na $q - 1$.
   D. W przeciwnym razie ($A[q] < x$), ustaw p na $q + 1$.

Zwróć NIE-ZNALEZIONO.

Pętla w kroku 2 niekoniecznie kończy się z powodu przekroczenia przez p wartości r. Może się skończyć w kroku 2B z powodu znalezienia elementu $A[q]$ równego x; zwracane jest wówczas q jako indeks w A, pod którym występuje x.

Aby wykazać, że procedura WYSZUKIWANIE-BINARNE działa poprawnie, wystarczy, jeśli pokażemy, że x nie występuje nigdzie w tablicy, jeśli WYSZUKIWANIE-BINARNE zwraca w kroku 3 wartość NIE-ZNALEZIONO. Posłużymy się następującym niezmiennikiem pętli:

> Na początku każdej iteracji pętli w kroku 2, jeśli x występuje gdzieś w tablicy A, to stanowi któryś z elementów podtablicy $A[p..r]$.

A oto krótkie uzasadnienie użycia tego niezmiennika pętli:

- **Zainicjowanie:** w kroku 1 indeksom p i r są nadawane wartości początkowe, odpowiednio: 1 i n, wobec tego niezmiennik pętli jest prawdziwy przy pierwszym wejściu procedury do pętli.
- **Utrzymanie:** uzasadniliśmy wyżej, że w krokach 2C i 2D wartość p albo q jest określana poprawnie.
- **Zakończenie:** jeśli x nie ma w tablicy, to procedura w końcu dotrze do miejsca, w którym p i r są równe. Wtedy w kroku 2A wartość q zostanie obliczona jako równa p i r. Jeśli w kroku 2C dochodzi do podstawienia pod r wartości $q - 1$, to na początku następnej iteracji r będzie równe $p - 1$, zatem p będzie większe niż r. Jeśli w kroku 2D p otrzymuje wartość $q + 1$, to na początku następnej iteracji p stanie się równe $r + 1$, więc znowu będzie większe niż r. W obu przypadkach warunek pętli sprawdzany w kroku 2 stanie się fałszywy i pętla zostanie zakończona. Ponieważ $p > r$, podtablica $A[p..r]$ będzie pusta, nie może więc w niej wystąpić wartość x. Przyjmując antytezę niezmiennika pętli, otrzymujemy, że jeśli x nie występuje w podtablicy $A[p..r]$, to nie ma go w całej tablicy A. Tym samym procedura poprawnie zwraca NIE-ZNALEZIONO w kroku 3.

Wyszukiwanie binarne możemy również zapisać w postaci procedury rekurencyjnej:

Procedura REKURENCYJNE-WYSZUKIWANIE-BINARNE(A, p, r, x)

Wejście i wyjście: dane wejściowe A i x oraz wyjście są takie same jak w WYSZUKIWANIU-LINIOWYM. Dane wejściowe p i r ograniczają rozpatrywaną podtablicę $A[p..r]$.

1. Jeśli $p >$ r, to zwróć NIE-ZNALEZIONO.
2. W przeciwnym razie ($p \leq r$) wykonaj, co następuje:
   A. Nadaj q wartość $\lfloor (p + r) / 2 \rfloor$.
   B. Jeśli $A[q] = x$, to zwróć q.
   C. W przeciwnym razie ($A[q] \neq x$), jeśli $A[q] > x$, to zwróć REKURENCYJNE-WYSZUKIWANIE-BINARNE(A, p, $q - 1$, x).
   D. W przeciwnym razie ($A[q] < x$) zwróć REKURENCYJNE-WYSZUKIWANIE-BINARNE(A, $q + 1$, r, x).

Wywołanie inicjujące ma postać REKURENCYJNE-WYSZUKIWANIE-BINARNE (A, 1, n, x).

Zobaczmy teraz, jak to się dzieje, że wyszukiwanie binarne osiąga dla n-elementowej tablicy czas $O(\lg n)$. Przede wszystkim zauważmy, że rozmiar $r - p + 1$ rozważanej podtablicy jest w przybliżeniu połowiony w każdej iteracji pętli (lub w każdym rekurencyjnym wywołaniu wersji rekurencyjnej, skupmy się jednak na wersji iteracyjnej w WYSZUKIWANIU-BINARNYM). Jeśli sprawdzisz wszystkie przypadki, to zauważysz, że gdy pewna iteracja zaczyna od podtablicy s-elementowej, wówczas następna ma do czynienia z $\lfloor s / 2 \rfloor$ albo $\lfloor s / 2 - 1 \rfloor$ elementami, zależnie od tego, czy s jest parzyste, czy nieparzyste i czy $A[q]$ jest większe niż x, czy mniejsze. Widzieliśmy już, że gdy rozmiar podtablicy zmaleje do 1, procedura kończy się w następnej iteracji. Możemy więc zapytać, ile razy musimy iterować pętlę, połowiąc podtablicę, nim początkowy rozmiar n zmaleje do 1? Wyniesie to tyle samo co liczba podwojeń, które musielibyśmy wykonać, aby, zaczynając od rozmiaru 1, dojść do rozmiaru n. A to jest po prostu potęgowanie: ustawiczne mnożenie przez 2. Inaczej mówiąc, dla jakiego x wartość 2^x osiąga n? Gdyby n było potęgą 2, to zobaczyliśmy już, że odpowiedzią jest $\lg n$. Oczywiście n może nie być dokładnie potęgą 2, a wtedy odpowiedź będzie odbiegać o 1 od $\lg n$. Zauważamy na koniec, że każda iteracja pętli zabiera skończoną ilość czasu, a więc czas jednej iteracji nie zależy od rozmiaru n pierwotnej tablicy ani od rozmiaru rozpatrywanej podtablicy. Posłużmy się notacją asymptotyczną, aby pominąć czynniki stałe i składniki niskiego rzędu. (Czy liczba iteracji pętli wynosi $\lg n$, czy $\lfloor \lg n \rfloor + 1$? Któż by się tym przejmował?). Przyjmujemy, że czas wykonania wyszukiwania binarnego wynosi $O(\lg n)$.

Użyłem tu notacji O, ponieważ zależało mi na wyrażeniu całościowego twierdzenia ujmującego wszystkie przypadki. W przypadku najgorszym, gdy wartość x nie występuje w tablicy, połowimy i połowimy, i połowimy — aż rozważana podtablica stanie się pusta, co daje czas działania $\Theta(\ln n)$. W przypadku najlepszym, gdy x zostaje znalezione w pierwszej iteracji pętli, czas działania wynosi $\Theta(1)$. Żadna notacja Θ nie ujmuje wszystkich przypadków, lecz czas działania $O(\lg n)$ jest zawsze poprawny dla wyszukiwania binarnego, o ile tylko tablica jest już posortowana.

Można pokonać czas $\Theta(\lg n)$ najgorszego przypadku wyszukiwania, lecz tylko wówczas, gdy zorganizujemy dane bardziej pracochłonnymi metodami i poczynimy pewne założenia odnośnie do kluczy.

Sortowanie przez wybieranie

Powracamy obecnie do **sortowania**, czyli takiego zreorganizowania (poprzestawiania) elementów tablicy, nazywanego również *permutowaniem* tablicy, aby każdy element był mniejszy lub równy swojemu następnikowi. Pierwszy algorytm sortowania, z którym się zapoznamy — sortowanie przez wybieranie (ang. *selection sort*) — uważam za najprostszy, bo to on przyszedł mi do głowy, gdy po raz pierwszy musiałem obmyślić jakiś algorytm sortowania. Daleko mu do najszybszego.

Oto jak wyglądałoby sortowanie przez wybieranie zastosowane do posortowania książek na półce według nazwisk autorów. Przejdź całą półkę i znajdź książkę

autora, którego nazwisko występuje alfabetycznie najwcześniej. Powiedzmy, że jest to książka Luizy May Alcott. (Jeśli półka zawiera dwie lub więcej książek tej autorki, to wybierz jedną z nich). Zamień miejscami tę książkę z książką w przegródce 1. Książka w przegródce 1 jest teraz książką autora o nazwisku pierwszym w porządku alfabetycznym. Przejdź teraz półkę od lewej do prawej, zaczynając od książki w przegródce 2, poszukując w przegródkach od 2 do *n* książki autora o nazwisku najwcześniejszym w alfabecie. Przypuśćmy, że jest to dzieło Jane Austen. Zamień miejscami tę książkę z książką w przegródce 2. — obecnie przegródki 1 i 2 zawierają pierwszą i drugą książkę w uporządkowaniu alfabetycznym. Zrób następnie to samo w odniesieniu do przegródki 3 i tak dalej. Kiedy wstawimy właściwą książkę do przegródki $n - 1$ (być może H.G. Wellsa), jesteśmy po robocie, ponieważ została tylko jedna książka (powiedzmy, Oscara Wilde'a), lecz ona znajduje się na swoim miejscu, w przegródce *n*.

Przekształcając to podejście w algorytm komputerowy, zamieniamy półkę na tablicę, a książki na elementy tablicy. Oto rezultat:

Procedura SORTOWANIE-PRZEZ-WYBIERANIE(*A*, *n*)

Dane wejściowe:

- *A*: tablica.
- *n*: liczba elementów w *A* do posortowania.

Wynik: elementy tablicy *A* są posortowane w porządku niemalejącym.

1. Dla $i = 1$ do $n - 1$:
   A. Przypisz zmiennej *najmniejszy* indeks najmniejszego elementu w podtablicy $A[i..n]$.
   B. Zamień $A[i]$ z $A[najmniejszy]$.

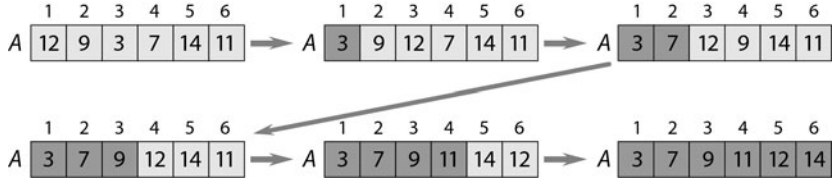
Znajdowanie najmniejszego elementu w $A[i..n]$ jest odmianą przeszukiwania liniowego. Najpierw przyjmujemy, że najmniejszym dotychczas oglądanym elementem w podtablicy jest $A[i]$. Następnie podążamy w głąb tablicy, aktualizując indeks najmniejszego elementu, ilekroć napotkamy element mniejszy niż aktualnie najmniejszy. Dokładniej wyrażona procedura przedstawia się następująco:

Procedura SORTOWANIE-PRZEZ-WYBIERANIE(*A*, *n*)

Dane wejściowe i wynik: takie same jak poprzednio.

1. Dla $i = 1$ do $n - 1$:
   A. Przypisz zmiennej *najmniejszy* wartość *i*.
   B. Dla $j = i + 1$ do *n*:
      i. Jeśli $A[j] < A[najmniejszy]$, to podstaw *j* do *najmniejszy*.
   C. Zamień $A[i]$ z $A[najmniejszy]$.

Ta procedura ma pętle „zagnieżdżone", tzn. pętla w kroku 1B występuje w pętli z kroku 1. Wewnętrzna pętla wykonuje wszystkie swoje iteracje dla każdej kolejnej iteracji pętli zewnętrznej. Zauważmy, że początkowa wartość *j* w pętli wewnętrznej zależy od bieżącej wartości *i* z pętli zewnętrznej. Poniżej przedstawiamy ilustrację działania sortowania przez wybieranie na tablicy sześciu elementów:



Początkową tablicę pokazano w lewym górnym rogu, a każdy następny krok przedstawia tablicę po iteracji pętli zewnętrznej. Elementy na ciemniejszym tle stanowią podtablicę już posortowaną.

Gdybyś chciał skorzystać z niezmiennika pętli do uzasadnienia, że procedura SORTOWANIE-PRZEZ-WYBIERANIE sortuje tablicę poprawnie, musiałbyś użyć po jednym dla każdej pętli. Procedura jest na tyle prosta, że nie będziemy podawać całych dowodów z użyciem niezmienników pętli, przytoczymy jednak te niezmienniki:

Na początku każdej iteracji pętli w kroku 1 podtablica $A[1..i-1]$ zawiera $i-1$ najmniejszych elementów całej tablicy *i*, przy czym są one posortowane.

Na początku każdej iteracji pętli w kroku 1B $A[najmniejszy]$ jest najmniejszym elementem w podtablicy $A[i..j-1]$.

Jaki jest czas działania SORTOWANIA-PRZEZ-WYBIERANIE? Pokażemy, że wynosi on $\Theta(n^2)$. Zasadniczy pomysł polega na przeanalizowaniu, ile iteracji jest wykonywanych w wewnętrznej pętli, przy założeniu że każda iteracja zabiera $\Theta(1)$ czasu. (Stałe czynniki dolnego i górnego ograniczenia w notacji Θ mogą być tutaj różne, gdyż przypisanie do zmiennej *najmniejszy* może, lecz nie musi występować w danej iteracji). Policzmy, ile jest iteracji, opierając się na wartości *i* w zewnętrznej pętli. Kiedy *i* równa się 1, treść wewnętrznej pętli jest powtarzana dla *j* przebiegającego wartości od 2 do *n*, czyli $n-1$ razy. Gdy *i* równa się 2, wewnętrzna pętla działa dla *j* zmieniającego się od 3 do *n*, czyli $n-2$ razy. Po każdym zwiększeniu w zewnętrznej pętli wartości *i* pętla wewnętrzna jest wykonywana o jeden raz mniej. Ogólnie biorąc, wewnętrzna pętla działa $n-i$ razy. W ostatniej iteracji pętli zewnętrznej, gdy *i* równa się $n-1$, pętla wewnętrzna jest iterowana tylko raz. Wobec tego łączna liczba iteracji pętli wewnętrznej wynosi:

$$(n-1) + (n-2) + (n-3) + \cdots + 2 + 1.$$

Ta suma nosi nazwę **postępu arytmetycznego**, a o szeregach arytmetycznych wiadomo — i jest to fakt podstawowy — że dla dowolnego nieujemnego *k*:

$$k + (k-1) + (k-2) + \cdots + 2 + 1 = \frac{k(k+1)}{2}.$$

Podstawiając $n - 1$ za k, widzimy, że sumaryczna liczba iteracji pętli wewnętrznej wynosi $(n - 1)\ n / 2$, czyli $(n^2 - n) / 2$. Użyjmy notacji asymptotycznej, aby pozbyć się składnika niskiego rzędu $(-n)$ i stałego czynnika $(1/2)$. Możemy wówczas powiedzieć, że łączny czas iteracji wewnętrznej pętli jest $\Theta(n^2)$. Dlatego czas działania SORTOWANIA-PRZEZ-WYBIERANIE wynosi $\Theta(n^2)$. Zauważmy, że ten czas działania wyraża stwierdzenie całościowe, obejmujące wszystkie przypadki. Niezależnie od rzeczywistych wartości elementów wewnętrzna pętla działa $\Theta(n^2)$ razy.

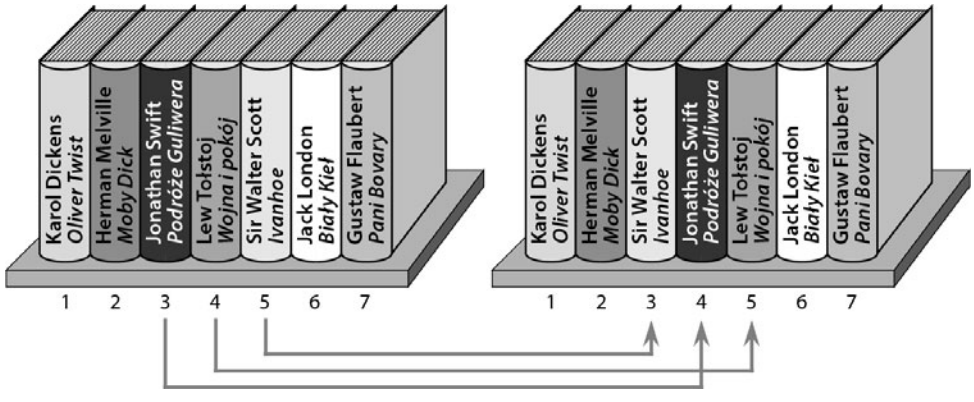
A oto inny sposób wykazania, że czas działania wynosi $\Theta(n^2)$, bez używania szeregów arytmetycznych. Pokażemy osobno, że czas działania wynosi zarówno $O(n^2)$, jak i $\Omega(n^2)$; złożenie asymptotycznych ograniczeń — górnego i dolnego — da nam czas $\Theta(n^2)$. Żeby wykazać, że czas działania wynosi $O(n^2)$, zauważmy, że każda iteracja zewnętrznej pętli powoduje najwyżej $(n - 1)$-krotne wykonanie treści pętli wewnętrznej, a to jest $O(n)$, gdyż każda iteracja pętli wewnętrznej zajmuje stałą ilość czasu. Ponieważ pętla zewnętrzna jest iterowana $n - 1$ razy, co także równa się $O(n)$, łączny czas spędzany w pętli wewnętrznej wynosi $O(n) \times O(n)$, czyli $O(n^2)$. Żeby zobaczyć, że czas działania wynosi $\Omega(n^2)$, zauważmy, że w każdej z pierwszych $n/2$ iteracji pętli zewnętrznej wykonujemy treść pętli wewnętrznej co najmniej $n/2$ razy, co łącznie stanowi przynajmniej $n/2 \times n/2$, czyli $n^2/4$ razy. Ponieważ każda iteracja pętli wewnętrznej zajmuje stałą ilość czasu, widzimy, że czas działania jest równy przynajmniej stałej pomnożonej przez $n^2/4$, czyli $\Omega(n^2)$.

Na koniec dwie uwagi dotyczące sortowania przez wybieranie. Po pierwsze, zobaczymy, że jego asymptotyczny czas działania rzędu $\Theta(n^2)$ jest najgorszy wśród algorytmów sortujących, które przebadamy. Po drugie, jeśli starannie przyjrzysz się sposobowi działania sortowania przez wybieranie, to zauważysz, że czas działania $\Theta(n^2)$ jest powodowany porównaniami w kroku 1Bi. Jednak liczba *przemieszczeń* elementów tablicy wynosi tylko $\Theta(n)$, jako że krok 1C jest wykonywany tylko $n - 1$ razy. Jeśli przemieszczanie elementów jest szczególnie czasochłonne — być może z powodu ich wielkości lub przechowywania na wolnym urządzeniu, takim jak dysk — to sortowanie przez wybieranie może się okazać algorytmem do przyjęcia.

Sortowanie przez wstawianie

Sortowanie przez wstawianie (ang. *insertion sort*) różni się nieco od sortowania przez wybieranie, choć ma podobny charakter. W sortowaniu przez wybieranie, kiedy zdecydowaliśmy, którą książkę wstawić do przegródki i, posortowane alfabetycznie według nazwisk autorów książki w pierwszych i przegródkach stanowiły pierwsze i książek *z całego ich zbioru*. W sortowaniu przez wstawianie książki w pierwszych i przegródkach będą tymi, *które od początku zajmowały pierwsze i przegródek*, obecnie posortowanymi według nazwisk autorów.

Załóżmy na przykład, że książki w pierwszych czterech przegródkach są już posortowane według nazwisk autorów i że są to kolejno książki: Karola Dickensa, Hermana Melville'a, Jonathana Swifta i Lwa Tołstoja. Powiedzmy, że autorem książki w przegródce 5 jest Sir Walter Scott. W sortowaniu przez wstawianie przesuwamy książki Swifta i Tołstoja o jedną przegródkę w prawo, przenosząc je z przegródek 3 i 4 do przegródek 4 i 5, po czym wkładamy książkę Scotta do zwolnionej przegródki 3. Kiedy zajmujemy się książką Scotta, nie zwracamy uwagi na książki z jej prawej strony (Jacka Londona i Gustawa Flauberta na rysunku poniżej); zajmiemy się nimi później.



Aby przesunąć książki Swifta i Tołstoja, porównujemy nazwisko Tołstoj z nazwiskiem Scott. Widząc, że Tołstoj występuje po Scotcie, przesuwamy książkę Tołstoja o jedną przegródkę w prawo, z przegródki 4 do 5. Następnie porównujemy nazwiska Swift i Scott. Widząc, że Swift występuje po Scotcie, przesuwamy książkę Swifta o jedną przegródkę w prawo, z przegródki 3 do 4, która się zwolniła, gdy przenieśliśmy książkę Tołstoja. Dalej porównujemy nazwisko autora Hermana Melville'a ze Scottem. Tym razem widzimy, że Melville *nie* występuje po Scotcie. Wobec tego zaprzestajemy porównywania nazwisk autorów, ponieważ okazało się, że książka Scotta powinna stać na prawo od książki Melville'a i na lewo od książki Swifta. Możemy włożyć książkę Scotta do przegródki 3, zwolnionej po przesunięciu książki Swifta.

Aby przełożyć ten pomysł na sortowanie tablicy za pomocą sortowania przez wstawianie, w podtablicy $A[1..i-1]$ będą przechowywane tylko te elementy, które na początku zajmowały w tablicy pierwsze $i-1$ pozycji, i będą one uporządkowane. Żeby określić miejsce, na które przechodzi element znajdujący się początkowo w $A[i]$, algorytm sortowania przez wstawianie zaczyna od $A[i-1]$ i podąża w $A[1..i-1]$ w lewo, przesuwając każdy element większy od rozpatrywanego o jedną pozycję w prawo. Kiedy natkniemy się na element nie większy od $A[i]$ lub dojdziemy do lewego końca tablicy, pozostawiamy element, który początkowo był w $A[i]$, na nowym miejscu w tablicy.

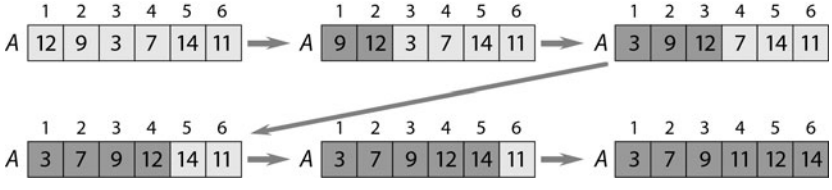
Procedura SORTOWANIE-PRZEZ-WSTAWIANIE(*A*, *n*)

Dane wejściowe i wynik: takie same jak w SORTOWANIU-PRZEZ-WYBIERANIE.

1. Dla *i* = 2 do *n*:
   A. Ustaw *klucz* na *A*[*i*] i podstaw do *j* wartość *i* – 1.
   B. Dopóki *j* > 0 i *A*[*j*] > *klucz*, wykonuj, co następuje:
      i. Podstaw *A*[*j*] do *A*[*j* + 1].
      ii. Zmniejsz *j* o 1 (tzn. podstaw do *j* wartość *j* – 1).
   C. Podstaw *klucz* do *A*[*j* + 1].

Sprawdzenie w kroku 1B zasadza się na potraktowaniu operatora „i" (ang. *and*) jako **krótko spinającego** (ang. *short circuiting*): jeśli wyrażenie po lewej, *j* > 0, jest fałszywe, to nie oblicza się wyrażenia po prawej, tj. *A*[*j*] > *klucz*. Gdyby to zrobiono, próba sięgnięcia po *A*[*j*], gdy *j* ≤ 0, spowodowałaby błąd indeksowania tablicy.

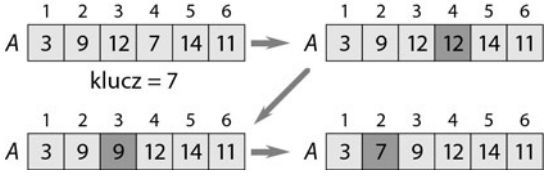
Oto jak działa sortowanie przez wstawianie na tej samej tablicy, której używaliśmy przy sortowaniu przez wybieranie:



Również tutaj początkowa tablica znajduje się w lewym górnym rogu, a każdy krok ukazuje tablicę po iteracji zewnętrznej pętli z kroku 1. Elementy mocniej zacieniowane zawierają podtablicę już posortowaną. Niezmiennik zewnętrznej pętli (jak poprzednio, nie będziemy go dowodzić) jest następujący:

> Na początku każdej iteracji pętli z kroku 1 podtablica *A*[1..*i* – 1] składa się z elementów występujących pierwotnie w *A*[1..*i* – 1], lecz uporządkowanych.

Następny rysunek przedstawia działanie w powyższym przykładzie wewnętrznej pętli z kroku 1B, gdy *i* równa się 4. Zakładamy, że podtablica *A*[1..3] zawiera elementy występujące początkowo na trzech pierwszych pozycjach w tablicy, lecz są one obecnie posortowane. Aby określić, gdzie umieścić element będący początkowo w *A*[4], przechowujemy go w zmiennej o nazwie *klucz*, po czym przesuwamy każdy element w *A*[1..3] większy niż *klucz* o jedną pozycję w prawo:



Ciemniejsze pozycje wskazują miejsca, do których elementy są przemieszczane. W ostatnim pokazanym kroku wartość $A[1]$, czyli liczba 3, nie jest większa niż wartość *klucza*, czyli 7, toteż wewnętrzna pętla się kończy. Wartość *klucza* wpada na pozycję na prawo od $A[1]$, jak pokazano w ostatnim kroku. Oczywiście w kroku 1A musimy przechować pierwotną wartość $A[i]$ w *kluczu*, ponieważ w pierwszej iteracji wewnętrznej pętli wartość $A[i]$ jest zastępowana inną.

Jest także możliwe, że pętla wewnętrzna zakończy się z powodu niespełnienia warunku $j > 0$. Dochodzi do tego, jeśli klucz jest mniejszy od wszystkich elementów w $A[1..i-1]$. Gdy j staje się równe 0, każdy element w $A[1..i-1]$ został przesunięty w prawo, zatem w kroku 1C klucz trafia do $A[1]$ — właśnie tam, gdzie chcieliśmy.

Przeanalizowanie czasu działania SORTOWANIA-PRZEZ-WSTAWIANIE przysparza nieco więcej trudności niż w wypadku SORTOWANIA-PRZEZ-WYBIERANIE. Liczba iteracji wewnętrznej pętli w procedurze SORTOWANIE-PRZEZ-WYBIERANIE zależy tylko od indeksu i zewnętrznej pętli, bez jakiegokolwiek powiązania z samymi elementami. Natomiast w procedurze SORTOWANIE-PRZEZ-WSTAWIANIE liczba iteracji wewnętrznej pętli zależy od indeksu i zewnętrznej pętli, *a także* od wartości w tablicy.

Z najlepszym przypadkiem SORTOWANIA-PRZEZ-WSTAWIANIE mamy do czynienia wtedy, gdy za każdym razem treść pętli wewnętrznej jest wykonywana zero razy. Aby tak było, wynik sprawdzenia $A[j] > klucz$ musi być fałszywy za pierwszym razem dla każdego i. Innymi słowy, musi być $A[i-1] \leq A[i]$ w każdym wykonaniu kroku 1B. Kiedy powstaje taka sytuacja? Tylko wówczas, gdy tablica A jest posortowana już w chwili rozpoczynania procedury. W tym przypadku zewnętrzna pętla iteruje $n-1$ razy, a każda taka iteracja zajmuje stałą ilość czasu, tak więc SORTOWANIE-PRZEZ-WSTAWIANIE zabiera tylko $\Theta(n)$ czasu.

Przypadek najgorszy występuje wtedy, gdy za każdym razem wewnętrzna pętla wykonuje maksymalną możliwą liczbę powtórzeń. Test $A[j] > klucz$ musi teraz za każdym razem wypadać pomyślnie, a pętla musi się kończyć z powodu niespełnienia warunku $j > 0$. Każdy element $A[i]$ musi wędrować na lewy koniec tablicy. Kiedy do tego dochodzi? Tylko wówczas, gdy tablica A na początku jest uporządkowana w odwrotnej kolejności, tzn. posortowana w porządku *nierosnącym*. W tym przypadku w każdej iteracji pętli zewnętrznej wewnętrzna pętla iteruje $i-1$ razy. Ponieważ treść pętli zewnętrznej jest powtarzana dla i zmieniającego się od 2 do n, liczba iteracji pętli wewnętrznej tworzy szereg arytmetyczny:

$$1 + 2 + 3 + \ldots + (n-2) + (n-1),$$

który, jak widzieliśmy w sortowaniu przez wybieranie, jest rzędu $\Theta(n^2)$. Skoro każda iteracja pętli wewnętrznej zajmuje stałą ilość czasu, czas działania sortowania przez wstawianie w najgorszym przypadku wynosi $\Theta(n^2)$. Dlatego w najgorszym przypadku sortowanie przez wybieranie i sortowanie przez wstawianie mają czasy działania asymptotycznie jednakowe.

Czy ma sens rozpatrywanie, co się dzieje z sortowaniem przez wstawianie w przypadku średnim? To zależy od tego, jak wyglądają „średnie" dane wejściowe. Jeśli uporządkowanie elementów w tablicy wejściowej jest naprawdę losowe, to możemy oczekiwać, że każdy element będzie większy od około połowy poprzedzających go elementów i od około połowy tych elementów mniejszy, wobec czego w każdym

wykonaniu pętli wewnętrznej nastąpi około $(i - 1) / 2$ iteracji. Skraca to czas działania o połowę w porównaniu z najgorszym przypadkiem. Lecz 1/2 jest tylko stałym czynnikiem, toteż asymptotycznie nie robi to różnicy w stosunku do czasu działania w najgorszym przypadku — nadal wynosi on $\Theta(n^2)$.

Sortowanie przez wstawianie jest świetnym wyborem, gdy na początku tablica jest „prawie posortowana". Przypuśćmy, że każdy element w sortowanej tablicy występuje w odległości najwyżej k pozycji od swojego docelowego miejsca. Wówczas łączna liczba przesunięć danego elementu we wszystkich iteracjach wewnętrznej pętli wynosi najwyżej k. Stąd łączna liczba przesunięć wszystkich elementów we wszystkich iteracjach wewnętrznej pętli wyniesie najwyżej $k\,n$, co daje do zrozumienia, że łączna liczba iteracji wewnętrznej pętli wynosi również najwyżej $k\,n$ (gdyż w każdej iteracji wewnętrznej pętli przesuwa się dokładnie jeden element o jedną pozycję). Jeśli k jest stałą, to ogólny czas działania sortowania przez wstawianie wyniesie tylko $\Theta(n)$, ponieważ notacja Θ wchłania stały czynnik k. Rzeczywiście, możemy nawet dopuścić, że niektóre elementy w tablicy będą przesuwane na duże odległości, o ile tylko nie będzie ich zbyt wiele. W szczególności gdy l elementom wolno przemieszczać się w tablicy w dowolne miejsce (czyli każdy z nich może być przesuwany nawet o $n - 1$ pozycji), a pozostałe $n - l$ elementów wolno przesuwać o najwyżej k pozycji, wówczas łączna liczba przesunięć wyniesie najwyżej $l\,(n - 1) + (n - l)\,k = (k + l)\,n - (k + 1)\,l$, czyli $\Theta(n)$, jeżeli k i l są stałymi.

Jeśli porównamy asymptotyczne czasy działania sortowania przez wstawianie i sortowania przez wybieranie, to zauważymy, że w najgorszym przypadku są one takie same. Sortowanie przez wstawianie jest lepsze, gdy tablica jest prawie uporządkowana. Jednakże sortowanie przez wybieranie pod jednym względem jest lepsze niż sortowanie przez wstawianie: sortowanie przez wybieranie przemieszcza elementy $\Theta(n)$ razy niezależnie od sytuacji, podczas gdy sortowanie przez wstawianie może przesuwać elementy aż $\Theta(n^2)$ razy, gdyż każde wykonanie w SORTOWANIU-PRZEZ-WSTAWIANIE kroku 1B powoduje przesunięcie elementu. W odniesieniu do sortowania przez wybieranie, jeżeli przesunięcie elementu jest szczególnie czasochłonne i nie spodziewasz się, że dane do sortowania przez wstawianie będą zbliżone do przypadku najlepszego, to niewykluczone, że postąpisz lepiej, wykonując sortowanie przez wybieranie zamiast sortowania przez wstawianie.

Sortowanie przez scalanie

Nasz następny algorytm sortowania — sortowanie przez scalanie — ma we wszystkich przypadkach czas działania wynoszący tylko $\Theta(n \lg n)$. Porównując jego czas działania z czasami działania $\Theta(n^2)$ sortowania przez wybieranie i sortowania przez wstawianie, wymieniamy czynnik n na czynnik wynoszący tylko $\lg n$. Jak zauważyliśmy w rozdziale 1, jest to rodzaj transakcji wartej zachodu.

Sortowanie przez scalanie (ang. *merge sort*) ma kilka wad w porównaniu z oglądanymi już przez nas dwoma algorytmami sortowania. Po pierwsze, stały czynnik, który ukrywamy w notacji asymptotycznej, jest większy niż w tamtych dwóch algorytmach. Rzecz jasna, gdy rozmiar n tablicy staje się dostatecznie duży, przestaje to mieć zna-

czenie. Po drugie, sortowanie przez scalanie nie działa **na miejscu** (ang. *in place*)[7] — musi wykonywać kopie całej tablicy wejściowej. Zestaw to z sortowaniem przez wybieranie i sortowaniem przez wstawianie, które podczas działania utrzymują dodatkową kopię tylko jednego wpisu tablicy, a nie kopie wszystkich wpisów tablicy. Jeżeli liczy się pamięć, to sortowanie przez scalanie możesz sobie podarować.

W sortowaniu przez scalanie stosujemy popularny paradygmat algorytmiczny, zwany **dziel i zwyciężaj** (ang. *divide-and-conquer*). W metodzie „dziel i zwyciężaj" dzielimy problem na podproblemy podobne do pierwotnego problemu, następnie rozwiązujemy te podproblemy rekurencyjnie, po czym łączymy ich rozwiązania, aby rozwiązać problem wyjściowy. Przypomnijmy za rozdziałem 2, że do poprawnego działania rekursji każde rekurencyjne wywołanie musi być mniejszym egzemplarzem tego samego problemu, który na koniec staje się przypadkiem bazowym. Oto ogólny schemat działania algorytmu typu „dziel i zwyciężaj":

1. **Podziel** problem na pewną liczbę podproblemów będących mniejszymi egzemplarzami problemu tego samego rodzaju.
2. **Zwyciężaj** podproblemy, rozwiązując je rekurencyjnie. Jeśli są wystarczająco małe, rozwiąż je jako przypadki bazowe.
3. **Połącz** rozwiązania podproblemów w rozwiązanie pierwotnego problemu.

Kiedy porządkujemy książki na półce, stosując sortowanie przez scalanie, wtedy każdy podproblem składa się z sortowania książek w grupie sąsiadujących przegródek półki. Początkowo chcemy posortować wszystkie n książek w przegródkach od 1 do n, lecz w ogólnym podproblemie będziemy sobie życzyli posortowania wszystkich książek w przegródkach od p do r. Oto jak stosujemy metodę „dziel i zwyciężaj":

1. **Dziel** przez znalezienie numeru q przegródki między p a r. Robimy to tak samo jak wtedy, gdy znajdowaliśmy punkt środkowy w wyszukiwaniu binarnym: dodajemy p i q, dzielimy przez 2 i bierzemy z tego podłogę.
2. **Zwyciężaj** przez rekurencyjne sortowanie książek w każdym z dwóch podproblemów utworzonych w kroku „dziel": sortujemy rekurencyjnie książki w przegródkach od p do q i to samo robimy z książkami w przegródkach od $q + 1$ do r.
3. **Połącz**, scalając książki posortowane w przegródkach od p do q i od $q + 1$ do r, tak aby wszystkie książki w przegródkach od p do r były posortowane. Jak połączyć książki, zobaczymy za chwilę.

Przypadek bazowy zachodzi, gdy trzeba posortować mniej niż dwie książki (tzn. gdy $p \geq r$), ponieważ zbiór niezawierający żadnej książki lub zawierający jedną jest już — co oczywiste — posortowany.

Aby zamienić ten pomysł na sortowanie tablicy, książkom w przegródkach od p do r będzie odpowiadać podtablica $A[p..r]$. Oto jak wygląda procedura sortowania przez scalanie, która wywołuje procedurę SCALANIE(A, p, q, r) w celu połączenia posortowanych podtablic $A[p..q]$ i $A[q + 1..r]$ w jedną posortowaną podtablicę $A[p..r]$:

[7] Używa się też określenia „w miejscu" (łac. *in situ*) — *przyp. tłum.*

Procedura SORTOWANIE-PRZEZ-SCALANIE(A, p, r)

Dane wejściowe:

- A: tablica.
- p, r: początkowy i końcowy indeks podtablicy A.

Wynik: elementy podtablicy $A[p..r]$ posortowane w porządku niemalejącym.

1. Jeśli $p \geq r$, to podtablica $A[p..r]$ ma najwyżej 1 element, jest więc już posortowana. Powróć zatem, nie wykonując niczego.
2. W przeciwnym razie wykonaj, co następuje:
   A. Nadaj q wartość $\lfloor (p + r) / 2 \rfloor$.
   B. Wywołaj rekurencyjnie SORTOWANIE-PRZEZ-SCALANIE(A, p, q).
   C. Wywołaj rekurencyjnie SORTOWANIE-PRZEZ-SCALANIE(A, $q + 1$, r).
   D. Wywołaj SCALANIE(A, p, q, r).

Mimo że nie zobaczyliśmy jeszcze, jak działa procedura SCALANIE, spójrzmy najpierw na przykład działania procedury SORTOWANIE-PRZEZ-SCALANIE. Zacznijmy od tablicy:

1	2	3	4	5	6	7	8	9	10
12	9	3	7	14	11	6	2	10	5

Wywołanie inicjujące ma postać SORTOWANIE-PRZEZ-SCALANIE(A, 1, 10). W kroku 2A określa się q jako równe 5, zatem rekurencyjne wywołania w krokach 2B i 2C przyjmują postać: SORTOWANIE-PRZEZ-SCALANIE(A, 1, 5) i SORTOWANIE-PRZEZ-SCALANIE(A, 6, 10).

1	2	3	4	5
12	9	3	7	14

6	7	8	9	10
11	6	2	10	5

Po tych dwu rekurencyjnych wywołaniach posortowane są następujące dwie podtablice:

1	2	3	4	5
3	7	9	12	14

6	7	8	9	10
2	5	6	10	11

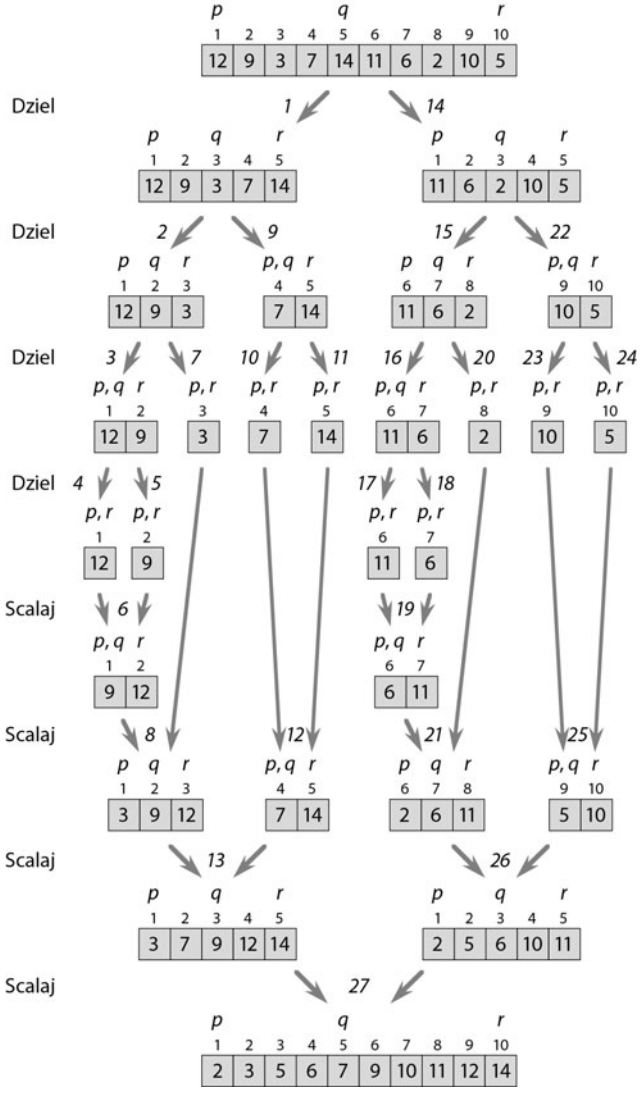
Na koniec, w kroku 2D, wywołanie SCALANIE(A, 1, 5, 10) łączy obie posortowane podtablice w jedną posortowaną podtablicę, która w tym wypadku stanowi całą tablicę:

1	2	3	4	5	6	7	8	9	10
2	3	5	6	7	9	10	11	12	14

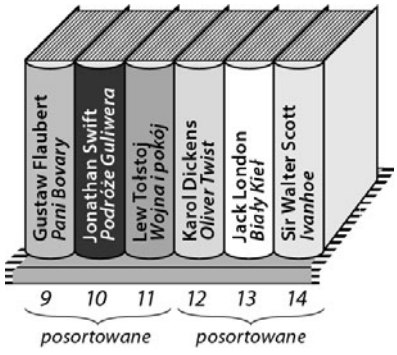
Gdybyśmy rozwinęli rekursję, otrzymalibyśmy rysunek przedstawiony na następnej stronie. Rozbiegające się strzałki wskazują kroki podziału, a strzałki zbiegające się ku sobie — kroki scalania. Zmienne p, q i r, widoczne nad każdą podtablicą,

są umieszczone nad indeksami, którym odpowiadają w każdym z rekurencyjnych wywołań. Numery zapisane kursywą podają kolejność, w której występują wywołania procedury po inicjującym wywołaniu SORTOWANIE-PRZEZ-SCALANIE(A, 1, 10). Na przykład wywołanie SCALANIE(A, 1, 3, 5) jest trzynastym wywołaniem procedury po wywołaniu inicjującym, a wywołanie SORTOWANIE-PRZEZ-SCALANIE(A, 6, 7) jest szesnastym z kolei.

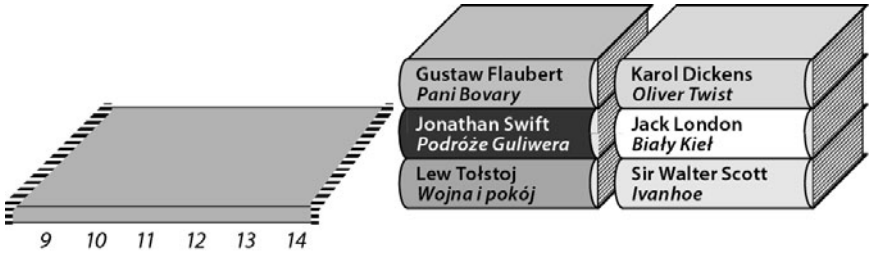
Prawdziwą robotę wykonuje procedura SCALAJ. Dlatego nie tylko musi ona działać poprawnie, lecz powinna być szybka. Jeśli łączymy ogółem n elementów, to możemy liczyć najwyżej na czas $\Theta(n)$, gdyż każdy element musi być dołączony na właściwe miejsce, faktycznie możemy więc osiągnąć liniowy czas scalania.



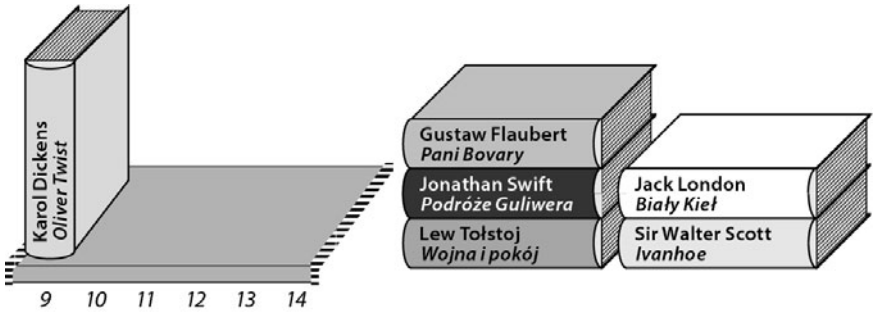
Wracając do przykładu z książkami, spójrzmy na fragment półki z przegródkami od 9 do 14. Załóżmy, że posortowaliśmy książki w przegródkach 9 – 11 i mamy je również posortowane w przegródkach 12 – 14.



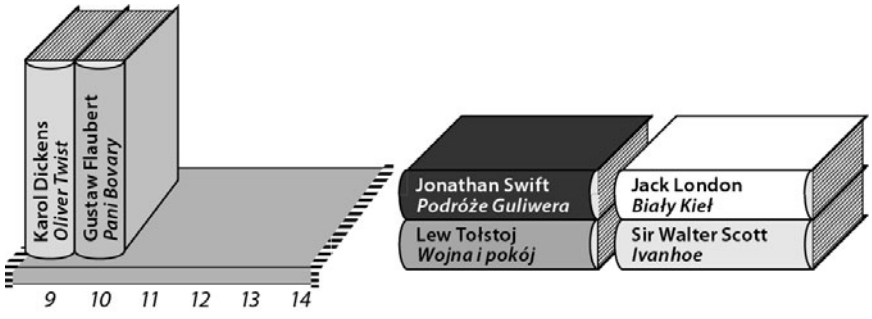
Wyjmujemy książki z przegródek 9 – 11 i układamy je na stosie, robiąc to tak, aby książka autora o nazwisku alfabetycznie pierwszym znalazła się na szczycie. To samo robimy z książkami w przegródkach 12 – 14, tworząc osobny stos:



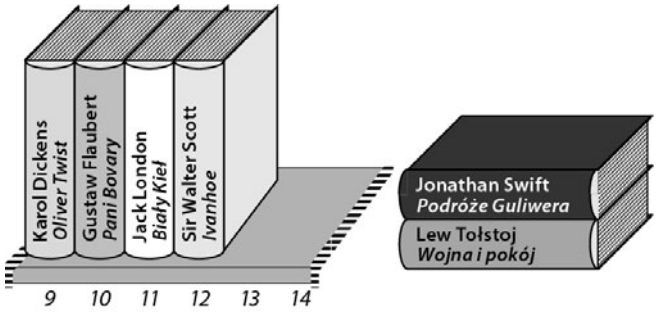
Ponieważ oba stosy są już posortowane, książka, która powinna trafić do przegródki 9, musi być jedną z leżących na szczycie stosu: będzie to książka albo Gustawa Flauberta, albo Karola Dickensa. I rzeczywiście, widząc, że książka Dickensa poprzedza alfabetycznie książkę Flauberta, przenosimy ją do przegródki 9:



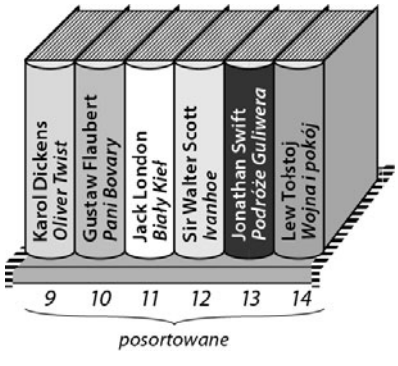
Po przeniesieniu książki Dickensa do przegródki 9 książką, która powinna trafić do przegródki 10, powinna być ta, która nadal pozostaje na szczycie pierwszego stosu, czyli Flaubert, albo książka, która teraz znalazła się na szczycie drugiego stosu — Jack London. Do przegródki 10 przenosimy książkę Flauberta:



Dalej porównujemy książki na szczycie stosów; są to obecnie książki Jonathana Swifta i Londona. Przenosimy książkę Londona do przegródki 11. Na szczycie prawego stosu zostaje więc książka Sir Waltera Scotta i po porównaniu jej z książką Swifta przenosimy ją do przegródki 12. W tym momencie prawy stos znika (staje się pusty).



Teraz pozostaje już tylko przemieścić po kolei książki z lewego stosu do pozostałych przegródek i wszystkie książki w przegródkach 9 – 14 będą posortowane:



Jak sprawna jest ta procedura scalania? Każdą książkę przenosimy dwa razy: raz z półki na stos i raz ze szczytu stosu na półkę. Ponadto ilekroć decydujemy o tym, którą książkę postawić z powrotem na półce, musimy porównać tylko dwie książki: te na wierzchu stosów. Do połączenia n książek przemieszczamy zatem książki $2n$ razy i porównujemy książki parami najwyżej n razy.

Po co wyjmować książki z półki? Czy nie moglibyśmy zostawić ich na półce i po prostu zapamiętać, które książki ustawiliśmy we właściwych przegródkach na półce, a które nie? Okazuje się, że byłoby z tym znacznie więcej roboty. Załóżmy na przykład, że każda książka z prawej połowy powinna stanąć przed każdą książką z lewej połowy. Przed przesunięciem pierwszej książki z prawej połowy do pierwszej przegródki lewej połowy musielibyśmy przesunąć każdą książkę z lewej połowy o jedną przegródkę w prawo, aby zrobić miejsce. Potem musielibyśmy zrobić to samo, żeby postawić następną książkę z prawej połowy w drugiej przegródce lewej połowy. I to samo trzeba by wykonać z kolejnymi książkami z prawej połowy. Musielibyśmy przesuwać połowę książek — wszystkie książki w lewej połowie — ilekroć chcielibyśmy dołożyć tam na właściwe miejsce książkę z prawej połowy.

To uzasadnia, dlaczego nie wykonujemy scalania na miejscu[8]. Powróćmy do sposobu scalenia posortowanych podtablic $A[p..q]$ i $A[q + 1..r]$ w podtablicę $A[p..r]$. Zaczynamy od skopiowania elementów do scalenia z tablicy A do tymczasowych tablic, a następnie scalamy je z powrotem w A. Niech $n_1 = q - p + 1$ będzie liczbą elementów w $A[p..q]$, a $n_2 = r - q$ oznacza liczbę elementów w $A[q + 1..r]$. Tworzymy tymczasowe tablice B o n_1 elementach i C o n_2 elementach, po czym kopiujemy kolejne elementy z $A[p..q]$ do B, i podobnie — elementy z $A[q + 1..r]$ do C. Teraz możemy scalić te elementy z powrotem w $A[p..r]$ bez obawy, że wskutek ponownego zapisywania uszkodzimy ich jedyne kopie.

Elementy tablic scalamy tak samo, jak łączyliśmy książki. Kopiujemy elementy z tablic B i C z powrotem do podtablicy $A[p..r]$, bacząc, aby indeksy w każdej z nich wskazywały najmniejszy, jeszcze nie skopiowany element. W stałym czasie określamy, który element jest mniejszy, kopiujemy go z powrotem na właściwe miejsce w $A[p..r]$ i uaktualniamy indeksy do tablic.

W końcu wszystkie elementy którejś z dwóch tablic zostaną przekopiowane z powrotem do $A[p..r]$. Odpowiada to sytuacji, w której został tylko jeden stos książek. Stosujemy tu jednak trik, żeby uniknąć sprawdzania za każdym razem, czy jedna z tablic się opróżniła: umieszczamy na prawym końcu w obu tablicach B i C dodatkowy element, większy od wszystkich innych. Przypominasz sobie sztuczkę z wartownikiem, którą posłużyliśmy się w WYSZUKIWANIU-LINIOWYM-Z-WARTOWNIKIEM w rozdziale 2? Jest to podobny pomysł. Tutaj w charakterze wartującego klucza sortowania używamy ∞ (nieskończoności), abyśmy mogli sobie odpuścić sprawdzanie, w której tablicy został mniejszy element, ilekroć wykryjemy element, którego

[8] Prawdę mówiąc, możliwe jest scalenie na miejscu w czasie liniowym, lecz procedura jest bardzo skomplikowana.

klucz sortowania ∞ jest najmniejszym elementem pozostającym w tablicy[9]. Kiedy już wszystkie elementy obu tablic B i C zostaną skopiowane z powrotem, w obu tablicach wartownicy pozostają jako elementy najmniejsze. W tym momencie nie trzeba już porównywać wartowników, ponieważ „prawdziwe" elementy (niebędące wartownikami) są już na nowo w $A[p..r]$. Ponieważ z góry wiemy, że będziemy kopiować z powrotem elementy od $A[p]$ do $A[r]$, możemy zakończyć po skopiowaniu elementu do $A[r]$. Wystarczy, abyśmy wykonali pętlę tylko dla indeksu tablicy A zmieniającego się od p do r.

Oto procedura SCALAJ(A, p, q, r). Wydaje się długa, lecz realizuje właśnie metodę opisaną wyżej.

Procedura SCALAJ(A, p, q, r)

Dane wejściowe:

- A: tablica.
- p, q, r: indeksy do tablicy A. W przypadku każdej z podtablic: $A[p..q]$ i $A[q + 1, r]$, zakłada się, że jest już posortowana.

Wynik: podtablica $A[p..r]$ zawiera elementy występujące pierwotnie w $A[p..q]$ i $A[q + 1, r]$, obecnie jednak cała podtablica $A[p..r]$ jest posortowana.

1. Nadaj n_1 wartość $q - p + 1$, a n_2 — wartość $r - q$.
2. Niech $B[1..n_1 + 1]$ i $C[1..n_2 + 1]$ będą nowymi tablicami.
3. Skopiuj $A[p..q]$ do $B[1..n_1]$ oraz $A[q + 1..r]$ do $C[1..n_2]$.
4. Ustaw zarówno $B[n_1 + 1]$, jak i $C[n_2 + 1]$ na ∞.
5. Ustaw i oraz j na 1.
6. Dla $k = p$ do r:
   A. Jeśli $B[i] \leq C[j]$, to przypisz $B[i]$ do $A[k]$ i zwiększ i o 1.
   B. W przeciwnym razie ($B[i] > C[j]$) przypisz $C[i]$ do $A[k]$ i zwiększ j o 1.

Po przydzieleniu w krokach 1 – 4 tablic B i C, skopiowaniu $A[p..q]$ do B i $A[q + 1..r]$ do C oraz wstawieniu do tych tablic wartowników każda iteracja pętli głównej w kroku 6 powoduje skopiowanie najmniejszego z pozostałych elementów z powrotem na następną pozycję w $A[p..r]$, kończąc działanie po skopiowaniu wszystkich elementów z B i C. W tej pętli i indeksuje najmniejszy element pozostający w B, j indeksuje najmniejszy element pozostający w C, a k indeksuje miejsce w A, na które element zostanie z powrotem skopiowany.

[9] W praktyce nieskończoność (∞) reprezentujemy za pomocą wartości, która jest większa od każdego z porównywanych kluczy sortowania. Gdyby na przykład chodziło o klucze sortowania rozumiane jako nazwiska autorów, to jako ∞ można by użyć napisu ZZZZ, zakładając oczywiście, że nigdy nie istniał autor o takim nazwisku.

Jeżeli scalamy n elementów (tzn. $n = n_1 + n_2$), to skopiowanie elementów do tablic B i C zajmuje $\Theta(n)$ czasu plus stały czas przypadający na skopiowanie elementu z powrotem do $A[p..r]$, wskutek czego łączny czas scalania wynosi tylko $\Theta(n)$.

Twierdziliśmy wcześniej, że cały algorytm sortowania przez scalanie zajmuje $\Theta(n \lg n)$ czasu. Poczynimy teraz upraszczające założenie, że rozmiar n tablicy jest potęgą 2, aby przy każdym dzieleniu tablicy rozmiary podtablic były równe. (W ogólności n może nie być potęgą 2, toteż rozmiary tablic mogą nie być jednakowe w danym wywołaniu rekurencyjnym. W ścisłej analizie można zwrócić uwagę na ten szczegół techniczny, lecz dajmy sobie z nim spokój).

Oto jak analizujemy sortowanie przez scalanie. Powiedzmy, że posortowanie podtablicy n-elementowej zabiera $T(n)$ czasu, stanowiąc funkcję rosnącą z n (bo przypuszczalnie posortowanie większej liczby elementów trwa dłużej). Czas $T(n)$ pochodzi z trzech komponentów składających się na wzorzec postępowania „dziel i zwyciężaj", których czasy sumujemy:

1. Podział zajmuje stałą ilość czasu, ponieważ sprowadza się do obliczenia indeksu q.
2. Na zwyciężanie składają się dwa rekurencyjne wywołania dotyczące podtablic, z których każda ma $n / 2$ elementów. Zważywszy, jak zdefiniowaliśmy czas sortowania podtablicy, każde z rekurencyjnych wywołań zajmuje $T(n / 2)$ czasu.
3. Łączenie wyników dwóch rekurencyjnych wywołań w drodze scalania posortowanych podtablic zajmuje $\Theta(n)$ czasu.

Ponieważ stały czas podziału jest składnikiem niskiego rzędu w porównaniu z czasem $\Theta(n)$ łączenia, możemy utopić czas podziału w czasie łączenia i powiedzieć, że etapy dzielenia i scalania zabierają łącznie $\Theta(n)$ czasu. Krok zwyciężania kosztuje $T(n / 2) + T(n / 2)$, czyli $2T(n / 2)$. Możemy teraz zapisać równanie dla $T(n)$:

$$T(n) = 2T(n / 2) + f(n),$$

gdzie $f(n)$ reprezentuje czas dzielenia i scalania, który — co właśnie zauważyliśmy — wynosi $\Theta(n)$. Typową praktyką w badaniu algorytmów jest właśnie umieszczanie notacji asymptotycznej wprost w równaniu i uważanie jej za pewną funkcję, której nie nazywamy. Przepiszemy więc to równanie tak:

$$T(n) = 2T(n / 2) + \Theta(n).$$

Chwileczkę, coś tu nie gra! Zdefiniowaliśmy funkcję T, która opisuje czas działania sortowania przez scalanie za pomocą tej samej funkcji! Równanie tego rodzaju nazywamy **zależnością rekurencyjną** (ang. *recurrence equation*) lub po prostu **rekurencją**. Problem polega na tym, że chcemy wyrazić funkcję $T(n)$ w sposób nierekurencyjny, czyli nie z użyciem jej samej. Zamiana funkcji wyrażonej rekurencyjnie na postać nierekurencyjną może przyprawić o prawdziwy ból głowy, lecz dla rozległej klasy równań rekurencyjnych możemy zastosować przepis kucharski znany

jako **metoda rekurencji uniwersalnej**[10]. Metodę rekurencji uniwersalnej stosuje się do wielu (lecz nie do wszystkich) rekurencji postaci $T(n) = aT(n/b) + f(n)$, gdzie a i b są dodatnimi stałymi całkowitymi. Na szczęście można jej użyć do rekurencji w naszym sortowaniu przez scalanie i wynika z niej, że $T(n)$ wynosi $\Theta(n \lg n)$.

Ów czas działania $\Theta(n \lg n)$ odnosi się do wszystkich przypadków sortowania przez scalanie: najlepszego, najgorszego i przypadków pośrednich. Każdy element jest kopiowany $\Theta(n \lg n)$ razy. Jak możesz się przekonać, sprawdzając metodę SCALAJ, po wywołaniu z $p = 1$ i $r = n$ tworzy ona kopie wszystkich n elementów, toteż sortowanie przez scalanie w żadnym wypadku nie działa na miejscu.

Sortowanie szybkie

Podobnie jak w sortowaniu przez scalanie, w sortowaniu szybkim stosuje się schemat „dziel i zwyciężaj" (a więc i rekursję). **Sortowanie szybkie** (ang. *quicksort*) używa „dzielenia i zwyciężania" nieco inaczej, niż to się dzieje w sortowaniu przez scalanie. Od sortowania przez scalanie różni się jeszcze paroma innymi istotnymi szczegółami:

- Sortowanie szybkie działa na miejscu[11].
- Asymptotyczny czas działania sortowania szybkiego jest różny w przypadku najgorszym i w przypadku średnim. W szczególności czas działania sortowania szybkiego w przypadku najgorszym wynosi $\Theta(n^2)$, lecz jego czas działania w przypadku średnim jest lepszy — wynosi $\Theta(n \lg n)$.

Sortowanie szybkie ma również dobre czynniki stałe (lepsze niż w sortowaniu przez scalanie) i często jest dobrym algorytmem sortowania do stosowania w praktyce.

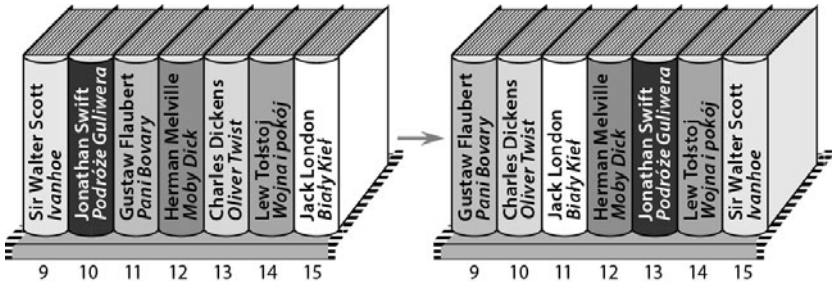
Oto jak metoda „dziel i zwyciężaj" jest wdrożona w sortowaniu szybkim. Raz jeszcze pomyślmy o sortowaniu książek na półce. Podobnie jak w sortowaniu przez scalanie, na początku chcemy posortować wszystkie n książek z przegródek od 1 do n i rozpatrujemy ogólny problem sortowania książek w przegródkach od p do r.

1. **Podziel**, wybierając najpierw dowolną książkę z przegródki między p a r (włącznie). Nazwijmy tę książkę **elementem osiowym** (rozdzielającym, ang. *pivot*). Zreorganizujmy książki na półce w ten sposób, aby wszystkie książki z nazwiskami autorów występującymi [alfabetycznie] przed autorem książki osiowej lub tego samego autora wystąpiły na lewo od elementu osiowego, a wszystkie książki autorów, których nazwiska następują po autorze osiowym, stały na prawo od elementu osiowego.

[10] W oryginale: *master method*, stosujemy jednak terminologię zgodną z polskimi wydaniami *Wprowadzenia do algorytmów* (zob. „Wstęp") — *przyp. tłum.*

[11] To znaczy nie wymaga dodatkowej pamięci poza sortowaną tablicą — *przyp. tłum.*

W tym przykładzie za element osiowy podczas reorganizacji książek w przegródkach od 9 do 15 obieramy książkę stojącą na prawym skraju, napisaną przez Jacka Londona:



Po reorganizacji — którą w sortowaniu szybkim nazywamy **rozdzielaniem** (ang. *partitioning*) — książki Flauberta i Dickensa, występujące alfabetycznie przed Londonem, stoją na lewo od Londona, a wszystkie książki autorów, którzy alfabetycznie występują po Londonie, są z prawej strony. Zauważmy, że po rozdzieleniu książki na lewo od książki Londona nie stoją w żadnym konkretnym porządku (to samo odnosi się do książek stojących na prawo od niej).

2. **Zwyciężaj** przez rekurencyjne sortowanie książek na lewo i na prawo od elementu osiowego. Oznacza to, że jeśli krok podziału przesuwa element osiowy do przegródki q (przegródka 11 w przykładzie powyżej), to powoduje rekurencyjne sortowanie książek w przegródkach od p do $q - 1$ i rekurencyjne sortowanie ich w przegródkach od $q + 1$ do r.
3. **Połącz** — tu nie masz nic do roboty! Z chwilą gdy w kroku zwyciężania nastąpi rekurencyjne posortowanie, wszystko jest gotowe. Dlaczego? Wszystkie książki na lewo od elementu osiowego (w przegródkach od p do $q - 1$) poprzedzają go alfabetycznie lub są tego samego, co on, autora i są posortowane, a wszystkie książki na prawo od elementu osiowego (w przegródkach od $q + 1$ do r) występują alfabetycznie po nim i również są posortowane. Książkom w przegródkach od p do r nie pozostaje nic innego jak być posortowanymi!

Jeśli zamienisz półkę z książkami na tablicę, a książki na elementy tablicy, to strategię sortowania szybkiego masz jak na dłoni. Podobnie jak w sortowaniu przez scalanie, przypadek bazowy występuje wówczas, gdy podtablica do posortowania ma mniej niż dwa elementy.

W procedurze sortowania szybkiego zakłada się, że możemy wywołać procedurę ROZDZIELANIE(A, p, r), która rozdziela podtablicę $A[p..r]$, zwracając indeks q miejsca z elementem osiowym.

Procedura SORTOWANIE-SZYBKIE(A, p, r)

Dane wejściowe i wynik: takie same jak w SORTOWANIU-PRZEZ-SCALANIE.

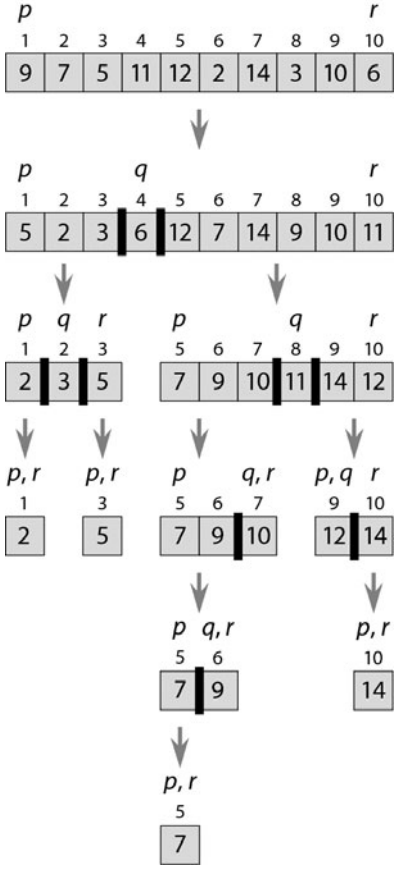
1. Jeśli $p \geq r$, to powróć, nie robiąc niczego.
   1. W przeciwnym razie wykonaj, co następuje:

   A. Wywołaj ROZDZIELANIE(A, p, r) i określ q według jego wyniku.

   B. Wywołaj rekurencyjnie SORTOWANIE-SZYBKIE(A, p, $q - 1$).

   C. Wywołaj rekurencyjnie SORTOWANIE-SZYBKIE(A, $q + 1$, r).

Początkowe wywołanie ma postać SORTOWANIE-SZYBKIE(A, 1, n), podobnie jak w przypadku procedury SORTOWANIE-PRZEZ-SCALANIE. Oto przykład, w jaki sposób rozwija się rekursja, z indeksami p, q i r pokazanymi dla każdej podtablicy, w której $p \leq r$:



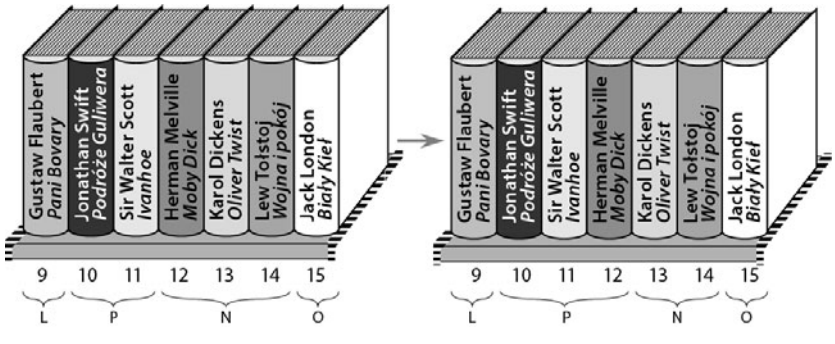
Dla każdej pozycji tablicy wartość pokazana na samym dole stanowi ostateczny, przechowany w niej element. Czytając tablicę od lewej do prawej i spoglądając w każdej jej pozycji na wartość uwidocznioną na samym dole, przekonasz się, że tablica jest naprawdę posortowana.

Najważniejszą rzeczą w sortowaniu szybkim jest rozdzielanie. Zupełnie tak samo jak potrafiliśmy scalić n elementów w czasie $\Theta(n)$, potrafimy rozdzielić n elementów w czasie $\Theta(n)$. Oto jak rozdzielamy książki na półce w przegródkach od p do r. Wybieramy jako element osiowy książkę stojącą na prawym skraju zbioru, czyli książkę w przegródce r. W dowolnej chwili każda książka będzie w jednej z czterech grup, a grupy te będą w przegródkach od p do r, idąc od lewej do prawej:

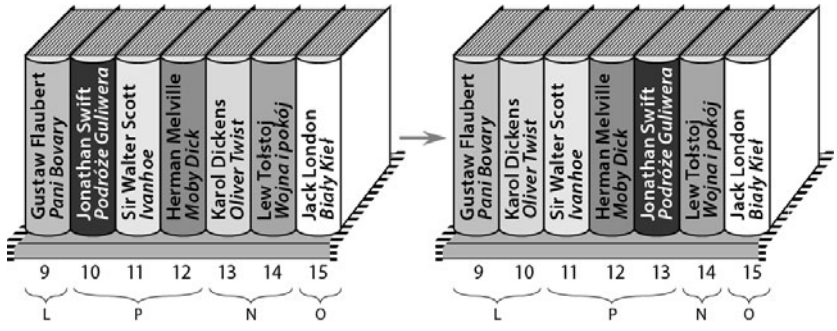
- grupa L (lewa grupa): książki autorów występujących alfabetycznie przed autorem książki osiowej lub napisane przez tego autora; po niej następuje:
- grupa P (prawa grupa): książki autorów następujących alfabetycznie po autorze książki osiowej; po której następuje:
- grupa N (nieznana grupa): książki, których jeszcze nie sprawdziliśmy, nie wiemy więc, jak mają się ich autorzy do autora osiowego; po której następuje:
- grupa O (element osiowy): ta jedna książka obrana za element osiowy.

Przechodzimy przez książki w grupie N od strony lewej do prawej, porównując każdą z elementem osiowym i przenosząc albo do grupy L, albo do P, i zatrzymując się po dotarciu do elementu osiowego. Książka, którą porównujemy z elementem osiowym, jest zawsze pierwszą z lewej w grupie N.

- Jeśli autor książki występuje po autorze książki osiowej, to książka ta staje się skrajną z prawej w grupie P. Ponieważ ta książka była skrajną z lewej w grupie N, a grupa N następuje bezpośrednio po grupie P, musimy poprowadzić linię podziału między grupami P i N, o jedną przegródkę w prawo, bez przesuwania jakiejkolwiek książki:



- Jeśli autor książki występuje [alfabetycznie] przed autorem osiowym lub nim jest, to czynimy tę książkę skrajną prawą w grupie L. Zamieniamy ją ze skrajną lewą książką w grupie P i przesuwamy linie podziału między grupami L i P oraz grupami P i N o jedną przegródkę w prawo:



Gdy dotrzemy do elementu osiowego, zamieniamy go ze skrajną lewą książką w grupie P. W naszym przykładzie kończymy na ustawieniu książek pokazanym na pierwszym rysunku w tym podrozdziale.

Każdą książkę porównujemy z elementem osiowym tylko raz, a każda książka, której autor występuje przed autorem osiowym lub jest tym autorem, powoduje jedną zamianę. Dlatego w celu rozdzielenia n książek wykonujemy najwyżej $n - 1$ porównań (ponieważ elementu osiowego nie musimy porównywać ze sobą) i najwyżej n zamian. Zauważmy, że — w odróżnieniu od scalania — możemy rozdzielić książki bez usuwania ich z półki. To znaczy, że dokonujemy rozdzielenia na miejscu:

Aby przejść od rozdzielania książek do rozdzielania podtablicy $A[p..r]$, wybieramy najpierw $A[r]$ (element z prawego skraju) jako element osiowy. Następnie przechodzimy przez podtablicę od lewej do prawej, porównując każdy element z osiowym. Utrzymujemy indeksy q i u podtablicy, które dzielą ją następująco:

- Podtablica $A[p..q - 1]$ odpowiada grupie L — każdy element jest mniejszy lub równy elementowi osiowemu.
- Podtablica $A[q..u - 1]$ odpowiada grupie P — każdy element jest większy niż element osiowy.
- Podtablica $A[u..r - 1]$ odpowiada grupie N — nie wiemy jeszcze, jak mają się jej elementy do elementu osiowego.
- Podtablica $A[r]$ odpowiada grupie O — przechowuje element osiowy.

Te podziały są w istocie niezmiennikami pętli. (Nie będziemy ich wszakże dowodzić).

W każdym kroku porównujemy $A[u]$, skrajny z lewej element w grupie N, z elementem osiowym. Jeśli $A[u]$ jest większy od elementu osiowego, to zwiększamy u, aby przesunąć w prawo linię podziału między grupami P i N. Jeśli natomiast $A[u]$ jest mniejszy lub równy elementowi osiowemu, to zamieniamy element w $A[q]$ (położony z lewego skraju element w grupie P) z elementem $A[u]$, po czym zwiększamy zarówno q, jak i u, aby przesunąć w prawo linie podziału między grupami L i P oraz P i N. Oto procedura ROZDZIELANIE:

Procedura ROZDZIELANIE(*A*, *p*, *r*)

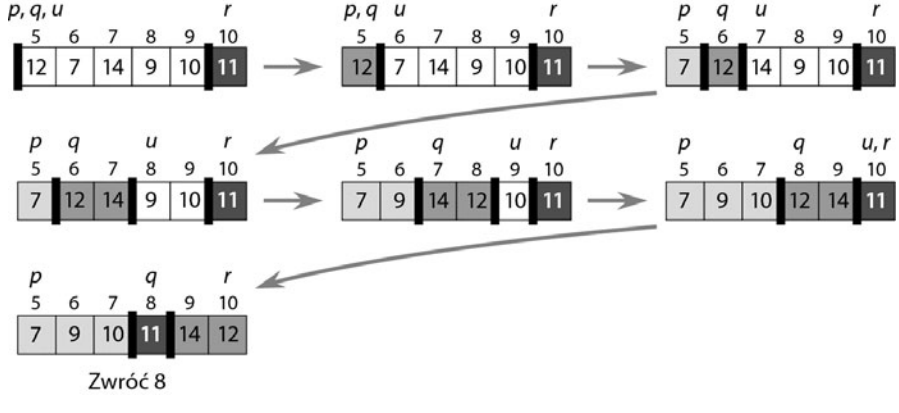
Dane wejściowe: takie same jak w SORTOWANIU-PRZEZ-SCALANIE.

Wynik: reorganizuje elementy *A*[*p*..*r*] tak, aby każdy element w *A*[*p*..*q* – 1] był mniejszy lub równy *A*[*q*] i każdy element w *A*[*q* + 1..*r*] był większy niż *q*. Zwraca wywołującemu indeks *q*.

1. Nadaj *q* wartość *p*.
2. Dla *u* = *p* do *r* – 1 wykonuj:
   A. Jeśli *A*[*u*] ≤ *A*[*r*], to zamień *A*[*q*] z *A*[*u*] i zwiększ *q* o 1.
   B. Zamień *A*[*q*] z *A*[*r*] i zwróć *q*.

Wskutek rozpoczęcia od nadania obu indeksom *q* i *u* wartości *p* grupy L (*A*[*p*.. *q* – 1]) i P (*A*[*q*..*u* – 1]) są początkowo puste, a grupa N (*A*[*u*..*r* – 1]) zawiera wszystkie elementy z wyjątkiem osiowego. W pewnych sytuacjach, jeśli na przykład *A*[*p*] ≤ A[*r*], element może być zamieniany z samym sobą, co nie powoduje w tablicy żadnej zmiany. Krok 3 kończy się zamianą elementu osiowego z lewym skrajnym elementem grupy P, co skutkuje przesunięciem elementu osiowego na właściwe miejsce w rozdzielanej tablicy i zwróceniem nowego indeksu *q* elementu osiowego.

Dalej opisujemy krok po kroku, jak działa procedura ROZDZIELANIE na podtablicy *A*[5..10] utworzonej przez pierwsze rozdzielenie w przykładzie sortowania szybkiego. Grupę N oznaczono na biało, grupa L jest jasnoszara, grupa P ma ciemniejszy odcień szarości, a najciemniejszym odcieniem zaznaczono element osiowy, czyli grupę O. Pierwsza część rysunku ukazuje początkowy stan tablicy i indeksów, pięć następnych przedstawia tablicę i indeksy po każdej iteracji pętli z kroku 2 (łącznie ze zwiększaniem indeksu *u* na końcu każdej iteracji), a ostatnia część przedstawia końcową, rozdzieloną tablicę:



Jak wtedy gdy rozdzielaliśmy książki, porównujemy każdy element z elementem osiowym i wykonujemy najwyżej jedną zamianę dla każdego elementu, który porównaliśmy. Ponieważ każde porównanie zajmuje stały czas i każda zamiana zajmuje stały czas, łączny czas ROZDZIELANIA na podtablicy n-elementowej wynosi $\Theta(n)$.

Ile zatem trwa procedura SORTOWANIE-SZYBKIE? Podobnie jak w sortowaniu przez scalanie, powiedzmy, że sortowanie podtablicy n-elementowej zajmuje czas $T(n)$ i jest funkcją rosnącą z n. Podział wykonywany przez procedurę ROZDZIELANIE zajmuje $\Theta(n)$ czasu. Czas SORTOWANIA-SZYBKIEGO zależy jednak i od tego, na ile zrównoważone okazuje się rozdzielanie.

W najgorszym przypadku rozmiary podziału są naprawdę niezrównoważone. Jeśli każdy element, poza samym osiowym, jest mniejszy od osiowego, to ROZDZIELANIE kończy się zostawieniem elementu osiowego w $A[r]$ i zwraca procedurze SORTOWANIE-SZYBKIE indeks r, który zapamiętuje ona w zmiennej q. W tym przypadku część $A[q + 1..r]$ jest pusta, a część $A[p..q - 1]$ jest tylko o jeden element mniejsza od $A[p..r]$. Wywołanie rekurencyjne z podtablicą pustą zabiera $\Theta(1)$ czasu (czas spowodowania wywołania i ustalenia w kroku 1, że podtablica jest pusta). Możemy ten czas $\Theta(1)$ włączyć po prostu do czasu rozdzielania $\Theta(n)$. Jeśli jednak $A[p..r]$ ma n elementów, to $A[p..q - 1]$ ma $n - 1$ elementów, przeto rekurencyjne wywołanie na $A[p..q - 1]$ zabiera $T(n - 1)$ czasu. Otrzymujemy rekurencję:

$$T(n) = T(n - 1) + \Theta(n).$$

Nie uda nam się rozwiązać tej zależności metodą rekurencji uniwersalnej, lecz ma ona rozwiązanie: $T(n)$ wynosi $\Theta(n^2)$. To nie lepiej niż w sortowaniu przez wybieranie! Jak dochodzi do tak nierównego podziału? Jeśli każdy element osiowy jest większy od wszystkich innych elementów, to tablica musiała być już na początku prawie posortowana. Okazuje się również, że nierówny podział otrzymujemy zawsze wtedy, kiedy tablica jest na początku posortowana w odwrotnej kolejności.

Z drugiej strony, gdybyśmy za każdym razem otrzymywali równy podział, to każda podtablica miałaby najwyżej $n / 2$ elementów. Rekurencja byłaby taka sama jak dla sortowania przez scalanie:

$$T(n) = 2T(n / 2) + \Theta(n),$$

z tym samym co tam rozwiązaniem: $T(n)$ wynosi $\Theta(n \lg n)$. Oczywiście musielibyśmy naprawdę mieć szczęście albo dane wejściowe musiałyby być bardzo wydumane, żeby otrzymać idealnie równy podział za każdym razem.

Typowy przypadek mieści się gdzieś pomiędzy przypadkiem najlepszym i najgorszym. Analiza techniczna jest zawiła i nie chcę Cię w nią wciągać, jeśli jednak elementy tablicy wejściowej wykazują układ losowy, to otrzymujemy średnio podziały na tyle bliskie równym, że SORTOWANIE-SZYBKIE zajmuje $\Theta(n \lg n)$ czasu.

Poszalejmy teraz. Przypuśćmy, że Twój najgorszy wróg dał Ci do posortowania pewną tablicę i wiedząc, że jako element osiowy w każdej podtablicy zawsze wybierasz ostatni, tak zaaranżował jej zawartość, abyś zawsze otrzymywał najgorszy przypadek podziału. Jak mógłbyś pokrzyżować mu plany? Mógłbyś najpierw

sprawdzić, czy tablica jest na początku posortowana lub uporządkowana w odwrotnej kolejności, i zrobić w takich przypadkach coś specjalnego. Wtedy Twój wróg mógłby się znów postarać i spreparować tablicę, dla której podziały są zawsze złe, choć nie w stopniu maksymalnym. Nie chciałoby Ci się sprawdzać każdego z możliwych złych przypadków.

Na szczęście istnieje znacznie prostsze rozwiązanie: nie wybierać jako osiowego zawsze ostatniego elementu. Lecz wówczas nasza śliczna procedura ROZDZIELANIE przestanie działać, ponieważ grupy będą nie takie, jakich się oczekuje. Nie jest to jednak problemem. Przed wykonaniem procedury ROZDZIELANIE zamień $A[r]$ z losowo wybranym elementem z $A[p..r]$. Teraz już wybrałeś element osiowy losowo i *możesz* wykonać procedurę ROZDZIELANIE.

Dokładając trochę więcej wysiłku, mógłbyś w istocie jeszcze podwyższyć szansę uzyskania podziału bliskiego równemu. Zamiast losowego wybierania elementu z $A[p..r]$ wybierz losowo trzy elementy i zamień $A[i]$ z medianą tych trzech. Przez medianę trzech wartości rozumiemy element, którego wartość mieści się między pozostałymi dwoma. (Jeśli dwa lub więcej losowo wybranych elementów jest równych, to postąp dowolnie). Tu również nie chcę Cię wciągać w forsowną analizę, musiałbyś jednak mieć wyjątkowego pecha z każdorazowym losowym wyborem elementów, aby SORTOWANIE-SZYBKIE zajęło więcej czasu niż $\Theta(n \lg n)$. Co więcej, o ile Twój wróg nie dostałby się do Twojego generatora liczb losowych, nie zdołałby przejąć żadnej kontroli nad stopniem równości podziałów.

Ile razy SORTOWANIE-SZYBKIE zamienia elementy? To zależy od tego, czy uwzględniasz jako zamianę sytuację, w której element jest „zamieniany" z samym sobą, na tej samej pozycji. Mógłbyś oczywiście sprawdzać, czy nie zachodzi ten przypadek, i unikać takich zamian. Mówmy zatem o zamianie tylko wtedy, kiedy w jej wyniku element naprawdę przemieszcza się po tablicy, to znaczy gdy $q \neq u$ w kroku 2A lub gdy $q \neq r$ w kroku 3 ROZDZIELANIA. Najlepszy przypadek, jeśli chodzi o minimalizowanie zamian, jest zarazem jednym z najgorszych przypadków asymptotycznego czasu działania — gdy tablica jest prawie posortowana. Nie ma wtedy zamian. Najwięcej zamian występuje wtedy, kiedy n jest parzyste, a tablica wejściowa jest czymś w rodzaju n, $n - 2$, $n - 4$,..., 4, 2, 1, 3, 5,..., $n - 3$, $n - 1$. Dochodzi wtedy do $n^2/4$ zamian, a asymptotyczny czas działania wciąż jest przypadkiem najgorszym: $\Theta(n^2)$.

Podsumowanie

W tym i w poprzednim rozdziale obejrzeliśmy cztery algorytmy wyszukiwania i cztery algorytmy sortowania. Zestawmy ich cechy w kilku tabelach. Zważywszy, że trzy algorytmy wyszukiwania z rozdziału 2 były jedynie wariacjami na temat, jako reprezentanta wyszukiwania liniowego możemy wziąć pod uwagę albo LEPSZE-WYSZUKIWANIE-LINIOWE, albo WYSZUKIWANIE-LINIOWE-Z-WARTOWNIKIEM.

Algorytmy wyszukiwania

Algorytm	Czas działania w przypadku najgorszym	Czas działania w przypadku najlepszym	Wymaga tablicy posortowanej?
Wyszukiwanie liniowe	$\Theta(n)$	$\Theta(1)$	nie
Wyszukiwanie binarne	$\Theta(\ln n)$	$\Theta(1)$	tak

Algorytmy sortowania

Algorytm	Czas działania w przypadku najgorszym	Czas działania w przypadku najlepszym	Najgorszy przypadek wymian	Na miejscu?
Sortowanie przez wybieranie	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	tak
Sortowanie przez wstawianie	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	tak
Sortowanie przez scalanie	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	nie
Sortowanie szybkie	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n^2)$	tak

W tych tabelach nie ujęto czasów działania w przypadku średnim, ponieważ — z zasługującym na uwagę wyjątkiem w postaci sortowania szybkiego — są one równe czasom działania w przypadku najgorszym. Jak widzieliśmy, jeśli założyć, że tablica ma na początku uporządkowanie losowe, czas działania sortowania szybkiego w przypadku średnim wynosi tylko $\Theta(n \lg n)$.

Jak przedstawia się porównanie tych algorytmów w praktyce? Zakodowałem je w C++ i wykonałem na tablicach 4-bajtowych liczb całkowitych, używając dwóch różnych maszyn: mojego macbooka pro (na którym napisałem tę książkę) z 2,4-gigahercowym procesorem Intel Core 2 Duo i 4 GB pamięci RAM, pracującym pod kontrolą systemu operacyjnego Mac OS 10.6.8, oraz komputera Dell PC (serwera mojej witryny sieciowej) z 3,2-gigahercowym procesorem Intel Pentium 4 i 1 GB RAM, pracującym pod nadzorem systemu operacyjnego Linux w wersji 2.6.22.14. Do kompilacji kodu użyłem kompilatora g++ na poziomie optymalizacji -03. Każdy algorytm wykonałem na tablicach o rozmiarach do 50 000 elementów, przy czym każda tablica była początkowo posortowana w odwrotnym porządku. Uśredniłem czasy 20 wykonań każdego algorytmu na każdej z tablic.

Zaczynając zawsze od tablicy odwrotnie uporządkowanej, wymuszałem wystąpienie najgorszych przypadków asymptotycznych czasów działania, zarówno w algorytmie sortowania przez wstawianie, jak i w sortowaniu szybkim. Biorąc to pod uwagę, wykonałem dwie wersje sortowania szybkiego: „zwyczajne" sortowanie szybkie, zawsze obierające jako osiowy element $A[r]$, ostatni w rozdzielanej podtablicy $A[p..r]$, i randomizowane sortowanie szybkie, zamieniające przed rozdzielaniem losowo wybrany element z $A[p..r]$ z elementem $A[r]$. (Nie wykonywałem metody

mediany trzech wartości). „Zwyczajne" sortowanie szybkie jest także nazywane **deterministycznym**, ponieważ nie używa losowości; z chwilą otrzymania na wejściu tablicy do posortowania wszystko odbywa się w nim w sposób z góry określony.

Randomizowane sortowanie szybkie okazało się najlepsze dla $n \geq 64$ na obu komputerach. Oto ujęte w postaci proporcji porównanie czasów działania innych algorytmów z czasami działania randomizowanego sortowania szybkiego dla różnych rozmiarów danych wejściowych:

MacBook Pro

	n						
Algorytm	50	100	500	1000	5000	10 000	50 000
Sortowanie przez wybieranie	1,34	2,13	8,04	13,13	59,07	114,24	537,42
Sortowanie przez wstawianie	1,08	2,02	6,15	11,35	51,86	100,38	474,29
Sortowanie przez scalanie	7,58	7,64	6,93	6,87	6,35	6,20	6,27
Deterministyczne sortowanie szybkie	1,02	1,63	6,09	11,51	52,02	100,57	475,34

Dell PC

	n						
Algorytm	50	100	500	1000	5000	10 000	50 000
Sortowanie przez wybieranie	0,76	1,60	5,46	12,23	52,03	100,79	496,94
Sortowanie przez wstawianie	1,01	1,66	7,68	13,90	68,34	136,20	626,44
Sortowanie przez scalanie	3,21	3,38	3,57	3,33	3,36	3,37	3,15
Deterministyczne sortowanie szybkie	1,12	1,37	6,52	9,30	47,60	97,45	466,83

Randomizowane sortowanie szybkie wygląda nieźle, lecz możemy je przebić. Przypomnijmy, że sortowanie przez wstawianie działa dobrze, jeśli nie trzeba przesuwać elementów w tablicy zbyt daleko. Gdyby zatem sprowadzić rozmiary podproblemów w algorytmach rekurencyjnych do pewnego k, to żaden element nie musiałby być przesuwany dalej niż o $k - 1$ pozycji. Co by się stało, gdybyśmy po osiągnięciu małych rozmiarów podproblemu, zamiast rekurencyjnie kontynuować wywoływanie randomizowanego sortowania szybkiego, wykonali sortowanie przez wstawianie odpowiednio zmodyfikowane do sortowania podtablicy, a nie całej tablicy? W istocie, posługując się taką hybrydową metodą, możemy sortować jeszcze sprawniej niż za pomocą randomizowanego sortowania szybkiego. Zaobserwo-

wałem, że na moim macbooku pro optymalnym punktem przejścia była podtablica o rozmiarze 22, a na PC-cie punktem takim była podtablica o rozmiarze 17. Oto proporcje czasów działania algorytmu hybrydowego i randomizowanego sortowania szybkiego na obu maszynach dla problemów tych samych rozmiarów:

	n						
Maszyna	50	100	500	1000	5000	10 000	50 000
MacBook Pro	0,55	0,56	0,60	0,60	0,62	0,63	0,66
PC	0,53	0,58	0,60	0,58	0,60	0,64	0,64

Czy dałoby się pokonać czas sortowania $\Theta(n \lg n)$? To zależy. W rozdziale 4 zobaczymy, że gdy jedynym sposobem określenia miejsca umieszczania elementów jest ich porównywanie, względnie gdy chodzi o robienie czegoś innego opartego na wynikach porównywania — to nie, nie przebijemy czasu $\Theta(n \lg n)$. Jeśli jednak o elementach wiemy coś więcej, co moglibyśmy wykorzystać, to możemy to zrobić lepiej.

Co czytać dalej

Materiał zawarty w CLRS [CLRS09] obejmuje sortowanie przez wstawianie, sortowanie przez scalanie i oba sortowania szybkie: deterministyczne i randomizowane. Protoplastą książek o sortowaniu i wyszukiwaniu pozostaje jednak tom 3 *The Art of Computer Programming* autorstwa Knutha [Knu98b]; przy czym zachowuje tu ważność uwaga z rozdziału 1: *TAOCP* jest dziełem głębokim i wymagającym dużej mobilizacji[12].

[12] *Sztuka programowania*, WNT, Warszawa 2002 – 2008 — *przyp. tłum.*

4 Dolne ograniczenie sortowania i sposoby jego przezwyciężenia

W poprzednim rozdziale poznaliśmy cztery algorytmy sortowania n elementów w tablicy. Czasy działania dwóch z nich: sortowania przez wybieranie i sortowania przez wstawianie, wynoszą w najgorszym przypadku $\Theta(n^2)$, co nie jest zadowalające. Jeden z nich, sortowanie szybkie, również ma w najgorszym przypadku czas działania $\Theta(n^2)$, jednak w przypadku średnim potrzebuje znacznie mniej czasu: $\Theta(n \lg n)$. Sortowanie przez scalanie zabiera czas $\Theta(n \lg n)$ we wszystkich przypadkach. W praktyce sortowanie szybkie jest najszybsze z tych czterech, lecz gdyby Ci bezwzględnie zależało na uchronieniu się przed działaniem występującym w najgorszym przypadku, powinieneś wybrać sortowanie przez scalanie.

Czy czas $\Theta(n \lg n)$ jest tak dobry, jak to możliwe? Czy można obmyślić algorytm sortowania, który pobiłby czas $\Theta(n \lg n)$ w przypadku najgorszym? Odpowiedź zależy od reguł gry, tj. sposobu, w jaki algorytmowi sortowania wolno używać kluczy sortowania do ustalania porządku.

W tym rozdziale zobaczymy, że stosując pewien zbiór reguł, nie przezwyciężymy $\Theta(n \lg n)$. Potem poznamy dwa algorytmy sortowania — sortowanie przez zliczanie i sortowanie pozycyjne, które wychodzą poza te reguły i dzięki temu potrafią sortować w czasie zaledwie $\Theta(n)$.

Reguły sortowania

Jeśli przyjrzysz się sposobowi, za pomocą którego cztery algorytmy z poprzedniego rozdziału używają kluczy sortowania, to przekonasz się, że wyznaczają one uporządkowanie wyłącznie na podstawie porównywania par kluczy. Wszystkie podejmowane przez nie decyzje mają następującą postać: „Jeśli ten element ma klucz sortowania mniejszy niż klucz sortowania tamtego, to zrób coś, a jeśli nie, to zrób coś innego albo nie rób nic". Można by dojść do wniosku, że algorytm sortowania może podejmować decyzje *tylko* w ten sposób. Czy algorytm sortowania mógłby podejmować decyzje inaczej? Jak?

Aby zobaczyć, jakie inne rodzaje decyzji są możliwe, rozważmy naprawdę prostą sytuację. Przypuśćmy, że o sortowanych elementach wiemy dwie rzeczy: każdy klucz sortowania jest równy 1 lub 2, a elementy składają się tylko z kluczy sortowania — nie mają żadnych danych towarzyszących. W tej prostej sytuacji zdołamy posortować n elementów już w czasie $\Theta(n)$, pokonując algorytmy z czasem $\Theta(n \lg n)$ z poprzedniego rozdziału. W jaki sposób? Najpierw przechodzimy wszystkie elementy i zliczamy, ile z nich ma wartość 1. Powiedzmy, że jest k takich elementów.

Następnie przechodzimy tablicę, wypełniając ją wartościami 1 na k pierwszych pozycjach i wartościami 2 na pozostałych $n - k$ pozycjach. Oto procedura:

Procedura NAPRAWDĘ-PROSTE-SORTOWANIE(A, n)

Dane wejściowe:

- A: tablica, której elementami są tylko 1 lub 2.
- n: liczba elementów w A do posortowania.

Wynik: elementy A są posortowane w porządku niemalejącym.

1. Nadaj k wartość 0.
2. Dla i = 1 do n:
   A. Jeśli $A[i] = 1$, to zwiększ k o 1.
3. Dla i = 1 do k:
   A. Podstaw 1 do $A[i]$.
4. Dla $i = k + 1$ do n:
   A. Podstaw 2 do $A[i]$.

Kroki 1 i 2 zliczają jedynki, zwiększając licznik k dla każdego elementu $A[i]$, który równa się 1. W kroku 3 zapełnia się $A[1..k]$ jedynkami, a w kroku 4 pozostałe pozycje, $A[k + 1..n]$, są zapełniane dwójkami. Nietrudno wykazać, że ta procedura działa w czasie $\Theta(n)$: pierwsza pętla jest wykonywana n razy, dwie pozostałe pętle razem wzięte są iterowane n razy, a każda iteracja każdej pętli zajmuje stałą porcję czasu.

Zauważmy, że w NAPRAWDĘ-PROSTYM-SORTOWANIU nie ma żadnego porównywania elementów dwóch tablic *ze sobą*. Każdy element tablicy jest porównywany z wartością 1, lecz nigdy z innym elementem tablicy. Widzisz zatem, że przy takich obwarowaniach możemy sortować bez porównywania par kluczy sortowania.

Dolne ograniczenie sortowania przez porównania

Obecnie, kiedy masz już pojęcie, jak można zmieniać reguły gry, spójrzmy na dolne ograniczenie szybkości, z którą możemy sortować.

Sortowaniem przez porównania (ang. *comparison sort*) określimy dowolny algorytm, który wyznacza porządek wyłącznie przez porównywanie par elementów. Cztery algorytmy sortowania z poprzedniego rozdziału są sortowaniami przez porównania, natomiast NAPRAWDĘ-PROSTE-SORTOWANIE — nie.

Oto ograniczenie dolne:

> Każdy algorytm sortowania n elementów przez porównania wymaga w najgorszym przypadku $\Omega(n \lg n)$ porównań par elementów.

Przypomnijmy, że notacja Ω podaje ograniczenie dolne, czyli to, co wyrażamy słowami „dla dostatecznie dużego n algorytm sortowania wymaga w najgorszym przypadku przynajmniej $cn \lg n$ porównań, gdzie c jest pewną stałą". Ponieważ każde porównanie zajmuje przynajmniej stałą ilość czasu, daje to $\Omega(n \lg n)$ jako czas dolnego ograniczenia na posortowanie n elementów, przy założeniu, że używamy algorytmu sortowania przez porównania.

Ważne jest zrozumienie kilku spraw dotyczących ograniczenia dolnego. Po pierwsze, mówi ono coś tylko o przypadku najgorszym. Zawsze możesz skonstruować algorytm sortowania działający w czasie liniowym w przypadku najlepszym: umówmy się, że przypadek najlepszy występuje wówczas, gdy tablica jest już posortowana i wystarczy sprawdzić, że każdy element (z wyjątkiem ostatniego) jest mniejszy lub równy elementowi następującemu po nim w tablicy. Łatwo tego dokonać w czasie $\Theta(n)$, a gdy się już przekonasz, że każdy element jest mniejszy lub równy swojemu następnikowi, masz kłopot z głowy. Jednakże w *przypadku najgorszym* niezbędne jest wykonanie $\Omega(n \lg n)$ porównań. To ograniczenie dolne nazywany **egzystencjalnym**, ponieważ mówi ono, że istnieją dane wejściowe, które wymagają $\Omega(n \lg n)$ porównań. Innego rodzaju ograniczeniem dolnym jest **uniwersalne** ograniczenie dolne, które stosuje się do wszystkich danych wejściowych. Dla sortowania jedynym uniwersalnym ograniczeniem dolnym, jakie mamy, jest $\Omega(n)$, ponieważ na każdy element musimy spojrzeć choć raz. Zwróć uwagę, że w poprzednim zdaniu nie powiedziałem, *czego* ograniczeniem jest $\Omega(n)$. Czy miałem na myśli $\Omega(n)$ porównań, czy $\Omega(n)$ czasu? Myślałem o czasie $\Omega(n)$, ponieważ jest zrozumiałe, że musimy sprawdzić każdy element nawet wówczas, gdy nie porównujemy par elementów.

Druga istotna sprawa ma naprawdę doniosłe znaczenie: to ograniczenie dolne nie zależy od konkretnego algorytmu, jeżeli tylko jest to algorytm sortowania przez porównania. To ograniczenie dolne stosuje się do *każdego* algorytmu sortowania przez porównania, niezależnie od tego, jak byłby prosty czy skomplikowany. Ma ono zastosowanie do algorytmów sortujących przez porównania, które już wynaleziono, a także tych, które zostaną wynalezione w przyszłości. Odnosi się nawet do algorytmów sortujących przez porównania, których ludzkość nigdy nie wymyśli!

Pokonywanie ograniczenia dolnego w sortowaniu przez zliczanie

Zobaczyliśmy już, jak można pokonać ograniczenie dolne w warunkach mocno wygórowanych: klucze sortowania mogą mieć tylko dwie wartości i każdy element składa się wyłącznie z klucza sortowania, bez danych towarzyszących. W tym restrykcyjnym przypadku potrafimy posortować n elementów już w czasie $\Theta(n)$, bez porównywania elementów.

Możemy uogólnić metodę NAPRAWDĘ-PROSTEGO-SORTOWANIA na obsługiwanie m różnych wartości kluczy sortowania, o ile tylko są one kolejnymi m liczbami całkowitymi z pewnego przedziału, powiedzmy od 0 do $m - 1$, i możemy też zezwolić na posiadanie przez elementy danych towarzyszących.

Oto pomysł. Załóżmy, że wiemy, iż klucze sortowania są liczbami całkowitymi z przedziału od 0 do $m - 1$. Załóżmy dalej, że jest nam wiadome, iż dokładnie trzy elementy mają klucze sortowania równe 5 i dokładnie sześć elementów ma klucze sortowania mniejsze niż 5 (tzn. z przedziału od 0 do 4). Wiemy wówczas, że w posortowanej tablicy elementy z kluczami sortowania równymi 5 powinny zająć pozycje 7, 8 i 9. Uogólniając, jeśli wiemy, że k elementów ma klucz sortowania równy x, a l elementów ma klucze sortowania mniejsze niż x, to wiadomo nam również, że w posortowanej tablicy elementy o kluczach sortowania równych x powinny zająć pozycje od $l + 1$ do $l + k$. Dlatego dla każdej możliwej wartości klucza sortowania chcemy obliczyć, ile elementów ma klucz sortowania mniejszy od danej wartości, a ile ma klucz sortowania jej równy.

Obliczenia liczby elementów mających klucze sortowania mniejsze od każdej z możliwych wartości klucza możemy dokonać, wyliczając najpierw, ile elementów ma klucz sortowania równy danej wartości, zacznijmy więc od tego:

Procedura POLICZ-KLUCZE-RÓWNE(A, n, m)

Dane wejściowe:

- A: tablica liczb całkowitych z przedziału od 0 do $m - 1$.
- n: liczba elementów w A.
- m: określa przedział wartości w A.

Wynik: tablica *równe*[0..$m - 1$], taka że *równe*[j] zawiera liczbę elementów A równych j, dla $j = 0, 1, 2, \ldots, m - 1$.

1. Niech *równe*[0..$m - 1$] będzie nową tablicą.
2. Ustaw wszystkie wartości w *równe* na 0.
3. Dla i = 1 do n:
   A. Podstaw $A[i]$ do *klucz*.
   B. Zwiększ o 1 *równe*[*klucz*].
4. Zwróć tablicę *równe*.

Zwróćmy uwagę, że POLICZ-KLUCZE-RÓWNE nigdy nie porównuje kluczy sortowania między sobą. Używa kluczy sortowania tylko do indeksowania w tablicy *równe*. Ponieważ w pierwszej pętli (występującej niejawnie w kroku 2) jest m iteracji, druga pętla (krok 3) stanowi n iteracji, a każda iteracja każdej pętli zużywa stały czas, procedura POLICZ-KLUCZE-RÓWNE zużywa $\Theta(m + n)$ czasu. Jeśli m jest stałą, to POLICZ-KLUCZE-RÓWNE zużywa $\Theta(n)$ czasu.

Możemy obecnie zastosować tablicę *równe* do sumowania na bieżąco liczb elementów mających klucze sortowania mniejsze od poszczególnych wartości:

Procedura POLICZ-KLUCZE-MNIEJSZE(*równe*, *m*)

Dane wejściowe:

- *równe*: tablica zwrócona przez procedurę POLICZ-KLUCZE-RÓWNE.
- *m*: określa przedział indeksów tablicy *równe*: od 0 do *m* – 1.

Wynik: tablica *mniejsze*[0..*m* – 1], taka że dla *j* = 0, 1, 2,..., *m* – 1 element *mniejsze*[*j*] zawiera sumę *równe*[0] + *równe*[1] + ... + *równe*[*j* – 1].

1. Niech *mniejsze*[0..*m* – 1] będzie nową tablicą.
2. Nadaj *mniejsze*[0] wartość 0.
3. Dla *i* = 1 do *m* – 1:
   A. Nadaj *mniejsze*[*j*] wartość *mniejsze*[*j* – 1] + *równe*[*j* – 1].
4. Zwróć tablicę *mniejsze*.

Zakładając, że *równe*[*j*] podaje dokładną liczbę kluczy sortowania równych *j*, dla *j* = 0, 1,..., *m* – 1, mógłbyś użyć następującego niezmiennika pętli, aby wykazać, że po wyjściu z procedury POLICZ-KLUCZE-MNIEJSZE element *mniejsze*[*j*] zawiera liczbę kluczy sortowania mniejszych niż *j*:

> Na początku każdej iteracji pętli z kroku 3. *mniejsze*[*j* – 1] równa się liczbie kluczy sortowania mniejszych niż *j* – 1.

Pozostawiam Ci sformułowanie części dotyczących inicjowania, utrzymania i zakończenia. Z łatwością zauważysz, że procedura POLICZ-KLUCZE-MNIEJSZE działa w czasie $\Theta(m)$. Z pewnością też nie porównuje kluczy sortowania między sobą.

Spójrzmy na przykład. Przypuśćmy, że *m* = 7 (wobec tego wszystkie klucze sortowania są liczbami całkowitymi z przedziału od 0 do 6) oraz że mamy następującą tablicę *A* z *n* = 10 elementami: *A* = {4, 1, 5, 0, 1, 6, 5, 1, 5, 3}. Wtedy *równe* = {1, 3, 0, 1, 1, 3, 1}, a *mniejsze* = {0, 1, 4, 4, 5, 6, 9}. Ponieważ *mniejsze*[5] = 6, a *równe*[5] = 3 (pamiętajmy, że tablice *mniejsze* i *równe* indeksujemy, poczynając od 0, a nie 1), po skończeniu sortowania pozycje od 1 do 6 powinny zawierać wartości kluczy mniejsze niż 5, a pozycje 7, 8 i 9 powinny zawierać klucz o wartości 5.

Mając tablicę *mniejsze*, możemy utworzyć tablicę posortowaną, choć nie na miejscu:

Procedura REORGANIZUJ(*A*, *mniejsze*, *n*, *m*)

Dane wejściowe:

- *A*: tablica liczb całkowitych z przedziału od 0 do *m* – 1.
- *mniejsze*: tablica zwrócona przez POLICZ-KLUCZE-MNIEJSZE.
- *n*: liczba elementów w *A*.
- *m*: określa przedział wartości w *A*.

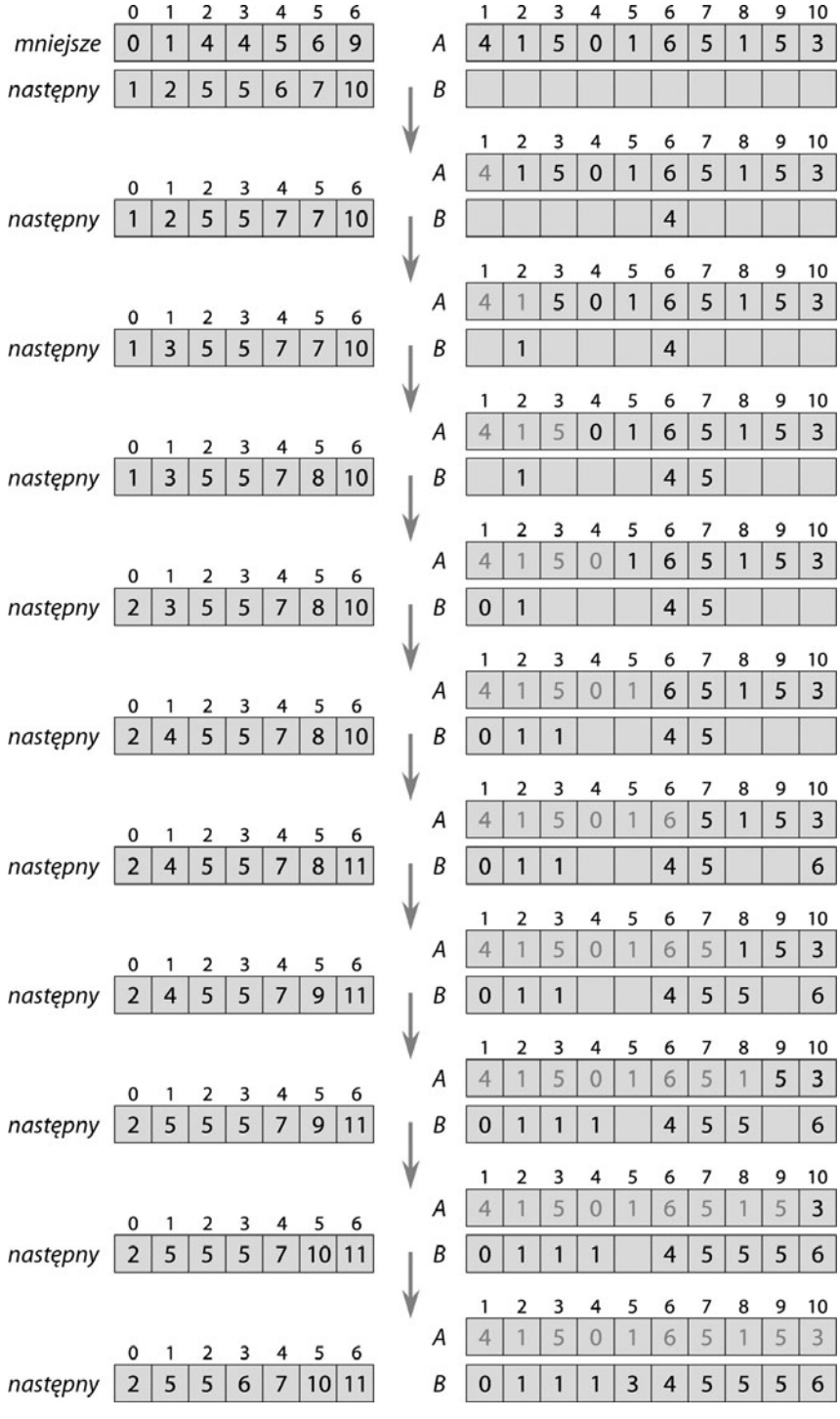
Wynik: tablica *B* zawierająca posortowane elementy *A*.

1. Niech *B*[1..*n*] i *następny*[0..*m* – 1] będą nowymi tablicami.
2. Dla *j* = 0 do *m* – 1:
   A. Podstaw do *następny*[*j*] wartość *mniejsze*[*j*] + 1.
3. Dla i = 1 do *n*:
   A. Ustaw *klucz* na wartość *A*[*i*].
   B. Ustaw *indeks* na wartość *następny*[*klucz*].
   C. Podstaw *A*[*i*] do *B*[*indeks*].
   D. Zwiększ o 1 *następny*[*klucz*].
4. Zwróć tablicę *B*.

Na rysunku zamieszczonym na następnej stronie pokazano, w jaki sposób procedura REORGANIZUJ przenosi elementy z tablicy *A* do tablicy *B*, aby skończyło się to ich uporządkowaniem w *B*. U góry pokazano tablice *mniejsze*, *następny*, *A* i *B* przed pierwszą iteracją pętli z kroku 3, po czym kolejno tablice *następny*, *A* i *B* po każdej iteracji. Elementy *A* przekopiowane do B zaznaczono szarymi numerami.

Pomysł jest następujący. Podczas przechodzenia przez tablicę *A* od początku do końca tablica *następny* podaje indeks w tablicy *B*, pod który powinien trafić następny element z *A* z kluczem *j*. Przypomnijmy, co już wcześniej powiedziano, że jeśli *l* elementów ma klucze sortowania mniejsze niż *x*, to *k* elementów o kluczach sortowania równych *x* powinno zająć pozycje od *l* + 1 do *l* + *k*. Pętla w kroku 2 tak ustawia tablicę *następny*, aby na początku było *następny*[*j*] = *l* + 1, gdzie *l* = *mniejszy*[*j*]. Pętla z kroku 3 przebiega przez tablicę *A* od początku do końca. Dla każdego elementu *A*[*i*] w kroku 3A następuje zapamiętanie *A*[*i*] w zmiennej *klucz*, krok 3B oblicza *indeks* w *B*, pod który powinien trafić element *A*[*i*], a w kroku 3C następuje przeniesienie *A*[*i*] na tę pozycję w *B*. Ponieważ następny element w tablicy *A* o takim samym kluczu sortowania co *A*[*i*] (jeśli taki istnieje) powinien trafić na następną pozycję w *B*, w kroku 3D następuje zwiększenie *następny*[*klucz*] o 1.

Ile trwa wykonanie procedury REORGANIZUJ? Pętla w kroku 2 działa w czasie $\Theta(m)$, a pętla w kroku 3 działa w czasie $\Theta(n)$. Wobec tego, podobnie jak w POLICZ-KLUCZE-RÓWNE, REORGANIZUJ działa w czasie $\Theta(m + n)$, czyli $\Theta(n)$, jeśli *m* jest stałą.


mniejsze 0 1 4 4 5 6 9
następny 1 2 5 5 6 7 10
A 4 1 5 0 1 6 5 1 5 3
B
następny 1 2 5 5 7 7 10
A 4 1 5 0 1 6 5 1 5 3
B 4
następny 1 3 5 5 7 7 10
A 4 1 5 0 1 6 5 1 5 3
B 1 4
następny 1 3 5 5 7 8 10
A 4 1 5 0 1 6 5 1 5 3
B 1 4 5
następny 2 3 5 5 7 8 10
A 4 1 5 0 1 6 5 1 5 3
B 0 1 4 5
następny 2 4 5 5 7 8 10
A 4 1 5 0 1 6 5 1 5 3
B 0 1 1 4 5
następny 2 4 5 5 7 8 11
A 4 1 5 0 1 6 5 1 5 3
B 0 1 1 4 5 6
następny 2 4 5 5 7 9 11
A 4 1 5 0 1 6 5 1 5 3
B 0 1 1 4 5 5 6
następny 2 5 5 5 7 9 11
A 4 1 5 0 1 6 5 1 5 3
B 0 1 1 1 4 5 5 6
następny 2 5 5 5 7 10 11
A 4 1 5 0 1 6 5 1 5 3
B 0 1 1 1 4 5 5 5 6
następny 2 5 5 6 7 10 11
A 4 1 5 0 1 6 5 1 5 3
B 0 1 1 1 3 4 5 5 5 6

Możemy teraz zestawić wszystkie trzy procedury razem, aby utworzyć **sortowanie przez zliczanie** (ang. *counting sort*):

Procedura SORTOWANIE-PRZEZ-ZLICZANIE(*A*, *n*, *m*)

Dane wejściowe:

- *A*: tablica liczb całkowitych z przedziału od 0 do *m* – 1.
- *n*: liczba elementów w *A*.
- *m*: określa przedział wartości w *A*.

Wynik: tablica *B* zawierająca posortowane elementy *A*.

1. Wywołaj POLICZ-KLUCZE-RÓWNE(*A*, *n*, *m*) i przypisz wynik do *równe*.
2. Wywołaj POLICZ-KLUCZE-MNIEJSZE(*równe*, *m*) i przypisz wynik do *mniejsze*.
3. Wywołaj REORGANIZUJ(*A*, *mniejsze*, *n*, *m*) i przypisz wynik do *B*.
4. Zwróć tablicę *B*.

Na podstawie czasów działania procedur POLICZ-KLUCZE-RÓWNE ($\Theta(m + n)$), POLICZ-KLUCZE-MNIEJSZE ($\Theta(m)$) i REORGANIZUJ ($\Theta(m + n)$) możesz zauważyć, że SORTOWANIE-PRZEZ-ZLICZANIE działa w czasie $\Theta(m + n)$, czyli w $\Theta(n)$, jeśli m jest stałą. Sortowanie przez zliczanie pokonuje ograniczenie dolne $\Omega(n \lg n)$ dla sortowania przez porównania, ponieważ nigdy nie porównuje kluczy sortowania ze sobą. W zamian używa kluczy sortowania do indeksowania tablic, co jest możliwe dlatego, że klucze sortowania są małymi liczbami całkowitymi. Gdyby klucze sortowania były liczbami rzeczywistymi z częściami ułamkowymi lub gdyby były napisami, to sortowania przez zliczanie nie dałoby się użyć.

Zauważasz być może, że w procedurze założono, iż elementy zawierają tylko klucze sortowania i nie ma tam żadnych danych towarzyszących. Wszakże obiecałem, że w odróżnieniu od NAPRAWDĘ-PROSTEGO-SORTOWANIA procedura SORTOWANIE-PRZEZ-ZLICZANIE dopuszcza dane towarzyszące. I jest tak w istocie, o ile zmodyfikujesz krok 3C w procedurze REORGANIZUJ, aby kopiowano w nim cały element, a nie tylko klucz sortowania.

Zauważyłeś też pewnie, że podane przeze mnie procedury są trochę niewydajne pod względem użytkowania tablic. Można by połączyć tablice *równe*, *mniejsze* i *następny* w jedną, zostawiam to jednak Twojej inwencji.

Wracam do uwagi, że czas działania wynosi $\Theta(n)$, jeśli m jest stałą. Kiedy m może być stałą? Przykładem może być sortowanie ocen egzaminacyjnych według punktacji. Punktacja mieści się między 0 a 100, lecz liczba studentów jest zmienna. Mógłbym zastosować sortowanie przez zliczanie do uporządkowania egzaminów n studentów w czasie $\Theta(n)$, ponieważ $m = 101$ (pamiętajmy, że sortowany przedział jest rozpatrywany od 0 do $m - 1$) jest stałą.

Jednak w praktyce sortowanie przez zliczanie okazuje się przydatne jako część jeszcze innego algorytmu sortowania — chodzi o sortowanie pozycyjne. Oprócz liniowego czasu działania, gdy m jest stałą, sortowanie przez zliczanie ma jeszcze inną ważną cechę: jest **stabilne**. W stabilnym sortowaniu elementy z takim samym kluczem sortowania występują w tablicy wyjściowej w porządku, w którym występowały w tablicy wejściowej. Innymi słowy, sortowanie stabilne rozstrzyga konflikt między dwoma elementami o równych kluczach sortowania, umieszczając najpierw w tablicy wynikowej ten element, który występuje w tablicy z danymi wejściowymi jako pierwszy. Możesz się przekonać, dlaczego sortowanie przez zliczanie jest stabilne, spoglądając na pętlę w kroku 3 procedury REORGANIZUJ. Jeśli dwa elementy z A mają ten sam klucz sortowania, powiedzmy — *klucz*, to procedura zwiększa *następny*[*klucz*] natychmiast po przeniesieniu do B elementu, który wcześniej występował w A; w ten sposób, kiedy dojdzie w niej do przenoszenia elementu, który występuje w A na dalszej pozycji, pojawi się on w tablicy B na dalszym miejscu.

Sortowanie pozycyjne

Załóżmy, że masz posortować napisy pewnej stałej długości. Na przykład, piszę ten podrozdział w samolocie, a kiedy dokonywałem rezerwacji, otrzymałem kod potwierdzający X17FS6. Linie lotnicze zaprojektowały wszystkie kody potwierdzające w postaci sześcioznakowych ciągów, w których każdy znak jest albo literą, albo cyfrą. Każdy znak może przyjmować jedną z 36 wartości (26 liter i 10 cyfr), istnieje więc 36^6 = 2 176 782 336 możliwych kodów potwierdzających. Chociaż jest to stała, jest ona na tyle duża, że linie lotnicze raczej nie będą polegały na sortowaniu przez zliczanie, żeby posortować kody potwierdzające.

Żeby pozostać przy konkretach, powiedzmy, że potrafimy przetłumaczyć każdy z 36 znaków na kod numeryczny od 0 do 35. Kodem cyfry jest reprezentowana przez nią liczba (czyli kodem cyfry 5 jest 5), a kody liter zaczynają się od 10 i kończą na 35, dla Z.

Uprośćmy teraz co nieco sprawę i załóżmy, że każdy kod potwierdzający składa się tylko z dwóch znaków. (Bez obawy! Wkrótce wrócimy do sześciu znaków). Aczkolwiek dla $m = 36^2 = 1296$ moglibyśmy wykonać sortowanie przez zliczanie, postąpimy inaczej, wykonując je dwukrotnie z $m = 36$. Za pierwszym razem wykonujemy je, używając jako klucza sortowania *prawego skrajnego* znaku. Następnie bierzemy wynik wykonania pierwszego sortowania przez zliczanie i wykonujemy je drugi raz, lecz obecnie używamy jak klucza sortowania *lewego skrajnego* znaku. Wybieramy sortowanie przez zliczanie, ponieważ działa ono dobrze, gdy m jest względnie małe, a także dlatego, że jest stabilne.

Załóżmy na przykład, że mamy dwuznakowe kody potwierdzające {F6, E5, R6, X6, X2, T5, F2, T3}. Po wykonaniu sortowania przez zliczanie na prawych skrajnych znakach otrzymujemy uporządkowanie {X2, F2, T3, E5, T5, F6, R6, X6}. Zauważmy, że skoro sortowanie przez zliczanie jest stabilne i w pierwotnym porządku X2 występuje przed F2 , to po przesortowaniu z użyciem tylko skrajnego

prawego znaku X2 nadal występuje przed F2. Teraz sortujemy wynik według lewych skrajnych znaków, znów za pomocą sortowania przez zliczanie, i otrzymujemy pożądany rezultat {E5, F2, F6, R6, T3, T5, X2, X6}.

Co by się stało, gdybyśmy wykonali najpierw sortowanie według znaków z lewej strony? Po wykonaniu sortowania przez zliczanie według lewych skrajnych znaków otrzymalibyśmy {E5, F6, F2, R6, T5, T3, X6, X2}, a po następnym przesortowaniu przez zliczanie otrzymanego wyniku według prawych skrajnych znaków otrzymalibyśmy {F2, X2, T3, E5, T5, F6, R6, X6}, co jest niepoprawne.

Dlaczego działanie od prawej do lewej daje poprawny wynik? Ważne jest użycie stabilnej metody sortowania; może to być sortowanie przez zliczanie lub jakakolwiek inna stabilna metoda sortowania. Przypuśćmy, że działamy na *i*-tej pozycji znakowej, i przyjmijmy, że jeśli popatrzymy na $i - 1$ pozycji znakowych z prawej strony, zobaczymy tablicę posortowaną. Rozważmy dowolne dwa klucze. Jeśli różnią się one na *i*-tej pozycji znakowej, to nie ma znaczenia, co znajduje się na $i - 1$ pozycjach z prawej strony: stabilny algorytm sortowania, sortujący według *i*-tej pozycji, ustawi je we właściwym porządku. Jeśli natomiast mają one ten sam znak na pozycji *i*, to pierwszy, który występuje na $i - 1$ pozycjach, powinien pojawić się najpierw, a stosując metodę stabilną, zapewniamy, że właśnie to nastąpi.

Powróćmy więc do 6-znakowych kodów potwierdzających i zobaczmy, jak posortować kody potwierdzające, które mają na początku kolejność {XI7FS6, PL4ZQ2, JI8FR9, XL8FQ6, PY2ZR5, KV7WS9, JL2ZV3, KI4WR2}. Ponumerujmy znaki od prawej do lewej liczbami od 1 do 6. Wyniki wykonania stabilnego sortowania według *i*-tego znaku z działaniem od prawej do lewej wyglądają wówczas następująco:

i	Porządek wynikowy
1	{PL4ZQ2, KI4WR2, JL2ZV3, PY2ZR5, XI7FS6, XL8FQ6, JI8FR9, KV7WS9}
2	{PL4ZQ2, XL8FQ6, KI4WR2, PY2ZR5, JI8FR9, XI7FS6, KV7WS9, JL2ZV3}
3	{XL8FQ6, JI8FR9, XI7FS6, KI4WR2, KV7WS9, PL4ZQ2, PY2ZR5, JL2ZV3}
4	{PY2ZR5, JL2ZV3, KI4WR2, PL4ZQ2, XI7FS6, KV7WS9, XL8FQ6, JI8FR9}
5	{KI4WR2, XI7FS6, JI8FR9, JL2ZV3, PL4ZQ2, XL8FQ6, KV7WS9, PY2ZR5}
6	{JI8FR9, JL2ZV3, KI4WR2, KV7WS9, PL4ZQ2, PY2ZR5, XI7FS6, XL8FQ6}

Uogólniając, w algorytmie **sortowania pozycyjnego** zakładamy, że możemy uważać każdy klucz sortowania za *d*-cyfrową liczbę, której każda cyfra jest z przedziału od 0 do $m - 1$. Idąc od prawej do lewej, wykonujemy stabilne sortowanie według każdej cyfry. Jeśli jako sortowania stabilnego używamy sortowania przez zliczanie, to czas sortowania według jednej cyfry wynosi $\Theta(m + n)$, a czas posortowania według wszystkich *d* cyfr wyniesie $\Theta(d(m + n))$. Jeśli *m* jest stałą (jak 36 w przykładzie z kodami potwierdzającymi), to czas sortowania pozycyjnego wynosi $\Theta(d\,n)$. Jeśli *d* jest także stałą (wynoszącą 6 w wypadku kodów potwierdzających), to czas sortowania pozycyjnego wynosi tylko $\Theta(n)$.

Jeśli w sortowaniu pozycyjnym zastosuje się sortowanie przez zliczanie do sortowania według każdej cyfry, to nigdy nie dochodzi do porównywania dwóch kluczy sortowania ze sobą. Poszczególne cyfry są używane do indeksowania tablic w sortowaniu przez zliczanie. To dlatego sortowanie pozycyjne, podobnie jak sortowanie przez zliczanie, pokonuje ograniczenie dolne $\Omega(n \lg n)$ sortowania przez porównania.

Co czytać dalej

Rozdział 8 podręcznika CLRS [CLRS09] zawiera rozszerzenie całego materiału pomieszczonego w tym rozdziale.

5 Skierowane grafy acykliczne

Przypomnijmy przypis z początku książki, w którym wyjawiłem, że grywałem w hokeja. Przez parę lat byłem bramkarzem, w końcu jednak moja gra pogorszyła się do tego stopnia, że nie mogłem już na to patrzeć. Można było odnieść wrażenie, że każdy strzał lądował w siatce. Zrobiłem więc sobie ponad siedmioletnią przerwę, po której wróciłem między słupki (tzn. znów stanąłem w bramce) w kilku spotkaniach.

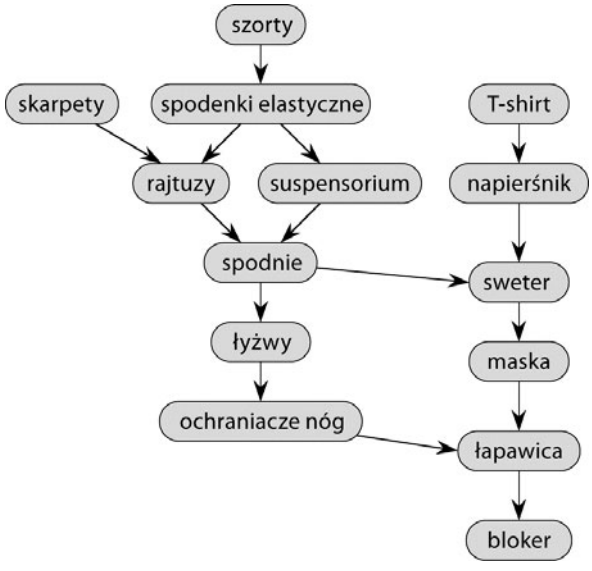
Moją największą troską było nie to, czy się sprawdzę — wiedziałem, że gram okropnie — lecz to, czy będę pamiętał, jak się nakłada cały bramkarski ekwipunek. W hokeju na lodzie bramkarze przywdziewają furę sprzętu (od 35 do 40 funtów[1]), a ubierając się do gry, musiałem to wszystko założyć we właściwym porządku. Na przykład, ponieważ jestem praworęczny, na lewej ręce noszę nadwymiarową rękawicę do łapania krążka; nazywa się ją „łapawicą" (ang. *catch glove*). Gdybym najpierw ubrał łapawicę, lewa ręka straciłaby zupełnie zręczność i nie zdołałbym już potem założyć żadnej z górnych części ubioru.

Przygotowując się do przywdziania bramkarskiego rynsztunku, sporządziłem sobie diagram ukazujący, w jakiej kolejności powinny być nakładane poszczególne elementy. Diagram ten jest przedstawiony na następnej stronie. Strzałka biegnąca od A do B wskazuje konieczność nałożenia A przed B. Na przykład ochraniacz piersiowy muszę nałożyć przed swetrem. Rzecz jasna konieczność [relacja] „musi być nałożone przed" jest **przechodnia**: jeśli rzecz A musi być nałożona przed rzeczą B, a rzecz B musi być nałożona przed rzeczą C, to rzecz A musi być nałożona przed rzeczą C. Dlatego muszę nałożyć napierśnik przed swetrem, maską, łapawicą i blokerem.

Dla kilku par w tym ekwipunku nie jest istotna kolejność, w której będę je nakładał. Skarpety mogę na przykład włożyć przed napierśnikiem albo po nim.

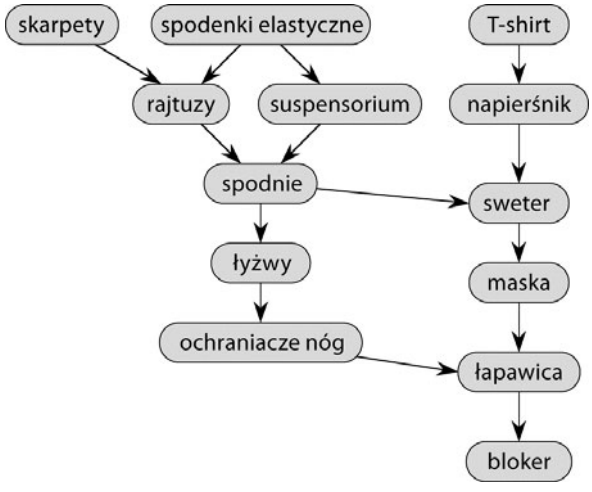
Musiałem określić kolejność ubierania się. Kiedy już przygotowałem diagram, należało sporządzić listę wszystkich rzeczy, które miałem do ubrania w jednym porządku, nienaruszającym żadnej z konieczności „musi być nałożone przed". Zauważyłem, że istnieje kilka takich uporządkowań; poniżej diagramu są podane trzy z nich.

[1] Funt = 453,6 g, zatem w przeliczeniu masa tej „zbroi" wynosi od 16 do 18 kg — *przyp. tłum.*

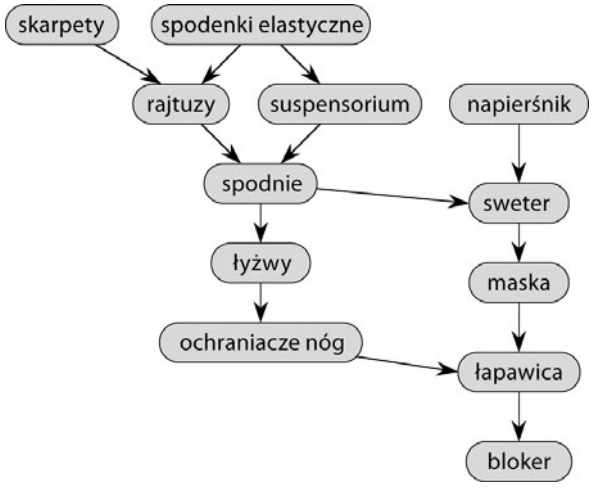


Porządek 1	Porządek 2	Porządek 3
szorty	szorty	skarpety
spodenki elastyczne	T-shirt	T-shirt
suspensorium	spodenki elastyczne	szorty
skarpety	suspensorium	napierśnik
rajtuzy	napierśnik	spodenki elastyczne
spodnie	skarpety	rajtuzy
łyżwy	rajtuzy	suspensorium
ochraniacze nóg	spodnie	spodnie
T-shirt	sweter	łyżwy
napierśnik	maska	ochraniacze nóg
sweter	łyżwy	sweter
maska	ochraniacze nóg	maska
łapawica	łapawica	łapawica
bloker	bloker	bloker

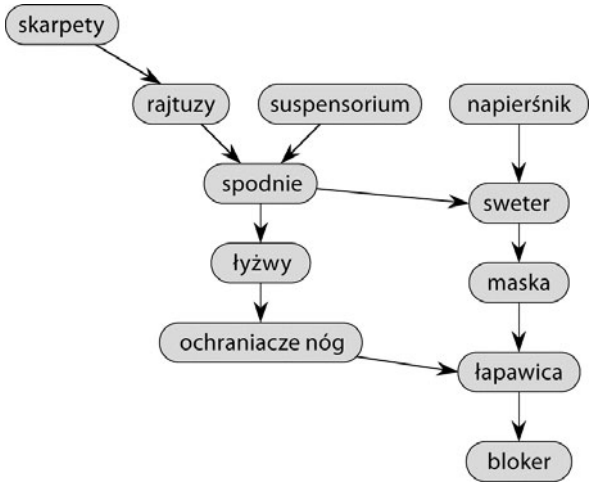
Jak ustaliłem te porządki? Oto jak doszedłem do porządku nr 2. Znalazłem rzecz, do której nie prowadzą żadne strzałki, ponieważ taka rzecz nie może być włożona po żadnej innej. Wybrałem szorty jako pierwszą rzecz w kolejności, a potem, mając już (teoretycznie) ubrane szorty, usunąłem je z diagramu, w wyniku czego powstał diagram przedstawiony na następnej stronie u góry.



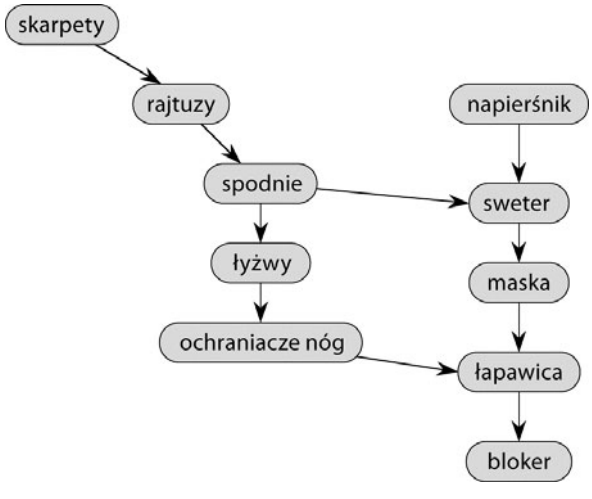
Następnie znowu wybrałem rzecz bez prowadzących do niej strzałek, tym razem T-shirt. Dodałem ją na końcu tworzonego porządku i usunąłem z diagramu, otrzymując taki diagram:



I znowu wybieram rzecz, do której nie prowadzą już strzałki — spodenki elastyczne, i jak poprzednio dokładam ją na koniec budowanego porządku i usuwam z diagramu, otrzymując diagram pokazany na następnej stronie u góry.



Dalej wybieram suspensorium:



Postępuję w ten sposób (tzn. wybieram rzecz, do której nie prowadzą strzałki, wstawiam ją na koniec porządku oraz usuwam z diagramu) tak długo, aż nie zostanie ani jedna rzecz. Trzy uporządkowania pokazane w tabeli powyżej powstają w wyniku różnych wyborów rzeczy, do których nie prowadzą żadne strzałki, zaczynając od diagramu z początku rozdziału.

Skierowane grafy acykliczne

Te diagramy są szczególnymi przykładami **grafów skierowanych** (digrafów, ang. *directed graphs*) zbudowanych z **wierzchołków** (ang. *vertices*)[2], odpowiadających elementom ekwipunku bramkarza, i **krawędzi skierowanych** (ang. *directed edges*)[3] przedstawionych za pomocą strzałek. Każda krawędź skierowana jest parą uporządkowaną postaci (u, v), gdzie u i v są wierzchołkami. Na przykład lewa skrajna krawędź w grafie skierowanym na początku rozdziału jest parą (skarpety, rajtuzy). Jeśli graf skierowany zawiera skierowaną krawędź (u, v), to mówimy, że v **sąsiaduje** z u i że (u, v) **wychodzi** z u i **wchodzi** do v, tak więc wierzchołek z etykietą rajtuzy sąsiaduje z wierzchołkiem o etykiecie skarpety, a krawędź (skarpety, rajtuzy) wychodzi z wierzchołka z etykietą skarpety i wchodzi do wierzchołka z etykietą rajtuzy.

Grafy skierowane, które oglądaliśmy, mają jeszcze inną właściwość: nie można w nich powrócić do tego samego wierzchołka, przechodząc jedną lub więcej krawędzi. Taki graf skierowany nazywamy **skierowanym grafem acyklicznym** (ang. *directed acyclic graph*) albo **dagiem**. Jest on acykliczny, gdyż nie ma sposobu, aby wykonać „cykl", wychodząc z jakiegoś wierzchołka i wracając do niego z powrotem. (Bardziej formalną definicję cyklu poznamy później w tym rozdziale).

Dagi nadają się świetnie do modelowania zależności polegających na tym, że jedno zadanie musi poprzedzić inne. Inne zastosowania znajdują dagi w planowaniu przedsięwzięć w rodzaju budowy domu: na przykład szkielet musi być ukończony przed rozpoczęciem robót dekarskich. Albo w gotowaniu — pewne kroki muszą wystąpić w określonej kolejności, choć kolejność innych jest bez znaczenia; w dalszej części rozdziału obejrzymy przykład dagu dotyczącego gotowania.

Sortowanie topologiczne

Kiedy potrzebowałem ustalić jeden liniowy porządek ubierania ekwipunku bramkarskiego, musiałem wykonać „sortowanie topologiczne". Aby wyrazić rzecz ściślej, **sortowanie topologiczne** dagu wytwarza porządek liniowy, taki że jeśli (u, v) jest krawędzią w dagu, to w porządku liniowym u występuje przed v. Sortowanie topologiczne różni się od sortowania w rozumieniu przyjętym przez nas w rozdziałach 3 i 4.

Porządek liniowy wytworzony przez sortowanie topologiczne nie musi być jednoznaczny. To jednak już wiesz, skoro wszystkie trzy kolejności wdziewania ubioru bramkarza przedstawione na początku rozdziału mogłyby być wynikiem sortowania topologicznego.

Inne zastosowanie sortowania topologicznego pojawiło się w zadaniu programistycznym, które wykonywałem dawno temu. Tworzyliśmy wspomagane komputerowo systemy projektowania i nasz system miał utrzymywać bibliotekę części. Części mogły zawierać inne części, nie były jednak dozwolone żadne zależności cykliczne:

---

[2] Liczba pojedyncza ang.: *vertex*; wierzchołki grafów są też nazywane **węzłami** (ang. *nodes*) — *przyp. tłum.*

[3] Inna nazwa polska: **łuki** (ang. *arcs*) — *przyp. tłum.*

nie mogło się okazać, że część zawiera samą siebie. Projekty części musieliśmy zapisywać na taśmie (*mówiłem*, że to była robota sprzed wielu lat) w ten sposób, że każda część poprzedzała wszystkie inne części, które ją zawierały. Jeśli każda część jest wierzchołkiem, a krawędź (*u*, *v*) wskazuje, że *v* zawiera część *u*, to pozostawało nam zapisać części w topologicznie posortowanym, liniowym porządku.

Który wierzchołek byłby dobrym kandydatem, żeby zająć pierwsze miejsce w porządku liniowym? Każdy, do którego nie wchodzą żadne krawędzie. Liczba krawędzi wchodzących do wierzchołka jest **stopniem wejściowym** (ang. *in-degree*) wierzchołka, moglibyśmy więc zacząć od dowolnego wierzchołka mającego stopień wejściowy 0. Na szczęście każdy dag musi mieć przynajmniej jeden wierzchołek o stopniu wejściowym 0 i przynajmniej jeden wierzchołek o **stopniu wyjściowym** (ang. *out-degree*) 0, w przeciwnym razie musiałby to być cykl.

Załóżmy więc, że wybieramy wierzchołek o stopniu wejściowym 0 — nazwijmy go wierzchołkiem *u* — i ustawmy go na początku porządku liniowego. Ponieważ postaraliśmy się, aby wierzchołek *u* wypadł na pierwszym miejscu, wszystkie inne wierzchołki będą w porządku liniowym występowały po *u*. W szczególności dowolny wierzchołek *v* sąsiadujący z *u* musi wystąpić w porządku liniowym gdzieś po *u*. Dlatego możemy bezpiecznie usunąć z dagu *u* i wszystkie krawędzie wychodzące z *u*, wiedząc, że zadbaliśmy o zależności definiowane przez te krawędzie. Kiedy usuniemy z dagu wierzchołek i wychodzące z niego krawędzie, co nam zostanie? Nowy dag! Przecież przez usunięcie wierzchołka i krawędzi nie moglibyśmy utworzyć cyklu. Możemy więc powtórzyć to postępowanie z pozostałym dagiem, znajdując jakiś wierzchołek o stopniu wejściowym 0, umieszczając go w porządku liniowym za wierzchołkiem *u*, usuwając krawędzie itd.

Ten pomysł znajduje zastosowanie w procedurze sortowania topologicznego przedstawionej na następnej stronie, jednak zamiast rzeczywistego usuwania wierzchołków i krawędzi z dagu pamięta ona tylko stopień wejściowy każdego wierzchołka, zmniejszając go wraz z każdą usuwaną teoretycznie krawędzią. Ponieważ indeksami tablicy są liczby całkowite, załóżmy, że każdy wierzchołek identyfikujemy unikatową liczbą całkowitą z przedziału od 0 do *n*. Ponieważ procedura wymaga szybkiego identyfikowania wierzchołka o stopniu wejściowym 0, utrzymuje ona stopnie wejściowe wszystkich wierzchołków w tablicy *stopień-we*, indeksowanej wierzchołkami, a także utrzymuje wykaz *następny* wszystkich wierzchołków o stopniu wejściowym 0. W krokach 1 – 3 tablica *stopień-we* jest inicjowana, w kroku 4 następuje zainicjowanie tablicy *następny*, a w kroku 5 — uaktualnienie tablic *stopień-we* i *następny* po teoretycznym[4] usunięciu wierzchołków i krawędzi. Procedura może wybrać dowolny wierzchołek w wykazie *następny* jako kolejny do umieszczenia w porządku liniowym.

[4] W oryg. *conceptual* (pojęciowy); trzymamy się za oryginałem tego podkreślenia, warto jednak zwrócić uwagę, że w komputerach zawsze stykamy się z abstrakcyjnymi odwzorowaniami; odpowiednie zaznaczenie stanu jakiegoś elementu w tablicy może być równoznaczne z usunięciem go poza obszar zainteresowań, choć element ten może nadal zajmować fizyczną pamięć, bo nie warto go w danej chwili fizycznie usuwać — *przyp. tłum.*

Procedura SORTOWANIE-TOPOLOGICZNE(*G*)

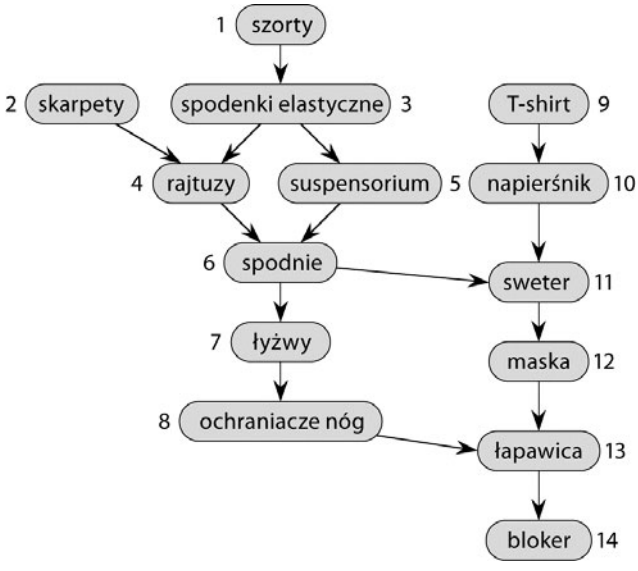
Dane wejściowe: G — skierowany graf acykliczny o wierzchołkach ponumerowanych od 1 do *n*.

Wynik: porządek liniowy wierzchołków taki, że *u* występuje przed *v* w porządku liniowym, jeśli (*u*, *v*) jest krawędzią w grafie.

1. Niech *stopień-we*[1..*n*] będzie nową tablicą i utwórzmy pusty porządek liniowy wierzchołków.
2. Nadaj wszystkim elementom w *stopień-we* wartość 0.
3. Dla każdego wierzchołka *u*:
   A. Dla każdego wierzchołka *v* sąsiadującego z *u*:
      i. Zwiększ o 1 *stopień-we*[*v*].
4. Utwórz wykaz *następny* składający się ze wszystkich takich wierzchołków *u*, że *stopień-we*[*u*] = 0.
5. Dopóki *następny* jest niepusty, wykonuj, co następuje:
   B. Usuń wierzchołek z *następny* i nazwij go wierzchołkiem *u*.
   C. Dodaj *u* na koniec porządku liniowego.
   D. Dla każdego wierzchołka *v* sąsiadującego z *u*:
      i. Zmniejsz *stopień-we*[*v*] o 1.
      ii. Jeśli *stopień-we*[*v*] = 0, to wstaw *v* do wykazu *następny*.
6. Zwróć porządek liniowy.

Zobaczmy, jak działa kilka pierwszych iteracji kroku 5 na dagu do nakładania ekwipunku bramkarza. Aby wykonać procedurę SORTOWANIE-TOPOLOGICZNE na tym dagu, musimy ponumerować wierzchołki, jak pokazano poniżej. Stopień wejściowy 0 mają tylko wierzchołki 1, 2 i 9, wobec tego po wejściu do pętli w kroku 5 wykaz *następny* zawiera tylko te trzy wierzchołki. Aby uzyskać uporządkowanie nr 1 z tabeli na początku rozdziału, kolejność wierzchołków w *następny* musiałaby być taka: 1, 2, 9. Następnie, w pierwszej iteracji pętli z kroku 5, wybieramy wierzchołek 1 (szorty) jako wierzchołek *u*, usuwamy go z *następny*, dodajemy ten wierzchołek na koniec początkowo pustego porządku liniowego, po czym zmniejszamy *stopień-we*[3] (spodenki elastyczne). Ponieważ ta operacja powoduje spadek wartości *stopień-we*[3] do 0, wstawiamy wierzchołek 3 do wykazu *następny*. Załóżmy, że gdy wstawiamy wierzchołek do *następny*, trafia on tam jako pierwszy element wykazu. Taki wykaz, do którego dodajemy i z którego ujmujemy zawsze tylko z tego samego końca, zwie się **stosem** (ang. *stack*), ponieważ jest niczym stos talerzy, z którego zawsze zabierasz talerz z wierzchu i odkładasz nowy talerz na wierzch. Porządek taki nazywamy **ostatni przychodzi, pierwszy wychodzi** (ang. *last in, first out*, LIFO). Przy tym założeniu *następny* przyjmuje elementy 3, 2, 9,

a w następnej iteracji pętli wybieramy wierzchołek 3 jako wierzchołek *u*. Usuwamy go z *następny*, dodajemy na końcu porządku liniowego, który teraz przybiera postać „szorty, spodenki elastyczne" i zmniejszamy *stopień-we*[4] (z 2 na 1) oraz *stopień-we*[5] (z 1 na 0). Wstawiamy wierzchołek 5 (suspensorium) do *następny*, wskutek czego *następny* składa się z 5, 2 i 9. W następnej iteracji wybieramy wierzchołek 5 jako wierzchołek *u*, usuwamy go z *następny* i dokładamy na koniec porządku liniowego (teraz stanowionego przez „szorty, spodenki elastyczne, suspensorium") i zmniejszamy *stopień-we*[6] (z 2 na 1). Tym razem żaden wierzchołek nie zostaje dodany do *następny*, toteż w kolejnej iteracji wybieramy wierzchołek 2 jako wierzchołek *u* itd.



Aby przeanalizować procedurę SORTOWANIE-TOPOLOGICZNE, musimy najpierw zrozumieć, w jaki sposób można reprezentować graf skierowany i wykaz taki jak *następny*. Reprezentując graf, nie będziemy wymagać jego acykliczności, ponieważ nieobecność lub obecność cykli nie ma wpływu na sposób reprezentowania grafu.

Jak reprezentować graf skierowany

Graf skierowany możemy reprezentować w komputerze na kilka sposobów. Przyjmiemy, że graf ma *n* wierzchołków i *m* krawędzi. Utrzymujemy w mocy założenie, że każdy wierzchołek ma własny numer od 1 do *n*, dzięki czemu możemy używać wierzchołka jako indeksu tablicy lub nawet jako numeru wiersza lub kolumny macierzy.

Na razie chcemy tylko określić, które wierzchołki i krawędzie są obecne (później związemy również z każdą krawędzią wartość liczbową). Moglibyśmy użyć **macierzy sąsiedztwa** (ang. *adjacency matrix*) o wymiarach $n \times n$, w której każdy wiersz i każda kolumna odpowiada jednemu wierzchołkowi, a element na skrzyżowaniu wiersza

odpowiadającego wierzchołkowi u i kolumny odpowiadającej wierzchołkowi v jest równy 1, jeśli krawędź (u, v) występuje w grafie, lub 0, jeśli graf nie zawiera krawędzi (u, v). Ponieważ macierz sąsiedztwa ma n^2 elementów, musi być prawdą, że $m \leq n^2$. Moglibyśmy też utrzymywać wykaz wszystkich m krawędzi grafu bez zachowywania jakiegoś specjalnego porządku. Czymś pośrednim między macierzą sąsiedztwa a wykazem nieuporządkowanym jest **reprezentacja list sąsiedztwa** (ang. *adjacency-list represention*) z n-elementową tablicą indeksowaną wierzchołkami, w której wpis dotyczący każdego wierzchołka u jest listą wszystkich wierzchołków sąsiadujących z u. Listy mają łącznie m wierzchołków, gdyż istnieje po jednej pozycji listowej dla każdej z m krawędzi. Oto macierz sąsiedztwa i listy sąsiedztwa dla grafu skierowanego z poprzedniej strony:

Macierz sąsiedztwa

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	1	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Listy sąsiedztwa

1	3
2	4
3	4, 5
4	6
5	6
6	7, 11
7	8
8	13
9	10
10	11
11	12
12	13
13	14
14	(nic)

Nieuporządkowany wykaz krawędzi i reprezentacja list sąsiedztwa nasuwa pytanie, w jaki sposób reprezentować listę. Najlepszy sposób przedstawiania listy zależy od typu operacji, które chcemy na liście wykonywać. W wypadku nieuporządkowanych wykazów i list sąsiedztwa wiemy z góry, ile krawędzi będzie na każdej liście, a zawartość listy się nie zmienia, dlatego każdą listę możemy przechowywać w tablicy. Tablicy możemy też użyć do przechowywania listy nawet wówczas, gdy treść listy zmienia się z biegiem czasu, o ile tylko znamy maksymalną liczbę elementów, które mogą się pojawić na liście. Jeśli nie musimy wstawiać elementu pośrodku listy lub usuwać go ze środka listy, to reprezentowanie listy w postaci tablicy jest równie efektywne jak każdy inny sposób.

Jeśli jednak musimy dokonywać wstawek w środku listy, to możemy się posłużyć **listą powiązaną** (ang. *linked list*)[5], w której każdy element listy zawiera dane o umiejscowieniu swojego następnika na liście, co ułatwia wplatanie nowego elementu po danym elemencie. Jeśli potrzebujemy również usuwać ze środka listy, to każdy element na liście powiązanej powinien też zawierać lokalizację swojego poprzednika,

[5] Inna nazwa polska: **lista z dowiązaniami** — *przyp. tłum.*

abyśmy mogli szybko wykluczyć z niej element. Poczynając od tej chwili, będziemy zakładać, że potrafimy wstawiać na listę powiązaną lub usuwać z niej w stałym czasie. Lista powiązana, która ma odniesienia tylko do następników, jest **listą jednokierunkową** (ang. *singly linked list*). Dodanie odniesień do poprzedników czyni z niej **listę dwukierunkową** (ang. *doubly linked list*).

Czas działania sortowania topologicznego

Jeśli przyjmiemy, że do przedstawienia dagu użyto reprezentacji list sąsiedztwa i wykaz *następny* jest listą powiązaną, to możemy udowodnić, że procedura SORTOWANIE-TOPOLOGICZNE zużywa $\Theta(n + m)$ czasu. Ponieważ wykaz *następny* ma postać listy powiązanej, operacje wstawiania i usuwania możemy na nim wykonywać w stałym czasie. Krok 1 zajmuje stały czas, a ponieważ tablica *stopień-we* ma n elementów, krok 2 nadaje jej początkowe wartości 0 w czasie $\Theta(n)$. Krok 3 zużywa $\Theta(n + m)$ czasu. Składnik $\Theta(n)$ w kroku 3 pojawia się dlatego, że zewnętrzna pętla sprawdza każdy z n wierzchołków, a składnik $\Theta(m)$ dlatego, że wewnętrzna pętla w kroku 3A odwiedza każdą z m krawędzi dokładnie raz w trakcie wszystkich iteracji pętli zewnętrznej. Krok 4 zabiera $\Theta(n)$ czasu, gdyż wykaz *następny* zaczyna się od najwyżej n wierzchołków. Większość pracy skupia się w kroku 5. Ponieważ każdy wierzchołek jest wstawiany do *następny* dokładnie jeden raz, pętla główna jest iterowana n razy. Kroki 5A i 5B zajmują stałą ilość czasu w każdej iteracji. Podobnie jak w kroku 3A, treść pętli w kroku 5C jest wykonywana łącznie m razy, raz na krawędź. Kroki 5Ci i 5Cii zajmują stałą ilość czasu w iteracji, toteż wszystkie iteracje z kroku 5C zużywają $\Theta(m)$ czasu i dlatego pętla w kroku 5 zużywa $\Theta(n + m)$ czasu. Oczywiście krok 6 zajmuje stałą ilość czasu, kiedy zatem zsumujemy czas wszystkich kroków, otrzymamy $\Theta(n + m)$.

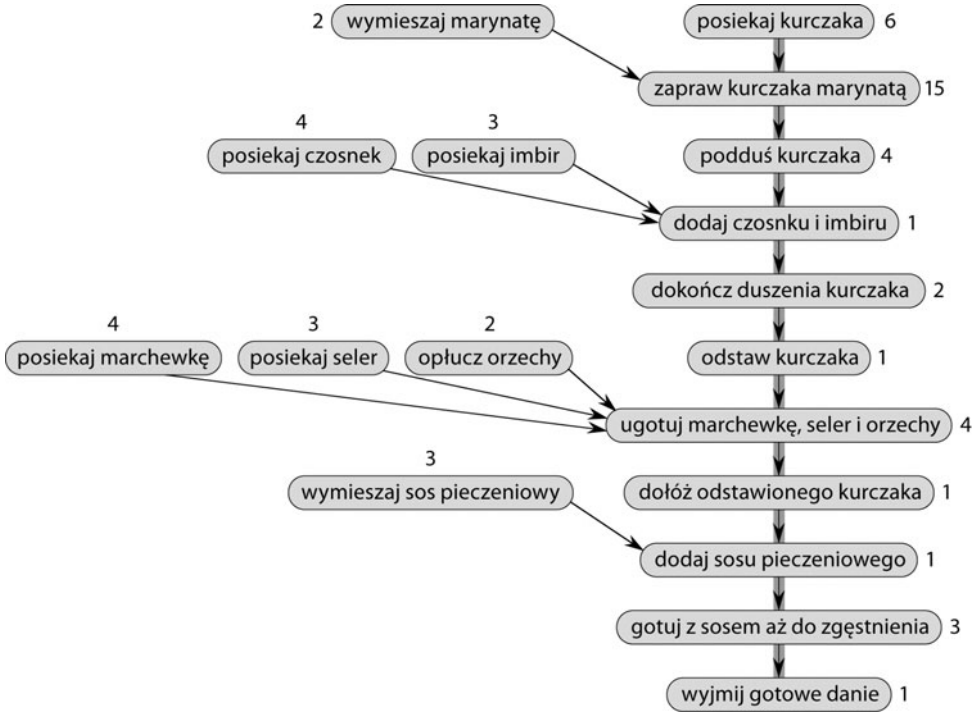
Ścieżka krytyczna w diagramie PERT

Po dniu pracy lubię zrelaksować się przy kuchni i zawsze sprawia mi frajdę przygotowywanie i spożywanie kurczaka à la kung pao[6]. Muszę sprawić kurczaka, posiekać warzywa, wymieszać marynatę i sos pieczeniowy, po czym ugotować całe danie. Podobnie jak podczas ubierania bramkarskiego uniformu, pewne kroki muszą tu być wykonane przed innymi, dlatego mogę posłużyć się dagiem, aby zamodelować procedurę przyrządzania tej potrawki z kurczaka po chińsku. Stosowny dag przedstawiono na następnej stronie.

Obok każdego wierzchołka dagu występuje liczba, która wskazuje, ile minut muszę wykonywać odpowiadającą mu czynność. Na przykład cztery minuty zajmuje mi siekanie czosnku (ponieważ obieram najpierw każdy ząbek i używam *mnóstwo* czosnku). Jeśli zsumujesz czasy wszystkich czynności, zauważysz, że gdybym wykonywał je po kolei, przyrządzenie kurczaka kung pao zabrałoby mi godzinę.

[6] Pokawałkowany kurczak z jarzynami, orzechami, w sosie chili — *przyp. tłum.*

Gdybym jednak miał kogoś do pomocy, moglibyśmy wykonać kilka czynności jednocześnie. Na przykład jedna osoba mogłaby mieszać marynatę, a druga siekać kurczaka. Gdyby się zebrało dostatecznie dużo osób do pomocy i znalazło odpowiednio dużo miejsca, noży, desek do krajania i misek, wiele z tych czynności moglibyśmy wykonać jednocześnie. Jeśli przyjrzysz się dowolnym dwóm czynnościom na diagramie i zauważysz, że nie ma między nimi żadnych łączących je strzałek, oznacza to, że każdą taką czynność mógłbym przydzielić innej osobie, aby była wykonywana jednocześnie z innymi.



Ile trwałoby przyrządzenie kurczaka po chińsku, gdybyśmy dysponowali nieograniczonymi zasobami (ludzi, miejsca, sprzętu kuchennego)? Ten dag jest przykładem **diagramu PERT** — skrót pochodzi od słów *program evaluation and review technique* (z ang. technika oceny i przeglądu programów). Czas potrzebny do wykonania całego zadania, nawet z jednoczesnym wykonaniem tak wielu czynności, jak to możliwe, jest zadany przez „ścieżkę krytyczną" w diagramie PERT. Nim pojmiemy, czym jest ścieżka krytyczna, musimy zrozumieć, co to znaczy „ścieżka". Dopiero wtedy będziemy mogli zdefiniować ścieżkę krytyczną.

Ścieżka (ang. *path*) jest ciągiem wierzchołków i krawędzi, który umożliwia przejście od jednego wierzchołka do innego (lub powrót do tego samego wierzchołka); mówimy, że ścieżka zawiera zarówno wierzchołki, jak i przebyte krawędzie. Na przykład jedna ze ścieżek w dagu przyrządzania potrawki z kurczaka ma wierzchołki z etykietami, kolejno: „posiekaj czosnek", „dodaj czosnku i imbiru", „dokończ du-

szenia kurczaka" i „odstaw kurczaka" wraz z łączącymi je krawędziami. Ścieżka powracająca z danego wierzchołka do niego samego jest **cyklem** (ang. *cycle*), lecz oczywiście dag nie ma cykli.

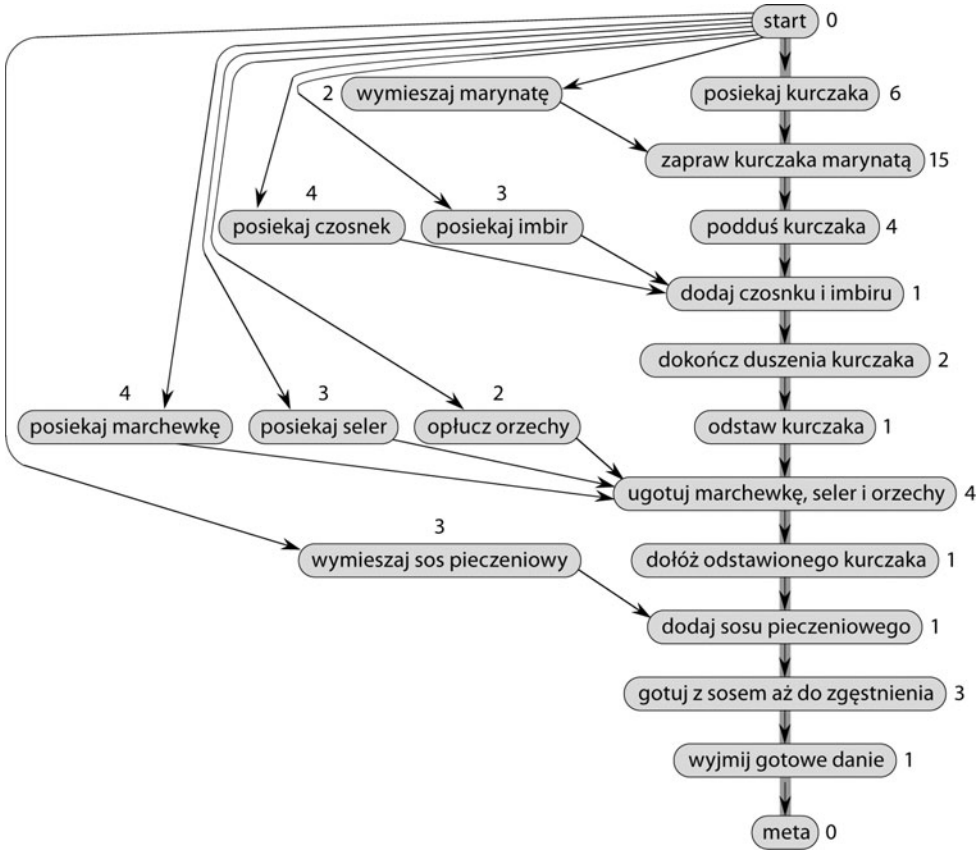
Ścieżką krytyczną (ang. *critical path*) w diagramie PERT jest ścieżka, dla której suma czasów czynności (zadań) jest maksymalna spośród wszystkich ścieżek. Suma czasów czynności wzdłuż ścieżki krytycznej daje najmniejszy możliwy czas wykonania całego zadania, niezależnie od tego, ile czynności jest wykonywanych równocześnie. W diagramie PERT gotowania potrawki z kurczaka ścieżkę krytyczną zacieniowałem. Jeśli zsumujesz czasy czynności wzdłuż ścieżki krytycznej, zobaczysz, że żebym miał nie wiem jaką pomoc, przyrządzenie kurczaka kung pao zabierze mi co najmniej 39 minut[7].

Zakładając, że wszystkie czasy są dodatnie, ścieżka krytyczna w diagramie PERT musi zaczynać się od któregoś wierzchołka ze stopniem wejściowym 0 i kończyć na wierzchołku ze stopniem wyjściowym 0. Zamiast sprawdzać ścieżki między wszystkimi parami wierzchołków, z których jeden ma stopień wejściowy 0, a drugi stopień wyjściowy 0, możemy po prostu dodać dwa „puste" wierzchołki: „start" i „meta", jak na rysunku na następnej stronie. Ponieważ są to wierzchołki puste, przypisujemy im zera jako czasy czynności. W diagramie PERT dodajemy krawędź od startu do każdego wierzchołka ze stopniem wejściowym 0 oraz dodajemy po krawędzi od każdego wierzchołka ze stopniem wyjściowym 0 do mety. W ten sposób jedynym wierzchołkiem ze stopniem wejściowym 0 staje się start, a jedynym wierzchołkiem ze stopniem wyjściowym 0 — meta. Ścieżka od startu do mety z maksymalną sumą czasów czynności w jej wierzchołkach (zacieniowana) stanowi ścieżkę krytyczną w diagramie PERT — oczywiście po odjęciu pustych wierzchołków start i meta.

Po dodaniu pustych wierzchołków ścieżki krytycznej poszukujemy jako najkrótszej ścieżki od starty do mety, opierając się na czasach czynności. W tym miejscu możesz pomyśleć, że się pomyliłem w poprzednim zdaniu, ponieważ ścieżka krytyczna powinna odpowiadać najdłuższej ścieżce, a nie najkrótszej. Rzeczywiście, tak właśnie jest, lecz skoro diagram PERT nie ma cykli, możemy pozmieniać czasy czynności, tak aby najkrótsza ścieżka dawała ścieżkę krytyczną. W szczególności negujemy czas każdej czynności i znajdujemy ścieżkę od startu do mety z *minimalną* sumą czasów czynności.

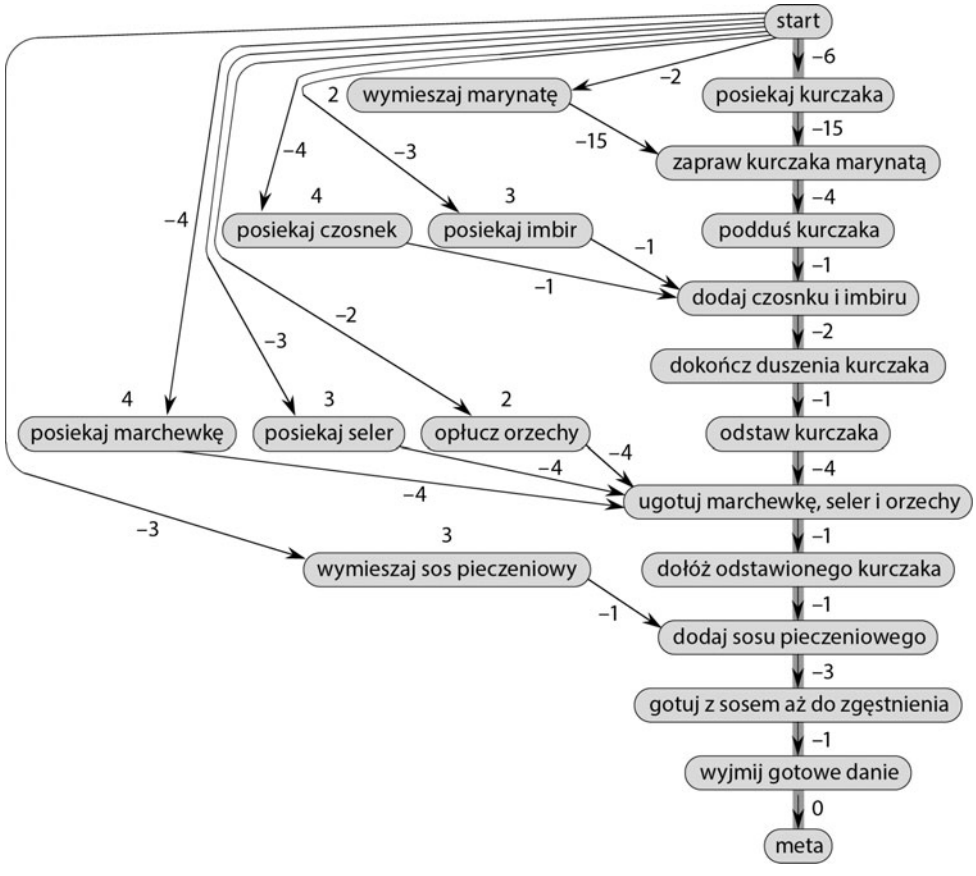
Po co negować czasy czynności i znajdować ścieżkę z minimalną sumą czasów czynności? Ponieważ rozwiązanie tego problemu jest specjalnym przypadkiem znajdowania najkrótszych ścieżek, a mamy mnóstwo algorytmów znajdowania najkrótszych ścieżek. Kiedy jednak mówimy o najkrótszych ścieżkach, wartości wyznaczające długość ścieżki są kojarzone z krawędziami, a nie z wierzchołkami. Wartość, którą przypisujemy każdej krawędzi, nazywamy **wagą** (ang. *weight*). Graf skierowany z wagami krawędziowymi określa się jako **ważony graf skierowany** (ang. *weighted*

[7] Jeśli Cię dziwi, jak to się dzieje, że chińskie restauracje serwują kurczaka kung pao w znacznie krótszym czasie, to weź pod uwagę, że mają oni wiele składników przygotowanych na zapas, a na ich komercyjnych kuchniach można gotować szybciej niż na mojej domowej kuchence.



directed graph). „Waga" jest pojęciem ogólnym na określenie wartości kojarzonych z krawędziami. Jeśli ważony graf skierowany reprezentuje sieć drogową, to każda krawędź reprezentuje jeden kierunek ruchu na drodze między dwoma skrzyżowaniami, a waga krawędzi mogłaby reprezentować długość drogi, czas wymagany na przebycie danej drogi lub myto, które trzeba zapłacić za skorzystanie z drogi. **Wagą ścieżki** (ang. *weight of path*) jest suma wag krawędzi na tej ścieżce, jeśli więc wagi krawędzi reprezentują odległości drogowe, to waga ścieżki mogłaby wskazywać sumaryczną długość dróg składających się na ścieżkę. **Najkrótsza ścieżka** (ang. *shortest of path*) od wierzchołka u do v jest ścieżką, której suma wag krawędzi stanowi minimum spośród wszystkich ścieżek prowadzących z u do v. Najkrótsze ścieżki niekoniecznie są unikatowe, jako że graf skierowany z u do v może zawierać wiele ścieżek, których wagi osiągają minimum.

Aby zamienić diagram PERT z zanegowanymi czasami czynności w ważony graf skierowany, przesuwamy zanegowany czas czynności z każdego wierzchołka na każdą z wchodzących do niego krawędzi. To znaczy, że jeśli wierzchołek v ma (niezanegowany) czas czynności t, to każdej krawędzi (u, v) wchodzącej do v nadajemy wagę $-t$. Oto dag, który otrzymujemy, z wagami krawędziowymi umieszczonymi obok krawędzi:



Pozostaje nam już tylko znaleźć w tym dagu najkrótszą ścieżkę (zacieniowaną) od startu do mety, opierając się na wagach krawędziowych. Ścieżka krytyczna w oryginalnym diagramie PERT będzie odpowiadać wierzchołkom położonym na najkrótszej ścieżce w dagu z pominięciem wierzchołków start i meta. Zobaczmy zatem, jak znaleźć w dagu najkrótszą ścieżkę.

Najkrótsza ścieżka w skierowanym grafie acyklicznym

Z nauki znajdowania najkrótszej ścieżki w dagu wynika inna korzyść: kładziemy w ten sposób podstawy pod znajdowanie najkrótszych ścieżek w dowolnych grafach skierowanych, również tych z cyklami. Przyjrzymy się temu ogólniejszemu problemowi w rozdziale 6. Tak jak to zrobiliśmy w przypadku sortowania topologicznego dagu, założymy, że dag jest zapamiętany w reprezentacji list sąsiedztwa i że z każdą krawędzią (u, v) zapamiętaliśmy również jej wagę jako $waga(u, v)$.

W dagu, który wyprowadzamy z diagramu PERT, jest nam potrzebna najkrótsza ścieżka z **wierzchołka źródłowego** (ang. *source vertex*), który nazywamy „startowym", do konkretnego **wierzchołka docelowego** (ang. *target vertex*), czyli „mety". Rozwiążemy tu ogólniejszy problem znalezienia **najkrótszych ścieżek z jednym**

źródłem (ang. *single-source shortest paths*), w którym poszukujemy najkrótszych ścieżek z wierzchołka źródłowego do *wszystkich* innych wierzchołków. Na zasadzie umowy wierzchołek źródłowy oznaczamy jako *s*, a dla każdego wierzchołka *v* chcemy obliczyć dwie rzeczy. Po pierwsze, wagę najkrótszej ścieżki z *s* do *v*, którą oznaczamy jako *ns*(*s*, *v*). Po drugie, **poprzednika** (ang. *predecessor*) *v* na najkrótszej ścieżce z *s* do *v*, tj. taki wierzchołek *u*, że najkrótsza ścieżka z *s* do *v* jest ścieżką z *s* do *u*, a potem jeszcze jedną krawędzią (*u*, *v*). Ponumerujemy *n* wierzchołków od 1 do *n*, aby nasze algorytmy dla najkrótszych ścieżek — ten tutaj i ten w rozdziale 6 — mogły przechować te wyniki w tablicach, odpowiednio: *najkrótsza*[1..*n*] i *poprz*[1..*n*]. W trakcie postępowania algorytmów wartości w *najkrótsza*[*v*] i *poprz*[*v*] mogą nie być poprawnymi ostatecznymi wartościami, lecz gdy algorytmy dokończą działania, będą takie.

Musimy zająć się paroma przypadkami, które mogą się pojawić. Przede wszystkim, co będzie, jeśli od *s* do *v* nie ma żadnej ścieżki? Wówczas definiujemy *ns*(*s*, *v*) = ∞, tzn. powinno się okazać, że *najkrótsza*[*v*] wynosi ∞. Ponieważ *v* nie powinien mieć poprzedników na najkrótszej ścieżce z *s*, mówimy również, że *poprz*[*v*] powinien przybrać specjalną wartość NULL. Co więcej, wszystkie najkrótsze ścieżki z *s* zaczynają się od *s*, a ponieważ *s* też nie ma poprzednika, więc powiemy, że *poprz*[*s*] również powinien być NULL. Drugi przypadek powstaje tylko w grafach, które mają zarówno cykle, jak i ujemne wagi przy krawędziach: co się dzieje, jeśli waga cyklu jest ujemna? Moglibyśmy wtedy krążyć po cyklu nieustannie, zmniejszając każdorazowo wagę ścieżki. Jeśli możemy przejść z *s* do cyklu o ujemnej wadze, a potem do *v*, to *ns*(*s*, *v*) jest nieokreślone. Na razie jednak interesujemy się tylko grafami acyklicznymi, nie musimy więc martwić się cyklami, a tym bardziej cyklami o ujemnych wagach.

Aby obliczyć najkrótsze ścieżki z wierzchołka źródłowego *s*, rozpoczynamy od *najkrótsza*[*s*] = 0 (ponieważ nie musimy nigdzie się wybierać, aby dotrzeć z wierzchołka do niego samego), *najkrótsza*[*v*] = ∞ dla wszystkich innych wierzchołków *v* (bo nie wiemy z góry, które wierzchołki są osiągalne z *s*) i *poprz*[*v*] = NULL dla wszystkich wierzchołków *v*. Następnie stosujemy ciąg *kroków osłabiających* (ang. *relaxation steps*)[8] do krawędzi grafu:

Procedura OSŁABIAJ(*u*, *v*)

Dane wejściowe: u, *v* — wierzchołki takie, że istnieje krawędź (*u*, *v*).

Wynik: wartość *najkrótsza*[*v*] może się zmniejszyć, a jeśli tak, to *poprz*[*v*] staje się *u*.

1. Jeśli *najkrótsza*[*u*] + *waga*(*u*, *v*) < *najkrótsza*[*v*], to nadaj *najkrótsza*[*v*] wartość *najkrótsza*[*u*] + *waga*(*u*, *v*) i ustaw *poprz*[*v*] na *u*.

[8] Występujący tu termin „osłabianie" (relaksacja) nie jest pierwszym przykładem terminu w ścisłej dziedzinie, którego uzasadnienie jest... historyczne; zob. dalej T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: *Wprowadzenie do algorytmów*, WNT 2004, s. 596 — *przyp. tłum.*

Kiedy wywołujemy OSŁABIAJ(u, v), ustalamy, czy możemy poprawić bieżącą najkrótszą ścieżkę z s do v, biorąc (u, v) jako ostatnią krawędź. Porównujemy wagę bieżącej najkrótszej ścieżki do u zwiększoną o wagę krawędzi (u, v) z wagą bieżącej najkrótszej ścieżki do v. Jeśli okaże się, że lepiej wziąć krawędź (u, v), to uaktualniamy *najkrótsza*[v] według tej nowej wagi i jako poprzednika v na najkrótszej ścieżce ustalamy u.

Jeśli będziemy kolejno osłabiać krawędzie wzdłuż najkrótszej ścieżki, otrzymamy poprawne wyniki. Możesz się dziwić, skąd ta nasza pewność co do kolejnego osłabiania krawędzi wzdłuż najkrótszej ścieżki, skoro nie wiemy nawet, o którą ścieżkę chodzi — w końcu to jej poszukujemy — okaże się to jednak łatwe w wypadku dagu. Dążymy do osłabienia wszystkich krawędzi w dagu, a kolejne krawędzie każdej najkrótszej ścieżki będą rozrzucone gdzieś po drodze, w miarę jak będziemy przechodzili wszystkie krawędzie i każdą osłabiali.

Oto nieco ściślejsze sformułowanie postępowania dotyczącego osłabiania krawędzi wzdłuż najkrótszej ścieżki, które stosuje się do każdego grafu skierowanego — z cyklami lub bez:

> Zacznij od wykonania dla wszystkich wierzchołków *najkrótsza*[u] = ∞ i *poprz*[u] = NULL, z wyjątkiem *najkrótsza*[s] = 0 dla wierzchołka źródłowego s.
>
> Następnie osłabiaj krawędzie wzdłuż najkrótszej ścieżki z s do dowolnego wierzchołka v, *kolejno, zaczynając od krawędzi wychodzącej z s i kończąc na krawędzi wchodzącej do v*. Osłabianie innych krawędzi może się swobodnie splatać z osłabianiem wzdłuż tej najkrótszej ścieżki, lecz tylko osłabienia mogą zmienić którekolwiek z wartości *najkrótsza* lub *poprz*.
>
> Wartości *najkrótsza* i *poprz* wierzchołka v są po osłabieniu krawędzi poprawne: *najkrótsza*[v] = *ns*(s, v), a *poprz*[v] jest wierzchołkiem poprzedzającym v na pewnej najkrótszej ścieżce z s.

Nietrudno zauważyć, dlaczego osłabianie krawędzi wzdłuż najkrótszej ścieżki działa należycie. Przypuśćmy, że najkrótsza ścieżka z s do v przechodzi kolejno wierzchołki s, v_1, v_2, v_3,..,v_k, v. Po osłabieniu krawędzi (s, v_1) *najkrótsza*[v_1] musi mieć poprawną wagę najkrótszej ścieżki dla v_1, a *poprz*[v_1] musi być s. Po osłabieniu (v_1, v_2) *najkrótsza*[v_2] i *poprz*[v_2] muszą być poprawne. I tak dalej, aż do osłabienia (v_k, v), po którym poprawne stają się wartości *najkrótsza*[v] i *poprz*[v].

To bardzo dobra nowina. W dagu naprawdę łatwo osłabić każdą krawędź dokładnie raz, ale osłabić krawędzie wzdłuż *każdej* najkrótszej ścieżki kolejno — jak? Po pierwsze, posortuj dag topologicznie. Następnie rozważ każdy wierzchołek w liniowym porządku sortowania topologicznego i osłabiaj wszystkie krawędzie wychodzące z wierzchołka. Ponieważ każda krawędź musi wychodzić z wierzchołka wcześniejszego w porządku liniowym i wchodzić do wierzchołka późniejszego w tym uporządkowaniu, każda ścieżka w dagu musi odwiedzać wierzchołki w kolejności spójnej z porządkiem liniowym.

Procedura NAJKRÓTSZE-SCIEŻKI-DAGU(*G*, *s*)

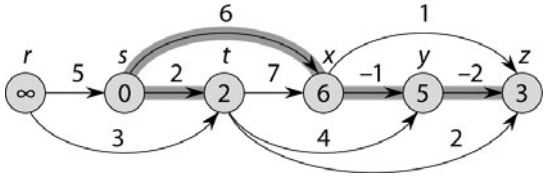
Dane wejściowe:

- *G*: ważony, acykliczny graf skierowany zawierający zbiór *V n* wierzchołków i zbiór *E m* krawędzi skierowanych.
- *s*: wierzchołek źródłowy w *V*.

Wynik: dla każdego nieźródłowego wierzchołka *v* w *V*, *najkrótsza*[*v*] jest wagą *ns*(*s*, *v*) najkrótszej ścieżki z *s* do *v*, a *poprz*[*v*] jest wierzchołkiem poprzedzającym *v* na pewnej najkrótszej ścieżce. Dla wierzchołka źródłowego *s najkrótsza*[*s*] = 0, a *poprz*[*s*] = NULL. Jeśli nie istnieje ścieżka z *s* do *v*, to *najkrótsza*[*v*] = ∞, a *poprz*[*v*] = NULL.

1. Wywołaj SORTOWANIE-TOPOLOGICZNE(*G*) i określ *l* jako porządek liniowy wierzchołków zwrócony przez to wywołanie.
2. Nadaj *najkrótsza*[*v*] wartość ∞ dla każdego wierzchołka *v* z wyjątkiem *s*, nadaj *najkrótsza*[*s*] wartość 0 i ustaw *poprz*[*v*] na NULL dla każdego wierzchołka *v*.
3. Dla każdego wierzchołka *u* pobranego w porządku ustalonym przez *l*:
   A. Dla każdego wierzchołka *v* sąsiadującego z *u*:
      i. Wywołaj OSŁABIAJ(*u*, *v*).

Na następnej stronie pokazano dag z wagami obok krawędzi. Wartości tablicy *najkrótsza* z wykonania NAJKRÓTSZYCH-SCIEŻEK-DAGU od wierzchołka źródłowego *s* występują wewnątrz wierzchołków, a zacieniowane krawędzie wskazują wartości *poprz*. Wierzchołki są ułożone od lewej do prawej w porządku liniowym zwróconym przez sortowanie topologiczne, tak więc wszystkie krawędzie idą od lewej do prawej. Jeżeli krawędź (*u*, *v*) jest zacieniowana, to *poprz*[*v*] jest *u*, a *najkrótsza*[*v*] = *najkrótsza*[*u*] + *waga*(*u*, *v*). Na przykład, skoro (*x*, *y*) jest zacieniowana, *poprz*[*y*] = *x* i wartość *najkrótsza*[*y*] (wynosząca 5) równa się *najkrótsza*[*x*] (wynosząca 6) + *waga*(*x*, *y*) (wynosząca –1). Nie ma ścieżki z *s* do *r*, dlatego *najkrótsza*[*r*] = ∞, a *poprz*[*r*] = NULL (żadne zacieniowane krawędzie nie wchodzą do *r*).



Pierwsza iteracja pętli w kroku 3 osłabia krawędzie (*r*, *s*) i (*r*, *t*), zostawiając *r*, lecz ponieważ *najkrótsza*[*r*] = ∞, osłabienia te niczego nie wnoszą. Następna iteracja pętli osłabia krawędzie (*s*, *t*) i (*s*, *x*), zostawiając *s* i powodując ustawienie *najkrótsza*[*t*] na 2, *najkrótsza*[*x*] na 6 oraz *poprz*[*t*] i *poprz*[*x*] na *s*. Kolejna iteracja osłabia krawędzie (*t*, *x*), (*t*, *y*) i (*t*, *z*), zostawiając *t*. Wartość *najkrótsza*[*x*] się nie zmienia, ponieważ

najkrótsza[*t*] + *waga*(*t*, *x*), która wynosi 2 + 7 = 9, jest większa niż *najkrótsza*[*x*], która wynosi 6; *najkrótsza*[*y*] przyjmuje jednak wartość 6, *najkrótsza*[*z*] — wartość 4 i zarówno *poprz*[*y*], jak i *poprz*[*z*] zostają ustawione na *t*. Następna iteracja osłabia (*x*, *y*) i (*x*, *z*), zostawiając *x* i powodując ustawienie *najkrótsza*[*y*] na 5 i *poprz*[*y*] na *x*; *najkrótsza*[*z*] i *poprz*[*z*] pozostają bez zmian. Ostatnia iteracja osłabia krawędź (*y*, *z*), zostawiając *y* i powodując ustawienie *najkrótsza*[*z*] na 3 i *poprz*[*z*] na *y*.

Łatwo możesz się przekonać, że NAJKRÓTSZE-SCIEŻKI-DAGU działają w czasie $\Theta(n + m)$. Jak widzieliśmy, krok 1 zabiera $\Theta(n + m)$ czasu i oczywiście w kroku 2 są nadawane dwie wartości początkowe dla każdego wierzchołka, co zabiera $\Theta(n)$ czasu. Jak widzieliśmy wcześniej, zewnętrzna pętla z kroku 3 sprawdza każdy wierzchołek dokładnie raz we wszystkich iteracjach. Ponieważ każde wywołanie OSŁABIAJ w kroku 3Ai zużywa stałą ilość czasu, krok 3 zużywa $\Theta(n + m)$ czasu. Dodając czasy działania tych kroków, otrzymujemy czas $\Theta(n + m)$ procedury.

Wracając do diagramów PERT, łatwo teraz zobaczyć, że znalezienie ścieżki krytycznej zajmuje $\Theta(n + m)$ czasu, jeżeli diagram PERT ma *n* wierzchołków i *m* krawędzi. Dodajemy oba wierzchołki, start i meta, i dodajemy najwyżej *m* krawędzi wychodzących z wierzchołka start i najwyżej *m* krawędzi dochodzących do mety, co daje łącznie najwyżej 3*m* krawędzi w dagu. Negowanie wag i wyciąganie ich z wierzchołków na krawędzie zabiera $\Theta(m)$ czasu, zatem znajdowanie najkrótszej ścieżki w wynikowym dagu zajmuje $\Theta(n + m)$ czasu.

Co czytać dalej

W rozdziale 22 CLRS [CLRS09] przedstawiono inny algorytm sortowania topologicznego dagu niż zaprezentowany w tym rozdziale, który występuje w tomie 1 *The Art of Computer Programming* Knutha [Knu97]. Na pierwszy rzut oka metoda w CLRS jest trochę prostsza, lecz mniej intuicyjna niż podejście przedstawione w tym rozdziale i opiera się na technice odwiedzania wierzchołków w grafie, znanej jako „przeszukiwanie w głąb". Algorytm znajdowania w dagu najkrótszych ścieżek z jednym źródłem występuje w rozdziale 24 CLRS.

Więcej o diagramach PERT, będących w użyciu od lat 50. XX wieku, możesz poczytać w dowolnej z wielu książek o zarządzaniu projektami.

6 Najkrótsze ścieżki

W rozdziale 5 zobaczyliśmy, jak znajdywać najkrótsze ścieżki z jednym źródłem w skierowanym grafie acyklicznym. Algorytm, który to robił, był oparty na założeniu, że graf jest acykliczny — nie ma cykli — dzięki czemu mogliśmy najpierw posortować topologicznie wierzchołki grafu.

Większość grafów modelujących sytuacje spotykane w życiu ma jednak cykle. Na przykład w grafie modelującym sieć dróg każdy wierzchołek reprezentuje skrzyżowanie, a każda krawędź skierowana — drogę, którą możesz podróżować w jednym kierunku między skrzyżowaniami. (Drogi dwukierunkowe byłyby reprezentowane za pomocą dwóch osobnych krawędzi, biegnących w przeciwnych kierunkach). Takie grafy muszą mieć cykle, inaczej z chwilą opuszczenia skrzyżowania nie dałoby się na nie wrócić. Dlatego gdy Twój GPS oblicza najkrótsze lub najszybsze drogi do celu, graf, na którym działa, ma cykle — ma ich całe mnóstwo.

Gdy Twój GPS znajduje najszybszą drogę z Twojego bieżącego miejsca pobytu do określonego celu, rozwiązuje problem **najkrótszej ścieżki między jedną parą wierzchołków** (ang. *single-pair shortest path*). Żeby to zrobić, używa prawdopodobnie algorytmu, który znajduje wszystkie najkrótsze ścieżki z jednego źródła, lecz koncentruje się wyłącznie na najkrótszej ścieżce, którą wytycza do określonego miejsca przeznaczenia.

Twój GPS działa na ważonym grafie skierowanym, w którym wagi krawędzi reprezentują odległość albo czas podróży. Ponieważ nie możesz przebywać ujemnych odległości lub docierać do celu, nim rozpoczniesz podróż, wszystkie wagi grafu w Twoim GPS-ie są dodatnie. Przypuszczam, że niektóre z nich mogą być równe 0 z pewnych niejasnych powodów, powiedzmy więc, że wagi krawędzi są nieujemne. Kiedy wszystkie wagi krawędzi są nieujemne, nie musimy się martwić o cykle z ujemnymi wagami i wszystkie najkrótsze ścieżki są dobrze określone.

Jako inny przykład najkrótszych ścieżek z jednym źródłem rozważmy grę „sześć stopni Kevina Bacona", w której gracze starają się łączyć aktorów filmowych z Kevinem Baconem. Każdy wierzchołek w grafie reprezentuje aktora, a graf zawiera krawędzie (u, v) i (v, u), jeśli aktorzy reprezentowani przez wierzchołki u i v wystąpili kiedyś w tym samym filmie. Mając jakiegoś aktora, gracz próbuje znaleźć najkrótszą ścieżkę z wierzchołka z tym aktorem do wierzchołka z Kevinem Baconem. Liczba krawędzi w najkrótszej ścieżce (mówiąc inaczej, waga najkrótszej ścieżki, gdy wszystkie wagi krawędzi wynoszą 1) jest „liczbą Kevina Bacona" dla danego aktora. Oto przykład: Renée Adorée grała w filmie z Bessie Love, która grała w filmie z Eli Wallach, która kręciła film z Kevinem Baconem, wobec tego dla Renée Adorée liczba Kevina Bacona wynosi 3. Matematycy posługują się podobnym pojęciem w związku

z liczbą Erdösa, która podaje najkrótszą ścieżkę od wielkiego Paula Erdösa do dowolnego innego matematyka poprzez łańcuch powiązań współautorów[1].

A co z grafami o ujemnych wagach krawędzi? Jak one się mają do rzeczywistego świata? Zobaczymy, że możemy sformułować problem określenia, czy istnieje możliwość arbitrażu w handlu walutami na zasadzie rozstrzygania, czy graf, w którym dopuszcza się ujemne wagi krawędzi, ma cykl z ujemną wagą.

Przechodząc do algorytmów, najpierw zbadamy algorytm Dijkstry znajdowania najkrótszych ścieżek z jednego źródłowego wierzchołka do wszystkich innych wierzchołków. Algorytm Dijkstry działa na grafach, które pod dwoma względami istotnie różnią się od grafów oglądanych przez nas w rozdziale 5: wszystkie wagi krawędzi muszą być nieujemne, a grafy mogą zawierać cykle. Ma to zasadnicze znaczenie dla sposobu, w jaki Twój GPS poszukuje tras. Przyjrzymy się też kilku możliwościom realizacji algorytmu Dijkstry. Następnie obejrzymy algorytm Bellmana-Forda, wyjątkowo prostą metodę znajdowania najkrótszych ścieżek z jednym źródłem — nawet wtedy, kiedy występują krawędzie z wagami ujemnymi. Wynik algorytmu Bellmana-Forda możemy zastosować do określania, czy graf zawiera cykl z wagą ujemną, i jeśli zawiera — do identyfikowania wierzchołków i krawędzi tego cyklu. Zarówno algorytm Dijkstry, jak i algorytm Bellmana-Forda datuje się na koniec lat 50. XX wieku, algorytmy te przetrwały więc próbę czasu. Zakończymy na skonstruowanym przez Floyda i Warshalla algorytmie problemu wszystkich par, w którym chcemy znaleźć najkrótszą ścieżkę między każdą parą wierzchołków.

Jak to uczyniliśmy w rozdziale 5 dla znajdowania najkrótszych ścieżek w dagu, założymy, że mamy dany wierzchołek źródłowy s i wagę $waga(u, v)$ każdej krawędzi (u, v) oraz że dla każdego wierzchołka v chcemy obliczyć wagę najkrótszej ścieżki $ns(u, v)$ z s do v, a także wierzchołek poprzedzający v na pewnej najkrótszej ścieżce z s. Wyniki będziemy przechowywali w tablicach, odpowiednio: $najkrótsza[v]$ i $poprz[v]$.

Algorytm Dijkstry

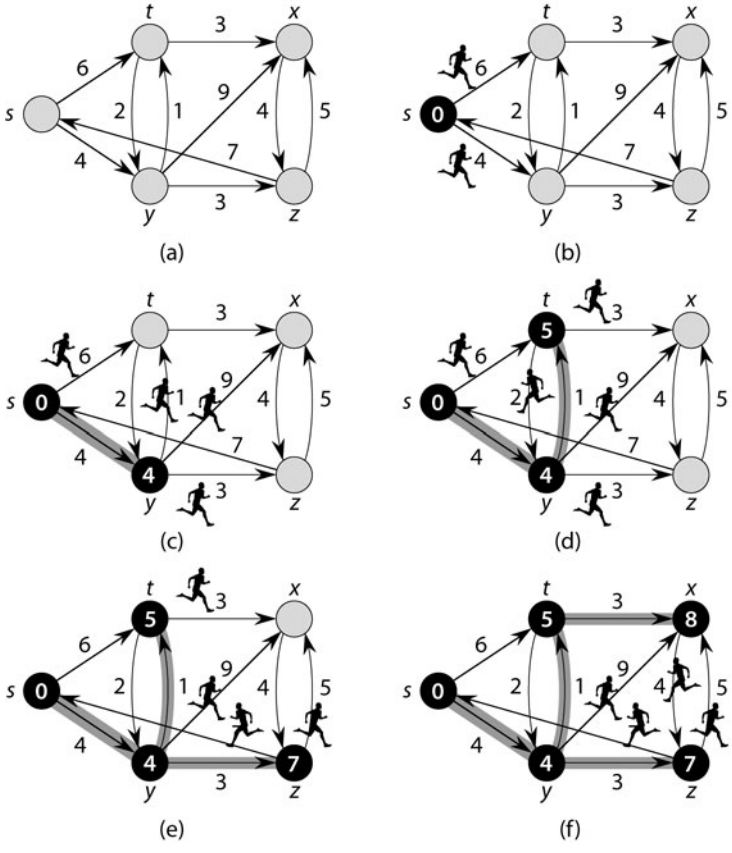
Lubię myśleć o algorytmie Dijkstry[2] jak o symulacji, w której wysyła się biegaczy w grafie.

W warunkach idealnych symulacja ta działa następująco (aczkolwiek zobaczymy, że algorytm Dijkstry działa nieco inaczej). Rozpoczyna się od wysłania biegaczy od wierzchołka źródłowego do wszystkich sąsiadujących wierzchołków. Gdy biegacz dociera do danego wierzchołka po raz pierwszy, wierzchołek ten natychmiast opuszczają nowi biegacze, kierując się do wierzchołków z nim sąsiadujących. Spójrzmy na część (a) poniższego rysunku:

---

[1] Możesz wierzyć lub nie, ale istnieje nawet coś takiego jak liczba Erdösa-Bacona, będącą sumą liczb Erdösa i Bacona, i niemała grupa ludzi mających skończone liczby Erdösa-Bacona, w tym sam Paul Erdös!

[2] Nazwanym tak od nazwiska Edsgera Dijkstry, który zaprezentował ten algorytm w 1959 r.



Widać na nim graf skierowany z wierzchołkiem źródłowym *s* i wagami obok krawędzi. Możemy uważać wagę wierzchołka za liczbę minut, które zajmuje biegaczowi przebycie krawędzi.

W części (b) widać początek symulacji w chwili 0. W tej chwili, pokazanej wewnątrz wierzchołka *s*, biegacze wyruszają z *s* i kierują się ku swoim sąsiadującym wierzchołkom *t* i *y*. Zaczerniony wierzchołek *s* wskazuje, że wiemy, iż *najkrótsza*[*s*] = 0.

Kilka minut potem, w czasie 4, biegacz dociera do wierzchołka *y*, co pokazano w części (c). Ponieważ biegacz ten przybywa do *y* jako pierwszy, wiemy, że *najkrótsza*[*y*] = 4, więc *y* jest zaczerniony na rysunku. Zacieniowana krawędź (*s*, *y*) wskazuje, że pierwszy biegacz, który dotarł do *y*, pochodził z wierzchołka *s*, dlatego *poprz*[*y*] = *s*. W czwartej minucie biegacz przemieszczający się z wierzchołka *s* do wierzchołka *t* wciąż jest w drodze, a z wierzchołka *y* wyruszają biegacze, kierując się do wierzchołków *t*, *x* i *z*.

Do następnej sytuacji, uwidocznionej w części (d), dochodzi w minutę później, w czasie 5, gdy biegacz z wierzchołka *y* przybywa do wierzchołka *t*. Biegacz z *s* do *t* jeszcze tam nie dotarł. Ponieważ jako pierwszy do wierzchołka *t* dociera w minucie 5 biegacz z wierzchołka *y*, ustawiamy *najkrótsza*[*t*] na 5 i *poprz*[*t*] na *y* (co wskazano

przez zacieniowanie krawędzi (y, t)). Kolejni biegacze wyruszają z wierzchołka t, kierując się tym razem ku wierzchołkom x i y.

Biegacz z wierzchołka s dociera wreszcie do wierzchołka t w czasie 6, lecz biegacz z wierzchołka y był tam minutę wcześniej; wysiłek biegacza z s do t idzie więc na marne.

W 7 minucie, przedstawionej w części (e), dwaj biegacze osiągają swoje cele. Biegacz z wierzchołka t dociera do wierzchołka y, lecz biegacz z s do y był tam już w 4 minucie, więc w symulacji zapomina się o biegaczu z t do y. W tym samym czasie biegacz z y pojawia się w wierzchołku z. Nadajemy *najkrótsza*[z] wartość 7 i *poprz*[z] wartość y, po czym nowi biegacze opuszczają wierzchołek z, dążąc do wierzchołków s i x.

Następne zdarzenie, z 8 minuty, pokazano w części (f), kiedy to biegacz z wierzchołka t dociera do wierzchołka x. Nadajemy *najkrótsza*[x] wartość 8 i podstawiamy t do *poprz*[x], a z wierzchołka x wyrusza biegacz, kierując się do wierzchołka z.

W tym momencie każdy wierzchołek został już odwiedzony przez jakiegoś biegacza i symulacja może być zakończona. Wszyscy biegacze pozostający w drodze dotrą do swoich docelowych wierzchołków po którymś z innych biegaczy, którzy tam się już pojawili. Z chwilą gdy do każdego wierzchołka dotrze jakiś biegacz, wartość *najkrótsza* dla każdego wierzchołka równa się wadze najkrótszej ścieżki z wierzchołka s, a wartość *poprz* dla każdego wierzchołka jest poprzednikiem na najkrótszej ścieżce z s.

Oto jak przebiegałaby symulacja idealna. Zależała ona od czasu zużywanego przez biegacza na pokonanie krawędzi, równego wadze krawędzi. Algorytm Dijkstry działa trochę inaczej. Traktuje wszystkie krawędzie tak samo, tzn. podczas rozpatrywania krawędzi wychodzących z wierzchołka przetwarza wierzchołki sąsiadujące razem, bez żadnego konkretnego porządku. Na przykład gdy algorytm Dijkstry przetwarza krawędzie wychodzące z wierzchołka s, *zawczasu* deklaruje, że *najkrótsza*[y] = 4, *najkrótsza*[t] = 6 oraz oba elementy *poprz*[y] i *poprz*[t] jako równe s. Kiedy algorytm Dijkstry uwzględni potem krawędź (y, t), zmniejsza znalezioną dotychczas wagę najkrótszej ścieżki do wierzchołka t, tak więc *najkrótsza*[t] maleje z 6 na 5, a *poprz*[t] przełącza się z s na y.

Algorytm Dijkstry działa na zasadzie wywoływania omówionej w poprzednim rozdziale procedury OSŁABIAJ, po razie na krawędź. Osłabienie krawędzi (u, v) odpowiada przybyciu biegacza z wierzchołka u do wierzchołka v. Algorytm utrzymuje zbiór Q wierzchołków, dla których nie są jeszcze znane ostateczne wartości w *najkrótsza* i *poprz*; wartości w *najkrótsza* i *poprz* odnoszące się do wszystkich wierzchołków *spoza* Q są już ostateczne. Po zainicjowaniu *najkrótsza*[s] na 0 dla wierzchołka źródłowego s, *najkrótsza*[v] na ∞ dla wszystkich innych wierzchołków i *poprz*[v] na NULL dla wszystkich wierzchołków algorytm powtarza poszukiwanie wierzchołka u w zbiorze Q, mającego najmniejszą wartość *najkrótsza*, usuwa ten wierzchołek z Q i osłabia wszystkie krawędzie wychodzące z u. Oto ta procedura:

Procedura DIJKSTRA(*G*, *s*)

Dane wejściowe:

- *G*: graf skierowany zawierający zbiór *V n* wierzchołków i zbiór *E m* krawędzi skierowanych z nieujemnymi wagami.
- *s*: wierzchołek źródłowy w *V*.

Wynik: dla każdego nieźródłowego wierzchołka *v* w *V*, *najkrótsza*[*v*] jest wagą *ns*(*s*, *v*) najkrótszej ścieżki z *s* do *v*, a *poprz*[*v*] jest wierzchołkiem poprzedzającym *v* na pewnej najkrótszej ścieżce. Dla wierzchołka źródłowego *s najkrótsza*[*s*] = 0, a *poprz*[*s*] = NULL. Jeśli nie istnieje ścieżka z *s* do *v*, to *najkrótsza*[*v*] = ∞, a *poprz*[*v*] = NULL. (Tak samo jak w NAJKRÓTSZYCH-SCIEŻKACH-DAGU).

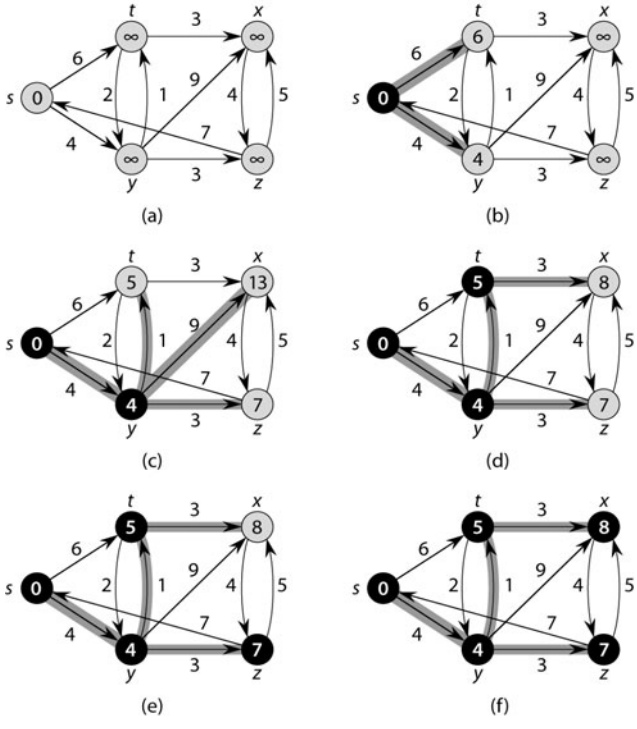
1. Nadaj *najkrótsza*[*v*] wartość ∞ dla każdego wierzchołka *v* z wyjątkiem *s*, nadaj *najkrótsza*[*s*] wartość 0 i ustaw *poprz*[*v*] na NULL dla każdego wierzchołka *v*.
2. Umieść wszystkie wierzchołki w zbiorze *Q*.
3. Dopóki *Q* jest niepusty, wykonuj, co następuje:
   A. Znajdź w *Q* wierzchołek *u* z najniższą wartością *najkrótsza* i usuń go z *Q*.
   B. Dla każdego wierzchołka *v* sąsiadującego z *u*:
      i. Wywołaj OSŁABIAJ(*u*, *v*).

Na następnym rysunku każda jego część ukazuje wartość *najkrótsza* (umieszczoną wewnątrz każdego wierzchołka) i wartość *poprz* (wskazywaną przez zacienione krawędzie) oraz zbiór *Q* (wierzchołki zacieniowane, ale nie zaczernione) tuż przed iteracją pętli z kroku 3.

W każdej części rysunku wierzchołek świeżo zaczerniony jest wierzchołkiem wybranym jako wierzchołek *u* w kroku 3A. W symulacji z biegaczami, z chwilą gdy wierzchołek otrzyma wartości *najkrótsza* i *poprz*, nie mogą one zostać zmienione, lecz tutaj wierzchołek mógłby otrzymać wartości *najkrótsza* i *poprz* z osłabienia krawędzi, a późniejsze osłabienie innej krawędzi mogłoby zmienić te wartości. Na przykład po osłabieniu krawędzi (*y*, *x*) w części (c) rysunku, wartość *najkrótsza*[*x*] zmniejsza się z ∞ do 13, a *poprz*[*x*] staje się *y*. Następna iteracja pętli w kroku 3 (część (d)) osłabia krawędź (*t*, *x*) i dalej zmniejsza *najkrótsza*[*x*], do 8, a *poprz*[*x*] staje się *t*. W następnej iteracji (część (e)) jest osłabiana krawędź (*z*, *x*), lecz tym razem *najkrótsza*[*x*] się nie zmienia, ponieważ jej wartość 8 jest mniejsza niż *najkrótsza*[*z*] + *waga*(*z*, *x*), co jest równe 12.

Algorytm Dijkstry utrzymuje następujący niezmiennik pętli:

> Na początku każdej iteracji pętli w kroku 3 *najkrótsza*[*v*] = *ns*(*s*, *v*) dla każdego wierzchołka *v* niebędącego w zbiorze *Q*. To znaczy, że dla każdego wierzchołka *v* spoza *Q* wartość *najkrótsza*[*v*] jest wagą najkrótszej ścieżki z *s* do *v*.



Oto uproszczona wersja uzasadnienia niezmiennika pętli. (Dowód formalny jest nieco bardziej złożony). Na początku wszystkie wierzchołki są w zbiorze Q, wobec tego niezmiennik pętli nie stosuje się do żadnego wierzchołka na wejściu do pierwszej iteracji pętli w kroku 3. Załóżmy, że gdy rozpoczynamy iterację tej pętli, wszystkie wierzchołki poza zbiorem Q mają w swoich wartościach *najkrótsza* poprawne wagi najkrótszej ścieżki. Wtedy każda krawędź wychodząca z tych wierzchołków została osłabiona w którymś wykonaniu kroku 3Bi. Rozważmy wierzchołek u ze zbioru Q z najniższą wartością *najkrótsza*. Jego wartość *najkrótsza* już nigdy się nie zmniejszy. Dlaczego? Ponieważ jedynymi krawędziami pozostającymi do osłabienia są krawędzie wychodzące z wierzchołków w Q, a każdy wierzchołek w Q ma wartość *najkrótsza* przynajmniej tak dużą jak *najkrótsza*[u]. Ponieważ wagi wszystkich krawędzi są nieujemne, musi być *najkrótsza*[u] ≤ *najkrótsza*[v] + *waga*(u, v) dla każdego wierzchołka v w Q, więc żaden przyszły krok osłabiania nie zmniejszy *najkrótsza*[u]. Dlatego *najkrótsza*[u] jest tak mała, jak to możliwe, i możemy usunąć wierzchołek u z Q i osłabić wszystkie krawędzie wychodzące z u. Kiedy pętla z kroku 3 się kończy, zbiór Q jest pusty, więc wszystkie wierzchołki mają poprawne wagi najkrótszych ścieżek w wartościach tablicy *najkrótsza*.

Możemy przystąpić do analizy czasu działania procedury DIJKSTRA, lecz żeby zanalizować ją w pełni, musimy najpierw rozstrzygnąć kilka detali realizacyjnych. Przypomnijmy za rozdziałem 5, że oznaczamy liczbę wierzchołków przez n, a liczbę krawędzi przez m i że $m < n^2$. Wiemy, że krok 1 zabiera $\Theta(n)$ czasu. Wiemy również, że treść pętli z kroku 3 jest powtarzana dokładnie n razy, ponieważ zbiór Q począt-

kowo zawiera wszystkie n wierzchołków, każda iteracja pętli usuwa jeden wierzchołek z Q, a wierzchołki nigdy nie są dodawane ponownie do Q. W trakcie algorytmu pętla w kroku 3A przetwarza każdy wierzchołek i każdą krawędź dokładnie raz (widzieliśmy ten sam pomysł w procedurach SORTOWANIE-TOPOLOGICZNE i NAJKRÓTSZE-SCIEŻKI-DAGU w rozdziale 5).

Co pozostaje do zanalizowania? Musimy zrozumieć, jak długo trwa umieszczanie wszystkich n wierzchołków w zbiorze Q (krok 2), jak długo trwa ustalenie, który wierzchołek w Q ma najniższą wartość *najkrótsza*, i usunięcie tego wierzchołka z Q (krok 3A) oraz jakie czynności administracyjne musimy wykonać — jeśli w ogóle jakieś są potrzebne — gdy wartości *najkrótsza* i *poprz* wierzchołka zmieniają się wskutek wywołania OSŁABIAJ. Nazwijmy te operacje:

- WSTAW(Q, v) wstawia wierzchołek v do zbioru Q. (Algorytm Dijkstry wywołuje WSTAW n razy).
- WYDOBĄDŹ-MIN(Q) usuwa z Q wierzchołek z minimalną wartością *najkrótsza* i zwraca ten wierzchołek wywołującemu. (Algorytm Dijkstry wywołuje WYDOBĄDŹ-MIN n razy).
- ZMNIEJSZ-KLUCZ(Q, v) wykonuje niezbędne prace administracyjne w Q, aby odnotować, że element *najkrótsza*[v] został zmniejszony przez wywołanie OSŁABIAJ. (Algorytm Dijkstry wywołuje ZMNIEJSZ-KLUCZ najwyżej n razy).

Te trzy operacje wzięte razem definiują **kolejkę priorytetową** (ang. *priority queue*).

Opisy operacji kolejki priorytetowej określają tylko, *co* robią operacje, nie wyjaśniając, *jak* to robią. W projektowaniu oprogramowania oddzielenie tego, *co* wykonują operacje, od tego, *jak* one to robią, jest nazywane **abstrahowaniem**. Zbiór operacji określających, *co* jest robione, lecz nie *jak*, nazywamy **abstrakcyjnym typem danych** (ang. *abstract data type*, ADT), tak więc kolejka priorytetowa jest ADT[3].

Operacje kolejki priorytetowej — czyli *jak* — możemy zrealizować za pomocą dowolnej z kilku struktur danych. **Struktura danych** (ang. *data structure*) oznacza specyficzny sposób przechowywania danych w komputerze i dostępu do nich — przykładem jest tablica. W wypadku kolejek priorytetowych poznamy trzy różne struktury danych, z użyciem których można realizować te operacje. Projektanci oprogramowania powinni umieć zaprząc w tym celu dowolną strukturę danych zdatną do implementacji operacji ADT. Jednak gdy myślimy o algorytmach, nie jest to takie proste. Wynika to z tego, że dla różnych struktur danych sposób, w jaki implementują one operacje, może powodować różnice w ilości czasu. Rzeczywiście, zobaczmy, że trzy struktury danych, których użyjemy do implementacji kolejki priorytetowej ADT, będą dawały różne czasy działania w algorytmie Dijkstry.

Na następnej stronie podano przepisaną wersję procedury DIJKSTRA, w której operacje kolejki priorytetowej są wywołane jawnie. Zbadajmy przydatność trzech struktur danych do realizacji kolejki priorytetowej i zobaczmy, jak one wpływają na czas działania algorytmu Dijkstry.

---

[3] Tak to jest definiowane w książkach o algorytmach; w podręcznikach inżynierii oprogramowania spotkamy inną definicję, wiążącą operacje i struktury danych — *przyp. tłum.*

Procedura DIJKSTRA(*G*, *s*)

Dane wejściowe i wynik: takie same jak poprzednio.

1. Nadaj *najkrótsza*[*v*] wartość ∞ dla każdego wierzchołka *v* z wyjątkiem *s*, nadaj *najkrótsza*[*s*] wartość 0 i ustaw *poprz*[*v*] na NULL dla każdego wierzchołka *v*.
2. Utwórz *Q*, pustą kolejkę priorytetową.
3. Dla każdego wierzchołka *v*:
   A. Wywołaj WSTAW(*Q*, *v*).
4. Dopóki *Q* jest niepusty, wykonuj, co następuje:
   A. Wywołaj WYDOBĄDŹ-MIN(*Q*) i niech *u* przechowuje zwrócony wierzchołek.
   B. Dla każdego wierzchołka *v* sąsiadującego z *u*:
      i. Wywołaj Osłabiaj(*u*, *v*).
      ii. Jeśli wywołanie Osłabiaj(*u*, *v*) zmniejszyło wartość *najkrótsza*[*v*], to wywołaj Zmniejsz-Klucz(*Q*, *v*).

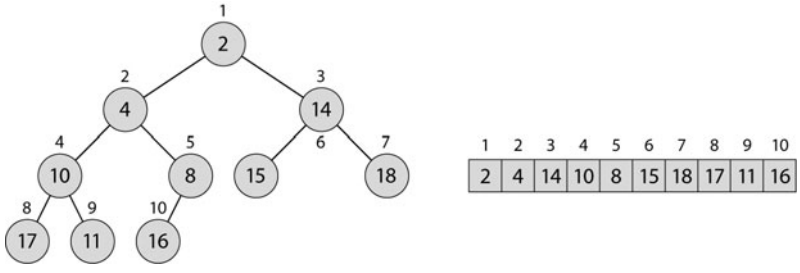
Prosta realizacja tablicowa

Najprostszy sposób zrealizowania operacji kolejki priorytetowej polega na zapamiętaniu wierzchołków na *n* pozycjach w tablicy. Jeśli kolejka priorytetowa zawiera aktualnie *k* wierzchołków, to znajdują się one na pierwszych *k* pozycjach tablicy, bez konkretnego uporządkowania. Wraz z tablicą potrzebujemy utrzymywać licznik aktualnie obecnych w niej wierzchołków. Operacja WSTAW jest łatwa: wystarczy umieścić wierzchołek na następnej nieużywanej pozycji w tablicy i zwiększyć licznik. Operacja ZMNIEJSZ-KLUCZ jest jeszcze łatwiejsza: nie trzeba robić nic! Obie te operacje zużywają stały czas. Operacja WYDOBĄDŹ-MIN zajmuje jednak $O(n)$ czasu, ponieważ musimy przejrzeć wszystkie wierzchołki aktualnie pamiętane w tablicy, aby znaleźć wierzchołek z najmniejszą wartością *najkrótsza*. Z chwilą gdy taki wierzchołek zidentyfikujemy, jego usunięcie będzie łatwe: wystarczy przenieść wierzchołek z ostatniej pozycji na pozycję usuwanego wierzchołka i zmniejszyć licznik. Tych *n* wywołań WYDOBĄDŹ-MIN zajmuje $O(n^2)$ czasu. Choć wywołania OSŁABIAJ zajmują $O(m)$ czasu, przypomnijmy, że $m \leq n^2$. Dlatego w tej implementacji kolejki priorytetowej algorytm Dijkstry zużywa $O(n^2)$ czasu, przy czym dominuje tu czas spędzany w procedurze WYDOBĄDŹ-MIN.

Realizacja z kopcem binarnym

Kopiec binarny organizuje dane w postaci drzewa binarnego przechowywanego w tablicy. **Drzewo binarne** (ang. *binary tree*) jest typem grafu, lecz w odniesieniu do jego wierzchołków używamy określenia **węzły** (ang. *nodes*), krawędzie są nieskierowane, a każdy węzeł ma pod sobą 0, 1 lub 2 węzły, nazywane jego **potomkami**

(dziećmi, ang. *children*). Po lewej stronie rysunku widnieje przykład drzewa binarnego z ponumerowanymi węzłami. Węzły bez potomków, jak węzły od 6 do 10, są **liśćmi** (ang. *leaves*)[4].



Kopiec binarny (ang. *binary heap*) jest drzewem binarnym z dodatkowymi właściwościami. Po pierwsze, drzewo takie jest w pełni zapełnione na wszystkich poziomach, z wyjątkiem być może najniższego, który jest zapełniony od lewej do pewnego miejsca. Po drugie, każdy węzeł zawiera klucz, pokazany na rysunku wewnątrz węzła. Po trzecie, klucze spełniają **własność kopca** (ang. *heap property*): klucz każdego węzła jest mniejszy lub równy kluczom jego potomków. Drzewo binarne na rysunku jest również kopcem binarnym.

Kopiec binarny możemy zapamiętać w tablicy, jak pokazano po prawej stronie rysunku. Ze względu na własność kopca węzeł o minimalnym kluczu musi zawsze być na pozycji 1. Potomkowie węzła na pozycji i występują na pozycjach $2i$ i $2i + 1$, a węzeł powyżej węzła na pozycji i — jego **ojciec** (przodek, ang. *parent*) — jest na pozycji $\lfloor i/2 \rfloor$. Łatwo nawigować w górę i w dół kopca binarnego, jeśli zapamiętamy go w tablicy.

Kopiec binarny ma jeszcze jedną istotną cechę: jeśli składa się z n węzłów, to jego **wysokość** (ang. *height*), czyli liczba krawędzi od korzenia w dół do najdalszego liścia, wynosi tylko $\lfloor \lg n \rfloor$. Dlatego możemy przebyć ścieżkę od korzenia w dół do liścia lub od liścia do korzenia w czasie zaledwie $O(\lg n)$.

Ponieważ kopce binarne mają wysokość $\lfloor \lg n \rfloor$, każdą z trzech operacji kolejki priorytetowej możemy wykonać w czasie $O(\lg n)$. Dla WSTAW dodajemy nowy liść na pierwszej dostępnej pozycji. Następnie, dopóki klucz w węźle jest mniejszy niż klucz jego ojca, dopóty zamieniamy treść[5] węzła z treścią jego ojca i przesuwamy ją o jeden poziom w kierunku korzenia. Innymi słowy, „bąbelkujemy" treść w stronę korzenia, aż dojdzie do przywrócenia własności kopca. Ponieważ ścieżka do korzenia ma najwyżej $\lfloor \lg n \rfloor$ krawędzi, wystąpi najwyżej $\lfloor \lg n \rfloor - 1$ zamian, więc operacja WSTAW zużywa czas $O(\lg n)$. Do wykonania ZMNIEJSZ-KLUCZ korzystamy z tego

[4] Informatykom łatwiej jest rysować drzewa z korzeniami u góry i gałęziami u dołu, niż rysować je na podobieństwo prawdziwych drzew, z korzeniem na dole i gałęziami zwróconymi do góry.

[5] Treść węzła obejmuje klucz i wszelkie inne informacje skojarzone z kluczem, na przykład to, który wierzchołek jest skojarzony z danym węzłem.

samego pomysłu: zmniejszamy klucz i bąbelkujemy treść w górę, do korzenia, aż do uzyskania własności kopca, co znów zabiera $O(\lg n)$ czasu. Żeby wykonać WYDOBĄDŹ-MIN, przechowujemy treść korzenia w celu zwrócenia wywołującemu. Następnie bierzemy ostatni liść (węzeł o największym numerze) i umieszczamy jego treść na pozycji korzeniowej. Następnie „bąbelkujemy w dół" treść korzenia, zamieniając treść węzła i potomka, którego klucz jest mniejszy, aż do uzyskania własności kopca. Na koniec zwracamy przechowaną treść węzła. I znowu, ponieważ ścieżka od korzenia w dół do liścia ma najwyżej $\lfloor \lg n \rfloor$ krawędzi, wystąpi najwyżej $\lfloor \lg n \rfloor - 1$ zamian, toteż WYDOBĄDŹ-MIN zużywa $O(\lg n)$ czasu.

Gdy algorytm Dijkstry używa do implementacji kolejki priorytetowej kopca binarnego, spędza na wstawianiu wierzchołków $O(n \lg n)$ czasu, $O(n \lg n)$ czasu na operacjach WYDOBĄDŹ-MIN i $O(n \lg n)$ czasu na operacjach ZMNIEJSZ-KLUCZ. (W gruncie rzeczy wstawianie n wierzchołków zabiera tylko $O(n)$ czasu, ponieważ na początku tylko wierzchołek źródłowy s ma wartość *najkrótsza* równą 0, a wszystkie inne wierzchołki mają wartości *najkrótsza* ustawione na ∞). Gdy graf jest **rzadki** (ang. *sparse*) — liczba m krawędzi jest znacznie mniejsza od n^2 — realizacja kolejki priorytetowej za pomocą kopca binarnego jest sprawniejsza niż z użyciem zwykłej tablicy. Grafy modelujące sieci drogowe są rzadkie, ponieważ średnio ze skrzyżowania wychodzi około czterech dróg, więc m wynosi około $4n$. Z drugiej strony, gdy graf jest **gęsty** (ang. *dense*) — m jest bliskie n^2, więc graf zawiera wiele krawędzi — to czas $O(n \lg n)$ spędzany przez algorytm Dijkstry na wywołaniach ZMNIEJSZ-KLUCZ może spowodować, że będzie on działał wolniej niż na kolejce priorytetowej w zwykłej tablicy.

I jeszcze jedno odnośnie do kopców binarnych: możemy ich używać do sortowania w czasie $O(n \lg n)$:

Procedura SORTOWANIE-NA-KOPCU(A, n) [6]

Dane wejściowe:

- A: tablica.
- n: liczba elementów w A do posortowania.

Wynik: tablica B zawierająca posortowane elementy A.

1. Zbuduj kopiec binarny Q z elementów A.
2. Niech $B[1..n]$ będzie nową tablicą.
3. Dla $i = 1$ do n:
   A. Wywołaj WYDOBĄDŹ-MIN(Q) i podstaw zwróconą wartość pod $B[i]$.
4. Zwróć tablicę B.

[6] Nazwa angielska tego algorytmu: *heapsort — przyp. tłum.*

W kroku 1 tablica wejściowa jest zamieniana na kopiec binarny, co możemy zrobić jednym z dwóch sposobów. Pierwszy polega na rozpoczęciu od pustego kopca binarnego i wstawianiu kolejnych elementów tablicy, co zajmuje $O(n \lg n)$ czasu. Drugi sposób polega na zbudowaniu kopca binarnego wprost wewnątrz tablicy: dzieje się to od dołu do góry i wymaga tylko czasu $O(n)$. Jest również możliwe sortowanie na miejscu z użyciem kopca, abyśmy nie musieli używać dodatkowej tablicy B.

Realizacja z użyciem kopca Fibonacciego

Możemy też zrealizować kolejkę priorytetową za pomocą skomplikowanej struktury danych zwanej „kopcem Fibonacciego" lub „F-kopcem". Za pomocą F-kopca n wywołań WSTAW i WYDOBĄDŹ-MIN zużywa łącznie czas $O(n \lg n)$, a m wywołań ZMNIEJSZ-KLUCZ zużywa czas $\Theta(m)$, więc algorytm Dijkstry zużywa tylko $O(n \lg n + m)$ czasu. W praktyce F-kopce są rzadko stosowane — z kilku powodów. Jeden to ten, że poszczególne operacje mogą trwać znacznie dłużej niż średnio, choć ogółem operacje zajmują czas podany wyżej. Drugim powodem jest to, że F-kopce są trochę bardziej skomplikowane, więc czynniki stałe, ukryte w notacji asymptotycznej, nie są tak dobre jak dla kopców binarnych.

Algorytm Bellmana-Forda

Jeśli niektóre wagi krawędzi są ujemne, to algorytm Dijkstry może zwrócić niepoprawne wyniki. Algorytm Bellmana-Forda [7] może operować ujemnymi wagami krawędzi i możemy użyć jego wyników do wykrywania i pomocy w identyfikowaniu cyklu o ujemnej wadze.

Algorytm Bellmana-Forda jest wyjątkowo prosty. Po nadaniu wartości początkowych tablicom *najkrótsza* i *poprz* osłabia on po prostu wszystkie m krawędzi $n - 1$ razy. Procedura znajduje się na następnej stronie, a poniższy rysunek przedstawia działanie algorytmu na małym grafie. Wierzchołkiem źródłowym jest s, wartości tablicy *najkrótsza* są pokazane wewnątrz wierzchołków, a zacieniowane krawędzie wskazują wartości *poprz*: jeśli krawędź (u, v) jest zacieniowana, to $poprz[v] = u$. W tym przykładzie zakładamy, że każde przejście po wszystkich krawędziach osłabia je w ustalonym porządku (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y). Część (a) ukazuje sytuację tuż przed pierwszym przejściem, a części (b) do (e) pokazują sytuację po każdym kolejnym przejściu. Wartości *najkrótsza* i *poprz* w części (e) są wartościami ostatecznymi.

Jakimż to sposobem algorytm tej prostoty może produkować poprawną odpowiedź? Rozważmy najkrótszą ścieżkę ze źródła s do dowolnego wierzchołka v. Przypomnijmy, że jeśli osłabiamy krawędzie kolejno wzdłuż najkrótszej ścieżki z s do v, to $najkrótsza[v]$ i $poprz[v]$ są poprawne. Jeśli więc teraz cykle o ujemnych wagach nie są dozwolone, to zawsze istnieje najkrótsza ścieżka z s do v, która nie zawiera cyklu. Dlaczego? Załóżmy, że najkrótsza ścieżka z s do v zawiera cykl. Ponieważ ten

[7] Oparty na osobnych algorytmach opracowanych przez Richarda Bellmana z 1958 i Lestera Forda z 1962 r.

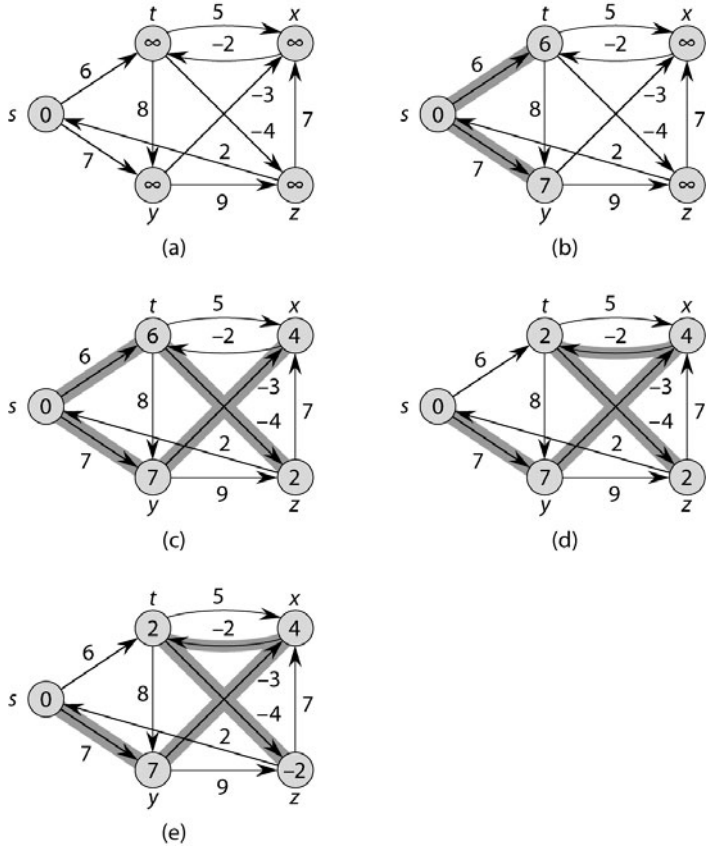
Procedura BELLMAN-FORD(*G*, *s*)

Dane wejściowe:

- *G*: graf skierowany zawierający zbiór *V* *n* wierzchołków i zbiór *E* *m* krawędzi skierowanych z dowolnymi wagami.
- *s*: wierzchołek źródłowy w *V*.

Wynik: taki sam jak w algorytmie DIJKSTRA.

1. Nadaj *najkrótsza*[*v*] wartość ∞ dla każdego wierzchołka *v* z wyjątkiem *s*, nadaj *najkrótsza*[*s*] wartość 0 i ustaw *poprz*[*v*] na NULL dla każdego wierzchołka *v*.
2. Dla *i* = 1 do *n* – 1:
   A. Dla każdej krawędzi (*u*, *v*) w *E*:
      i. Wywołaj OSŁABIAJ(*u*, *v*).



cykl musi mieć nieujemną wagę, możemy go wyciąć i zakończyć na ścieżce z s do v, której waga jest nie większa niż ścieżka zawierająca cykl. Każda acykliczna ścieżka musi zawierać najwyżej $n - 1$ krawędzi, bo gdyby zawierała n krawędzi, musiałaby odwiedzać któryś wierzchołek dwukrotnie, co spowodowałoby cykl. Jeśli zatem istnieje najkrótsza ścieżka z s do v, to jest nią taka, która zawiera najwyżej $n - 1$ krawędzi. Za pierwszym razem podczas osłabiania w kroku 2A wszystkich krawędzi musi nastąpić osłabienie pierwszej krawędzi na tej najkrótszej ścieżce. Kiedy w kroku 2A wszystkie krawędzie są osłabiane po raz drugi, osłabienie musi dotyczyć drugiej krawędzi na najkrótszej ścieżce i tak dalej. Po $n - 1$ razach wszystkie krawędzie na najkrótszej ścieżce zostaną kolejno osłabione, toteż *najkrótsza*[v] i *poprz*[v] są poprawne. Sprytnie wykombinowane!

Załóżmy teraz, że graf zawiera cykl z wagą ujemną i że puściliśmy już w ruch procedurę BELLMAN-FORD. Możesz chodzić w kółko po ujemnym cyklu, otrzymując za każdym okrążeniem coraz niższe wagi. Oznacza to, że w cyklu istnieje co najmniej jedna krawędź (u, v), dla której *najkrótsza*[v] będzie maleć, jeśli ponowisz jej osłabianie, mimo że osłabiono ją już $n - 1$ razy.

Oto jak można znaleźć cykl z ujemną wagą — o ile istnieje — po wykonaniu algorytmu Bellmana-Forda. Przejdźmy krawędzie jeszcze raz. Jeśli znajdziemy krawędź (u, v), dla której *najkrótsza*[u] + *waga*(u, v) < *najkrótsza*[v], to wiemy, że wierzchołek v leży w obrębie cyklu z ujemną wagą albo jest z niego osiągalny. Wierzchołek w cyklu z ujemną wagą możemy znaleźć, śledząc wstecz wartości *poprz* z v i utrzymując ślad wierzchołków, które odwiedziliśmy, nim dotarliśmy do wierzchołka x, który odwiedziliśmy wcześniej. Następnie możemy wywieść wartości *poprz* z x, aż wrócimy do x — wtedy wszystkie wierzchołki pomiędzy, wraz z x, utworzą cykl z ujemną wagą. Procedura ZNAJDŹ-CYKL-Z-WAGĄ-UJEMNĄ na następnej stronie pokazuje, jak ustalić, czy graf ma cykl z wagą ujemną, i jak go wyodrębnić, jeśli istnieje.

Łatwo przeanalizować czas działania algorytmu Bellmana-Forda. Treść pętli w kroku 2 jest powtarzana $n - 1$ razy, a za każdym jej powtórzeniem pętla z kroku 2A iteruje m razy, po razie na krawędź. Łączny czas działania wyniesie zatem $\Theta(n\,m)$. Aby upewnić się, czy istnieje cykl z wagą ujemną, osłabiamy jeszcze raz każdą krawędź, aż osłabianie zmieni wartość *najkrótsza* albo wszystkie krawędzie zostaną osłabione, co zabiera $O(m)$ czasu. Jeśli istnieje cykl z ujemną wagą, to może on zawierać najwyżej n krawędzi, wobec czego czas potrzebny do jego wykrycia wynosi $O(n)$.

Na początku tego rozdziału obiecałem pokazać, jak cykle z ujemnymi wagami mają się do możliwości arbitrażu w handlu walutami. Kursy wymian walut zmieniają się gwałtownie. Wyobraźmy sobie, że w którymś momencie dochodzi do następującej relacji kursów:

- 1 dolar amerykański ma siłę nabywczą 0,7292 euro,
- 1 euro ma siłę nabywczą 105,374 jenów japońskich,
- 1 jen japoński ma siłę nabywczą 0,3931 rubli rosyjskich,
- 1 rubel rosyjski ma siłę nabywczą 0,0341 dolara amerykańskiego.

Procedura ZNAJDŹ-CYKL-Z-WAGĄ-UJEMNĄ(*G*)

Dane wejściowe: G — graf skierowany zawierający zbiór *V n* wierzchołków i zbiór *E m* krawędzi skierowanych z dowolnymi wagami, na którym wykonano już procedurę Bellman-Ford.

Wynik: lista kolejnych wierzchołków w cyklu z ujemną wagą albo pusta lista, jeśli graf nie ma cykli z ujemnymi wagami.

1. Przejdź wszystkie krawędzie w poszukiwaniu takiej krawędzi (*u*, *v*) , że *najkrótsza*[*u*] + *waga*(*u*, *v*) < *najkrótsza*[*v*].
2. Jeśli taka krawędź nie istnieje, zwróć pustą listę.
3. W przeciwnym razie (istnieje krawędź (*u*, *v*), dla której *najkrótsza*[*u*] + *waga*(*u*, *v*) < *najkrótsza*[*v*]) wykonaj, co następuje:
   A. Niech *odwiedzone* będzie nową tablicą z jednym elementem dla każdego wierzchołka. Ustaw wszystkie elementy *odwiedzone* na FAŁSZ.
   B. Ustaw *x* na *v*.
   C. Dopóki *odwiedzone*[*x*] jest równe FAŁSZ, wykonuj, co następuje:
      i. Nadaj *odwiedzone*[*x*] wartość PRAWDA.
      ii. Nadaj *x* wartość *poprz*[*x*].
   D. W tym miejscu wiemy, że *x* jest wierzchołkiem w cyklu z ujemną wagą. Nadaj *v* wartość *poprz*[*x*].
   E. Utwórz listę *cykl* wierzchołków, początkowo zawierającą tylko *x*.
   F. Dopóki *v* nie jest *x*, wykonuj, co następuje:
      i. Wstaw wierzchołek *v* na początek *cyklu*.
      ii. Nadaj *v* wartość *poprz*[*v*].
   G. Zwróć *cykl*.

Możesz więc za 1 dolara kupić 0,7292 euro, za 0,7292 euro kupić 76,8387 jenów (ponieważ 0,7292 × 105,374 = 76,8387, z dokładnością do czterech miejsc dziesiętnych), za 76,8397 jenów kupić 30,2053 rubli (ponieważ 76,8387 × 0,3931 = 30,2053, z dokładnością do czterech miejsc dziesiętnych) i na koniec za 30,2053 rubli kupić 1,03 dolara (ponieważ 30,2053 × 0,0341 = 1,0300, z dokładnością do czterech miejsc dziesiętnych). Gdyby udało Ci się wykonać wszystkie cztery transakcje przed zmianą kursów wymian, uzyskałbyś 3% zysku ze swojej jednodolarowej inwestycji. Zacznij z milionem dolarów, a zarobisz 30 tys. dolarów za nic![8]

[8] Do realizmu zabrakło w tym przykładzie pewnego szczegółu: opłat pobieranych przy każdej takiej transakcji niezależnie od fluktuacji kursu — *przyp. tłum.*

Scenariusz taki jest **możliwością arbitrażu** (ang. *arbitrage opportunity*). Oto jak znaleźć możliwość arbitrażu, poszukując cyklu z ujemną wagą. Załóżmy, że spoglądasz na n walut c_1, c_2, c_3,..., c_n i dysponujesz wszystkimi kursami wymian między parami walut. Przypuśćmy, że za jednostkę waluty c_i możesz kupić r_{ij} jednostki waluty c_j, czyli r_{ij} jest kursem wymiany między walutami c_i i c_j. Tutaj zarówno i, jak j zmienia się od 1 do n. (Przypuszczalnie $r_{ii} = 1$ dla każdej waluty c_i).

Możliwość arbitrażu odpowiadałaby ciągowi k takich walut (c_{j1}, c_{j2},...,c_{jk}), że mnożąc kursy wymian, otrzymasz iloczyn ostro większy niż 1: $r_{j_1,j_2} \cdot r_{j_2,j_3} \cdots r_{j_{k-1},j_k} \cdot r_{j_k,j_1} > 1$.

Zlogarytmujmy to stronami. Nie ma znaczenia, jakiej podstawy użyjemy, postąpmy więc jak przystało na informatyków i użyjmy podstawy 2. Ponieważ logarytm iloczynu jest sumą poszczególnych logarytmów, tzn. $\lg(x \cdot y) = \lg x + \lg y$, otrzymujemy sytuację, w której $\lg r_{j_1,j_2} + \lg r_{j_2,j_3} + ... + \lg r_{j_{k-1},j_k} + \lg r_{j_k,j_1} > 0$.

Zanegowanie obu stron tej nierówności daje

$$(-\lg r_{j_1,j_2}) + (-\lg r_{j_2,j_3}) + ... + (-\lg r_{j_{k-1},j_k}) + (-\lg r_{j_k,j_1}) < 0,$$

co odpowiada cyklowi z wagami krawędziowymi, które są negacjami logarytmów kursów wymian.

Aby znaleźć możliwość arbitrażu, jeśli taka istnieje, budujemy graf skierowany z jednym wierzchołkiem v_i dla każdej waluty c_i. Dla każdej pary walut c_i i c_j tworzymy krawędzie skierowane (v_i, v_j) i (v_j, v_i) z wagami, odpowiednio: lg r_{ij} i –lg r_{ji}. Dodajemy nowy wierzchołek s z krawędzią (s, v_i) o zerowej wadze do każdego z wierzchołków: v_1 do v_n. Na tym grafie, biorąc s jako wierzchołek źródłowy, wykonujemy algorytm Bellmana-Forda i korzystamy z wyniku, aby określić, czy zawiera on cykl z ujemną wagą. Jeśli zawiera, to wierzchołki tego cyklu odpowiadają walutom w możliwości arbitrażu. Łączna liczba krawędzi m wynosi $n + n(n - 1) = n^2$, wobec czego algorytm Bellmana-Forda zużywa $O(n^3)$ czasu plus dodatkowe $O(n^2)$ na sprawdzenie, czy istnieje cykl z ujemną wagą, i dodatkowe $O(n)$ na wyśledzenie go, jeśli istnieje. Chociaż czas $O(n^3)$ wydaje się długi, w praktyce nie jest zły, ponieważ czynniki stałe w czasach działania pętli są małe. Zakodowałem program arbitrażu na swoim 2,4-gigahercowym macbooku pro i wykonałem go dla 182 walut, czyli wszystkich notowanych na świecie. Po załadowaniu kursów wymian (wybrałem w ich charakterze wartości losowe) program działał około 0,02 s.

Algorytm Floyda-Warshalla

Załóżmy teraz, że chcesz znaleźć najkrótszą ścieżkę z każdego wierzchołka do każdego innego. Jest to problem **najkrótszych ścieżek między wszystkimi parami wierzchołków** (ang. *all-pairs shortest-paths*).

Klasycznym przykładem wyszukiwania najkrótszych ścieżek między wszystkimi parami wierzchołków, do którego — co miałem możność stwierdzić — odwołuje się kilku autorów, jest tablica, którą widzisz w atlasie samochodowym, podająca odległości między pewną liczbą miast. Odnajdujesz wiersz z jednym miastem, znajdujesz kolumnę z drugim, a odległość między nimi leży na przecięciu wiersza i kolumny.

Z tym przykładem jest pewien kłopot: *nie dotyczy on wszystkich par*. Gdyby to były wszystkie pary, tablica musiałaby mieć po jednym wierszu i jednej kolumnie dla każdego skrzyżowania, a nie tylko dla każdego miasta. Liczba wierszy i kolumn dla samych Stanów Zjednoczonych poszłaby w miliony. Nie, w sposobie utworzenia tablicy, którą oglądasz w atlasie, kryje się odnalezienie najkrótszych ścieżek z jednym źródłem z każdego miasta i zamknięcie w tablicy podzbioru wyników: najkrótszych ścieżek do innych miast, a nie do wszystkich skrzyżowań.

Jak wyglądałaby aplikacja uzasadniająca odnalezienie najkrótszych ścieżek między wszystkimi parami wierzchołków? Znalezienie **średnicy** sieci — najdłuższej ze wszystkich najkrótszych ścieżek. Załóżmy na przykład, że graf skierowany reprezentuje sieć komunikacyjną, a waga krawędzi określa czas przekazywania komunikatu łączem komunikacyjnym. Wówczas średnica podaje najdłuższy możliwy czas przejścia komunikatu w sieci.

Oczywiście możemy obliczyć najkrótsze ścieżki między wszystkimi parami, obliczając po kolei najkrótsze ścieżki z każdego wierzchołka. Jeśli wszystkie wagi krawędzi są nieujemne, to możemy wykonać algorytm Dijkstry z każdego z n wierzchołków, przy czym każde wywołanie zajmie $O(m \lg n)$ czasu, jeśli użyjemy kopca binarnego, lub $O(n \lg n + m)$ czasu, jeśli posłużymy się kopcem Fibonacciego, co da łącznie czas $O(n\,m \lg n)$ lub $O(n^2 \lg n + n\,m)$. Jeśli graf jest rzadki, takie podejście sprawuje się dobrze. Jeśli jednak graf jest gęsty, tzn. m wynosi prawie n^2, to $O(n\,m \lg n)$ równa się $O(n^3 \lg n)$. Nawet w wypadku grafu gęstego i kopca Fibonacciego $O(n^2 \lg n + m\,n)$ wyniesie $O(n^3)$, a czynnik stały, wynikający z użycia kopca Fibonacciego, może być istotny. Oczywiście, jeśli dopuszczamy w grafie ujemne wagi krawędzi, to nie możemy użyć algorytmu Dijkstry, a wykonanie algorytmu Bellmana-Forda z każdego z n wierzchołków na grafie gęstym daje czas działania $\Theta(n^2 m)$, co się równa $\Theta(n^4)$.

Zamiast tego, korzystając z algorytmu Floyda-Warshalla[9], możemy rozwiązać problem wszystkich par w czasie $\Theta(n^3)$, niezależnie od tego, czy graf jest rzadki, gęsty, czy jakiś pośredni i nawet godząc się na występowanie w nim ujemnych wag krawędzi, lecz bez cykli z ujemnymi wagami, przy czym stały czynnik ukryty w notacji Θ będzie mały. Ponadto algorytm Floyda-Warshalla ilustruje sprytną technikę algorytmiczną zwaną „programowaniem dynamicznym".

Algorytm Floyda-Warshalla korzysta z oczywistej właściwości najkrótszych ścieżek. Załóżmy, że jedziesz z centrum Nowego Jorku do Seattle po najkrótszej drodze i że ta najkrótsza droga z Nowego Jorku do Seattle przechodzi przez Chicago, a następnie — nim dotrzesz do Seattle — przez Spokane. Wówczas część najkrótszej drogi z Nowego Jorku do Seattle wiodąca z Chicago do Spokane sama musi być najkrótszą drogą z Chicago do Spokane. Dlaczego? Ponieważ gdyby istniała krótsza droga z Chicago do Spokane, to skorzystalibyśmy z niej w najkrótszej trasie z Nowego Jorku do Seattle! Jak rzekłem — jest to oczywiste. Stosując tę zasadę do grafów skierowanych:

[9] Od nazwisk Roberta Floyda i Stephena Warshalla.

Jeśli najkrótsza ścieżka, nazwijmy ją p, z wierzchołka u do wierzchołka v wiedzie z wierzchołka u przez wierzchołek x i wierzchołek y do wierzchołka v, to część p zawarta między x i y jest najkrótszą ścieżką między x i y. To znaczy, że *każda najkrótsza podścieżka najkrótszej ścieżki jest najkrótszą ścieżką*.

Algorytm Floyda-Warshalla utrzymuje ślad wag ścieżek i poprzedników wierzchołków w tablicach indeksowanych nie w jednym, a w trzech wymiarach. Możesz myśleć o tablicy jednowymiarowej jako o strukturze, którą oglądaliśmy na początku rozdziału 2. Tablica dwuwymiarowa byłaby jak macierz, taka jak macierz sąsiedztwa w rozdziale 5; potrzebujesz dwóch indeksów (wiersza i kolumny), żeby zlokalizować element. Możesz również uważać tablicę dwuwymiarową za tablicę jednowymiarową, której każdy element jest tablicą jednowymiarową. Tablica trójwymiarowa może być traktowana jako jednowymiarowa tablica tablic dwuwymiarowych; do zidentyfikowania wpisu potrzebujesz indeksu w każdym z trzech wymiarów. Indeksując wielowymiarową tablicę, do oddzielania wymiarów będziemy używać przecinków.

W algorytmie Floyda-Warshalla zakładamy, że wierzchołki są ponumerowane od 1 do n. Numery wierzchołków nabierają znaczenia, ponieważ algorytm Floyda-Warshalla korzysta z następującej definicji:

najkrótsza[u, v, x] jest wagą najkrótszej ścieżki z wierzchołka u do wierzchołka v, której każdy wierzchołek wewnętrzny — wierzchołek na ścieżce inny niż u i v — ma numer od 1 do x.

(Myślmy więc o u, v i x jako o liczbach całkowitych z przedziału od 1 do n, reprezentujących wierzchołki). Ta definicja nie wymaga, aby w skład wierzchołków wewnętrznych wchodziły *wszystkie* z puli x wierzchołków ponumerowanych od 1 do x; wymaga się tylko, aby każdy wierzchołek wewnętrzny — niezależnie od tego, ile ich jest — miał numer nie większy niż x. Ponieważ wszystkie wierzchołki nie przekraczają numeru n, musi zachodzić przypadek, że *najkrótsza*[u, v, n] równa się *ns*(u, v), czyli jest wagą najkrótszej ścieżki z u do v.

Rozważmy dwa wierzchołki u i v, i wybierzmy liczbę x z przedziału od 1 do n. Rozważmy wszystkie ścieżki z u do v, w których wszystkie wierzchołki wewnętrzne są ponumerowane liczbami nie większymi niż x. Ze wszystkich tych ścieżek niech ścieżka p ma najmniejszą wagę. Ścieżka p albo zawiera wierzchołek x, albo nie, przy czym wiemy, że inaczej niż w przypadku u lub v, nie zawiera ona żadnego wierzchołka z numerem większym niż x. Są więc dwie możliwości:

- Pierwsza możliwość: x nie jest wierzchołkiem wewnętrznym na ścieżce p. Wówczas wszystkie wierzchołki wewnętrzne ścieżki p mają numery najwyżej $x - 1$. Co to oznacza? Oznacza to, że waga najkrótszej ścieżki z u do v ze wszystkimi wierzchołkami wewnętrznymi o numerach co najwyżej x jest taka sama jak waga najkrótszej ścieżki z u do v ze wszystkimi wierzchołkami wewnętrznymi z numerami co najwyżej $x - 1$. Innymi słowy, *najkrótsza*[u, v, x] równa się *najkrótsza*[u, v, $x - 1$].

- Druga możliwość: x występuje jako wierzchołek wewnętrzny na ścieżce p. Ponieważ dowolna podścieżka najkrótszej ścieżki jest też najkrótszą ścieżką, część ścieżki p biegnąca od u do x jest najkrótszą ścieżką od u do x. Podobnie część ścieżki p biegnąca od x do v jest najkrótszą ścieżką z x do v. Ponieważ wierzchołek x jest punktem końcowym każdej z tych podścieżek, nie jest wierzchołkiem wewnętrznym żadnej z nich, wobec tego wierzchołki wewnętrzne w każdej z tych podścieżek mają numery nieprzekraczające $x - 1$. Dlatego waga najkrótszej ścieżki z u do v ze wszystkimi wierzchołkami wewnętrznymi o numerach nie większych niż x jest sumą wag dwóch najkrótszych ścieżek: wiodącej z u do x przez wszystkie wierzchołki wewnętrzne ponumerowane co najwyżej do $x - 1$ i takiej, która biegnie z x do v, również ze wszystkimi wierzchołkami wewnętrznymi ponumerowanymi co najwyżej do $x - 1$. Mówiąc inaczej, *najkrótsza*[u, v, x] równa się *najkrótsza*[u, x, $x - 1$] + *najkrótsza*[x, v, $x - 1$].

Ponieważ x jest wierzchołkiem wewnętrznym na najkrótszej ścieżce z u do v albo nim nie jest, możemy podsumować, że *najkrótsza*[u, v, x] jest mniejszą z *najkrótsza*[u, x, $x - 1$] + *najkrótsza*[x, v, $x - 1$] i *najkrótsza*[u, v, $x - 1$].

Najlepszym sposobem reprezentowania grafu w algorytmie Floyda-Warshalla jest odmiana reprezentacji macierzy sąsiedztwa z rozdziału 5. Zamiast ograniczania każdego z elementów macierzy do wartości 0 lub 1 wpis dotyczący krawędzi (u, v) zawiera wagę krawędzi, a wartość ∞ wskazuje nieobecność krawędzi. Ponieważ *najkrótsza*[u, v, 0] oznacza wagę najkrótszej ścieżki z u do v ze wszystkimi wierzchołkami wewnętrznymi mającymi numery co najwyżej 0, taka ścieżka nie ma wierzchołków wewnętrznych. To znaczy, że składa się ona po prostu z jednej krawędzi, więc macierz ta jest dokładnie tym, czego nam trzeba dla *najkrótsza*[u, v, 0].

Mając wartości *najkrótsza*[u, v, 0] (będące wagami krawędzi), algorytm Floyda-Warshalla oblicza wartości *najkrótsza*[u, v, x] najpierw dla wszystkich par wierzchołków u i v z x ustawionym na 1. Następnie algorytm oblicza wartości najkrótsza[u, v, x] dla wszystkich par wierzchołków u i v z x ustawionym na 2. Potem dla x ustawionego na 3 i tak dalej, aż do n.

A jeśli chodzi o utrzymywanie śladów poprzedników? Zdefiniujmy *poprz*[u, v, x] analogicznie do tego, jak zdefiniowaliśmy *najkrótsza*[u, v, x], jako poprzednika wierzchołka v na najkrótszej ścieżce z wierzchołka u, na której wszystkie wierzchołki wewnętrzne są ponumerowane najwyżej do wartości x. Możemy uaktualnić *poprz*[u, v, x] podczas obliczania wartości *najkrótsza*[u, v, x] w sposób następujący. Jeśli *najkrótsza*[u, v, x] jest taka sama jak *najkrótsza*[u, v, $x - 1$], to najkrótsza ścieżka, którą znaleźliśmy z u do v ze wszystkimi wierzchołkami wewnętrznymi ponumerowanymi najwyżej do x, jest tym samym co ścieżka ze wszystkimi wierzchołkami wewnętrznymi ponumerowanymi najwyżej do $x - 1$. Poprzednik wierzchołka v musi być ten sam na obu ścieżkach, możemy więc określić *poprz*[u, v, x] jako to samo co *poprz*[u, v, $x - 1$]. A co w sytuacji, gdy *najkrótsza*[u, v, x] jest mniejsza niż *najkrótsza*[u, v, $x - 1$]? Zdarza się to wówczas, gdy znajdziemy ścieżkę z u do v, która ma wierzchołek x jako wierzchołek wewnętrzny i niższą wagę niż najkrótsza ścieżka z u do v ze wszystkimi wierzchołkami wewnętrznymi ponumerowanymi najwyżej

do $x - 1$. Ponieważ x musi być wierzchołkiem wewnętrznym na tej nowo znalezionej ścieżce, poprzednik v na ścieżce z u musi być taki sam jak poprzednik v na ścieżce z x. W tym przypadku ustawiamy *poprz*[u, v, x] na tę samą wartość co *poprz*[u, v, $x - 1$].

Mamy obecnie wszystkie części potrzebne do zestawienia algorytmu Floyda-Warshalla. Oto ta procedura:

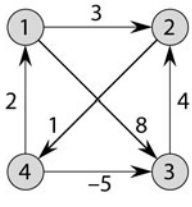
Procedura FLOYD-WARSHALL(G)

Dane wejściowe: G — graf reprezentowany przez macierz sąsiedztwa W o n wierszach i n kolumnach (po jednym wierszu i kolumnie na wierzchołek). Element na skrzyżowaniu wiersza u i kolumny v, oznaczany jak w_{uv}, jest wagą krawędzi (u, v), jeśli krawędź występuje w G, w przeciwnym razie wynosi ∞.

Wynik: dla każdej pary wierzchołków u i v wartość *najkrótsza*[u, v, n] zawiera wagę najkrótszej ścieżki z u do v, a *poprz*[u, v, n] jest poprzednikiem wierzchołka v na najkrótszej ścieżce z u.

1. Niech *najkrótsza* i *poprz* będą nowymi tablicami o wymiarach $n \times n \times (n + 1)$.
2. Dla każdego u i v przedziału od 1 do n:
   A. Nadaj *najkrótsza*[u, v, 0] wartość w_{uv}.
   B. Jeśli (u, v) jest krawędzią w G, to nadaj *poprz*[u, v, 0] wartość u. W przeciwnym razie ustaw *poprz*[u, v, 0] na NULL.
3. Dla $x = 1$ do n:
   A. Dla $u = 1$ do n:
      i. Dla $v = 1$ do n:
         a. Jeśli *najkrótsza*[u, v, $x - 1$] > *najkrótsza*[u, x, $x - 1$] + *najkrótsza*[x, v, $x - 1$], to nadaj *najkrótsza*[u, v, x] wartość *najkrótsza*[u, x, $x - 1$] + *najkrótsza*[x, v, $x - 1$] i ustaw *poprz*[u, v, x] na *poprz*[x, v, $x - 1$].
         b. W przeciwnym razie nadaj *najkrótsza*[u, v, x] wartość *najkrótsza*[u, v, $x - 1$] i ustaw *poprz*[u, v, x] na *poprz*[u, v, $x - 1$].
4. Zwróć tablice *najkrótsza* i *poprz*.

Dla następującego grafu:



macierz sąsiedztwa *W* zawierająca wagi krawędzi ma postać:

$$\begin{pmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & \infty \\ 2 & \infty & -5 & 0 \end{pmatrix},$$

która również podaje wartości *najkrótsza*[*u*, *v*, 0][10] (wagi ścieżek mających co najmniej jedną krawędź). Na przykład *najkrótsza*[2, 4, 0] równa się 1, ponieważ możemy przejść z wierzchołka 2 do wierzchołka 4 bezpośrednio, bez wierzchołków wewnętrznych, biorąc krawędź (2, 4) z wagą 1. Podobnie *najkrótsza*[4, 3, 0] wynosi –5. Oto macierz wartości *poprz*[*u*, *v*, 0]:

$$\begin{pmatrix} \text{NULL} & 1 & 1 & \text{NULL} \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & \text{NULL} \\ 4 & \text{NULL} & 4 & \text{NULL} \end{pmatrix}.$$

Na przykład *poprz*[2, 4, 0] wynosi 2, ponieważ poprzednikiem wierzchołka 4 jest wierzchołek 2, jeśli skorzysta się z krawędzi (2, 4) z wagą 1, a *poprz*[2, 3, 0] wynosi NULL, ponieważ nie ma krawędzi (2, 3).

Po wykonaniu pętli z kroku 3 dla $x = 1$ (w celu zbadania ścieżek, które mogą zawierać wierzchołek 1 jako wewnętrzny) wartości *najkrótsza*[*u*, *v*, 1] i *poprz*[*u*, *v*, 1] wynoszą:

$$\begin{pmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & \infty \\ 2 & \infty & -5 & 0 \end{pmatrix} \text{ i } \begin{pmatrix} \text{NULL} & 1 & 1 & \text{NULL} \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & \text{NULL} \\ 4 & 1 & 4 & \text{NULL} \end{pmatrix}.$$

Po tym jak pętla zadziała dla $x = 2$, wartości *najkrótsza*[*u*, *v*, 2] i *poprz*[*u*, *v*, 2] przyjmą postać:

$$\begin{pmatrix} 0 & 3 & 8 & 4 \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & 5 \\ 2 & 5 & -5 & 0 \end{pmatrix} \text{ i } \begin{pmatrix} \text{NULL} & 1 & 1 & 2 \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & 2 \\ 4 & 1 & 4 & \text{NULL} \end{pmatrix}.$$

[10] Ponieważ tablica trójwymiarowa jest jednowymiarową tablicą tablic dwuwymiarowych, dla ustalonej wartości *x* możemy uważać *najkrótsza*[*u*, *v*, *x*] za tablicę dwuwymiarową.

Po wykonaniu dla $x = 3$:

$$\begin{pmatrix} 0 & 3 & 8 & 4 \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & 5 \\ 2 & -1 & -5 & 0 \end{pmatrix} \text{ i } \begin{pmatrix} \text{NULL} & 1 & 1 & 2 \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & 2 \\ 4 & 3 & 4 & \text{NULL} \end{pmatrix}.$$

Nasze finalne wartości *najkrótsza*[*u*, *v*, 4] i *poprz*[*u*, *v*, 4] przedstawiają się po wykonaniu pętli dla $x = 4$ następująco:

$$\begin{pmatrix} 0 & 3 & -1 & 4 \\ 3 & 0 & -4 & 1 \\ 7 & 4 & 0 & 5 \\ 2 & -1 & -5 & 0 \end{pmatrix} \text{ i } \begin{pmatrix} \text{NULL} & 1 & 1 & 2 \\ 4 & \text{NULL} & 4 & 2 \\ 4 & 3 & \text{NULL} & 2 \\ 4 & 3 & 4 & \text{NULL} \end{pmatrix}.$$

Możemy na przykład zauważyć, że najkrótsza ścieżka z wierzchołka 1 do wierzchołka 3 ma wagę –1. Ta ścieżka biegnie z wierzchołka 1 do wierzchołka 2, następnie do wierzchołków 4 i 3, co można zaobserwować, idąc wstecz: *poprz*[1, 3, 4] wynosi 4, *poprz*[1, 4, 4] jest 2, a *poprz*[1, 2, 4] równa się 1.

Zaznaczyłem, że algorytm Floyda-Warshalla działa w czasie $\Theta(n^3)$, o czym można łatwo się przekonać. Mamy pętle zanurzone do stopnia trzy, a każda iteruje n razy. W każdej iteracji pętli z kroku 3 pętla z kroku 3A iteruje n razy. Podobnie w każdej iteracji pętli z kroku 3A pętla z kroku 3Ai iteruje n razy. Ponieważ zewnętrzna pętla z kroku 3 również iteruje n razy, pętla zanurzona najgłębiej (krok 3Ai) iteruje w sumie n^3 razy. Każda iteracja najbardziej wewnętrznej pętli zajmuje stały czas, więc algorytm zużywa $\Theta(n^3)$ czasu.

Wygląda na to, że algorytm ten zużywa również $\Theta(n^3)$ pamięci. W końcu tworzy on dwie tablice o wymiarach $n \times n \times (n + 1)$. Ponieważ każdy element tablicy zajmuje stałą ilość miejsca, tablice te zajmują $\Theta(n^3)$ obszaru pamięci. Okazuje się jednak, że poradzimy sobie w przestrzeni $\Theta(n^2)$. W jaki sposób? Po prostu tworzymy tablice *najkrótsza* i *poprz* jako tablice o wymiarach $n \times n$ i całkowicie zapominamy o trzecim indeksie do *najkrótsza* i *poprz*. Choć w krokach 3Aia i 3Aib są aktualizowane te same wartości *najkrótsza*[*u*, *v*] i *poprz*[*u*, *v*], okazuje się, że tablice te mają na końcu wartości poprawne!

Wspomniałem wcześniej, że algorytm Floyda-Warshalla ilustruje technikę zwaną **programowaniem dynamicznym** (ang. *dynamic programming*). Technika ta znajduje zastosowanie tylko wówczas, gdy:

1. Próbujemy znaleźć optymalne rozwiązanie problemu.
2. Możemy wydzielić w problemie egzemplarz lub egzemplarze reprezentujące mniejsze podproblemy.
3. Używamy rozwiązań podproblemu lub podproblemów do rozwiązania problemu oryginalnego.

4. Jeśli używamy rozwiązania podproblemu w ramach rozwiązania optymalnego problemu oryginalnego, to używane rozwiązanie podproblemu musi być optymalne dla podproblemu.

Możemy zebrać te warunki pod wspólnym parasolem o nazwie **podstruktura optymalna** i, wyrażając się zwięźlej, powiedzieć, że optymalne rozwiązanie problemu zawiera w sobie optymalne rozwiązania podproblemów. W programowaniu dynamicznym operujemy pewnym pojęciem „rozmiaru" podproblemu i często rozwiązujemy podproblemy w rosnącej kolejności rozmiaru, tzn. najpierw rozwiązujemy podproblemy najmniejsze i dopiero potem, kiedy już mamy optymalne rozwiązania podproblemów mniejszych, próbujemy optymalnie rozwiązać większe podproblemy, korzystając z optymalnych rozwiązań mniejszych podproblemów.

Ten opis programowania dynamicznego brzmi cokolwiek abstrakcyjnie, zobaczmy więc, jaki użytek z niego ma algorytm Floyda-Warshalla. Podproblem określamy następująco:

> Oblicz *najkrótsza*[*u*, *v*, *x*], czyli wagę najkrótszej ścieżki z wierzchołka *u* do wierzchołka *v*, której każdy węzeł wewnętrzny nosi numer od 1 do *x*.

Tutaj „rozmiarem" podproblemu jest wierzchołek o największym numerze, który dopuszczamy jako wierzchołek wewnętrzny w najkrótszej ścieżce — innymi słowy: wartość *x*. Optymalna podstruktura wchodzi do gry, ponieważ mamy następującą własność:

> Rozważmy najkrótszą ścieżkę *p* z wierzchołka *u* do wierzchołka *v* i niech *x* będzie wierzchołkiem wewnętrznym o najwyższym numerze na tej ścieżce. Wtedy część *p*, która biegnie od *u* do *x*, jest najkrótszą ścieżką z *u* do *x*, ze wszystkimi wierzchołkami wewnętrznymi ponumerowanymi od 1 do $x - 1$, a część *p* biegnąca od *x* do *v* jest najkrótszą ścieżką z *x* do *v*, ze wszystkimi wierzchołkami wewnętrznymi ponumerowanymi od 1 do $x - 1$.

Problem obliczenia *najkrótsza*[*u*, *v*, *x*] rozwiązujemy, obliczając najpierw *najkrótsza*[*u*, *v*, $x - 1$], n*ajkrótsza*[*u*, *x*, $x - 1$] i *najkrótsza*[*x*, *v*, $x - 1$], po czym używając mniejszej z *najkrótsza*[*u*, *v*, $x - 1$] i *najkrótsza*[*u*, *x*, $x - 1$] + *najkrótsza*[*x*, *v*, $x - 1$]. Ponieważ mamy obliczone wszystkie wartości *najkrótsza*, w których trzeci indeks równa się $x - 1$, zanim zaczniemy obliczać dowolną z wartości *najkrótsza*, w której trzeci indeks równa się *x*, mamy wszystkie informacje potrzebne do tego, żeby obliczyć *najkrótsza*[*u*, *v*, *x*].

Typową praktyką w programowaniu dynamicznym jest zapamiętywanie optymalnych rozwiązań podproblemów (*najkrótsza*[*u*, *v*, $x - 1$], *najkrótsza*[*u*, *x*, $x - 1$] i *najkrótsza*[*x*, *v*, $x - 1$]) w tablicy, a następnie przeglądanie ich podczas obliczania optymalnego rozwiązania problemu oryginalnego (*najkrótsza*[*u*, *v*, *x*]). Podejście takie nazywamy „wstępującym" (ang. „*bottom-up*"), ponieważ prace przebiegają od mniejszych do większych podproblemów. Innym podejściem jest rozwiązywanie podproblemów „zstępująco" (ang. „*top-down*"), kiedy rozpoczynamy od większych podproblemów i przechodzimy do mniejszych, również zapamiętując wynik każdego podproblemu w tablicy.

Programowanie dynamiczne stosuje się do rozległego obszaru problemów optymalizacyjnych, przy czym tylko w niektórych mamy do czynienia z grafami. Spotkamy się z nim ponownie w rozdziale 7, gdy będziemy znajdować najdłuższy podciąg w dwóch napisach.

Co czytać dalej

W rozdziale 24 CLRS [CLRS09] są przedstawione algorytmy Dijkstry i Bellmana-Forda. W rozdziale 25 CLRS omówiono algorytmy wyznaczania najkrótszych ścieżek między parami wierzchołków, w tym algorytm Floyda-Warshalla, działający w czasie $\Theta(n^3 \lg n)$ algorytm najkrótszych ścieżek między wszystkimi parami oparty na mnożeniu macierzy i sprytny algorytm Donalda Johnsona, zaprojektowany do znajdowania najkrótszych ścieżek między wszystkimi parami wierzchołków w grafach rzadkich w czasie $O(n^2 \lg n + n\,m)$.

Jeśli wagi krawędzi są małymi liczbami nieujemnymi, nie większymi niż znana wartość C, to nieco bardziej skomplikowana implementacja kolejki priorytetowej w algorytmie Dijkstry daje lepsze asymptotyczne czasy działania niż kopiec Fibonacciego. Na przykład Ahuja, Mehlhorn, Orlin i Tarjan [AMOT90] włączają do algorytmu Dijkstry „kopiec redystrybucyjny", uzyskując czas działania $O(m + n\sqrt{\lg C})$.

7 Algorytmy napisowe

Napis (łańcuch, ang. *string*) jest ciągiem znaków z pewnego przyjętego zbioru znaków. Na przykład ta książka składa się ze znaków, wśród których można wyróżnić zbiór liter, cyfr, znaków interpunkcyjnych i symboli matematycznych, które — choć jest ich dość dużo — tworzą skończony zbiór znaków. Biolodzy kodują nici DNA w postaci napisów złożonych tylko z czterech znaków: `A`, `C`, `G` i `T`, które reprezentują podstawowe molekuły: adeninę, cytozynę, guaninę i tyminę.

W związku z napisami możemy formułować wszelkiego rodzaju pytania, lecz w tym rozdziale skoncentrujemy się na algorytmach dotyczących trzech problemów, w których napisy są danymi wejściowymi:

1. Znajdź najdłuższy wspólny podciąg dwóch napisów.
2. Mając zbiór operacji, które mogą przekształcać jeden napis w inny, i znając koszt każdej operacji, znajdź najtańszy sposób przekształcenia jednego napisu w drugi.
3. Znajdź wszystkie wystąpienia napisu wzorcowego w tekście.

Dwa pierwsze problemy mają zastosowanie w biologii obliczeniowej. Im dłuższy wspólny podciąg w dwóch niciach DNA potrafimy znaleźć, tym większe jest między nimi podobieństwo. Jednym ze sposobów układania nici DNA jest przekształcanie jednej nici w drugą; im niższy koszt transformacji, tym bardziej nici są do siebie podobne. Ostatni problem, znajdowanie wystąpień wzorca w tekście, jest również znany pod nazwą **dopasowywania napisów** (ang. *string matching*). Pojawia się we wszelkiego rodzaju programach, kiedy tylko użyjesz polecenia *Znajdź*. Występuje również w biologii obliczeniowej, ponieważ możemy poszukiwać jednej nici DNA w innej.

Najdłuższy wspólny podciąg

Zacznijmy od tego, co nazywamy „ciągiem" i „podciągiem". **Ciąg** (ang. *sequence*) jest wykazem elementów, w którym kolejność elementów jest istotna. Dany element może występować w ciągu wiele razy. Konkretne ciągi, na których będziemy pracować w tym rozdziale, są łańcuchami znaków, będziemy więc używać zamiast „ciąg" nazwy „napis". Przyjmiemy też na podobnej zasadzie, że elementy tworzące ciąg są znakami. Na przykład napis `GACA` zawiera ten sam znak (`A`) kilka razy i różni się od napisu `CAAG`, który składa się z tych samych znaków, lecz ustawionych w innej kolejności. **Podciągiem** (ang. *subsequence*) Z napisu X jest X, być może z usuniętymi elementami. Jeśli na przykład X jest napisem `GAC`, to ma osiem podciągów: `GAC` (nie usunięto żadnego znaku), `GA` (usunięty `C`), `GC` (usunięty `A`), `AC` (usunięty `G`), `G` (usunięte `A` i `C`), `A` (usunięte `G` i `C`), `C` (usunięte `G` i `A`) oraz napis pusty (usunięte wszystkie znaki).

Jeśli *X* i *Y* są napisami, to *Z* jest **wspólnym podciągiem** (ang. *common subsequence*) *X* i *Y*, o ile jest podciągiem każdego z nich. Na przykład, jeśli *X* jest napisem `CATCGA`, a *Y* jest napisem `CTACCGTCA`, to `CCA` jest wspólnym podciągiem *X* i *Y* składającym się z trzech znaków. Nie jest on jednak **najdłuższym wspólnym podciągiem** (ang. *longest common subsequence*, **LCS**), ponieważ najdłuższy wspólny podciąg, `CTCA`, ma cztery znaki. Rzeczywiście, `CTCA` jest najdłuższym wspólnym podciągiem, lecz nie jedynym, ponieważ `TCGA` jest innym wspólnym podciągiem z czterema znakami. Pojęcia podciągu i podnapisu różnią się od siebie: **podnapis** (ang. *substring*) jest podciągiem napisu utworzonym przez znaki wyjęte z ciągłego fragmentu napisu. W napisie `CATCGA` podciąg `ATCG` jest podnapisem, lecz podciąg `CTCA` — nie.

Naszym celem jest odnalezienie w dwóch ciągach *X* i *Y* najdłuższego, występującego w nich wspólnego podciągu *Z*. Do rozwiązania tego problemu skorzystamy z techniki programowania dynamicznego, z którą zetknęliśmy się w rozdziale 6.

Najdłuższy wspólny podciąg możesz znaleźć bez odwoływania się do programowania dynamicznego, nie polecam jednak tego. Możesz brać każdy podciąg *X* i sprawdzać, czy jest podciągiem *Y*, poczynając od najdłuższego podciągu *X* i idąc ku najmniejszym podciągom *X*, porównując każdy z *Y* i zatrzymując się, gdy znajdziesz podciąg zarówno w *X*, jak i w *Y*. (Masz pewność, że w końcu jakiś znajdziesz, gdyż napis pusty jest podciągiem wszystkich napisów). Jeśli *X* ma długość *m*, to istnieje 2^m podciągów, więc jeśli nawet pominiemy czas sprawdzania każdego podciągu z *Y*, w przypadku najgorszym czas znalezienia LCS zależałby co najmniej wykładniczo od długości *X*.

Przypomnijmy za rozdziałem 6, że aby zastosować programowanie dynamiczne, musimy dysponować podstrukturą optymalną: optymalne rozwiązanie problemu zawiera optymalne rozwiązania jego podproblemów. Aby znaleźć LCS dwu napisów za pomocą programowania dynamicznego, musimy najpierw zdecydować, czym jest podproblem. Sprawdzają się tu przedrostki. Jeśli *X* jest napisem $x_1x_2x_3\cdots x_m$, to **i-tym przedrostkiem** (prefiksem, ang. *ith prefix*) *X* jest napis $x_1x_2x_3\cdots x_i$ i zapisujemy go jako X_i. Wymagamy tutaj, aby *i* należało do przedziału od 0 do *m*, przy czym X_0 jest napisem pustym. Na przykład, jeśli *X* jest `CATCGA`, to X_4 jest `CATC`.

Możemy zauważyć, że LCS[1] dwóch napisów zawiera w sobie LCS przedrostków tych dwu napisów. Rozważmy dwa napisy: $X = x_1x_2x_3\cdots x_m$ i $Y = y_1y_2y_3\cdots y_n$. Mają one pewien LCS, powiedzmy *Z*, gdzie $Z = z_1z_2z_3\cdots z_k$ dla pewnej długości *k*, która może się mieścić gdziekolwiek między zerem a mniejszą z wartości *m* lub *n*. Co możemy wydedukować odnośnie do *Z*? Spójrzmy na ostatnie znaki w *X* i *Y*: x_m i y_n. Są one równe albo nie.

Jeśli są równe, to ostatni znak z_k napisu *Z* musi być taki sam jak tamte. Co wiemy o reszcie *Z*, którą stanowi $Z_{k-1} = z_1z_2z_3\cdots z_{k-1}$? Wiemy, że Z_{k-1} musi być LCS tego, co zostaje z *X* i *Y*, mianowicie $X_{m-1} = x_1x_2x_3\cdots x_{m-1}$ i $Y_{n-1} = y_1y_2y_3\cdots y_{n-1}$. Z naszego

[1] Przypominamy, że LCS oznacza najdłuższy wspólny podciąg; nie chcemy go nazywać NWP — *przyp. tłum.*

poprzedniego przykładu, w którym X = `CATCGA` i Y = `GTACCGTCA`, a LCS jest Z = `CTCA`, ostatni znak w X i w Y, czyli `A`, jest ostatnim znakiem Z i widzimy, że Z_3 = `CTC` musi być LCS przedrostków X_5 = `CATCG` i Y_8 = `GTACCGTC`.

- Jeśli nie są równe, to z_k może być taki sam jak ostatni znak x_m w X albo ostatni znak y_n w Y, ale nie może równać się obu. Względnie może być różny od ostatniego znaku i w X, i w Y. Jeżeli z_k nie jest taki sam jak x_m, to pomijamy ostatni znak X: Z musi być LCS X_{m-1} i Y. Podobnie, jeżeli z_k nie jest równy y_n, to pomijamy ostatni znak Y: Z musi być LCS X i Y_{n-1}. Kontynuując powyższy przykład, niech X = `CATCG`, Y = `GTACCGTC` i Z = `CTC`. Teraz z_3 jest równy y_8 (`C`), lecz nie x_5 (`G`), więc Z jest LCS przedrostka X_4 = `CATC` i Y.

Ten problem ma zatem podstrukturę optymalną: LCS dwu napisów zawiera w sobie LCS przedrostków tych dwu napisów.

Co dalej? Musimy rozwiązać jeden lub dwa podproblemy, co zależy od tego, czy ostatnie znaki X i Y są takie same. Jeśli są, to rozwiązujemy tylko jeden podproblem — znajdujemy LCS (najdłuższy wspólny podciąg) X_{m-1} i Y_{n-1}, po czym dodajemy ten ostatni znak, aby otrzymać LCS X i Y. Jeśli ostatnie znaki X i Y nie są takie same, to musimy rozwiązać dwa podproblemy: znaleźć LCS X_{m-1} i Y oraz znaleźć LCS X i Y_{n-1}, po czym użyć dłuższego z tych dwóch najdłuższych wspólnych podciągów jako LCS X i Y. Jeśli te dwa najdłuższe podciągi mają taką samą długość, to stosujemy albo jeden, albo drugi z nich — wszystko jedno który.

Do problemu znalezienia LCS X i Y podejdziemy w dwu krokach. Najpierw znajdziemy długość LCS X i Y, a także długości najdłuższych wspólnych podciągów we wszystkich przedrostkach X i Y. Może Cię dziwić, że potrafimy znaleźć długość LCS, nie znając samego LCS. Po obliczeniu długości LCS „rozłożymy na części" metodę, za pomocą której obliczyliśmy te długości, aby znaleźć faktyczny LCS X i Y.

Aby sprawy nieco uściślić, oznaczmy długość LCS przedrostków X_i i Y_j jako $l[i, j]$. Długość LCS X i Y jest dana przez $l[m, n]$. Możemy zacząć operowanie indeksami i oraz j od 0, ponieważ gdy któryś z przedrostków ma długość 0, to znamy jego LCS — jest nim napis pusty. Innymi słowy, $l[0, j]$ i $l[i, 0]$ równa się 0 dla wszystkich wartości i oraz j. Kiedy zarówno i, jak j są dodatnie, wtedy określamy $l[i, j]$, zaglądając pod mniejsze wartości i oraz (lub) j:

- Jeśli i oraz j są dodatnie i x_i jest równe y_j, to $l[i, j]$ równa się $l[i - 1, j - 1] + 1$.
- Jeśli i oraz j są dodatnie i x_i jest różne od y_j, to $l[i, j]$ równa się większej z wartości $l[i, j - 1]$ i $l[i - 1, j]$.

Pomyślmy o wartościach $l[i, j]$ jako pamiętanych w tabeli. Musimy obliczyć te wartości w rosnącym porządku indeksów i oraz j. Oto tabela $l[i, j]$ dla naszych przykładowych napisów (za chwilę dowiemy się, co oznaczają zacieniowane fragmenty):

		j	0	1	2	3	4	5	6	7	8	9
		y_j		G	T	A	C	C	G	T	C	A
i	x_i	$l[i, j]$										
0			0	0	0	0	0	0	0	0	0	0
1	C		0	0	0	0	1	1	1	1	1	1
2	A		0	0	0	1	1	1	1	1	1	2
3	T		0	0	1	1	1	1	1	2	2	2
4	C		0	0	1	1	2	2	2	2	3	3
5	G		0	1	1	1	2	2	3	3	3	3
6	A		0	1	1	2	2	2	3	3	3	4

Na przykład $l[5, 8]$ wynosi 3, co oznacza, że LCS X_5 = `CATCG` i Y_8 = `GTACCGTC` ma długość 3, jak widzieliśmy na poprzedniej stronie.

Aby obliczyć tabelę wartości we wzrastającym porządku indeksów, nim obliczymy konkretny wpis $l[i, j]$, gdzie zarówno i, jak j są dodatnie, musimy obliczyć elementy: $l[i, j - 1]$ (występujący tuż na lewo od $l[i, j]$), $l[i - 1, j]$ (występujący tuż powyżej $l[i, j]$) i $l[i - 1, j - 1]$ (powyżej i na lewo od $l[i, j]$)[2]. Łatwo obliczyć tabelę elementów tym sposobem: możemy obliczać je albo wierszami, w każdym wierszu od lewej do prawej, albo kolumnami, w każdej kolumnie od góry do dołu.

Następująca procedura traktuje tabelę jak dwuwymiarową tablicę $l[0..m, 0..n]$. Po zapełnieniu zerami skrajnej lewej kolumny i górnego wiersza, wypełnia resztę tablicy wiersz po wierszu.

Procedura OBLICZ-TABELĘ-LCS(X, Y)

Dane wejściowe: X i Y – dwa napisy długości, odpowiednio, m i n.

Wynik: tablica $l[0..m, 0..n]$. Wartość $l[m, n]$ jest długością najdłuższego wspólnego podciągu X i Y.

1. Niech $l[0..m, 0..n]$ będzie nową tablicą.
2. Dla $i = 0$ do m:
   A. Nadaj $l[i, 0]$ wartość 0.
3. Dla $j = 0$ do n:
   A. Nadaj $l[0, j]$ wartość 0.
4. Dla $i = 1$ do m:
   A. Dla $j = 1$ do n:
      i. Jeśli x_i jest takie samo jak y_j, to nadaj $l[i, j]$ wartość $l[i - 1, j - 1] + 1$.
      ii. W przeciwnym razie (x_i różni się od y_j) nadaj $l[i, j]$ wartość większą z $l[i, j - 1]$ i $l[i - 1, j]$. Jeśli $l[i, j - 1]$ równa się $l[i - 1, j]$, to nie ma znaczenia, którą wybierzesz.
5. Zwróć tablicę l.

[2] Nawet wspominanie o $l[i - 1, j - 1]$ jest zbyteczne, ponieważ musimy obliczyć go przed obliczeniem zarówno $l[i, j - 1]$, jak $l[i - 1, j]$.

Ponieważ wypełnienie każdego elementu tablicy zajmuje stałą ilość czasu, a tablica zawiera $(m+1)\cdot(n+1)$ elementów, czas działania procedury OBLICZ-TABELĘ-LCS wynosi $\Theta(m\,n)$.

Dobrą nowiną jest to, że po obliczeniu tabeli $l[i, j]$ jej prawy dolny wpis $l[m, n]$ daje nam długość LCS X i Y. Zła wieść to ta, że z żadnego pojedynczego elementu w tabeli nie dowiemy się, jakie znaki są naprawdę w LCS. Możemy zastosować tę tabelę wraz z napisami X i Y do zbudowania LCS, zużywając dodatkowo $O(m + n)$ czasu. To, jak otrzymaliśmy wartość $l[i, j]$, określamy za pomocą rozłożenia na czynniki pierwsze tego obliczenia, opierając się na $l[i, j]$ i wartościach, od których ona zależy: x_i, y_j, $l[i - 1, j - 1]$, $l[i, j - 1]$ i $l[i - 1, j]$.

Lubię zapisywać tę procedurę rekurencyjnie, kiedy to zestawiamy LCS od tyłu do przodu. Procedura zagnieżdża się w sobie i kiedy znajdzie takie same znaki w X i Y, dodaje znak na końcu budowanego LCS. Początkowe wywołanie przybiera postać ZESTAWIAJ-LCS(X, Y, l, m, n).

Procedura ZESTAWIAJ-LCS(X, Y, l, i, j)

Dane wejściowe:

- X i Y: dwa napisy.
- l: tablica wypełniona przez procedurę OBLICZ-TABELĘ-LCS.
- i oraz j: indeksy do, odpowiednio, X i Y, a także do l.

Wynik: LCS (najdłuższy wspólny podciąg) X_i i Y_j.

1. Jeśli $l[i, j]$ równa się 0, zwróć napis pusty.
2. W przeciwnym razie (ponieważ $l[i, j]$ jest dodatnie, więc również i oraz j są dodatnie), jeżeli x_i jest taki sam jak y_j, zwróć napis utworzony najpierw przez rekurencyjne wywołanie ZESTAWIAJ-LCS(X, Y, l, $i - 1$, $j - 1$), a następnie dodanie na jego końcu x_i (lub y_j).
3. W przeciwnym razie (x_i różni się od y_j), jeśli $l[i, j - 1]$ jest większe niż $l[i - 1, j]$, zwróć napis przekazany przez rekurencyjne wywołanie ZESTAWIAJ-LCS(X, Y, l, i, $j - 1$).
4. W przeciwnym razie (x_i różni się od y_j i $l[i, j - 1]$ jest mniejsze lub równe $l[i - 1, j]$), zwróć napis przekazany przez rekurencyjne wywołanie ZESTAWIAJ-LCS(X, Y, l, $i - 1$, j).

W tabeli zacieniowane elementy $l[i, j]$ są tymi, które w trakcie rekurencji są odwiedzane w początkowym wywołaniu ZESTAWIAJ-LCS(X, Y, l, 6, 9), a zacieniowane znaki x_i są tymi, które zostały dołączone do konstruowanego LCS. Aby zrozumieć, jak działa ZESTAWIAJ-LCS, zacznijmy od $i = 6$ i $j = 9$. Zauważamy wtedy,

że zarówno x_6, jak i y_9, to litera A. Wobec tego A będzie ostatnim znakiem w LCS X_6 i Y_9, a my przechodzimy z rekursją do kroku 2. W tym wywołaniu rekurencyjnym $i = 5$, a $j = 8$. Tym razem widzimy, że znaki x_5 i x_8 są różne, oraz zauważamy, że $l[5, 7]$ równa się $l[4, 8]$, przechodzimy więc z rekursją do kroku 4. Teraz wywołanie rekurencyjne ma $i = 4$ oraz $j = 8$. I tak dalej. Jeśli przeczytasz zacieniowane znaki od góry do dołu, otrzymasz napis CTCA będący LCS. Gdybyśmy rozstrzygnęli wybór między $l[i, j – 1]$ i $l[i – 1, j]$, preferując pójście w lewo (krok 3) zamiast pójścia w górę (krok 4), to wyprodukowany LCS przybrałby postać TCGA.

Jakim sposobem procedura ZESTAWIAJ-LCS zajmuje $O(m + n)$ czasu? Zauważmy, że w każdym rekurencyjnym wywołaniu zmniejsza się albo indeks i, albo j, albo oba. Po $m + n$ wywołaniach rekurencyjnych mamy zatem zagwarantowane, że jeden lub drugi z tych indeksów osiągnie 0 i rekursja osiągnie najniższy poziom w kroku 1.

Zamiana napisu na inny

Zobaczmy teraz, jak przekształcić jakiś napis X w inny napis Y. Zaczniemy od X i będziemy zamieniać go znak po znaku na Y. Załóżymy, że X i Y mają, odpowiednio, m i n znaków. Jak poprzednio, będziemy zapisywać i-ty znak w każdym napisie, używając małej litery nazwy napisu z dolnym wskaźnikiem i, tak więc i-tym znakiem napisu X będzie x_i, a j-tym znakiem Y — symbol y_j.

Aby zamienić X na Y, zbudujemy napis, który nazwiemy Z, taki, że gdy skończymy, Z i Y będą jednakowe. Operujemy indeksem i w odniesieniu do X i indeksem j w odniesieniu do Z. Wolno nam wykonywać ciąg specjalnych operacji transformujących, które mogą zmieniać Z i wspomniane indeksy. Zaczynamy od i oraz j równych 1 i w trakcie postępowania musimy sprawdzić każdy znak w X, co oznacza, że zatrzymamy się, gdy tylko i osiągnie wartość $m + 1$.

Oto operacje, które bierzemy pod uwagę:

- **Kopiuj** (ang. *copy*) znak x_i z X do Z, umieszczając x_i na pozycji z_j i zwiększając oba indeksy: i oraz j.
- **Zastąp** (ang. *replace*) x_i X innym znakiem a, podstawiając na miejsce z_i znak a i zwiększając oba indeksy: i oraz j.
- **Usuń** (ang. *delete*) znak x_i z X, zwiększając i, lecz nie ruszając j.
- **Wstaw** (ang. *insert*) znak a do Z, ustawiając z_j na a i zwiększając j, zostawiając jednocześnie i bez zmian.

Są możliwe inne operacje, jak zamiana dwóch sąsiednich znaków lub usuwanie znaków od x_i do x_m za jednym razem, lecz tutaj będziemy rozpatrywać tylko operacje *kopiuj*, *zastąp*, *usuń* i *wstaw*.

Jako przykład weźmy pod uwagę ciąg operacji, które przekształcają napis ATGATCGGCAT w napis CAATGTGAATC, gdzie zacieniowanymi znakami są x_i i z_j po każdej operacji:

Operacja	*X*	*Z*
Napisy początkowe	ATGATCGGCAT	
usuń A	ATGATCGGCAT	
zastąp T *przez* C	ATGATCGGCAT	C
zastąp G *przez* A	ATGATCGGCAT	CA
kopiuj A	ATGATCGGCAT	CAA
kopiuj T	ATGATCGGCAT	CAAT
zastąp C *przez* G	ATGATCGGCAT	CAATG
zastąp G *przez* T	ATGATCGGCAT	CAATGT
kopiuj G	ATGATCGGCAT	CAATGTG
zastąp C *przez* A	ATGATCGGCAT	CAATGTGA
kopiuj A	ATGATCGGCAT	CAATGTGAA
kopiuj T	ATGATCGGCAT	CAATGTGAAT
wstaw C	ATGATCGGCAT	CAATGTGAATC

Inne ciągi operacji też by zadziałały. Moglibyśmy, na przykład, po prostu usunąć każdy kolejny znak z *X*, a potem wstawić każdy znak z *Y* do *Z*.

Każda operacja transformacji ma swój koszt, który jest stałą zależną tylko od typu operacji, a nie od znaku, którego ona dotyczy. Naszym celem jest znalezienie takiego ciągu operacji, który przekształca *X* w *Y* z minimalnym łącznym kosztem. Oznaczmy koszt operacji *kopiuj* przez c_C, koszt operacji *zastąp* przez c_R, koszt *usuń* przez c_D i koszt *wstaw* jako c_I. Dla ciągu operacji w powyższym przykładzie sumaryczny koszt wyniósłby $5c_C + 5c_R + c_D + c_I$. Powinniśmy założyć, że każdy z kosztów c_C i c_R jest mniejszy niż $c_D + c_I$, gdyż w przeciwnym razie, zamiast płacić c_C, żeby skopiować znak, lub płacić c_R, żeby go zastąpić, zapłacilibyśmy po prostu $c_D + c_I$ za usunięcie znaku i wstawienie tego samego (zamiast kopiowania) lub innego (zamiast zastępowania).

Po co Ci zamienianie jednego napisu na drugi? Biologia obliczeniowa stanowi jeden z przykładów zastosowania tego zabiegu. Biolodzy obliczeniowi często zestawiają ze sobą dwie sekwencje DNA, aby zmierzyć stopień ich podobieństwa. W jednym ze sposobów zestawiania dwóch sekwencji *X* i *Y* układamy kolejno tyle identycznych znaków, ile się da, wstawiając spacje do obu sekwencji (również na dowolnym z końców), tak że wynikowe ciągi, powiedzmy *X'* i *Y'*, mają tę samą długość, lecz nie zawierają spacji na tych samych pozycjach. To znaczy, że nie możemy mieć wyspacjowanych zarówno x'_i, jak i y'_i. Po ułożeniu przypisujemy punktację każdej pozycji:

- –1, jeśli x'_i i y'_i są takie same i nie są spacjami,
- +1, jeśli x'_i różni się od y'_i i żaden nie jest spacją,
- +2, jeśli albo x'_i, albo y'_i jest spacją.

Wynik ułożenia jest sumą wyników z poszczególnych pozycji. Im niższy wynik, tym bliższe sobie są dwa łańcuchy. Napisy z powyższego przykładu możemy ułożyć następująco, przy czym _ oznacza spację:

```
X': ATGATCG_GCAT_
Y': _CAAT_GTGAATC
    *++—*–*–+—*
```

Znak – pod daną pozycją wskazuje na tej pozycji wynik –1, znak + wskazuje wynik +1, a * — wynik +2. W tym konkretnym ułożeniu uzyskuje się sumaryczny wynik $(6\cdot -1)+(3\cdot 1)+(4\cdot 2)$, czyli 5.

Jest wiele sposobów wstawiania spacji i układania dwóch sekwencji. Aby znaleźć sposób prowadzący do najlepszego dopasowania — czyli takiego, które ma najniższą punktację — stosujemy przekształcenie napisu z kosztami $c_C = -1$, $c_R = +1$ i $c_D = c_I = +2$. Im więcej uda się dopasować identycznych znaków, tym lepsze ułożenie, a ujemny koszt operacji *kopiuj* zachęca do dopasowywania identycznych znaków. Spacja w Y' odpowiada znakowi usuniętemu, toteż w powyższym przykładzie pierwsza spacja w Y' odpowiada usunięciu pierwszego znaku (`A`) z X. Spacja w X' odpowiada znakowi wstawionemu, tak więc w powyższym przykładzie pierwsza spacja w X' odpowiada wstawieniu znaku `T`.

Zobaczmy zatem, jak przekształcić napis X w napis Y. Zastosujemy programowanie dynamiczne z podproblemami postaci: „zamień napis przedrostkowy X_i na napis przedrostkowy Y_j", gdzie i przebiega wartości od 0 do m, a j — od 0 do n. Nazwiemy ten podproblem „problemem $X_i \rightarrow Y_j$", a problem, od którego rozpoczynamy, jest problemem $X_m \rightarrow Y_n$. Oznaczmy koszt rozwiązania problemu optymalnego $X_i \rightarrow Y_j$ przez *koszt*[*i*, *j*]. Jako przykład weźmy X = `ACAAGC` i Y = `CCGT`, tak więc chcemy rozwiązać problem $X_6 \rightarrow Y_4$ i będziemy używać kosztów operacji z układania sekwencji DNA: $c_C = -1$, $c_R = +1$ i $c_D = c_I = +2$. Będziemy rozwiązywać podproblemy postaci $X_i \rightarrow Y_j$, gdzie i zmienia się od 0 do 6, a j od 0 do 4. Na przykład problemem $X_3 \rightarrow Y_2$ jest przekształcenie napisu przedrostkowego X_3 = `ACA` w napis przedrostkowy Y_2 = `CC`.

Łatwo określić *koszt*[*i*, *j*], gdy i lub j równa się 0, ponieważ X_0 i Y_0 są napisami pustymi. Zamieniamy pusty napis na Y_j za pomocą j operacji *wstaw*, tak więc *koszt*[*i*, *j*] równa się $j \cdot c_I$. Podobnie zamieniamy X_i na pusty napis za pomocą i operacji *usuń*, tym samym *koszt*[*i*, 0] wyniesie $i \cdot c_D$. Gdy zarówno i, jak j są równe 0, zamieniamy napis pusty na siebie, wobec czego *koszt*[0, 0] wynosi oczywiście 0.

Gdy i oraz j są dodatnie, wówczas musimy sprawdzić, jak podstruktura optymalna stosuje się do przekształcania jednego napisu w drugi. Załóżmy na chwilę, że znamy ostatnią operację użytą do zamiany X_i na Y_j. Mogła to być jedna z czterech operacji: *kopiuj*, *zastąp*, *usuń* lub *wstaw*.

- Jeśli jako ostatnia wykonana została operacja *kopiuj*, to x_i i y_j musiały być tym samym znakiem. Podproblemem, który pozostaje, jest zamiana X_{i-1} na Y_{j-1}, a optymalne rozwiązanie problemu $X_i \rightarrow Y_j$ musi zawierać optymalne rozwiązanie problemu $X_{i-1} \rightarrow Y_{j-1}$. Dlaczego? Ponieważ gdybyśmy mieli rozwiązanie problemu $X_{i-1} \rightarrow Y_{j-1}$, które nie miało minimalnego kosztu, to zamiast niego moglibyśmy posłużyć się rozwiązaniem o minimalnym koszcie, aby uzyskać lepsze rozwiązanie problemu $X_i \rightarrow Y_j$ niż to, które otrzymaliśmy. Dlatego, zakładając, że ostatnią operacją było *kopiuj*, wiemy, że *koszt*[*i*, *j*] równa się *koszt*[*i* – 1, *j* – 1] + c_C.

  Spójrzmy na problem $X_5 \rightarrow Y_3$ w naszym przykładzie. Zarówno x_5, jak i x_3 są tu znakiem `G`, więc jeśli ostatnią operacją było *kopiuj*, to ze względu na to, że $c_C = -1$, musimy mieć *koszt*[5, 3] = *koszt*[4, 2] – 1. Jeśli *koszt*[4, 2] równa się 4, to *koszt*[5, 3] musi być 3. Gdybyśmy mogli znaleźć rozwiązanie problemu $X_4 \rightarrow Y_2$ z kosztem mniejszym niż 4, to moglibyśmy skorzystać z tego rozwiązania do znalezienia rozwiązania problemu $X_5 \rightarrow Y_3$ z kosztem mniejszym niż 3.

- Jeżeli ostatnią operacją było *zastąp*, to przy dającym się przyjąć założeniu, że nie możemy „zastąpić" znaku nim samym, x_i i y_j muszą się różnić. Stosując to samo uzasadnienie z podstrukturą optymalną, którym posłużyliśmy się w wypadku operacji *kopiuj*, widzimy, że zakładając, iż ostatnią operacją było *zastąp*, *koszt*[*i*, *j*] równa się *koszt*[*i* – 1, *j* – 1] + c_R.

  W naszym przykładzie rozważmy problem $X_5 \rightarrow Y_4$. Tym razem x_5 i y_4 są różnymi znakami (odpowiednio: `G` i `T`), jeżeli więc ostatnią operacją było *zastąp* `G` przez `T`, to zważywszy że $c_R = +1$, musimy mieć *koszt*[5, 4] = *koszt*[4, 3] + 1. Jeśli *koszt*[4, 3] równa się 3, to *koszt*[5, 4] musi być 4.

- Jeśli ostatnią operacją było *usuń*, to nie mamy żadnych ograniczeń na x_i lub y_j. Pomyślmy o operacji *usuń* jak o przeskakiwaniu znaku x_i i zostawianiu przedrostka Y_j w spokoju, wobec czego podproblem, który mamy rozwiązać, jest problemem $X_{i-1} \rightarrow Y_j$. Zakładając, że ostatnia operacja była usuwaniem, wiemy, że *koszt*[*i*, *j*] = *koszt*[*i* – 1, *j*] + c_D.

  Rozważmy w naszym przykładzie problem $X_6 \rightarrow Y_3$. Jeśli ostatnią operacją było *usuń* (usuniętym znakiem musi być x_6, czyli `C`), to ponieważ $c_D = +2$, musimy mieć *koszt*[6, 3] = *koszt*[5, 3] + 2. Jeśli *koszt*[5, 3] wynosi 3, *koszt*[6, 3] musi wynieść 5.

- Wreszcie, gdy ostatnią operacją było *wstaw*, to zostawia ona X_i bez zmian, lecz dodaje znak y_j, a podproblemem do rozwiązania jest $X_i \rightarrow Y_{j-1}$. Zakładając, że ostatnią operacją było *wstaw*, wiemy, że *koszt*[*i*, *j*] = *koszt*[*i*, *j* – 1] + c_I.

  W naszym przykładzie rozważmy problem $X_2 \rightarrow Y_3$. Jeżeli ostatnią operacją było *wstaw* (wstawionym znakiem musi być y_3, którym jest `G`), to ponieważ $c_I = +2$, musimy mieć *koszt*[2, 3] = *koszt*[2, 2] + 2. Jeśli *koszt*[2, 2] równa się 0, to *koszt*[2, 3] musi być 2.

Oczywiście nie wiemy z góry, która z czterech operacji została użyta jako ostatnia. Chcemy użyć tej, która daje najmniejszą wartość *koszt*[*i*, *j*]. Dla danej kombinacji *i* oraz *j* nadają się trzy z czterech operacji. Operacje *usuń* i *wstaw* nadają się wtedy, kiedy *i* oraz *j* są dodatnie i stosuje się dokładnie jedno *kopiuj* lub *zastąp*, zależnie od tego, czy x_i i y_j są tym samym znakiem. Aby obliczyć *koszt*[*i*, *j*] na podstawie innych wartości *koszt*, ustalamy, która z operacji, *kopiuj* czy *zastąp*, jest stosowana, i bierzemy minimalną wartość *koszt*[*i*, *j*] generowaną przez te trzy możliwe operacje. To znaczy *koszt*[*i*, *j*] jest najmniejszą z następujących czterech wartości:

- *koszt*[*i* – 1, *i* –1] + c_C, lecz tylko wówczas, gdy x_i i y_j są tym samym znakiem,
- *koszt*[*i* – 1, *j* –1] + c_R, lecz tylko wówczas, gdy x_i i y_j się różnią,
- *koszt*[*i* – 1, *j*] + c_D,
- *koszt*[*i*, *j* – 1] + c_I.

Zupełnie tak samo, jak to zrobiliśmy, wypełniając tabelę *l* przy obliczaniu LCS, możemy wypełnić wiersz po wierszu tabelę *koszt*. Możemy tak postąpić, gdyż podobnie jak w tabeli *l*, każdy wpis *koszt*[*i*, *j*], gdzie *i* oraz *j* są dodatnie, zależy od już obliczonych wpisów leżących tuż na lewo, tuż powyżej i w lewym rogu powyżej.

Oprócz tabeli *koszt* wypełnimy tabelę *op*, gdzie *op*[*i*, *j*] podaje ostatnią operację użytą do zamiany X_i na Y_j. Element *op*[*i*, *j*] możemy wypełnić podczas wypełniania *koszt*[*i*, *j*]. Procedura OBLICZ-TABELE-PRZEKSZTAŁCEŃ na następnej stronie wypełnia tabele *koszt* i *op* wiersz po wierszu, traktując *koszt* i *op* jako dwuwymiarowe tablice.

Na następnych stronach przedstawiono tabele *koszt* i *op* obliczone przez procedurę OBLICZ-TABELE-PRZEKSZTAŁCEŃ dla naszego przykładowego przekształcenia X = `ACAAGC` w Y = `CCGT` z $c_C = -1$, $c_R = +1$ i $c_D = c_I = +2$. Na skrzyżowaniu wiersza *i* z kolumną *j* występują wartości *koszt*[*i*, *j*] i *op*[*i*, *j*] ze skróconymi nazwami operacji. Na przykład ostatnia operacja użyta podczas przekształcania X_5 = `ACAAG` w Y_2 = `CC` zastępuje `G` przez `C`, a łączny koszt optymalnego ciągu operacji zamiany `ACAAG` na `CC` wynosi 6.

Procedura OBLICZ-TABELE-PRZEKSZTAŁCEŃ wypełnia poszczególne elementy tabel w stałym czasie, podobnie jak procedura OBLICZ-TABELĘ-LCS. Ponieważ każda z tabel zawiera $(m+1)\cdot(n+1)$ elementów, OBLICZ-TABELE-PRZEKSZTAŁCEŃ działa w czasie $\Theta(n\,m)$.

Aby zbudować ciąg operacji przekształcających *X* w *Y*, pomagamy sobie tabelą *op*, zaczynając od ostatniego elementu *op*[*m*, *n*]. Zagnieżdżamy się rekurencyjnie, podobnie jak w procedurze ZESTAWIAJ-LCS, dodając każdą napotkaną operację z tabeli *op* na końcu ciągu operacji. Pod tabelami podano procedurę ZESTAWIAJ-PRZEKSZTAŁCENIE. Początkowe wywołanie ma postać ZESTAWIAJ-PRZEKSZTAŁCENIE(*op*, *m*, *n*). Ciąg operacji zamiany X = `ACAAGC` na napis *Z*, taki sam jak Y = `CCGT`, występuje poniżej tabel *koszt* i *op*.

Podobnie jak w ZESTAWIAJ-LCS, każde rekurencyjne wywołanie procedury ZESTAWIAJ-PRZEKSZTAŁCENIE zmniejsza i albo j, albo oba indeksy, więc rekursja schodzi do najniższego poziomu po najwyżej $m + n$ wywołaniach rekurencyjnych. Ponieważ każde wywołanie rekurencyjne zużywa stały czas przed i po rekurencyjnym zagnieżdżeniu, procedura ZESTAWIAJ-PRZEKSZTAŁCENIE działa w czasie $O(m + n)$.

Procedura OBLICZ-TABELE-PRZEKSZTAŁCEŃ(X, Y, c_C, c_R, c_D, c_I)

Dane wejściowe:

- X i Y: dwa napisy o długości, odpowiednio, m i n.
- c_C, c_R, c_D, c_I: koszty operacji, odpowiednio: *kopiuj*, *zastąp*, *usuń* i *wstaw*.

Wynik: tablice *koszt*[0..*m*, 0..*n*] i *op*[0..*m*, 0..*n*]. Wartość w *koszt*[*i*, *j*] jest minimalnym kosztem przekształcenia przedrostka X_i w przedrostek Y_j, tym samym *koszt*[*m*, *n*] jest minimalnym kosztem przekształcenia X w Y. Operacja w *op*[*i*, *j*] jest ostatnią operacją wykonaną podczas przekształcania X_i w Y_j.

1. Niech *koszt*[0..*m*, 0..*n*] i *op*[0..*m*, 0..*n*] będą nowymi tablicami.
2. Ustaw *koszt*[0, 0] na 0.
3. Dla $i = 1$ do m:
   A. Nadaj *koszt*[*i*, 0] wartość $i \cdot c_D$ i ustaw *op*[*i*, 0] na *usuń* x_i.
4. Dla $j = 1$ do m:
   A. Nadaj *koszt*[0, *j*] wartość $j \cdot c_I$ i ustaw *op*[0, *j*] na *wstaw* y_j.
5. Dla $i = 1$ do m:
   A. Dla $j = 1$ do n:
   (Określ, która z operacji: *kopiuj* czy *zastąp*, ma być użyta, i ustaw *koszt*[*i*, *j*] i *op*[*i*, *j*] stosownie do tego, która z trzech możliwych operacji minimalizuje *koszt*[*i*, *j*]).
      i. Określ *koszt*[*i*, *j*] i *op*[*i*, *j*] następująco:
         a. Jeśli x_i i y_j są jednakowe, to nadaj *koszt*[*i*, *j*] wartość *koszt*[$i - 1$, $j - 1$] + c_C i podstaw do *op*[*i*, *j*] operację *kopiuj* x_i.
         b. W przeciwnym razie (x_i i y_j są różne) nadaj *koszt*[*i*, *j*] wartość *koszt*[$i - 1$, $j - 1$] + c_R i podstaw do *op*[*i*, *j*] operację *zastąp* x_i przez y_j.
      ii. Jeśli *koszt*[$i - 1$, *j*] + c_D < *koszt*[*i*, *j*], to nadaj *koszt*[*i*, *j*] wartość *koszt*[$i - 1$, *j*] + c_D i podstaw do *op*[*i*, *j*] operację *usuń* x_i.
      iii. Jeśli *koszt*[*i*, $j - 1$] + c_I < *koszt*[*i*, *j*], to nadaj *koszt*[*i*, *j*] wartość *koszt*[*i*, $j - 1$] + c_I i podstaw do *op*[*i*, *j*] operację *wstaw* y_j.
6. Zwróć tablice *koszt* i *op*.

i	x_i	j 0	1	2	3	4
		y_j	C	C	G	T
0		0	2 *wstaw* C	4 *wstaw* C	6 *wstaw* G	8 *wstaw* T
1	A	2 *usuń* A	1 *zast* A C	3 *zast* A C	5 *zast* A G	7 *zast* A T
2	C	4 *usuń* C	1 *kopiuj* C	0 *kopiuj* C	2 *wstaw* G	4 *wstaw* T
3	A	6 *usuń* A	3 *usuń* A	2 *zast* A C	1 *zast* A G	3 *zast* A T
4	A	8 *usuń* A	5 *usuń* A	4 *zast* A C	3 *zast* A G	2 *zast* A T
5	G	10 *usuń* G	7 *usuń* G	6 *zast* G C	3 *kopiuj* G	4 *zast* G T
6	C	12 *usuń* C	9 *kopiuj* C	6 *kopiuj* C	5 *usuń* C	4 *zast* C T

Operacja	X	Z
Napisy początkowe	ACAAGC	
usuń A	ACAAGC	
kopiuj C	ACAAGC	C
usuń A	ACAAGC	C
zastąp A *przez* C	ACAAGC	CC
kopiuj G	ACAAGC	CCG
zastąp C *przez* T	ACAAGC	CCGT

Pewna subtelność w procedurze ZESTAWIAJ-PRZEKSZTAŁCENIE zasługuje na dokładniejsze przeanalizowanie. Rekursja kończy się tylko wtedy, kiedy i oraz j osiągną 0. Załóżmy, że jedna z wartości: i lub j, lecz nie obie, równa się 0. Każdy z trzech przypadków w krokach 2A, 2B i 2C wchodzi w rekursję z wartością i lub j (albo z obiema) zmniejszoną o 1. Czy byłoby możliwe wywołanie rekurencyjne, w którym i lub j ma wartość –1? Na szczęście odpowiedź brzmi: nie. Załóżmy, że w wywołaniu ZESTAWIAJ-PRZEKSZTAŁCENIE $j = 0$, a i jest dodatnie. Ze sposobu konstruowania tabeli op wiemy, że $op[i, 0]$ jest operacją *usuń*, tak więc dochodzi do wykonania kroku 2B. Wywołanie rekurencyjne w kroku 2B ma postać ZESTAWIAJ-PRZEKSZTAŁCENIE(op, $i - 1$, j), a więc wartość j w wywołaniu rekurencyjnym pozostaje równa 0. Podobnie, jeśli $i = 0$, a j jest dodatnie, to $op[0, j]$ jest operacją *wstaw*, wobec czego dochodzi do wykonania kroku 2C i rekurencyjnego wywołania ZESTAWIAJ-PRZEKSZTAŁCENIE(op, i, $j - 1$), więc wartość i pozostaje zerem.

Procedura ZESTAWIAJ-PRZEKSZTAŁCENIE(*op*, *i*, *j*)

Dane wejściowe:

- *op*: tabela operacji wypełniona przez OBLICZ-TABELE-PRZEKSZTAŁCEŃ.
- *i* oraz *j*: indeksy do tabeli *op*.

Wynik: ciąg operacji, które przekształcają napis *X* w napis *Y*, gdzie *X* i *Y* są napisami wejściowymi do OBLICZ-TABELE-PRZEKSZTAŁCEŃ.

1. Jeśli *i* oraz *j* są równe 0, to zwróć ciąg pusty.
2. W przeciwnym razie (przynajmniej jedna z wartości *i* lub *j* jest dodatnia) wykonaj, co następuje:
   A. Jeśli *op*[*i*, *j*] jest operacją *kopiuj* lub *zastąp*, to zwróć ciąg utworzony najpierw przez rekurencyjne wywołanie ZESTAWIAJ-PRZEKSZTAŁCENIE(*op*, *i* – 1, *j* – 1), a następnie dołączenie do niego wartości *op*[*i*, *j*].
   B. W przeciwnym razie (*op*[*i*, *j*] nie jest ani operacją *kopiuj*, ani *zastąp*), jeżeli *op*[*i*, *j*] jest operacją *usuń*, to zwróć ciąg utworzony najpierw przez rekurencyjne wywołanie ZESTAWIAJ-PRZEKSZTAŁCENIE(*op*, *i* – 1, *j*), a następnie dołączenie do niego wartości *op*[*i*, *j*].
   C. W przeciwnym razie (*op*[*i*, *j*] nie jest ani operacją *kopiuj*, ani *zastąp*, ani *usuń*, więc musi być operacją *wstaw*) zwróć ciąg utworzony najpierw przez rekurencyjne wywołanie ZESTAWIAJ-PRZEKSZTAŁCENIE(*op*, *i*, *j* – 1), a następnie dołączenie do niego wartości *op*[*i*, *j*].

Dopasowywanie napisów

W problemie dopasowywania napisów mamy do rozpatrzenia dwa napisy: **napis z tekstem** *T* (ang. *text string*) i **napis wzorcowy** *P* (ang. *pattern string*). Chcemy znaleźć *wszystkie* wystąpienia *P* w *T*. Skrócimy te nazwy do słów „tekst" i „wzorzec" i założymy, że tekst i wzorzec składają się z, odpowiednio, *n* i *m* znaków, gdzie $m \leq n$ (ponieważ nie ma sensu poszukiwać wzorca dłuższego niż zadany tekst). Znaki w *P* i *T* będziemy zapisywać w postaci, odpowiednio: $p_1 p_2 p_3 \cdots p_m$ i $t_1 t_2 t_3 \cdots t_n$.

Ponieważ chcemy znaleźć wszystkie wystąpienia wzorca *P* w tekście *T*, rozwiązaniem będą wszystkie wielkości przesunięć *P*, po wykonaniu których natrafimy na *P* w *T*. Ujmując to inaczej, powiemy, że wzorzec *P* **występuje z przesunięciem s** (ang. *occurs with shift s*) w tekście *T*, jeśli podnapis *T*, który zaczyna się od pozycji t_{s+1}, jest taki sam jak wzorzec *P*: $t_{s+1} = p_1$, $t_{s+2} = p_2$ itd., aż do $t_{s+m} = p_m$. Minimalnym możliwym przesunięciem będzie 0, a ponieważ wzorzec nie może wyjść poza koniec tekstu, maksymalne możliwe przesunięcie wyniesie $n - m$. Chcemy poznać wszystkie przesunięcia, z którymi *P* występuje w *T*. Na przykład, jeśli mamy tekst `GTAACAGTAAACG`, a wzorzec *P* jest równy `AAC`, to *P* występuje w *T* z przesunięciami 2 i 9.

Gdybyśmy chcieli sprawdzić, czy wzorzec *P* występuje w tekście *T* z zadanym przesunięciem *s*, to musielibyśmy porównać wszystkie *m* znaków *P* ze znakami w *T*. Przy założeniu, że porównanie jednego znaku w *P* z jednym znakiem w *T* zabiera stałą ilość czasu, sprawdzenie wszystkich *m* znaków zajmie w najgorszym przypadku $\Theta(m)$ czasu. Oczywiście, jeżeli wykryjemy niezgodność znaków *P* i *T*, to nie musimy sprawdzać pozostałych znaków. Najgorszy przypadek występuje w każdym przesunięciu, gdy *P* nie ma w *T*.

Byłoby dość łatwo sprawdzić po prostu wzorzec z tekstem dla każdego możliwego przesunięcia, działając od 0 do $n - m$. Tak wyglądałoby to w wypadku sprawdzania występowania wzorca `AAC` w tekście `GTAACAGTAAACG` dla każdego możliwego przesunięcia; znaki dopasowane zacieniowano:

Wielkość przesunięcia	Tekst i wzorzec	Wielkość przesunięcia	Tekst i wzorzec
0	GTAACAGTAAACG AAC	6	GTAACAGTAAACG AAC
1	GTAACAGTAAACG AAC	7	GTAACAGTAAACG AAC
2	GTAACAGTAAACG AAC	8	GTAACAGTAAACG AAC
3	GTAACAGTAAACG AAC	9	GTAACAGTAAACG AAC
4	GTAACAGTAAACG AAC	10	GTAACAGTAAACG AAC
5	GTAACAGTAAACG AAC		

To proste podejście jest jednak niezbyt efektywne: z $n - m + 1$ możliwymi przesunięciami, w których każdorazowo zużywa się $O(m)$ czasu na sprawdzenie, czas działania wyniósłby $O((n - m)m)$. Sprawdzalibyśmy niemal każdy znak w tekście *m* razy.

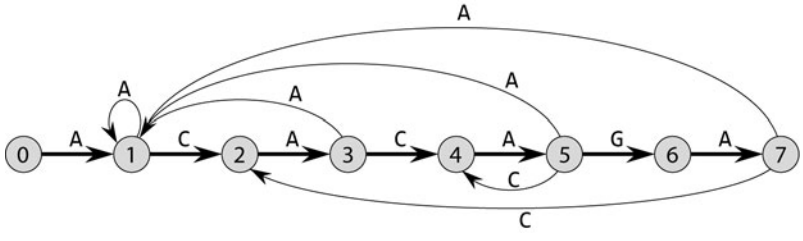
Możemy postąpić lepiej, ponieważ ta prosta metoda porównywania wzorca z tekstem dla każdego dopuszczalnego przesunięcia zdradza cenną informację. Kiedy w powyższym przykładzie zaczynaliśmy od pozycji $s = 2$, widzieliśmy wszystkie znaki w podnapisie $t_3t_4t_5 = \texttt{AAC}$. Lecz w następnym przesunięciu, $s = 3$, znowu patrzymy na pozycje t_4 i t_5. Byłoby ekonomiczniej uniknąć powtórnego patrzenia na te znaki, jeśli to możliwe. Przeanalizujmy sprytne podejście do dopasowywania napisów, w którym unika się marnowania czasu spowodowanego ponawianym analizowaniem tekstu. Zamiast *m*-krotnego sprawdzania znaków tekstu, każdy znak jest w nim badany tylko raz.

To ekonomiczniejsze podejście opiera się na **automacie skończonym** (ang. *finite automaton*). Choć nazwa brzmi dumnie, pomysł jest całkiem prosty. Istnieją liczne zastosowania automatów skończonych, lecz my skupimy się tutaj na użyciu

automatu skończonego do dopasowywania napisów. Automat skończony, w skrócie **FA**, jest po prostu zbiorem **stanów** (ang. *states*) i sposobem przechodzenia z jednego stanu do drugiego na podstawie ciągu znaków wejściowych. FA zaczyna od wyróżnionego stanu i konsumuje znaki z wejścia po jednym. Na podstawie stanu, w którym się znajduje, i znaku, który właśnie skonsumował, przechodzi do nowego stanu.

W naszym zastosowaniu z dopasowywaniem napisów ciąg wejściowy będzie składał się ze znaków tekstu *T*, a automat FA będzie miał *m* + 1 stanów, o jeden więcej niż wynosi liczba znaków we wzorcu *P*, ponumerowanych od 0 do *m*. („Skończoność" w nazwie „automat skończony" wywodzi się ze skończonej liczby stanów). Tak więc FA zaczyna w stanie 0. Kiedy jest w stanie *k*, ostatnich *k* znaków tekstu skonsumował na dopasowanie pierwszych *k* znaków wzorca. Wobec tego, jeśli tylko FA osiąga stan *m*, będzie to oznaczało, że obejrzał już cały wzorzec w tekście.

Spójrzmy na przykład, w którym są używane tylko znaki `A`, `C`, `G` i `T`. Załóżmy, że wzorzec ma postać `ACACAGA`, tak więc *m* = 7 znaków. Oto stosowny automat FA ze stanami od 0 do 7:

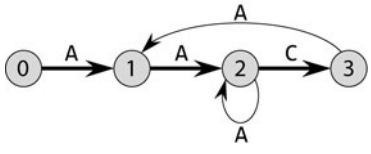


Kółka reprezentują stany, a etykietowane znakami strzałki pokazują, jak FA przechodzi ze stanu do stanu na podstawie znaków wejściowych. Na przykład strzałki ze stanu 5 mają etykiety `A`, `C` i `G`. Strzałka do stanu 1, z etykietą `A`, wskazuje, że jeśli FA jest w stanie 5 i konsumuje znak tekstu `A`, to przechodzi do stanu 1. Podobnie strzałka do stanu 4, zaetykietowana `C`, mówi nam, że jeśli FA jest w stanie 5 i konsumuje znak `C` z tekstu, to przechodzi do stanu 4. Zauważ, że nakreśliłem poziomy „szkielet" FA ze specjalnie grubymi strzałkami oraz że etykiety strzałek w szkielecie, czytane od lewej do prawej, składają się na wzorzec `ACACAGA`. Ilekroć ten wzorzec wystąpi w tekście, automat FA przesuwa się w prawo o jeden stan dla każdego znaku, aż osiągnie stan ostatni, w którym deklaruje, że znalazł wystąpienie wzorca w tekście. Zauważmy też, że niektóre strzałki są pominięte, na przykład wszelkie strzałki z etykietą `T`. Jeśli strzałka jest pominięta, to odpowiednie przejście prowadzi do stanu 0.

Automat skończony przechowuje wewnętrznie tabelę *następny-stan*, indeksowaną za pomocą wszystkich stanów i wszystkich możliwych znaków wejściowych. Wartość w *następny-stan*[*s*, *a*] jest numerem stanu, do którego ma nastąpić przejście, jeśli FA jest aktualnie w stanie *s* i właśnie konsumuje znak *a* z tekstu. Oto cała tabela *następny-stan* dla wzorca `ACACAGA`:

	Znak			
Stan	A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

Automat FA przesuwa się o jeden stan w prawo dla każdego znaku, który pasuje do wzorca, a dla każdego znaku, który nie pasuje do wzorca, przesuwa się w lewo lub pozostaje w tym samym stanie (*następny-stan*[1, A] wynosi 1). Później zobaczymy, jak zbudować tabelę *następny-stan*, lecz najpierw prześledźmy, co robi automat FA dla wzorca AAC na tekście wejściowym GTAACAGTAAACG. Oto ten automat:



Na podstawie tego rysunku potrafisz określić, że tabela *następny-stan* wygląda następująco:

	Znak			
Stan	A	C	G	T
0	1	0	0	0
1	2	0	0	0
2	2	3	0	0
3	1	0	0	0

Oto stany, do których przechodzi automat skończony, i znaki tekstu, które konsumuje, aby do nich przejść:

```
Stan   0  0  0  1  2  3  1  0  0  1  2  2  3  0
Znak      G  T  A  A  C  A  G  T  A  A  A  C  G
```

Zacieniowałem dwa przypadki, w których FA osiąga stan 3, ponieważ ilekroć osiągnie on stan 3, oznacza to, że znalazł wystąpienie wzorca AAC.

Oto procedura SKOŃCZONY-DOPASOWYWACZ-NAPISU służąca do dopasowywania napisów. Zakłada się w niej, że tabela *następny-stan* została już zbudowana.

Procedura SKOŃCZONY-DOPASOWYWACZ-NAPISU(*T*, *następny-stan*, *m*, *n*)

Dane wejściowe:

- *T*, *n*: napis z tekstem i jego długość.
- *następny-stan*: tabela przejść stanów, utworzona odpowiednio do dopasowywanego wzorca.
- *m*: długość wzorca. Tabela *następny-stan* ma wiersze indeksowane od 0 do *m*, a kolumny indeksowane znakami, które mogą pojawić się w tekście.

Wynik: drukuje wszystkie wielkości przesunięć, dla których wzorzec występuje w tekście.

1. Ustaw *stan* na 0.
2. Dla *i* = 1 do *n*:
   A. Nadaj zmiennej *stan* wartość *następny-stan*[*stan*, t_i].
   B. Jeśli *stan* równa się *m*, drukuj „Wzorzec występuje z przesunięciem" *i* – *m*.

Jeśli wykonamy SKOŃCZONY-DOPASOWYWACZ-NAPISU na powyższym przykładzie, w którym *m* równa się 3, to automat skończony osiągnie stan 3 po skonsumowaniu znaków t_5 i t_{12}. Dlatego procedura wydrukuje „Wzorzec występuje z przesunięciem 2" (2 = 5 – 3) i „Wzorzec występuje z przesunięciem 9" (9 = 12 – 3).

Ponieważ każda iteracja pętli w kroku 2 zużywa stałą ilość czasu, a pętla ta wykonuje dokładnie *n* iteracji, łatwo zauważyć, że czas działania SKOŃCZONEGO-DOPASOWYWACZA-NAPISU wynosi $\Theta(n)$.

To była łatwa część. Trudniejszą częścią jest skonstruowanie dla danego wzorca tabeli *następny-stan* automatu skończonego. Przypomnijmy pomysł:

> Jeśli automat skończony jest w stanie *k*, to *k* ostatnich znaków, które skonsumował, stanowi *k* pierwszych znaków wzorca.

Aby skonkretyzować ten pomysł, powróćmy do FA, dotyczącego wzorca `ACACAGA`, i zastanówmy się, dlaczego *następny-stan*[5, `C`] równa się 4. Gdyby FA znalazł się w stanie 5, to pięć ostatnich znaków, które skonsumował z tekstu, utworzyłoby napis `ACACA`, co możesz stwierdzić, spoglądając na szkielet FA. Gdyby następnym skonsumowanym znakiem był `C`, to nie pasowałby do wzorca i automat FA nie mógłby przejść do stanu 6. Lecz FA nie musi też wycofywać się do stanu 0. Dlaczego? Ponieważ teraz czterema ostatnio skonsumowanymi znakami są `ACAC`, a to są pierwsze cztery znaki wzorca `ACACAGA`. Oto dlaczego, gdy FA jest w stanie 5 i konsumuje `C`, przechodzi do stanu 4: widział ostatnio pierwsze cztery znaki wzorca.

Jesteśmy niemal gotowi, aby podać regułę konstrukcji tabeli *następny-stan*, lecz potrzebujemy najpierw kilku definicji. Przypomnijmy, że dla *i* z przedziału

od 0 do m przedrostek P_i wzorca P jest podnapisem składającym się z pierwszych i znaków wzorca P. (Jeśli i równa się 0, przedrostek jest napisem pustym). Zdefiniujemy odpowiednio **przyrostek** (sufiks, ang. *suffix*) wzorca jako podnapis złożony ze znaków z końca P. Na przykład `AGA` jest przyrostkiem wzorca `ACACAGA`. Zdefiniujemy też **złączenie** (konkatenację, ang. *concatenation*) napisu X i znaku a, jako nowy napis utworzony przez dodanie a na końcu X, i zapiszemy to w postaci Xa. Na przykład złączeniem napisu `CA` ze znakiem `T` jest napis `CAT`.

Jesteśmy wreszcie gotowi do zbudowania *następny-stan*[k, a], gdzie k jest numerem stanu z przedziału od 0 do m, natomiast a jest dowolnym znakiem mogącym się pojawić w tekście. W stanie k widzieliśmy już przedrostek P_k w tekście. Oznacza to, że k ostatnio oglądanych znaków jest tym samym, co pierwsze k znaków wzorca. Kiedy obejrzymy następny znak, powiedzmy a, obejrzeliśmy już w tekście $P_k\,a$ (złączenie P_k z a). W tym miejscu rodzi się pytanie: jak długi przedrostek P obejrzeliśmy? Innym sposobem sformułowania tego problemu jest następujące pytanie: jak długi przedrostek P występuje na końcu $P_k\,a$? Ta długość będzie numerem następnego stanu.

Uściślijmy:

> Weź przedrostek P_k (pierwsze k znaków P) i połącz go ze znakiem a. Oznacz wynikowy napis jako $P_k\,a$. Znajdź najdłuższy przedrostek P, który jest również przyrostkiem $P_k\,a$. Wówczas *następny-stan*[k, a] jest długością tego najdłuższego przedrostka.

Tak, rozpatrywanych jest kilka przedrostków i przyrostków, zobaczmy zatem, jak określamy, że *następny-stan*[5, `C`] wynosi 4 dla wzorca P = `ACACAGA`. Ponieważ w tym przykładzie k = 5, bierzemy przedrostek P_5, czyli `ACACA`, i łączymy go ze znakiem `C`, otrzymując `ACACAC`. Chcemy znaleźć najdłuższy przedrostek w `ACACAGA`, który jest również przyrostkiem `ACACAC`. Ponieważ napis `ACACAC` ma długość 6, a przyrostek nie może być dłuższy od napisu, którego jest przyrostkiem, zaczynamy od przyjrzenia się P_6, po czym schodzimy w dół do coraz krótszych przyrostków. Tutaj P_6 równa się `ACACAG` i nie jest przyrostkiem w `ACACAC`. Rozpatrujemy więc P_5, wynoszący `ACACA`: on również nie jest przyrostkiem w `ACACAC`. Dalej rozpatrujemy P_4, czyli `ACAC`. Tym razem ten przedrostek *jest* przyrostkiem w `ACACAC`, zatrzymujemy się więc i określamy, że *następny-stan*[5, `C`] równa się 4.

Możesz mieć wątpliwości, czy zawsze znajdziemy przedrostek P, który jest także przyrostkiem $P_k\,a$. Odpowiedź jest twierdząca, ponieważ napis pusty jest przedrostkiem i przyrostkiem w każdym napisie. Kiedy najdłuższy przedrostek P, będący również przyrostkiem w $P_k\,a$, okaże się napisem pustym, wtedy określamy *następny-stan*[k, a] jako 0. Pozostając dalej przy wzorcu P = `ACACAGA`, zobaczmy, jak określić *następny-stan*[3, `G`]. Złączenie P_3 z `G` daje napis `ACAG`. Pracujemy na przedrostkach P, zaczynając od P_4 (ponieważ długość `ACAC` wynosi 4) i schodząc w dół. Żaden z przedrostków `ACAC`, `ACA`, `AC` ani `A` nie jest przyrostkiem w `ACAG`, więc przyjmujemy, że najdłuższym przedrostkiem, który się sprawdza, jest napis pusty. Ponieważ napis pusty ma długość 0, nadajemy *następny-stan*[3, `G`] wartość 0.

Ile czasu zabiera zapełnienie całej tabeli *następny-stan*? Wiemy, że ma ona po jednym wierszu dla każdego stanu w FA, ma zatem $m + 1$ wierszy ponumerowanych od 0 do m. Liczba kolumn zależy od liczby znaków, które mogą wystąpić w tekście — nazwijmy ją q. Tak więc tabela *następny-stan* ma $q(m + 1)$ pozycji. Aby zapełnić *następny-stan*[k, a], wykonujemy, co następuje:

1. Tworzymy napis $P_k a$.
2. Nadajemy i mniejszą z wartości $k + 1$ (długość $P_k a$) i m (długość P).
3. Dopóki P_i nie jest przyrostkiem w $P_k a$, wykonujemy, co następuje:
   A. Nadaj i wartość $i - 1$.

Nie wiemy z góry, ile razy będzie wykonywana pętla z kroku 3, lecz wiemy, że wykona ona najwyżej $m + 1$ iteracji. Nie wiemy również z góry, ile znaków P_i i $P_k a$ trzeba będzie sprawdzić w kroku 3, lecz wiemy, że zawsze będzie to najwyżej i, czyli najwyżej m. Ponieważ pętla iteruje najwyżej $m + 1$ razy, a w każdej iteracji jest sprawdzanych najwyżej m znaków, daje to $O(m^2)$ jako czas zapełnienia pozycji *następny-stan*[k, a]. Ponieważ tabela *następny-stan* zawiera $q(m + 1)$ elementów, łączny czas jej zapełnienia wyniesie $O(m^3q)$.

W praktyce czas zapełnienia tabeli *następny-stan* nie jest taki zły. Zakodowałem algorytm dopasowywania napisów w języku C++ na moim 2,4-gigahercowym macbooku pro i skompilowałem go na poziomie optymalizacji -O3. Podałem mu wzorzec `a man, a plan, a canal, panama` ze 128-znakowym zbiorem znaków ASCII użytym jako alfabet. Program skonstruował tabelę *następny-stan* z 31 wierszami i 127 kolumnami (pominąłem kolumnę ze znakiem pustym) w około 1,35 milisekundy. Dla krótszego wzorca program oczywiście działał szybciej: zbudowanie tabeli dla wzorca `panama` zajęło mu około 0,07 ms.

Niemniej w niektórych zastosowaniach dopasowywanie napisów jest wykonywane często, toteż w takich aplikacjach czas $O(m^3q)$ zbudowania tabeli *następny-stan* może stanowić problem. Nie chcę wchodzić w szczegóły, lecz istnieje sposób na skrócenie tego czasu do $\Theta(mq)$, W rzeczywistości możemy osiągnąć nawet więcej. Algorytm „KMP" (opracowany przez Knutha, Morrisa i Pratta) korzysta z automatu skończonego, lecz unika tworzenia i zapełniania tabeli *następny-stan*. W zamian używa tablicy *przejdź-do* zawierającej tylko m numerów stanów, co umożliwia automatowi FA emulowanie tabeli *następny-stan* i zabiera tylko $\Theta(m)$ czasu na zapełnienie tablicy *przejdź-do*. Powtórzę, algorytm KMP jest nazbyt skomplikowany, aby się w niego zagłębiać, lecz wykonałem go na moim macbooku pro i dla wzorca `a man, a plan, a canal, panama` zbudowanie tablicy *przejdź-do* zajęło mu około jedną mikrosekundę. Dla krótszego wzorca `panama` program pracował około 600 nanosekund (0,0000006 s). Nieźle! Podobnie jak procedura SKOŃCZONY-DOPASOWYWACZ-NAPISU, po zbudowaniu tablicy *przejdź-do* algorytm KMP zużywa na dopasowanie wzorca do tekstu $\Theta(n)$ czasu.

Co czytać dalej

W rozdziale 15 CLRS [CLRS09] opisano szczegółowo programowanie dynamiczne, w tym znajdowanie najdłuższego wspólnego podciągu. Podany w tym rozdziale algorytm przekształcania jednego napisu w drugi stanowi część rozwiązania problemu z rozdziału 15 CLRS. (Problem w CLRS obejmuje dwie operacje: zamianę sąsiadujących znaków i usuwanie przyrostka *X*, których nie rozpatrywałem w tym rozdziale. Nie uważasz, że zdradzając całe rozwiązanie, zdenerwowałbym moich współautorów?).

Algorytmy dopasowywania napisów występują w rozdziale 32 CLRS. Podano tam algorytm oparty na automacie skończonym, a także pełny opis algorytmu KMP. W pierwszym wydaniu *Introduction to algorithms* [CLR90][3] pomieszczono algorytm Boyera-Moore'a, który jest szczególnie sprawny, gdy wzorzec jest długi, a liczba znaków w alfabecie duża.

[3] Por. T. H. Cormen, C. E. Leiserson, R. L. Rivest: *Wprowadzenie do algorytmów*, wyd. 1, WNT, Warszawa 1997, strona 980 i następne — *przyp. tłum.*

8 Podstawy kryptografii

Gdy kupujesz coś przez Internet, niewykluczone, że musisz podać numer swojej karty kredytowej serwerowi w witrynie sprzedawcy lub serwerowi w witrynie pośredniczącej w zapłacie. Aby przekazać numer karty kredytowej serwerowi, przesyłasz go Internetem. Internet jest siecią powszechnie dostępną i każdy może mieć wgląd w bity, które są w niej przesyłane. Gdyby więc numer Twojej karty kredytowej podróżował w Internecie niezamaskowany, ktoś mógłby go poznać i zacząć kupować towary i usługi na Twoje konto.

Trzeba przyznać, że jest mało prawdopodobne, że ktoś tam siedzi i czeka właśnie na to, że *Ty* prześlesz Internetem coś, co wygląda jak numer karty kredytowej. Bardziej prawdopodobna jest sytuacja, że ktoś czeka na *kogokolwiek*, kto to zrobi, a *Ty* możesz być akurat pechową ofiarą. Byłoby znacznie bezpieczniej dla Ciebie, gdyby dało się jakoś ukryć numer karty kredytowej, ilekroć przesyłasz go Internetem. I pewnie tak właśnie robisz. Jeśli korzystasz z bezpiecznych witryn — takich, których lokalizator URL zaczyna się od „https:", a nie od zwykłego „http:" — to Twoja przeglądarka maskuje przesyłane informacje w procesie zwanym **szyfrowaniem** (ang. *encryption*). (Protokół https umożliwia również „uwierzytelnianie" (ang. *authentication*), dzięki czemu masz pewność, że łączysz się właśnie z tą witryną, o którą Ci chodziło). W tym rozdziale przyjrzymy się szyfrowaniu i odwrotnemu procesowi: **deszyfrowaniu** (ang. *decryption*), w którym zaszyfrowane informacje są przywracane do ich pierwotnej formy. Rozpatrywane razem procesy szyfrowania i deszyfrowania tworzą podstawę dziedziny zwanej kryptografią.

Choć uważam, że numer mojej karty kredytowej jest informacją, której należy strzec, zdaję sobie sprawę, że nie jest to najważniejsze w wielkim porządku rzeczy. Z tego powodu, że ktoś mi ukradnie numer karty kredytowej, bezpieczeństwo narodowe nie dozna uszczerbku. Lecz jeśli ktoś zdoła podsłuchać instrukcje wydawane jakiemuś dyplomacie w Departamencie Stanu lub wyszpiegować informacje wojskowe, bezpieczeństwo narodowe mogłoby być naprawdę zagrożone. Dlatego potrzebujemy sposobów szyfrowania i deszyfrowania informacji, i to nie byle jakich, lecz skutecznych (czytaj: bardzo trudnych do pokonania).

W tym rozdziale rozważymy pewne podstawowe koncepcje dotyczące szyfrowania i deszyfrowania. Nowoczesna kryptografia sięga dalej, znacznie dalej poza to, co tu przedstawiam. Nie próbuj budować bezpiecznego systemu wyłącznie na podstawie materiału zawartego w tym rozdziale. Aby zbudować system bezpieczny zarówno w teorii, jak i w praktyce, należałoby zrozumieć nowoczesną kryptografię znacznie bardziej szczegółowo. Musiałbyś na przykład przestrzegać ustalonych standardów, takich jak te publikowane przez National Institute of Standard and Technology. Ron Rivest (jeden z wynalazców kryptosystemu RSA, który poznamy

później w tym rozdziale) napisał mi kiedyś: „Ogólnie biorąc, kryptografia ma w sobie coś z rywalizacji w sztukach walki. Aby korzystać z niej w praktyce, musisz zrozumieć najnowsze techniki przeciwnika". Niemniej jednak ten rozdział da Ci posmak pewnych algorytmów, które powstały po to, aby szyfrować i deszyfrować informacje.

W kryptografii informacje w postaci pierwotnej nazywamy **tekstem jawnym**[1] (ang. *plaintext*), a ich wersję zaszyfrowaną — **tekstem zaszyfrowanym** (kryptogramem, ang. *ciphertext*). Szyfrowanie zamienia zatem tekst jawny na zaszyfrowany, a deszyfrowanie zamienia tekst zaszyfrowany z powrotem na jawny. Informacja potrzebna do zamiany jest nazywana **kluczem** (ang. *key*) kryptograficznym.

Proste szyfry podstawieniowe

W **prostym szyfrze podstawieniowym** (ang. *simple substitution cipher*) szyfrujesz tekst, podstawiając po prostu jedną literę za inną, a tekst zaszyfrowany deszyfrujesz, odwracając to podstawienie. Juliusz Cezar mógł komunikować się ze swoimi dowódcami, używając **szyfru z przesunięciem** (ang. *shift cipher*), w którym nadawca zastępował każdą literę wiadomości literą występującą trzy miejsca dalej w alfabecie, dokonując na końcu alfabetu przejścia do początku. Na przykład w naszym 35-literowym alfabecie *A* byłoby zastąpione przez *C*, a *Ż* przez *B* (po *Ż* następuje *A*, a potem *Ą* i *B*). W przesunięciowym szyfrze Cezara dowódca potrzebujący wzmocnienia oddziałów mógłby zaszyfrować tekst jawny *Przyślijcie mi jeszcze stu żołnierzy* do postaci tekstu zaszyfrowanego *Stażvnlłelg ol łguaeag uwx bąńólgtaż*. Po otrzymaniu tego szyfrogramu, aby odkryć oryginalny tekst jawny *Przyślijcie mi jeszcze stu żołnierzy*, Cezar mógłby każdą literę zastąpić literą występującą trzy miejsca wcześniej w alfabecie, przechodząc u początku alfabetu na jego koniec. (Oczywiście w czasach Cezara ta wiadomość zostałaby zapisana po łacinie, z użyciem alfabetu łacińskiego z tamtych czasów).

Jeśli przechwycisz wiadomość i wiesz, że została ona zaszyfrowana za pomocą szyfru z przesunięciem, to rozszyfrowanie jej jest śmiesznie łatwe, nawet jeśli nie znasz z góry wielkości przesunięcia (klucza): wystarczy wypróbować wszystkie możliwe przesunięcia, aż odszyfrowany tekst przybierze sensowną postać tekstu jawnego. W wypadku alfabetu 35-znakowego potrzebujesz do tego tylko 34 przesunięć.

Możesz wzmocnić nieco bezpieczeństwo szyfru, zamieniając każdy znak na jakiś inny, unikatowy znak, lecz niekoniecznie ten, który występuje tyle samo pozycji dalej w alfabecie. To znaczy tworzysz permutację znaków i używasz jej jako klucza. Jest to nadal prosty szyfr podstawieniowy, lecz lepszy niż szyfr z przesunięciem. Jeśli masz w swoim alfabecie n znaków, to podsłuchiwacz, który przechwytuje wiadomość, musiałby rozpoznać, której z $n!$ (n silnia) permutacji użyłeś. Funkcja silnia rośnie bardzo szybko z n; w istocie — rośnie ona szybciej niż funkcja wykładnicza.

---

[1] W tym rozdziale stosujemy, z nielicznymi wyjątkami, terminologię zgodną z książką Douglasa R. Stinsona *Kryptografia w teorii i praktyce* (przekład W. Bartola), wydaną przez WNT w 2005 r. — *przyp. tłum.*

Dlaczego więc nie zamienić jednoznacznie każdego znaku na inny? Jeśli choć raz próbowałeś rozwiązać zagadkę typu „kryptocytat" (ang. *cryptoquote*) — pojawiają się takie w wielu czasopismach — to wiesz, że można zrobić użytek z częstości występowania liter i kombinacji liter, aby zawęzić liczbę wyborów. Przypuśćmy, że tekst jawny *Send me a hundred more soldiers*[2] wygląda następująco po zamianie na tekst zaszyfrowany: *Krcz sr h byczxrz sfxr kfjzgrxk*. W tekście zaszyfrowanym litera *r* występuje najczęściej i mógłbyś zgadnąć — trafnie — że odpowiadającym jej znakiem w tekście jawnym jest *e*, litera najczęściej występująca w tekstach angielskich. Potem mógłbyś zwrócić uwagę na dwuliterowe słowo *sr* w tekście zaszyfrowanym i odgadnąć, że tekstem jawnym odpowiadającym zaszyfrowanej literze *s* musi być jedna z liter: *b*, *h*, *m* lub *w*, ponieważ jedynymi dwuliterowymi angielskimi słowami kończącymi się na *e* są: *be*, *he*, *me* i *we*[3]. Mógłbyś również ustalić, że tekstowi zaszyfrowanemu *h* odpowiada tekst jawny *a*, ponieważ jedynym jednoliterowym słowem pisanym małą literą jest w angielskim *a*.

Oczywiście, gdybyś zaszyfrował numery kart kredytowych, to nie musiałbyś zbytnio martwić się o częstość liter lub ich kombinacje. Lecz dziesięć cyfr daje tylko 10! różnych sposobów zamiany cyfr liczby na inne, czyli 3 628 800. Dla komputera to niespecjalnie dużo, szczególnie gdy porównać to z liczbą 10^{16} możliwych numerów kart kredytowych (16 cyfr dziesiętnych), więc podsłuchiwacz mógłby zautomatyzować próby, próbując dokonać zakupów każdą z 10! możliwości, być może trafiając przy tym na numery kart kredytowych innych niż Twoja.

Być może zauważyłeś jeszcze inny problem z użyciem prostego szyfru podstawieniowego: zarówno nadawca, jak i odbiorca muszą uzgodnić klucz. Co więcej, jeśli wysyłasz różne wiadomości do różnych odbiorców i nie chcesz, aby każdy z nich mógł odszyfrować wiadomości przeznaczone dla kogoś innego, to musisz ustalić osobny klucz dla każdego z nich.

Kryptografia z kluczem symetrycznym

Jeśli nadawca i odbiorca używają tych samych kluczy, to stosują **kryptografię z kluczem symetrycznym** (ang. *symmetric-key cryptography*). Muszą w jakiś sposób uzgodnić z góry klucz, którego będą używać.

Podkładka jednorazowa

Zakładając na razie, że dobrze sobie radzisz z kryptografią z kluczem symetrycznym, skoro prosty szyfr podstawieniowy nie jest wystarczająco bezpieczny, jako inną możliwość można wskazać podkładkę jednorazową. Podkładka jednorazowa działa na bitach. Jak zapewne wiesz, *bit* jest skrótem od *binary digit* (z ang. cyfra dwójkowa) i może przyjmować tylko dwie wartości: 0 lub 1. Komputery cyfrowe przechowują

---

[2] Tym razem cytujemy tę „depeszę" w oryginale, aby nie komplikować jej dalszej analizy — *przyp. tłum.*

[3] Z ang.: być, on, mi i my — *przyp. tłum.*

informację w postaci ciągów bitów. Niektóre ciągi bitów reprezentują liczby, inne reprezentują znaki (z użyciem standardowych zestawów znaków: ASCII lub Unicode), a jeszcze inne reprezentują rozkazy wykonywane przez komputer.

Podkładka jednorazowa[4] polega na użyciu w odniesieniu do bitów operacji **alternatywy wykluczającej** (ang. *exclusive-or*), czyli **XOR**. Do zapisywania tej operacji używamy symbolu $\oplus$:

$0 \oplus 0 = 0$,
$0 \oplus 1 = 1$,
$1 \oplus 0 = 1$,
$1 \oplus 1 = 0$.

Najprostszy sposób pojmowania operacji XOR polega na przyjęciu, że jeśli x jest bitem, to $x \oplus 0 = x$, a $x \oplus 1$ daje przeciwieństwo x. Ponadto jeśli x i y są bitami, to $(x \oplus y) \oplus y = x$: wykonanie na x operacji XOR z użyciem tej samej wartości dwukrotnie daje x.

Załóżmy, że chcę Ci przesłać jednobitowy komunikat. Mógłbym Ci wysłać 0 lub 1 w postaci tekstu jawnego i moglibyśmy się umówić, czy posyłając Ci wartość bitu, posłałem Ci tę, którą chciałem, czy też wartość przeciwną. Patrząc przez pryzmat operacji XOR, moglibyśmy się umówić, czy potraktowałem ten bit w XOR zerem, czy jedynką. Gdybyś potem podziałał na odebrany, zaszyfrowany bit operacją XOR, jako drugiego argumentu używając bitu (klucza), którym ja podziałałem na niego za pomocą XOR, poznałbyś oryginalny tekst jawny.

Załóżmy teraz, że chcę Ci wysłać wiadomość dwubitową. Mógłbym zostawić oba bity nietknięte, mógłbym je odwrócić, mógłbym odwrócić pierwszy i nie ruszać drugiego lub odwrócić drugi, zostawiając pierwszy bez zmian. Znów moglibyśmy się porozumieć co do tego, które bity odwracam (jeśli w ogóle któreś). W kategoriach operacji XOR na dwóch bitach moglibyśmy uzgodnić, która z dwubitowych sekwencji: 00, 01, 10 czy 11 była kluczem, za pomocą którego „XOR-owałem" bity tekstu jawnego, żeby go przekształcić w kryptogram. I znowu, aby odkryć tekst oryginalny, mógłbyś potraktować operacją XOR ten dwubitowy kryptogram z użyciem tego samego dwubitowego klucza, którym „XOR-owałem" tekst jawny.

Gdyby tekst jawny wymagał b bitów — być może składając się ze znaków kodu ASCII lub Unicode i dając w sumie b bitów — to mógłbym wygenerować w charakterze klucza losowy ciąg b bitów, umożliwić Ci poznanie b bitów tego klucza i wykonać na tekście jawnym operację XOR z jego użyciem, bit po bicie, żeby utworzyć tekst zaszyfrowany. Po otrzymaniu b-bitowego kryptogramu mógłbyś potraktować go operacją XOR bit po bicie, stosując klucz do odkrycia

[4] Inaczej bloczek jednorazowy (ang. *one-time pad*); mówiąc mniej dosłownie: szyfr z kluczem jednorazowym — *przyp. tłum.*

b-bitowego tekstu jawnego. Ten system jest nazywamy **podkładką jednorazową** (ang. *one-time pad*)[5], a klucz nazywa się **podkładką** (ang. *pad*).

Dopóki bity klucza są wybierane losowo — a zajmiemy się tą kwestią później — odszyfrowanie przez podsłuchiwacza tekstu zaszyfrowanego na zasadzie odgadnięcia klucza jest prawie niemożliwe. Nawet jeśli podsłuchiwacz wie coś o tekście jawnym, na przykład że jest zapisany po angielsku, to dla dowolnego kryptogramu i dowolnego *potencjalnego* tekstu jawnego istnieje klucz zamieniający ów potencjalny tekst jawny w tekst zaszyfrowany[6], i ten klucz jest bitowym XOR potencjalnego tekstu jawnego i kryptogramu. (Wynika to z tego, że jeśli potencjalnym tekstem jawnym jest *t*, tekstem zaszyfrowanym — *c*, a kluczem — *k*, to nie tylko $t \oplus k = c$, lecz również $t \oplus c = k$; operację $\oplus$ stosuje się bit po bicie do *t*, *k* i *c*, tak więc *i*-ty bit *t* potraktowany za pomocą XOR z *i*-tym bitem *k* równa się *i*-temu bitowi *c*). W ten sposób szyfrowanie z użyciem podkładki jednorazowej chroni przed przejęciem przez podsłuchiwacza dodatkowych informacji o tekście jawnym.

Podkładki jednorazowe zapewniają wysoki poziom bezpieczeństwa, lecz klucze wymagają tyle bitów, ile ich zawiera tekst jawny; bity te powinny być wybierane losowo, a klucze muszą być wymieniane między komunikującymi się stronami zawczasu. Jak sugeruje nazwa, podkładkę jednorazową należy stosować tylko jeden raz. Jeśli użyjesz tego samego klucza *k* do tekstów jawnych t_1 i t_2, to $(t_1 \oplus k) \oplus (t_2 \oplus k) = t_1 \oplus t_2$, co może ujawnić miejsca, w których oba teksty jawne mają te same bity.

Szyfry blokowe i łańcuchowanie

Jeśli tekst jawny jest długi, to podkładka w schemacie z podkładką jednorazową musi być równie długa, co jest dość niewygodne. Zamiast tego w niektórych systemach z kluczami symetrycznymi łączy się dwie dodatkowe techniki: stosuje się krótszy klucz, a tekst jawny tnie się na pewną liczbę bloków, stosując klucz do każdego bloku z osobna. Tak więc rozważa się w nich tekst jawny jako złożony z l bloków t_1, t_2, t_3,..., t_l i szyfruje te bloki tekstu jawnego na *l* bloków c_1, c_2, c_3,..., c_l tekstu zaszyfrowanego. System taki jest określany jako **szyfr blokowy** (ang. *block cipher*).

---

[5] Nazwa pochodzi z czasów przedkomputerowej realizacji tego pomysłu, kiedy to każda komunikująca się strona miała bloczek papieru z kluczem wypisanym na każdej kartce i obie strony miały identyczne ciągi kluczy. Klucza wolno było użyć tylko raz, po czym wydzierano go razem z kartką bloczku, uwidaczniając następny klucz. W tym papierowym systemie stosowano szyfr z przesunięciem, lecz na zasadzie osobnych liter, w którym każda odpowiednia litera klucza wyznaczała wielkość przesunięcia, od 0 dla *a* do 25 dla *z*. Na przykład, ponieważ *z* oznacza przesunięcie o 25, *m* oznacza przesunięcie o 12, a *n* oznacza przesunięcie o 13, klucz *zmn* zamienia tekst jawny *dog* na tekst zaszyfrowany *cat*. Inaczej jednak niż w systemie opartym na XOR, przesuwanie liter tekstu zaszyfrowanego w tę samą stronę z tym samym kluczem nie daje z powrotem tekstu jawnego; w danym przypadku powstałoby *bmg*. Należało więc przesuwać litery tekstu zaszyfrowanego w odwrotnym kierunku.

[6] W schemacie kolejnych liter z poprzedniego przypisu klucz *zmn* zamienia tekst jawny *dog* na kryptogram *cat*, lecz możemy uzyskać ten kryptogram z innego tekstu jawnego, *elk* (z ang. łoś — *przyp. tłum.*), i innego klucza: *ypj*.

W praktyce w szyfrowaniu z użyciem szyfrów blokowych są stosowane systemy trochę bardziej skomplikowane niż wykonywanie zwykłego XOR z podkładki jednorazowej. W systemie AES (*Advanced Encryption Standard*, z ang. zaawansowany standard szyfrowania), jednym z często stosowanych kryptosystemów z kluczem symetrycznym, uwzględnia się szyfr blokowy. Nie będę wchodził w szczegóły AES, ograniczając się do stwierdzenia, że do produkowania szyfrogramów są w nim używane żmudne metody plasterkowania i kostkowania bloków tekstu jawnego. Stosowany w AES klucz ma rozmiar 128, 192 lub 256 bitów, a rozmiar bloku wynosi 128 bitów.

Z szyframi blokowymi wciąż wiąże się jednak pewien kłopot. Jeżeli ten sam blok pojawia się w tekście jawnym dwukrotnie, to w szyfrogramie ten sam zaszyfrowany blok również wystąpi dwukrotnie. Jeden ze sposobów rozwiązania tego problemu polega na zastosowaniu techniki **łańcuchowania bloków szyfru** (ang. *cipher block chaining*). Przypuśćmy, że chcesz mi wysłać zaszyfrowaną wiadomość. Siekasz tekst jawny t na l bloków t_1, t_2, t_3,..., t_l i tworzysz l bloków c_1, c_2, c_3,..., c_l tekstu zaszyfrowanego w następujący sposób. Powiedzmy, że szyfrujesz blok, stosując do niego pewną funkcję E, a ja będę deszyfrował blok tekstu zaszyfrowanego, traktując go pewną funkcją D. Tworzysz pierwszy blok c_1 tekstu zaszyfrowanego tak, jak można by oczekiwać: $c_1 = E(t_1)$. Zanim jednak zaszyfrujesz drugi blok, wykonujesz na nim bit po bicie XOR z c_1, czyli $c_2 = E(c_1 \oplus t_2)$. Trzeci blok traktujesz alternatywą wykluczającą z blokiem c_2: $c_3 = E(c_2 \oplus t_3)$. I tak dalej. Ogólnie biorąc, obliczasz i-ty blok tekstu zaszyfrowanego na podstawie $(i - 1)$-go bloku tekstu zaszyfrowanego i i-tego bloku tekstu jawnego: $c_i = E(c_{i-1} \oplus t_i)$. Ten wzór stosuje się nawet do obliczania c_1 z t_1, jeśli zaczniesz od c_0 złożonego z samych zer (ponieważ $0 \oplus x$ daje x). W celu odszyfrowania najpierw obliczam $t_1 = D(c_1)$. Na podstawie c_1 i c_2 mogę obliczyć t_2, obliczając najpierw $D(c_2)$, równe $c_1 \oplus t_2$, a potem „XOR-ując" wynik z c_1. Ogólnie, odszyfrowuję c_i w celu wyznaczenia t_i przez obliczenie $t_i = D(c_i) \oplus c_{i-1}$; jak w wypadku szyfrowania, schemat ten działa nawet dla obliczania t_1, jeśli zacznę od c_0 złożonego z samych zer.

Nie wyszliśmy jeszcze na prostą. Nawet w wypadku łańcuchowania bloków szyfru, jeśli wyślesz mi ten sam ciąg bloków tekstu zaszyfrowanego dwukrotnie, za każdym razem wyślesz ten sam ciąg bloków tekstu zaszyfrowanego. Podsłuchiwacz mógłby się zorientować, że wysyłasz mi ten sam komunikat dwa razy, co mogłoby być dla niego cenną informacją. Jedno rozwiązanie polega na nierozpoczynaniu od c_0 złożonego z samych zer. Generujesz natomiast losowo blok c_0 i używasz go do szyfrowania pierwszego bloku tekstu jawnego, a ja używam go do odszyfrowania pierwszego bloku tekstu zaszyfrowanego; ten losowo wygenerowany blok c_0 nazywamy **wektorem początkowym** (ang. *initialization vector*).

Uzgadnianie wspólnych informacji

Do poprawnego działania kryptografii z kluczem symetrycznym nadawca i odbiorca muszą uzgodnić klucz. Jeśli ponadto stosują szyfr blokowy z łańcuchowaniem bloków szyfru, to może im być również potrzebne uzgodnienie wektora początkowego. Jak łatwo sobie wyobrazić, uzgadnianie tych wartości z góry jest mało praktyczne.

Jak zatem nadawca i odbiorca mieliby uzgadniać klucz i wektor początkowy? Dalej w tym rozdziale zobaczymy, jak za pomocą kryptosystemu hybrydowego można je bezpiecznie przesyłać.

Kryptografia z kluczem jawnym

Jest oczywiste, że aby odbiorca zaszyfrowanego komunikatu mógł go odszyfrować, nadawca i odbiorca muszą znać klucz używany do szyfrowania. Prawda?

Nieprawda.

W **kryptografii z kluczem jawnym** (ang. *public key cryptography*) każda strona ma dwa klucze: **klucz publiczny** (klucz jawny, ang. *public key*) i **klucz tajny** (klucz prywatny, ang. *secret key*). Opiszę kryptografię z kluczem jawnym, odwołując się do dwóch stron: Ciebie i mnie. Będę przy tym oznaczał mój klucz publiczny jako *P*, a mój klucz tajny jako *S*. Ty masz własne klucze: publiczny i tajny. Inni uczestnicy komunikacji mają własne klucze publiczne i tajne.

Klucze tajne są utajnione, natomiast klucze publiczne może znać każdy. Mogą one nawet być przechowywane w scentralizowanym katalogu, który udostępnia każdemu możliwość poznania klucza publicznego dowolnej innej osoby. W odpowiednich warunkach Ty i ja możemy używać dowolnego z tych kluczy do szyfrowania i deszyfrowania. Przez „odpowiednie warunki" rozumiem, że istnieją funkcje korzystające z kluczy publicznych i tajnych, służące do szyfrowania tekstu jawnego na tekst zaszyfrowany lub deszyfrowania tekstu zaszyfrowanego na tekst jawny. Oznaczmy przez F_P funkcję, której używam z moim kluczem publicznym, a funkcję, której używam z moim kluczem tajnym — przez F_S.

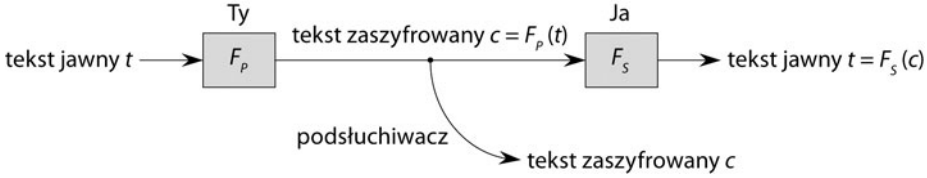
Klucze publiczne i tajne pozostają w specyficznym związku:

$$t = F_S(F_P(t)),$$

jeśli więc użyjesz mojego klucza publicznego do zaszyfrowania tekstu jawnego na tekst zaszyfrowany, a ja użyję mojego klucza tajnego, aby go odszyfrować, to otrzymam z powrotem tekst jawny. W pewnych innych zastosowaniach kryptografii z kluczem jawnym wymaga się, aby $t = F_P(F_S(t))$, tak więc gdy zaszyfruję tekst jawny moim kluczem tajnym, to każdy może odszyfrować tekst zaszyfrowany.

Każdemu powinno być wolno efektywnie obliczać moją funkcję F_P klucza publicznego, lecz tylko mnie powinno być wolno obliczać funkcję F_S mojego klucza tajnego z użyciem rozsądnej ilości czasu. Czas wymagany do udanego odgadnięcia mojej funkcji F_S bez znajomości mojego klucza powinien być zaporowo długi dla wszystkich innych. (Tak, wyrażam się tu niejasno, lecz wkrótce zobaczymy faktyczną implementację kryptografii z kluczem jawnym). To samo dotyczy kluczy publicznych i tajnych wszystkich innych osób: funkcja klucza publicznego jest obliczalna efektywnie, lecz tylko właściciel klucza tajnego może w rozsądnym czasie obliczyć funkcję klucza tajnego.

Oto jak możesz mi przesłać wiadomość, posługując się kryptografią z kluczem jawnym:



Zaczynasz od tekstu jawnego *t*. Znajdujesz mój klucz publiczny *P*; być może dostajesz go wprost ode mnie lub odnajdujesz w jakimś katalogu. Kiedy już masz *P*, szyfrujesz tekst jawny, aby otrzymać tekst zaszyfrowany $c = F_P(t)$, co możesz zrobić sprawnie. Wysyłasz mi tekst zaszyfrowany, tak więc ktokolwiek przechwyciłby to, co mi wysyłasz, zobaczyłby tylko tekst zaszyfrowany. Biorę ten zaszyfrowany tekst i odszyfrowuję go za pomocą mojego klucza tajnego, odtwarzając w ten sposób tekst jawny $t = F_S(c)$. Ty (lub ktokolwiek inny) możesz stosunkowo szybko szyfrować w celu utworzenia tekstu zaszyfrowanego, lecz tylko mnie uda się w rozsądnym czasie odszyfrować tekst zaszyfrowany, aby odtworzyć tekst jawny.

W praktyce musimy się upewnić, że funkcje F_P i F_S działają poprawnie. Chcemy, aby dla każdego możliwego tekstu jawnego F_P tworzyła inny tekst zaszyfrowany. Załóżmy bowiem, że F_P daje ten sam wynik dla dwóch różnych tekstów jawnych t_1 i t_2, tzn. $F_P(t_1) = F_P(t_2)$. Wówczas po otrzymaniu szyfrogramu $F_P(t_1)$ i odszyfrowaniu go za pomocą funkcji F_S nie wiedziałbym, czy otrzymałem t_1 czy t_2. Z drugiej strony, jest w porządku i w gruncie rzeczy — pożądane, aby szyfrowanie zawierało element losowości, tzn. żeby ten sam tekst jawny był szyfrowany na różne teksty zaszyfrowane za każdym razem, gdy jest przetwarzany funkcją F_P. (Kryptosystem RSA, który niebawem poznamy, jest znacznie bezpieczniejszy, gdy tekst jawny jest tylko małą porcją tego, co jest szyfrowane, a przeważająca część zaszyfrowanej informacji stanowi losowy „wypełniacz"). Oczywiście funkcja deszyfrująca F_S powinna być zaprojektowana tak, aby to kompensować, to znaczy zamieniać wiele tekstów [różnie] zaszyfrowanych na ten sam tekst jawny[7].

Powstaje jednak trudność. Tekst jawny *t* mógłby mieć dowolną liczbę wartości — mógłby być w istocie dowolnie długi – i liczba wartości tekstu zaszyfrowanego, na które F_P miałaby zamienić *t*, mogłaby być co najmniej tak duża jak liczba wartości, które mógłby przybrać *t*. Jak skonstruować funkcje F_P i F_S z dodatkowymi ograniczeniami, żeby F_P była łatwa do obliczenia dla każdego, a F_S — tylko dla mnie? Jest to trudne, lecz wykonalne, jeśli ograniczymy liczbę możliwych tekstów jawnych, to znaczy — zastosujemy szyfr blokowy.

[7] W bejsbolu jest stosowany podobny system. Menedżerowie i trenerzy informują zawodników, jak grać, stosując skomplikowane systemy gestów, zwanych „znakami" (ang. „*signs*"). Na przykład dotknięcie prawego ramienia może oznaczać wykonanie zagrywki uderz-i-biegnij, a dotknięcie lewego może oznaczać *bunt* (czyli blokadę piłki, tj. odbicie jej kijem bez zamachu — *przyp. tłum.*). Menedżer (lub trener) wykonuje długie serie znaków, lecz tylko niektóre z nich mają znaczenie; reszta służy zmyleniu przeciwnika. Nadawca i odbiorca znaków dysponują systemem, za pomocą którego uzgadniają, które znaki są istotne, niekiedy opartym na porządkowaniu ciągów znaków, a czasami na znaku „wywoławczym". Aby wskazać stosowny manewr gry, menedżer (lub trener) może dawać dowolnie długie serie znaków, przy czym większość z tych znaków jest bez znaczenia.

Kryptosystem RSA

Kryptografia z kluczem jawnym jest świetnym pomysłem, lecz zależy od możliwości znalezienia funkcji F_P i F_S, które poprawnie ze sobą współpracują. F_P jest łatwa do obliczenia przez każdego, a F_S jest łatwa do obliczenia tylko dla właściciela klucza tajnego. Schemat spełniający takie kryteria nazywamy **kryptosystemem z kluczem jawnym** (ang. *public-key cryptosystem*), a jednym z takich schematów jest **kryptosystem RSA** lub po prostu — **RSA**[8].

RSA zależy od kilku aspektów teorii liczb, z których znaczna część odnosi się do **arytmetyki modularnej** (arytmetyki modulo, ang. *modular arithmetic*). W arytmetyce modularnej wybieramy dodatnią liczbę całkowitą, powiedzmy n, i gdy tylko dojdziemy do n, natychmiast zawracamy na początek, do 0. Wygląda to jak zwykła arytmetyka liczb całkowitych, lecz zawsze z dzieleniem przez n i braniem reszty. Na przykład, jeśli pracujemy modulo 5, to jedynymi możliwymi wartościami są 0, 1, 2, 3, 4, a $3 + 4 = 2$, ponieważ 7 podzielone przez 5 daje w reszcie 2. Zdefiniujmy operator mod do obliczania reszt, abyśmy mogli rzec $7 \bmod 5 = 2$. Arytmetyka modularna przypomina arytmetykę zegara, lecz z zastępowaniem 12 przez 0 na tarczy zegara. Jeśli idziesz spać o 11 i śpisz osiem godzin, to budzisz się o 7: $(11 + 8) \bmod 12 = 7$.

Co jest szczególnie miłe w arytmetyce modularnej? Możność używania operacji mod wewnątrz wyrażeń bez zmiany wyniku:[9]

$$(a + b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n,$$
$$ab \bmod n = ((a \bmod n)(b \bmod n)) \bmod n,$$
$$a^b \bmod n = (a \bmod n)^b \bmod n.$$

Co więcej, dla dowolnej liczby całkowitej x zachodzi $x\ n \bmod n = 0$.

Ponadto do spełnienia przez RSA kryteriów kryptosystemu z kluczem jawnym muszą występować dwie własności dotyczące liczb pierwszych. Jak zapewne wiesz, **liczba pierwsza** (ang. *prime number*) jest liczbą całkowitą większą niż 1, która ma tylko dwa czynniki: 1 i samą siebie. Na przykład 7 jest liczbą pierwszą, lecz 6 nie, ponieważ daje się rozłożyć na czynniki $2 \cdot 3$. Pierwsza własność, od której zależy RSA, to ta, że jeśli masz liczbę będącą iloczynem dwóch wielkich, tajnych liczb pierwszych,

---

[8] Nazwa pochodzi od jego wynalazców: Ronalda Rivesta, Adiego Shamira i Leonarda Adlemana.

[9] Przykładowo, aby się przekonać, że $ab \bmod n = ((a \bmod n)(b \bmod n)) \bmod n$, załóżmy, że $a \bmod n = x$ i $b \bmod n = y$. Wtedy istnieją liczby całkowite i oraz j takie, że $a = ni + x$ i $b = nj + y$, zatem

$$\begin{aligned} ab \bmod n &= (ni + x)(nj + y) \bmod n \\ &= (n^2ij + xnj + yni + xy) \bmod n \\ &= ((n^2ij \bmod n) + (xnj \bmod n) + (yni \bmod n) + (xy \bmod n)) \bmod n \\ &= xy \bmod n \\ &= ((a \bmod n)(b \bmod n)) \bmod n. \end{aligned}$$

to nikt inny nie zdoła ustalić tych czynników w sensownym czasie. Przypomnijmy w ślad za rozdziałem 1, że ktoś mógłby sprawdzać wszystkie możliwe nieparzyste podzielniki aż do pierwiastka kwadratowego z danej liczby, lecz jeśli liczba jest duża — ma setki lub tysiące cyfr — to jej pierwiastek kwadratowy ma o połowę mniej cyfr, więc nadal bardzo dużo. Choć *teoretycznie* ktoś mógłby znaleźć któryś z czynników, wymagane do tego środki — czas i (lub) moc obliczeniowa — uczyniłyby poszukiwania czynnika nierealnymi[10].

Druga własność polega na tym, że choć rozkładanie na czynniki liczby będącej iloczynem dwóch wielkich liczb pierwszych jest trudne, nietrudno ustalić, czy duża liczba jest pierwsza. Mógłbyś pomyśleć, że określenie, iż liczba nie jest pierwsza — czyli że jest **złożona** (ang. *composite*) — jest niemożliwe bez znalezienia przynajmniej jednego niebanalnego czynnika (różnego od 1 i od niej samej). A jednak można to zrobić. Jednym ze sposobów jest test prymarności AKS[11], pierwszy algorytm określania w czasie $O(nc)$ z pewną stałą c, czy n-bitowa liczba jest pierwsza. Chociaż test prymarności AKS z teoretycznego punktu widzenia jest uważany za sprawny, nie jest jeszcze praktyczny dla dużych liczb. Zamiast niego możemy użyć testu prymarności Millera-Rabina. Ujemną stroną testu Millera-Rabina jest to, że może on wykazywać błędne działanie, uznając za pierwszą tę liczbę, która jest w rzeczywistości złożona. (Jeśli jednak test ów deklaruje, że liczba jest złożona, to na pewno jest złożona). Dobrą wiadomością jest to, że współczynnik błędu wynosi 1 na 2^s, przy czym za s możemy wybrać dowolną dodatnią wartość. Jeśli więc zgodzimy się zaryzykować błąd — powiedzmy — raz na 2^{50} testów, to możemy określić, czy liczba jest pierwsza z *niemal* całkowitą pewnością. Być może przypominasz sobie z rozdziału 1, że 2^{50} to około milion miliardów, czyli około 1 000 000 000 000 000. A jeśli wciąż czujesz się niekomfortowo z jednym błędem na 2^{50} testów, to z niewielkim dodatkowym nakładem możesz uzyskać 1 błąd na 2^{60} testów; 2^{60} to około 1000 razy więcej niż 2^{50}. Jest tak, ponieważ czas wykonania testu Millera-Rabina rośnie liniowo z parametrem s, więc zwiększenie s o 10, z 50 do 60, zwiększa czas działania tylko o 20%, lecz zmniejsza częstość błędów o czynnik 2^{10}, co wynosi 1024.

Oto jak zabrałbym się do stosowania kryptosystemu RSA. Po obejrzeniu działania RSA zajmiemy się kilkoma szczegółami.

1. Wybieram losowo dwie bardzo duże liczby pierwsze, p i q, różniące się od siebie. Co to znaczy „bardzo duże"? Przynajmniej po 1024 bity na każdą, czyli co najmniej 309 cyfr dziesiętnych. Lepiej, aby były jeszcze większe.
2. Obliczam $n = pq$. Jest to liczba co najmniej 2048-bitowa, czyli ma co najmniej 618 cyfr dziesiętnych.
3. Obliczam $r = (p - 1)(q - 1)$, która jest prawie tak duża jak n.

[10] Na przykład, gdyby liczba miała 1000 bitów, to jej pierwiastek kwadratowy miałby 500 bitów, będąc liczbą rzędu 2^{500}. Nawet gdyby ktoś sprawdzał bilion bilionów potencjalnych podzielników na sekundę, Słońce mogłoby się wypalić znacznie, znacznie wcześniej, nimby osiągnął 2^{500}.

[11] Nazwa pochodzi od jego wynalazców: Manidry Agrawala, Neeraja Kayala i Nitina Saxeny.

4. Wybieram małą liczbę nieparzystą e, **względnie pierwszą** (ang. *relative prime*) z r: jedynym wspólnym dzielnikiem e i r powinno być 1. Nadaje się do tego dowolna mała liczba całkowita.
5. Obliczam d jako **odwrotność multiplikatywną** (ang. *multiplicative inverse*) e modulo r. To znaczy, że ed mod r powinno być równe 1.
6. Za **klucz publiczny RSA** obieram parę $P = (e, n)$.
7. Parę $S = (d, n)$ ukrywam przed wszystkimi — jest to mój **klucz tajny RSA**.
8. Definiuję funkcje F_P i F_S:

   $F_P(x) = x^e \bmod n$,

   $F_S(x) = x^d \bmod n$.

   Te funkcje mogą działać na blokach tekstu jawnego lub na blokach tekstu zaszyfrowanego, których bity interpretujemy jako reprezentujące wielkie liczby całkowite.

Rozważmy przykład, lecz z użyciem małych liczb, abyśmy mogli zrozumieć, jak to działa.

1. Wybieramy liczby pierwsze $p = 17$ i $q = 29$.
2. Obliczamy $n = pq = 493$.
3. Obliczamy $r = (p - 1)(q - 1) = 448$.
4. Wybieramy liczbę $e = 5$, która jest względnie pierwsza z 448.

Obliczamy $d = 269$. Sprawdzenie: $ed = 5 \cdot 269 = 1345$, więc ed mod r = 1345 mod 448 = $(3 \cdot 448 + 1) \bmod 448 = 1$.

1. Przyjmuję jako mój klucz publiczny RSA parę $P = (5, 493)$.
2. Chowam $S = (269, 493)$ jako mój klucz tajny RSA.
3. Jako przykład obliczmy $F_P(327)$:

   $F_P(327) = 327^5 \bmod 493$

   $= 3\,738\,856\,210\,497 \bmod 493$

   $= 259$.

   Jeśli obliczymy $F_S(259) = 259^{269} \bmod 493$, powinniśmy otrzymać z powrotem 327. Otrzymujemy, lecz — szczerze radzę — daj sobie spokój z dociekaniem wszystkich cyfr wyrażenia 256^{269}. Możesz wyszukać w Internecie kalkulator liczący z dowolną precyzją i sprawdzić to tam[12]. (Sprawdziłem). I znów, skoro pracujemy w arytmetyce modularnej, nie musimy obliczać rzeczywistej wartości 256^{269}; wszystkie wyniki pośrednie możemy wyrażać modulo 493, gdybyś więc chciał, to mógłbyś zacząć od iloczynu 1, a potem 269 razy wykonać, co następuje: przemnóż to, co masz, przez 259 i weź iloczyn modulo 493. Otrzymasz w wyniku 327. (Ja to zrobiłem, tzn. zrobił to napisany przeze mnie program komputerowy).

[12] Por. np. *http://apfloat.appspot.com/ — przyp. tłum.*

A teraz kilka szczegółów, którym muszę poświęcić uwagę, aby przygotować i używać RSA:

- Jak mogę działać na liczbach z setkami cyfr?
- Choć sprawdzanie, czy liczba jest pierwsza, nie stanowi przeszkody, skąd wiem, że mogę znaleźć duże liczby pierwsze w sensownym czasie?
- Jak znajduję takie *e*, żeby *e* oraz *r* były względnie pierwsze?
- Jak obliczam liczbę *d*, aby była ona multiplikatywną odwrotnością *e* modulo *r*?
- Jeśli *d* jest duża, to jak obliczam x^d mod *n* w sensownym czasie?
- Skąd wiem, że funkcje F_P i F_S są wzajemnie odwrotne?

Jak wykonywać działania arytmetyczne na wielkich liczbach

Jest oczywiste, że liczby tak duże, jak tego wymaga RSA, nie zmieszczą się w rejestrach większości komputerów, przechowujących najwyżej 64 bity. Na szczęście kilka pakietów oprogramowania, a nawet niektóre języki programowania — na przykład Python — umożliwiają działania na liczbach całkowitych bez ograniczania ich wielkości.

Ponadto cała arytmetyka w RSA jest arytmetyką modularną, co umożliwia nam ograniczenie rozmiarów obliczanych liczb całkowitych. Na przykład gdy obliczamy x^d mod *n*, będziemy obliczać wyniki pośrednie, czyli podnosić *x* do różnych potęg, lecz wszystko to modulo *n*, co oznacza, że wszystkie obliczone wyniki pośrednie będą się mieścić w przedziale od 0 do $n - 1$. Co więcej, jeśli ustalasz maksymalne rozmiary *p* i *q*, to ustalasz maksymalny rozmiar *n*, a to z kolei oznacza, że realizacja RSA jest możliwa za pomocą specjalizowanego sprzętu.

Jak znajdować duże liczby pierwsze

Dużą liczbę pierwszą mogę znaleźć, powtarzając generowanie dużej liczby nieparzystej i stosowanie testu prymarności Millera-Rabina do określenia, czy jest to liczba pierwsza; zatrzymuję się, gdy znajdę liczbę pierwszą. W tym schemacie zakłada się z góry, że dużą liczbę pierwszą znajdę w niezbyt długim czasie. A jeśli liczby pierwsze pojawiają się wyjątkowo rzadko ze wzrostem wielkości liczb? Mógłbym wtedy spędzić mnóstwo czasu, poszukując liczby pierwszej niczym igły w stogu siana.

Nie muszę się martwić. **Twierdzenie o liczbach pierwszych** głosi, że gdy *m* dąży do nieskończoności, liczba liczb pierwszych mniejszych lub równych *m* dąży do $m/\ln m$, gdzie ln *m* jest logarytmem naturalnym z *m*. Jeśli wybiorę *m* losowo, to mam mniej więcej jedną szansę na ln *m*, że jest to liczba pierwsza. Teoria prawdopodobieństwa mówi, że średnio muszę wypróbować tylko około ln *m* liczb w otoczeniu *m*, nim znajdę taką, która jest pierwsza. Jeżeli poszukuję 1024-bitowych liczb pierwszych *p* i *q*, to *m* wynosi 2^{1024}, a ln *m* — około 710. Komputer szybko poradzi sobie z testem Millera-Rabina dla 710 liczb.

W praktyce mogę wykonać prostsze sprawdzenie niż test Millera-Rabina. **Małe twierdzenie Fermata** głosi, że jeśli *m* jest liczbą pierwszą, to x^{m-1} mod *m* równa się 1 dla dowolnej liczby *x* z przedziału od 1 do $m - 1$. Teza odwrotna: jeśli x^{m-1} mod

m równa się 1 dla dowolnej liczby x należącej do przedziału od 1 do $m - 1$, to m jest liczbą pierwszą, niekoniecznie jest prawdziwa, lecz wyjątki są bardzo rzadkie w wypadku wielkich liczb. W rzeczywistości prawie zawsze wystarcza wypróbować nieparzyste liczby m i uznać, że m będzie pierwsza, jeśli $2^{m-1} \bmod m$ równa się 1. Dalej zobaczymy, jak obliczyć $2^{m-1} \bmod m$ za pomocą tylko $\Theta(\lg m)$ mnożeń.

Jak znaleźć liczbę względnie pierwszą z inną

Muszę znaleźć małą liczbę nieparzystą e, która jest względnie pierwsza z r. Dwie liczby są względnie pierwsze, jeśli ich największy wspólny dzielnik wynosi 1. Posłużę się algorytmem obliczania największego wspólnego dzielnika dwóch liczb całkowitych, pochodzącym od starożytnego greckiego matematyka Euklidesa. W teorii liczb istnieje twierdzenie, które mówi, że jeśli a i b są liczbami całkowitymi, z których żadna nie jest zerem, to ich największy wspólny dzielnik g równa się $ai + bj$ dla pewnych liczb całkowitych i, j. (Co więcej, g jest najmniejszą liczbą, którą można utworzyć w ten sposób, lecz ten fakt nie ma dla nas znaczenia). Jeden ze współczynników i lub j może być ujemny; na przykład największy wspólny dzielnik liczb 30 i 18 równa się 6, a $6 = 30i + 18j$, gdy $i = -1$, a $j = 2$.

Na następnej stronie znajduje się algorytm Euklidesa w postaci, która podaje największy wspólny dzielnik g liczb a i b wraz ze współczynnikami i, j. Współczynniki te przydadzą się nieco później, gdy będę chciał znaleźć multiplikatywną odwrotność e modulo r. Jeśli mam wartość kandydującą do e, to wywołuję procedurę EUKLIDES(r, e). Jeśli pierwszy element trójki zwróconej przez wywołanie równa się 1, to wartość kandydująca do e jest względnie pierwsza z r. Jeśli pierwszy element jest inną liczbą, to r i wartość kandydująca do e mają wspólny dzielnik większy niż 1, więc nie są względnie pierwsze.

Procedura EUKLIDES(a, b)

Dane wejściowe: a i b – dwie liczby całkowite.

Wynik: trójka (g, i, j) o tej własności, że g jest największym wspólnym dzielnikiem liczb i oraz j, a także $g = ai + bj$.

1. Jeśli b równa się 0, to zwróć trójkę (a, 1, 0).
2. W przeciwnym razie (b nie jest zerem) wykonaj, co następuje:
   A. Wywołaj rekurencyjnie EUKLIDES(b, a mod b) i przypisz zwrócony wynik do trójki (g, i', j'). To znaczy nadaj g wartość pierwszego elementu zwróconej trójki, nadaj i' wartość drugiego elementu zwróconej trójki i nadaj j' wartość trzeciego elementu zwróconej trójki.
   B. Nadaj i wartość j'.
   C. Nadaj j wartość $i' - \lfloor a / b \rfloor j'$.
   D. Zwróć trójkę (g, i, j).

Nie chcę wnikać w to, dlaczego ta procedura działa[13], ani analizować jej czasu działania, lecz powiem Ci tylko, że jeśli wykonam wywołanie EUKLIDES(r, e), to liczba rekurencyjnych wywołań wyniesie $O(\lg e)$. Dzięki temu mogę szybko sprawdzić, czy 1 jest największym wspólnym dzielnikiem r i wartości kandydującej do e. (Pamiętajmy, że e jest mała). Jeśli nie, to mogę wypróbować inną wartość kandydującą do e i tak dalej, aż znajdę taką, która jest względnie pierwsza z r. Ile takich kandydatek przyjdzie mi sprawdzać? Niezbyt wiele. Jeżeli ograniczę wybory na e do nieparzystych liczb pierwszych mniejszych niż r (łatwo sprawdzalnych za pomocą testu Millera-Rabina lub testu opartego na małym twierdzeniu Fermata), to każdy wybór ma dużą szansę na to, że będzie względnie pierwszy z r. Jest tak dlatego, że na mocy twierdzenia o liczbach pierwszych w przybliżeniu $r / \ln r$ liczb pierwszych jest mniejszych niż r, lecz z innego twierdzenia wynika, że r nie może mieć więcej niż $\lg r$ czynników pierwszych. Dlatego mam niewielkie szanse natrafienia na czynnik pierwszy r.

Jak obliczyć odwrotność multiplikatywną w arytmetyce modularnej

Kiedy już mam r i e, muszę obliczyć d jako odwrotność e modulo r, aby $ed \bmod r$ równało się 1. Wiemy już, że wywołanie EUKLIDES(r, e) zwróciło trójkę postaci $(1, i, j)$, że 1 jest największym wspólnym dzielnikiem e i r (ponieważ są one względnie pierwsze) i że $1 = ri + ej$. Mogę więc po prostu nadać d wartość $j \bmod r$.[14] Wynika to z tego, że działamy modulo r, możemy więc wziąć obie strony modulo r:

$$\begin{aligned} 1 \bmod r &= (ri + ej) \bmod r \\ &= ri \bmod r + ej \bmod r \\ &= 0 + ej \bmod r \\ &= ej \bmod r \\ &= (e \bmod r) \cdot (j \bmod r) \bmod r \\ &= e(j \bmod r) \bmod r. \end{aligned}$$

(Ostatni wiersz wynika z tego, że $e < r$, co implikuje, że $e \bmod r = e$). Widzimy zatem, że $1 = e(j \bmod r) \bmod r$, co oznacza, że mogę nadać d wartość j z trójki zwracanej przez wywołanie EUKLIDES(r, e), wziętą modulo r. Korzystam z $j \bmod r$ zamiast z samej j, na wypadek gdyby j nie leżała w przedziale od 0 do $r - 1$.

---

[13] Wywołanie EUKLIDES(0, 0) zwraca trójkę (0, 1, 0), więc według niego 0 jest największym wspólnym dzielnikiem liczb 0 i 0. Może Cię razić ta specyfika (powiedziałbym „osobliwość", lecz to zły kontekst jak na „osobliwość" w takim znaczeniu), ponieważ jednak r jest dodatnie, parametr a w pierwszym wywołaniu EUKLIDES będzie dodatni i w każdym rekurencyjnym wywołaniu a musi być dodatnie. Nie ma więc dla nas znaczenia, co zwraca EUKLIDES (0, 0).

[14] Przypomnijmy, że liczba j mogłaby być ujemna. Jeden ze sposobów pojmowania $j \bmod r$, gdy j jest ujemna, a r dodatnia, polega na rozpoczęciu od j i dodawaniu r, aż uzyskana liczba stanie się nieujemna. Ta liczba równa się $j \bmod r$. Na przykład, aby określić $-27 \bmod 10$, bierzesz liczby –27, –17, –7 i 3. Kiedy otrzymasz 3, kończysz i stwierdzasz, że $-27 \bmod 10$ równa się 3.

Jak szybko podnieść liczbę do potęgi całkowitej

Chociaż e jest mała, d może być duża, a ja muszę obliczyć $x^d \bmod n$, aby obliczyć funkcję F_S. Co prawda mogę działać modulo n, co oznacza, że wszystkie wartości, z którymi mam do czynienia, będą w przedziale od 0 do $n - 1$, nie chcę jednak mnożyć liczb d razy. Na szczęście nie muszę. Mogę mnożyć liczby tylko $\Theta(\lg d)$ razy, stosując technikę o nazwie **powtarzane podnoszenie do kwadratu** (ang. *repeated squaring*). Mogę zastosować tę technikę do testu prymarności opartego na małym twierdzeniu Fermata.

Pomysł jest następujący. Wiemy, że d jest nieujemna. Załóżmy najpierw, że d jest parzysta. Wtedy x^d równa się $(x^{d/2})^2$. Teraz załóżmy, że d jest nieparzysta. Wtedy x^d równa się $(x^{(d-1)/2})^2 \cdot x$. Te obserwacje naprowadzają nas na przyjemny rekurencyjny sposób obliczenia x^d, w którym przypadek bazowy występuje wówczas, gdy d równa się 0: x^0 równa się 1. Następująca procedura ujmuje to podejście, wykonując wszystkie działania arytmetyczne modulo n:

Procedura POTĘGOWANIE-MODULARNE(x, d, n)

Dane wejściowe: x, d, n — trzy liczby całkowite, przy czym x i d są nieujemne, a n jest dodatnia.

Wynik: zwraca wartość $x^d \bmod n$.

1. Jeśli d równa się 0, to zwróć 1.
2. W przeciwnym razie (d jest dodatnia), jeśli d jest parzysta, to wywołaj rekurencyjnie POTĘGOWANIE-MODULARNE(x, d / 2, n), nadaj z wartość wyniku tego rekurencyjnego wywołania i zwróć $z^2 \bmod n$.
3. W przeciwnym razie (d jest dodatnia i nieparzysta) wywołaj rekurencyjnie POTĘGOWANIE-MODULARNE(x, $(d - 1)$ / 2, n), nadaj z wartość wyniku tego rekurencyjnego wywołania i zwróć $(z^2 \cdot x) \bmod n$.

W każdym rekurencyjnym wywołaniu parametr d zmniejsza się co najmniej o połowę. Po co najwyżej $\lfloor \lg d \rfloor + 1$ wywołaniach d zmniejsza się do zera i rekursja się kończy. Dlatego ta procedura mnoży liczby $\Theta(\lg d)$ razy.

Wykazanie, że funkcje F_P i F_S są wzajemnie odwrotnymi

Uwaga! *Dalej jest sporo teorii liczb i arytmetyki modularnej. Jeśli zadowolisz się uznaniem bez dowodu, że funkcje F_P i F_S są wzajemnie odwrotne, to pomiń następne pięć akapitów i podejmij czytanie od podrozdziału „Kryptosystemy hybrydowe".*

Aby RSA był kryptosystemem z kluczem jawnym, funkcje F_P i F_S muszą być wzajemnymi odwrotnościami. Jeśli bierzemy blok t tekstu jawnego, traktujemy go jako liczbę całkowitą mniejszą niż n i podajemy na wejście F_P, to otrzymujemy $t^e \bmod n$, a jeśli podamy ten wynik na wejście funkcji F_S, to otrzymamy $(t^e)^d \bmod n$, co jest równe $t^{ed} \bmod n$. Jeżeli odwrócimy kolejność, używając najpierw F_S, a potem

F_P, to otrzymamy $(t^d)^e \bmod n$, co znowu równa się $t^{ed} \bmod n$. Musimy pokazać, że dla dowolnego bloku tekstu jawnego t, interpretowanego jako liczba całkowita mniejsza niż n, $t^{ed} \bmod n$ jest równe t.

Oto szkic naszego postępowania. Przypomnijmy, że $n = pq$. Wykażemy, że $t^{ed} \bmod p = t \bmod p$ oraz że $t^{ed} \bmod q = t \bmod q$. Potem, opierając się na innym fakcie z teorii liczb, wywnioskujemy, że $t^{ed} \bmod pq = t \bmod pq$ — innymi słowy, że $t^{ed} \bmod n = t \bmod n$, co jest równe po prostu t, ponieważ t jest mniejsze niż n.

Znów musimy skorzystać z małego twierdzenia Fermata, które pomaga wyjaśnić, dlaczego określamy r jako iloczyn $(p - 1)(q - 1)$. (Nie dziwiło Cię, skąd to się wzięło?). Ponieważ p jest liczbą pierwszą, jeśli $t \bmod p$ jest różne od zera, to $(t \bmod p)^{p-1} \bmod p = 1$.

Przypomnijmy, że zdefiniowaliśmy e i d tak, aby były multiplikatywnymi odwrotnościami modulo r: $ed \bmod r = 1$. Mówiąc inaczej, $ed = 1 + h(p - 1)(q - 1)$ dla pewnej liczby całkowitej h. Jeśli $t \bmod p$ nie jest równe 0, to mamy, co następuje:

$$\begin{aligned} t^{ed} \bmod p &= (t \bmod p)^{ed} \bmod p \\ &= (t \bmod p)^{1+h(p-1)(q-1)} \bmod p \\ &= (t \bmod p) \cdot ((t \bmod p)^{p-1} \bmod p)^{h(q-1)} \bmod p \\ &= (t \bmod p) \cdot (1^{h(q-1)} \bmod p) \\ &= t \bmod p. \end{aligned}$$

Oczywiście, jeśli $t \bmod p$ równa się 0, to $t^{ed} \bmod p$ równa się 0.

W podobny sposób uzasadniamy, że jeśli $t \bmod q$ nie równa się 0, to $t^{ed} \bmod q$ równa się $t \bmod q$, a jeśli $t \bmod q$ jest zerem, to $t^{ed} \bmod q$ również jest zerem.

Aby zakończyć, potrzebujemy jeszcze jednego faktu z teorii liczb: ponieważ p i q są względnie pierwsze (każda jest liczbą pierwszą), więc jeśli zarówno $x \bmod p = y \bmod q$, jak i $x \bmod q = y \bmod q$, to $x \bmod pq = y \bmod pq$. (Ten fakt wynika z „chińskiego twierdzenia o resztach"). Podstawienie t^{ed} za x i t za y z uwzględnieniem, że $n = pq$ oraz że t jest mniejsze niż n, daje nam $t^{ed} \bmod n = t \bmod n = t$, czyli właśnie to, czego chcieliśmy dowieść. Uff!

Kryptosystemy hybrydowe

Choć potrafimy wykonywać działania arytmetyczne na dużych liczbach, w praktyce płacimy za to szybkością obliczeń. Szyfrowanie i deszyfrowanie długiego komunikatu, zawierającego setki lub tysiące bloków tekstu jawnego, mogłoby powodować widoczne opóźnienia. RSA jest często używany w systemie hybrydowym, będącym po części systemem z kluczem publicznym i po części systemem z kluczem symetrycznym.

Oto jak mógłbyś mi przesłać zaszyfrowaną wiadomość w systemie hybrydowym. Uzgadniamy, którego systemu z kluczem publicznym i którego systemu z kluczem symetrycznym będziemy używać. Powiedzmy, że będą to RSA i AES. Wybierasz klucz k dla AES i szyfrujesz go za pomocą mojego publicznego klucza RSA, tworząc $F_P(k)$. Kluczem k szyfrujesz następnie ciąg bloków tekstu jawnego za pomocą AES, wytwarzając bloki tekstu zaszyfrowanego. Wysyłasz mi $F_P(k)$ i ciąg bloków tekstu

zaszyfrowanego. Ja odszyfrowuję $F_P(k)$, obliczając F_S ($F_P(k)$), dzięki czemu otrzymuję klucz k szyfru AES, a następnie używam klucza k do odszyfrowania bloków tekstu zaszyfrowanego w AES, odzyskując w ten sposób bloki tekstu jawnego. Gdybyśmy korzystali z łańcuchowania bloków szyfru i potrzebowali wektora początkowego, mógłbyś go również zaszyfrować za pomocą RSA lub AES.

Obliczanie liczb losowych

Jak widzieliśmy, niektóre kryptosystemy wymagają od nas generowania liczb losowych — losowych dodatnich liczb całkowitych, wyrażając się ściślej. Ponieważ liczbę całkowitą przedstawiamy w postaci ciągu bitów, tym, czego nam naprawdę trzeba, jest generowanie bitów losowych, które można potem interpretować jako liczbę całkowitą.

Bity losowe mogą pochodzić tylko z procesów losowych. Jakim sposobem program działający w komputerze może stać się procesem losowym? Na ogół nie jest to możliwe, ponieważ program komputerowy, zbudowany z dobrze zdefiniowanych, deterministycznych instrukcji, będzie zawsze produkował ten sam wynik na podstawie tych samych danych początkowych. Aby wspomóc oprogramowanie kryptograficzne, niektóre nowoczesne procesory mają rozkazy generujące bity losowe na podstawie losowych procesów, takich jak szum termiczny w układach scalonych. Projektanci tych procesorów mierzą się z potrójnym wyzwaniem: generowaniem bitów wystarczająco szybko dla aplikacji, które żądają liczb losowych, zapewnieniem, że generowane bity spełniają testy statystyczne losowości, i zużywaniem rozsądnych ilości mocy podczas generowania i testowania losowych bitów.

Programy kryptograficzne zazwyczaj uzyskują bity z **generatora liczb pseudolosowych** (ang. *pseudorandom number generator*), czyli z **PRNG**. Generator PRNG jest programem deterministycznym, który wytwarza ciąg wartości na podstawie wartości początkowej, czyli **ziarna** (ang. *seed*), i deterministycznych reguł ujętych w programie, które określają, jak wygenerować następną wartość ciągu z wartości bieżącej. Jeśli za każdym razem uruchomisz PRNG z tym samym ziarnem, to otrzymasz zawsze ten sam ciąg wartości. To powtarzalne zachowanie jest dobre do wykrywania błędów, lecz złe dla kryptografii. Najnowsze standardy generatorów liczb losowych dla kryptosystemów wymagają specjalnych realizacji programów PRNG.

Jeśli używasz PRNG do generowania bitów sprawiających wrażenie losowych, to musisz zaczynać zawsze od różnego ziarna, które powinno być losowe. W szczególności ziarno powinno opierać się na bitach, które nie są tendencyjne (nie wykazują skłonności do preferowania 0 lub 1), niezależne (niezależnie od tego, co wiesz o poprzednio wygenerowanych bitach, masz tylko 50% szans na poprawne odgadnięcie następnego bitu) i nie do określenia przez przeciwnika, który próbuje złamać kryptosystem. Jeżeli Twój procesor ma rozkaz generujący bity losowe, to jest to dobry sposób utworzenia ziarna PRNG.

Co czytać dalej

Kryptografia stanowi jeden z elementów zapewniania bezpieczeństwa w systemach komputerowych. Książka Smitha i Marchesiniego [SM08] ujmuje zagadnienia bezpieczeństwa komputerowego szeroko, uwzględniając kryptografię i sposoby atakowania kryptosystemów.

Do głębszego zanurzenia się w kryptografii polecam książki Katza i Lindella [KL08] oraz Menezesa, van Oorschota i Vastone'a [MvOV96]. Rozdział 31 CLRS [CLRS] dostarcza elementarnych podstaw kilku teorii wiodących do kryptografii, a także opisów systemu RSA i testu prymarności Millera-Rabina. Diffie i Hellman [DH76] zaproponowali kryptografię z kluczem jawnym w 1976 roku, a oryginalny artykuł opisujący RSA, autorstwa Rivesta, Shamira i Adlemana [RSA78], ukazał się dwa lata później.

Więcej szczegółów na temat zaaprobowanych generatorów PRNG można znaleźć w dodatku C do Federal Information Processing Standards Publication 140-2 [FIP11]. O pewnej sprzętowej realizacji generatora liczb losowych opartej na szumie termicznym możesz poczytać w artykule Taylora i Coxa [TC11].

9 Kompresja danych

W poprzednim rozdziale zobaczyliśmy, jak przekształcać informacje, aby osłaniać je przed kimś, kto ma wobec nich wrogie zamiary. Ochrona informacji nie jest jednak jedynym powodem ich przekształcania. Czasami zależy Ci na ich ulepszeniu; na przykład chciałbyś wprowadzić zmiany na fotografii za pomocą narzędzia programowego typu Adobe Photoshop, aby usunąć efekt czerwonych oczu lub zmienić odcień skóry. Kiedy indziej chcesz je poszerzyć o pewną nadmiarowość, tak aby można było wykrywać i poprawiać partie uszkodzonych bitów.

W tym rozdziale prześledzimy inny sposób przekształcania informacji: ich kompresowanie. Zanim przejdziemy do pewnych metod stosowanych do kompresji i dekompresji informacji, musimy odpowiedzieć na trzy pytania:

1. *Dlaczego mielibyśmy kompresować informacje?*

   Zazwyczaj kompresujemy informacje z jednego z dwu powodów: aby zaoszczędzić na czasie lub na przestrzeni.

   Czas: im mniej bitów zostanie przetransmitowanych podczas przesyłania informacji siecią, tym szybsza będzie transmisja. Dlatego nadawca często kompresuje dane przed przesłaniem, wysyła dane skompresowane, a potem odbiorca dekompresuje otrzymane dane.

   Przestrzeń: kiedy ilość dostępnej pamięci zaczyna ograniczać ilość informacji, które możesz przechować, zdołasz zapamiętać więcej informacji, jeśli zostaną one skompresowane. Na przykład formaty MP3, JPEG służą do kompresji dźwięku i obrazów, przy czym jest to robione w taki sposób, że większość ludzi nie zauważa poważniejszych różnic — jeśli w ogóle jakieś wychwytuje — między materiałami oryginalnymi i skompresowanymi.

2. *Jaka jest jakość skompresowanej informacji?*

   Rozróżniamy bezstratne i stratne metody kompresji. W **kompresji bezstratnej** (kompresji bez strat, ang. *lossless compression*) skompresowana informacja jest po zdekompresowaniu identyczna z informacją oryginalną. W **kompresji stratnej** (kompresji ze stratami, ang. *lossy compression*) zdekompresowana informacja różni się od oryginalnej, lecz w przypadku idealnym — w znikomym stopniu. Kompresje MP3 i JPEG są stratne, natomiast metoda kompresji używana przez program zip jest bezstratna.

   Ogólnie rzecz ujmując, podczas kompresowania tekstu oczekujemy kompresji bez strat. Nawet różnica jednego bitu może mieć znaczenie. Następujące zdania różnią się zaledwie jednym bitem w kodzie ASCII liter, z których są utworzone:[1]

   ```
   Don't forget the pop.
   ```

[1] Kody ASCII liter `p` i `t` wynoszą, odpowiednio: 01110000 i 01110100.

```
Don't forget the pot.
```

Te zdania można zinterpretować jako przypomnienie[2] o lekkim drinku (przynajmniej na środkowym zachodzie USA) lub marihuanie — jeden bit robi dużą różnicę!

3. *Dzięki czemu kompresowanie informacji jest możliwe?*

   Na to pytanie łatwo odpowiedzieć w przypadku kompresji stratnej: po prostu tolerujesz spadek precyzji. A co z kompresją bez strat? Informacja cyfrowa często zawiera bity nadmiarowe lub zbyteczne. Na przykład w kodzie ASCII każdy znak zajmuje ośmiobitowy bajt, a wszystkie powszechnie używane znaki (nie włączając liter akcentowanych) mają 0 na najstarszym znaczącym (położonym na skrajnej lewej pozycji) bicie. Oznacza to, że znaki ASCII mają kody z przedziału od 0 do 255, lecz znaki powszechnie używane wpadają do przedziału od 0 do 127. W wielu wypadkach zatem jedna ósma bitów w tekście ASCII jest bezużyteczna, łatwo więc byłoby skompresować większość tekstów ASCII o 12,5%.

   Jako bardziej wyrazisty przykład wykorzystania nadmiarowości w kompresji bezstratnej rozważmy transmitowanie czarno-białego obrazu, jak to się dzieje w faksach. Faksy przesyłają obraz w postaci ciągu **peli** (ang. *pels*):[3] czarnych lub białych kropek, które tworzą obraz. Wiele faksów przesyła pele rzędami, od góry do dołu. Jeśli obraz zawiera głównie tekst, to większa jego część jest biała, więc każdy rząd zawiera prawdopodobnie wiele kolejnych białych peli. Jeśli rząd zawiera fragment poziomej czarnej linii, to może się składać z wielu kolejnych czarnych peli. Zamiast określać w odniesieniu do każdego w serii peli jego kolor, faksy kompresują informację, wskazując długość każdej serii i kolor peli w serii. Na przykład w jednym ze standardów faksu seria 140 białych peli jest kompresowana do jedenastu bitów: 10010001000.

Kompresja danych jest dobrze zbadaną dziedziną, tu zdołam poruszyć zaledwie niewielki jej fragment. Skupię się na kompresji bez strat, wszakże kilka dobrych odesłań do prac omawiających kompresję stratną znajdziesz w podrozdziale „Co czytać dalej".

W tym rozdziale, inaczej niż w poprzednich rozdziałach, nie będziemy się koncentrować na czasie działania. Będę o nim wspominał przy okazji, lecz znacznie bardziej niż czas kompresji i dekompresji będzie nas interesował rozmiar skompresowanej informacji.

Kody Huffmana

Powróćmy na chwilę do napisów reprezentujących DNA. Przypomnijmy za rozdziałem 7, że biolodzy reprezentują DNA w postaci napisów utworzonych z czterech znaków: `A`, `C`, `G` i `T`. Załóżmy, że otrzymaliśmy nić DNA reprezentowaną przez

[2] *Don't forget*, z ang.: nie zapomnij — *przyp. tłum.*

[3] Pele są podobne do piksli na ekranie. I „pel", i „piksel" są kuferkami z „elementem obrazu".

n znaków, przy czym 45% znaków to `A`, 5% to `C`, 5% przypada na `G` i 45% na `T`, lecz znaki występują w nici bez żadnego konkretnego uporządkowania. Gdybyśmy do przedstawienia tej nici użyli kodu ASCII, w którym każdy znak zajmuje osiem bitów, to reprezentacja całej nici zajęłaby 8*n* bitów. Jasne, że możemy zrobić to lepiej. Ponieważ reprezentujemy nici DNA z użyciem tylko czterech znaków, do zakodowania każdego znaku potrzebujemy naprawdę tylko dwóch bitów (00, 01, 10, 11), możemy więc zredukować przestrzeń do 2*n* bitów.

Możemy to jednak robić jeszcze lepiej, korzystając ze względnej częstości znaków. Zakodujmy znaki następującymi ciągami bitów: `A` = 0, `C` = 100, `G` = 101, `T` = 11. Częściej występujące znaki otrzymują krótsze ciągi bitów. 20-znakową nić `TAATTAGAAATTCTATTATA` moglibyśmy zakodować za pomocą 33-bitowego ciągu 11001111010100011111100110111110110. (Za chwilę zobaczymy, dlaczego wybrałem to szczególne kodowanie i jakie są jego właściwości). Znając częstości czterech znaków, do zakodowania *n*-znakowej nici potrzebujemy tylko $0{,}45 \cdot n \cdot 1 + 0{,}05 \cdot n \cdot 3 + 0{,}05 \cdot n \cdot 3 + 0{,}45 \cdot n \cdot 2 = 1{,}65n$ bitów. Zauważmy, że dla nici w powyższym przykładzie $33 = 1{,}65 \cdot 20$. Korzystając ze względnej częstości znaków, potrafimy osiągnąć lepszy rezultat niż 2*n* bitów!

W zastosowanym przez nas kodowaniu prócz tego, że częściej występujące znaki otrzymują krótsze ciągi bitów, jest jeszcze coś innego, co zasługuje na uwagę: żaden kod nie jest przedrostkiem innego kodu. Kod `A` równa się 0 i żaden inny kod nie zaczyna się od 0, kod `T` wynosi 11 i żaden inny kod nie zaczyna się od 11 — itd. Kod taki nazywamy **kodem bezprzedrostkowym** (ang. *prefix-free code*)[4].

Podstawowa zaleta kodów bezprzedrostkowych ujawnia się podczas dekompresji. Ponieważ żaden kod nie jest przedrostkiem innego kodu, możemy jednoznacznie skojarzyć skompresowane bity z ich oryginalnymi znakami, dekompresując je kolejno. Na przykład w skompresowanym ciągu 11001111010100011111100110111110110 żaden znak nie ma jednobitowego kodu 1 i tylko kod T zaczyna się od 11, wiemy więc, że pierwszym znakiem zdekompresowanego kodu musi być `T`. Po zdjęciu 11 zostaje 00111101010001111100110111110110. Tylko kod `A` zaczyna się od 0, zatem pierwszym znakiem w tym, co zostało, musi być `A`. Po zdjęciu 0, a potem bitów 011110, odpowiadających znakom `ATTA`, pozostają bity 10100011111100110111110110. Ponieważ tylko kod `G` zaczyna się od 101, następnym zdekompresowanym znakiem musi być `G`. I tak dalej.

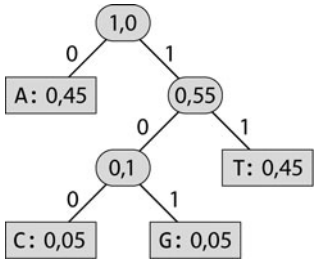
Jeśli mierzyć sprawność metod kompresji według średniej długości skompresowanej informacji, to z kodów bezprzedrostkowych najlepsze są kody Huffmana[5]. Pewną wadą tradycyjnego kodowania Huffmana jest to, że wymaga ono znajomości z góry częstości wszystkich znaków, dlatego kompresja często wymaga dwóch przejść przez nieskompresowany kod: jednego do wyznaczenia częstości znaków

[4] W CLRS nazywaliśmy to „kodami przedrostkowymi". Obecnie preferuję odpowiedniejszy termin: „bezprzedrostkowy" [a my nie nadużywamy zbędnych zapożyczeń (tu: prefiks) — *przyp. tłum.*].

[5] Nazwane tak od nazwiska wynalazcy, Davida Huffmana.

i drugiego do odwzorowania każdego znaku na jego kod. Nieco później zobaczymy, jak uniknąć pierwszego przebiegu kosztem dodatkowych obliczeń.

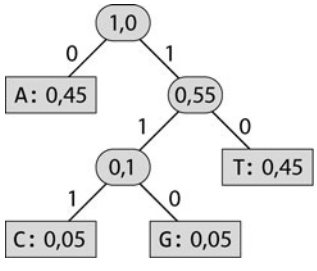
Kiedy już znamy częstości znaków, metoda Huffmana zasadza się na zbudowaniu drzewa binarnego. Drzewo to mówi nam, jak tworzyć kody, i jest również przydatne w procesie dekompresji. Dla naszego przykładu z kodowaniem DNA drzewo wygląda tak:



Liście drzewa, narysowane w postaci prostokątów, reprezentują znaki, obok których podano częstości ich występowania. Węzły, które nie są liśćmi, czyli **węzły wewnętrzne** (ang. *internal nodes*), są narysowane z zaokrąglonymi narożnikami, przy czym każdy węzeł wewnętrzny zawiera sumę częstości w liściach poniżej niego. Niedługo zobaczymy, dlaczego opłaca się zapamiętywać częstości w węzłach wewnętrznych.

Obok każdej krawędzi drzewa występuje 0 albo 1. Aby określić kod znaku, postępujemy ścieżką od korzenia w dół, do liścia znaku, i złączamy bity wzdłuż ścieżki. Na przykład, aby określić kod `G`, zaczynamy od korzenia i najpierw bierzemy krawędź z etykietą 1, prowadzącą do jego prawego potomka. Następnie wybieramy krawędź z etykietą 0, do lewego potomka (węzeł wewnętrzny z częstością 0,1) i na koniec krawędź z etykietą 1, wiodącą do prawego potomka (liść zawierający `G`). Złączenie tych bitów daje dla `G` kod 101.

Choć zawsze oznaczałem zerami krawędzie prowadzące do lewego potomka, a jedynkami te, które prowadziły do prawego, etykiety te nie są istotne. Mógłbym z powodzeniem nadać krawędziom etykiety w następujący sposób:



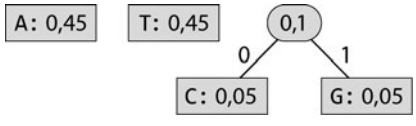
Na podstawie tego drzewa powstałyby następujące kody: `A` = 0, `C` = 111, `G` = 110, `T` =10. Byłyby one również bezprzedrostkowe, a liczba bitów w poszczególnych kodach byłaby taka sama jak poprzednio. Wynika to z faktu, że liczba bitów w kodzie

znaku jest taka sama jak **głębokość** (ang. *depth*) liścia znaku, tzn. liczba krawędzi na ścieżce od korzenia do liścia. Życie jest jednak prostsze, jeśli zawsze nadajemy krawędziom do lewych potomków etykietę 0, a krawędziom do prawych potomków — etykietę 1.

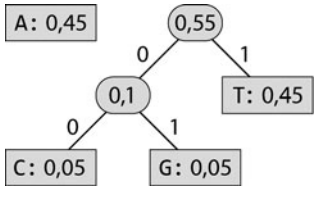
Kiedy już znamy częstości znaków, budujemy drzewo binarne od dołu do góry. Zaczynamy od poszczególnych z *n* liści, odpowiadających nie skompresowanym znakom, traktując je jak indywidualne drzewa; tak więc na początku każdy liść jest również korzeniem. Potem powtarzamy poszukiwanie dwóch węzłów korzeniowych o najniższych częstościach, tworzymy nowy korzeń z tymi węzłami w charakterze jego potomków i umieszczamy w tym nowym korzeniu sumę częstości jego potomków. Ten proces jest kontynuowany dotąd, aż wszystkie liście znajdą się pod jednym korzeniem. W miarę jego postępowania nadajemy każdej krawędzi do lewego potomka etykietę 0, a każdej krawędzi do prawego potomka — etykietę 1, choć po wybraniu obu korzeni z najmniejszymi częstościami nie ma znaczenia, który uczynimy lewym, a który prawym potomkiem nowego korzenia.

Oto jak przebiega ten proces dla naszego przykładu z DNA. Zaczynamy od czterech węzłów, każdy liść reprezentuje jeden znak:

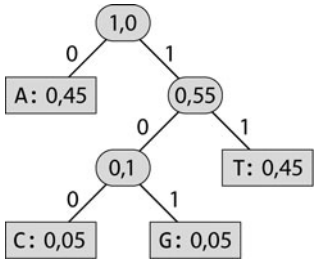
Węzły `C` i `G` mają najmniejsze częstości, tworzymy więc nowy węzeł, dane dwa węzły czynimy jego potomkami i nadajemy mu ich połączone częstości:



Z trzech pozostałych korzeni ostatni przez nas utworzony ma najmniejszą częstość (0,1), a oba pozostałe mają częstości 0,45. Jako następny korzeń możemy wybrać dowolny z tych dwu; wybieramy ten z `T` i czynimy z niego i z korzenia z częstością 0,1 potomków nowego węzła, którego częstość jest sumą tamtych: 0,55.



Zostały tylko dwa korzenie. Tworzymy nowy węzeł i czynimy z nich jego potomków, a za jego częstość (której nie potrzebujemy, gdyż zbliżamy się do końca) obieramy sumę ich częstości: 1,0.



Obecnie, gdy wszystkie liście znajdują się pod tym nowym korzeniem, budowa naszego drzewa dobiegła końca.

Aby rzecz nieco uściślić, zdefiniujemy procedurę budowania tego drzewa binarnego. Procedura BUDOWANIE-DRZEWA-HUFFMANA przyjmuje na wejściu dwie tablice *n*-elementowe: *znak* i *częst*, przy czym *znak*[*i*] zawiera *i*-ty nieskompresowany znak, a *częst*[*i*] podaje częstość tego znaku. Procedura przyjmuje także wartość wejściową *n*. Aby znaleźć dwa korzenie z najmniejszymi częstościami, procedura wywołuje procedury WSTAW i WYDOBĄDŹ-MIN dla kolejki priorytetowej.

Procedura BUDOWANIE-DRZEWA-HUFFMANA(*znak*, *częst*, *n*)

Dane wejściowe:

- *znak*: tablica *n* nieskompresowanych znaków.
- *częst*: tablica *n* częstości znaków.
- *n*: rozmiary tablic *znak* i *częst*.

Wynik: korzeń drzewa binarnego skonstruowanego dla kodów Huffmana.

1. Niech *Q* będzie pustą kolejką priorytetową.
2. Dla *i* = 1 do *n*:
   A. Zbuduj nowy węzeł *z* zawierający *znak*[*i*] i którego częstość wynosi *częst*[*i*].
   B. Wywołaj WSTAW(*Q*, *z*).
3. Dla *i* = 1 do *n* – 1:
   A. Wywołaj WYDOBĄDŹ-MIN(*Q*) i nadaj *x* wartość wydobytego węzła.
   B. Wywołaj WYDOBĄDŹ-MIN(*Q*) i nadaj *y* wartość wydobytego węzła.
   C. Zbuduj nowy węzeł *z*, którego częstość jest sumą częstości *x* i *y*.
   D. Uczyń *x* lewym potomkiem *z*, a *y* prawym potomkiem *z*.
   E. Wywołaj WSTAW(*Q*, *z*).
4. Wywołaj WYDOBĄDŹ-MIN(*Q*) i zwróć wydobyty węzeł.

Po osiągnięciu przez procedurę kroku 4 w kolejce priorytetowej pozostaje tylko jeden węzeł i jest nim korzeń całego drzewa binarnego.

Możesz prześledzić działanie tej procedury na drzewach binarnych z poprzedniej strony. Korzenie w kolejce priorytetowej na początku każdej iteracji pętli w kroku 3 występują u góry każdego rysunku.

Przeanalizujmy szybko czas działania procedury BUDOWANIE-DRZEWA-HUFFMANA. Przy założeniu, że kolejka priorytetowa jest zrealizowana za pomocą kopca binarnego, każda operacja WSTAW i WYDOBĄDŹ-MIN zużywa $O(\lg n)$ czasu. Procedura wywołuje każdą z tych operacji $2n - 1$ razy, co daje w sumie czas $O(n \lg n)$. Cała pozostała robota zajmuje $\Theta(n)$ czasu, więc BUDOWANIE-DRZEWA-HUFFMANA działa w czasie $O(n \lg n)$.

Wspomniałem wcześniej, że do dekompresji dobrze jest mieć drzewo binarne skonstruowane przez BUDOWANIE-DRZEWA-HUFFMANA. Zaczynając od korzenia drzewa binarnego, podróżujemy w dół drzewa zgodnie z bitami skompresowanej informacji. Zdejmujemy każdy bit, idąc w lewo, jeśli jest zerem, lub w prawo, jeśli jest jedynką. Po dotarciu do liścia zatrzymujemy się, emitujemy znak, po czym wznawiamy przeszukiwanie od korzenia. Wracając do przykładu z DNA, podczas dekompresowania ciągu bitów 1100111101010001111100111011110110 zdejmujemy pierwszą jedynkę i ruszamy z korzenia w prawo, potem zdejmujemy drugą jedynkę i znów idziemy w prawo, docierając do liścia z `T`. Wyprowadzamy `T` i wznawiamy przeszukiwanie od korzenia. Zdejmujemy następny bit, 0, i idziemy od korzenia w lewo, napotykając liść z `A`. Wyprowadzamy go na zewnątrz i wracamy do korzenia. Dekompresja jest w ten sposób kontynuowana, aż zostaną przetworzone wszystkie bity skompresowanej informacji.

Jeżeli drzewo binarne mamy już zbudowane przed dekompresją, to w jej trakcie przetworzenie każdego bitu zajmuje stały czas. Zatem w jaki sposób proces dekompresji uzyskuje dostęp do drzewa binarnego? Jedną z możliwości jest włączenie reprezentacji drzewa binarnego do skompresowanych informacji. Inna możliwość polega na dołączeniu tabeli dekodowania z przetworzonymi informacjami. Każdy wpis tabeli zawierałby znak, liczbę bitów w jego kodzie i sam kod. Na podstawie tej tabeli można zbudować drzewo binarne w czasie liniowym względem łącznej liczby bitów we wszystkich kodach.

Procedura BUDOWANIE-DRZEWA-HUFFMANA służy za przykład **algorytmu zachłannego** (ang. *greedy algorithm*), w którym podejmujemy decyzje wyglądające na najlepsze w danej chwili. Ponieważ chcemy, aby znaki o najmniejszych częstościach występowały daleko od korzenia drzewa binarnego, podejście zachłanne zawsze prowadzi do wybrania pod nowym węzłem — który może się stać później potomkiem innego węzła — dwóch korzeni o najmniejszej częstości. Algorytm Dijkstry jest przykładem innego algorytmu zachłannego, ponieważ zawsze osłabia krawędzie [wychodzące] z wierzchołka o najmniejszej wartości *najkrótsza* z tych, które pozostają w jego kolejce priorytetowej.

Zaimplementowałem kodowanie Huffmana i wykonałem je na dostępnej online wersji *Moby Dicka*. Oryginalny tekst książki liczy 1 193 826 bajtów, a wersja skompresowana zajęła tylko 673 579 bajtów, czyli 56,42% rozmiaru oryginału, nie wliczając samego kodowania. Ujmując to inaczej, zakodowanie każdego znaku zajęło średnio tylko 4,51 bitu. Nic dziwnego — najczęściej występującym znakiem była spacja (15,96%), a po niej litera `e` (9,56%). Najrzadziej występującymi znakami, pojawiającymi się tylko po dwa razy, były `$`, `&`, `[` i `]`.

Adaptacyjne kody Huffmana

Praktycy często dochodzą do wniosku, że wykonywanie dwóch przebiegów przetwarzania danych wejściowych: jednego w celu obliczenia częstości znaków i drugiego w celu zakodowania znaków, jest zbyt wolne. Zamiast tego programy kompresji i dekompresji działają adaptacyjnie, uaktualniając częstości znaków i drzewo binarne w trakcie kompresowania lub dekompresowania w jednym przebiegu.

Program kompresji zaczyna od pustego drzewa binarnego. Każdy czytany przez niego znak z wejścia jest albo nowy, albo znajduje się już w drzewie binarnym. Jeśli znak występuje już w drzewie binarnym, to program kompresji emituje kod znaku zgodnie z bieżącym drzewem binarnym, zwiększa częstość znaku i — w razie konieczności — uaktualnia drzewo binarne, aby odzwierciedlić nową częstość. Jeśli znaku nie było dotąd w drzewie binarnym, to program kompresji emituje znak *od*kodowany (w obecnej postaci), dodaje go do drzewa binarnego i odpowiednio je uaktualnia.

Program dekompresji wykonuje robotę bliźniaczo podobną do działania programu kompresji. Również utrzymuje drzewo binarne w trakcie przetwarzania skompresowanej informacji. Napotykając bity znaku, podąża w drzewie binarnym w dół, aby określić, jaki znak te bity kodują, wyprowadza ten znak, zwiększa częstość znaku i uaktualnia drzewo binarne. Widząc, że znaku nie ma jeszcze w drzewie, program dekompresji wyprowadza znak, dodaje go do drzewa binarnego i uaktualnia je.

Coś tu jednak nie gra. Bity są bitami niezależnie od tego, czy reprezentują znaki ASCII, czy bity w kodzie Huffmana. W jaki sposób program dekompresji ustala, że oglądane przez niego bity reprezentują zakodowany lub niezakodowany znak? Czy ciąg bitów 101 przedstawia znak aktualnie zakodowany jako 101, czy jest początkiem ośmiobitowego, zdekodowanego znaku? Rozwiązanie polega na poprzedzeniu każdego zdekodowanego znaku **kodem sygnałowym** (ang. *escape code*), tj. specjalnym kodem wskazującym, że następny komplet bitów reprezentuje znak zdekodowany. Jeśli tekst oryginalny zawiera k różnych znaków, to w skompresowanej informacji pojawi się tylko k kodów sygnałowych, z których każdy poprzedzi pierwsze wystąpienie znaku. Kody sygnałowe będą na ogół występować rzadko, nie musimy więc czynić starań o nadawanie im krótkich sekwencji bitowych kosztem częściej występujących znaków. Dobrym sposobem zadbania o to, aby kody sygnałowe nie były krótkie, jest włączenie znaku o kodzie sygnałowym do drzewa binarnego, lecz z przypisaniem mu na stałe częstości 0. Podczas aktualizowania drzewa binarnego ciąg bitów kodu sygnałowego będzie się zmieniał zarówno w czasie kompresji, jak i dekompresji, lecz jego liść pozostanie zawsze możliwie najdalej od korzenia.

Faksy

Wspomniałem uprzednio, że faksy kompresują informację, wskazując kolory i długości serii identycznych peli w rzędach (rządkach) przesyłanego obrazu. Ten schemat nosi nazwę **kodowanie seria-długość** (ang. *run-length encoding*). Faksy łączą

kodowanie seria-długość z kodami Huffmana. W standardzie faksowania używanym w zwykłych liniach telefonicznych 104 kody określają różne długości białych peli i 104 kody określają różne długości czarnych peli. Kody serii białych peli są bezprzedrostkowe i to samo odnosi się do kodów czarnych peli, aczkolwiek niektóre z kodów serii białych peli są przedrostkami kodów serii czarnych peli i odwrotnie.

Aby określić, które kody stosować do których serii, w komitecie standaryzacyjnym wybrano zestaw ośmiu reprezentatywnych dokumentów i policzono, jak często dana seria się pojawia. Następnie zbudowano dla tych serii kody Huffmana. Serie o największych częstościach, a więc mające najkrótsze kody, składały się z dwóch, trzech lub czterech czarnych peli; stosownie do tego przypisano im kody 11, 10 i 011. Do typowych serii zaliczono jeszcze: jeden czarny pel (010), pięć i sześć czarnych peli (0011 i 0010), dwa do siedmiu białych peli (każdy z kodem czterobitowym) i inne, względnie krótkie serie. Jedna z dość częstych serii składała się z 1664 białych peli i reprezentowała cały rząd białych peli. Inne krótkie kody przypisano seriom białych peli o długościach będących potęgami 2 lub sumami dwóch potęg 2 (jak 192, równe $2^7 + 2^6$). Serie mogą być kodowane przez złączanie kodowań krótszych serii. Wcześniej podałem przykład kodu serii złożonej ze 140 białych peli, 10010001000. Ten kod jest w rzeczywistości złączeniem kodów serii 128 białych peli (10010) i serii 12 białych peli (001000).

Prócz kompresowania informacji tylko w obrębie każdego rzędu obrazu niektóre faksy kompresują obraz w obu wymiarach. Serie peli tego samego koloru mogą występować pionowo oraz poziomo, to znaczy zamiast traktowania każdego rzędu w sposób wyizolowany, koduje się go na zasadzie wynajdywania różnic z rzędem poprzednim. Większość rzędów różni się od poprzednich zaledwie garstką peli. Z tym schematem wiąże się ryzyko przenoszenia się błędów: błąd kodowania lub transmisji powoduje, że kilka kolejnych rzędów staje się niepoprawnych. Z tego powodu faksy stosujące ten schemat i przesyłające dane za pośrednictwem linii telefonicznych ograniczają liczbę kolejnych rzędów, w których można go użyć, tak że po pewnej liczbie rzędów transmitują pełny rząd obrazu, używając schematu kodowania Huffmana zamiast przesyłania tylko różnic w stosunku do poprzedniego rzędu.

Kompresja LZW

W innym podejściu do kompresji bezstratnej, zwłaszcza w przypadku tekstów, wykorzystuje się informacje powtarzające się w tekście niekoniecznie w miejscach następujących po sobie. Rozważmy słynny cytat z przemówienia inauguracyjnego Johna F. Kennedy'ego:

Ask not what your country can do for you — ask what you can do for your country[6].

Z wyjątkiem słowa *not*, każde słowo w cytacie występuje dwukrotnie. Załóżmy, że sporządziliśmy tabelę słów:

[6] Z ang.: nie pytaj, co twój kraj może zrobić dla ciebie — pytaj, co ty możesz zrobić dla swojego kraju — *przyp. tłum.*

Indeks	Słowo
1	ask
2	not
3	what
4	your
5	country
6	can
7	do
8	for
9	you

Moglibyśmy wtedy zakodować ten cytat (pomijając pisownię dużymi literami i interpunkcję) w postaci:

1 2 3 4 5 6 7 8 9 1 3 9 6 7 8 4 5

Ponieważ cytat zawiera niewiele słów, a bajt może przechowywać liczby całkowite z przedziału od 0 do 255, możemy zapamiętać każdy indeks w jednym bajcie. W ten sposób możemy zapamiętać ten cytat już w 17 bajtach, po jednym bajcie na słowo plus miejsce potrzebne, abyśmy mogli zapamiętać tabelę. Oryginalny cytat pamiętany po jednym znaku w bajcie wymaga (bez znaków interpunkcyjnych, lecz z odstępami między słowami) 77 bajtów.

Oczywiście przestrzeń potrzebna do zapamiętania tabeli ma znaczenie, w przeciwnym razie moglibyśmy po prostu ponumerować wszystkie możliwe słowa i skompresować plik przez zapamiętanie tylko indeksów słów. Dla niektórych słów ten schemat powoduje rozszerzenie miast kompresji. Dlaczego? Bądźmy ambitni i załóżmy, że słów jest mniej niż 2^{32}, więc każdy indeks możemy zapamiętać w słowie 32-bitowym[7]. Moglibyśmy reprezentować każde słowo za pomocą czterech bajtów, wówczas jednak schemat ten przegrałby ze słowami trzyliterowymi lub krótszymi, które wymagają tylko po jednym bajcie na literę do zapamiętania w postaci nieskompresowanej.

Prawdziwa trudność w ponumerowaniu wszystkich możliwych słów tkwi jednak w tym, że rzeczywiste teksty zawierają „słowa", które nie są słowami, a przynajmniej nie w języku angielskim. Jako skrajny przykład rozważmy początkowy czterowiersz z poematu Lewisa Carrolla *Jabberwocky*:

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe[8].

Weź również pod uwagę programy komputerowe, w których często używa się nazw zmiennych niebędących angielskimi słowami. Dorzuć duże i małe litery,

[7] Tu: w słowie maszynowym; zarówno w języku polskim, jak angielskim wyraz słowo (ang. *word*) ma w informatyce różne przypisania — *przyp. tłum.*

[8] Prawie wszystkie słowa w tym czterowierszu są zmyślone — *przyp. tłum.*

interpunkcję i *naprawdę* wielkoformatowe nazwy[9], a zobaczysz, że gdybyśmy próbowali kompresować tekst przez ponumerowanie wszelkich możliwych słów, brnęlibyśmy w kierunku używania *mnóstwa* indeksów. Z pewnością więcej niż 2^{32} i — zważywszy że w tekście *mogłaby* się pojawić każda kombinacja znaków — w rzeczywistości ilość nieograniczona.

Jednak nie wszystko stracone, ponieważ wciąż możemy zrobić użytek z tego, że informacje się powtarzają. Wystarczy, abyśmy nie czepiali się aż tak bardzo powtarzanych *słów*. Przydać się może dowolny powtarzający się ciąg znaków. Na powtarzających się ciągach znaków opiera się kilka schematów kompresji. Ten, któremu się przejrzymy, jest znany pod nazwą **LZW**[10] i leży u podstaw wielu programów kompresji używanych w praktyce.

LZW działa na kompresowanych i dekompresowanych danych jednoprzebiegowo. W obu przypadkach buduje słownik oglądanych przez siebie ciągów znaków i używa słownikowych indeksów do ich reprezentowania. Pomyśl o słowniku jak o tablicy napisów. Tablicę tę możemy indeksować, wolno więc nam mówić o *i*-tym elemencie. Bliżej początku danych wejściowych ciągi są raczej krótkie i reprezentowanie ich za pomocą indeksów mogłoby nawet powodować rozrost zamiast kompresji. Jednak w miarę pokonywania przez LZW kolejnych partii danych ciągi w słowniku się wydłużają i reprezentowanie ich przez indeksy zaczyna oszczędzać sporo miejsca. Potraktowałem na przykład *Moby Dicka* kompresorem LZW, a on wyprodukował na wyjściu indeks reprezentujący 10-znakowy ciąg *_from_the_* 20 razy (każdy znak _ symbolizuje jedną spację). Wyprowadził również indeks reprezentujący ośmioznakowy ciąg *_of_the_* 33-krotnie.

Zarówno kompresor, jak i dekompresor przesiewają słownik jednoznakowym ciągiem dla każdego znaku ze zbioru znaków. W wypadku użycia pełnego zbioru znaków ASCII słownik zaczyna się od 256 ciągów jednoznakowych; *i*-ty wpis w słowniku zawiera znak, którego kod ASCII równa się *i*.

Zanim przejdziemy do ogólnego opisu działania kompresora, zwróćmy uwagę na kilka obsługiwanych przez niego sytuacji. Kompresor buduje napisy, wstawiając je do słownika i produkując na wyjściu indeksy do słownika. Załóżmy, że kompresor zaczyna budować napis ze znakiem `T`, który przeczytał z wejścia. Ponieważ słownik zawiera każdy ciąg jednoznakowy, kompresor znajduje `T` w słowniku. Ilekroć kompresor znajdzie w słowniku napis, który buduje, pobiera następny znak z wejścia i dołącza go do budowanego napisu. Przypuśćmy więc, że następnym znakiem wejściowym jest `A`. Kompresor dołącza `A` do budowanego napisu, powstaje więc napis `TA`. Załóżmy, że `TA` również jest w słowniku. Kompresor czyta więc następny znak z wejścia — niech to będzie `G`. Dodaje `G` do budowanego napisu, tworząc `TAG`. Tym razem załóżmy, że `TAG` *nie* występuje w słowniku. Kompresor wykonuje trzy czynności: (1) wyprowadza indeks słownikowy napisu `TA`; (2) wstawia napis `TAG` do słownika i (3) rozpoczyna składanie nowego napisu, początkowo zawierającego tylko znak (`G`), który spowodował, że napisu `TAG` nie odnaleziono w słowniku.

---

[9] Takie jak Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch, walijska wieś.

[10] Jak zapewne zgadujesz, nazwa upamiętnia wynalazców: twórcą LZW jest Terry Welch, który zmodyfikował schemat kompresji LZ78 zaproponowany przez Abrahama Lempela i Jacoba Ziva.

Jak ten kompresor działa w ogólnym przypadku? Produkuje ciąg indeksów do słownika. Złączenie napisów spod tych indeksów daje tekst oryginalny. Kompresor buduje napisy w słowniku po jednym znaku, więc zawsze gdy wstawia napis do słownika, napis ów jest taki jak pewien napis już występujący w słowniku, lecz wydłużony o jeden znak. Kompresor obrabia napis złożony z *s* kolejnych znaków z wejścia, utrzymując niezmienniczo, że słownik zawsze zawiera *s* w którymś z wpisów. Nawet jeśli *s* jest pojedynczym znakiem, pojawia się w słowniku, ponieważ słownik jest przesiewany z użyciem ciągu jednoznakowego dla każdego znaku ze zbioru znaków. Na początku *s* jest po prostu pierwszym znakiem danych wejściowych. Po przeczytaniu nowego znaku *c* kompresor sprawdza, czy napis *s c*, utworzony z dołączenia *c* na końcu *s*, jest obecny w słowniku. Jeśli jest, to dołącza *c* na końcu *s* i określa ten wynik jako *s*. Innymi słowy, nadaje *s* wartość *s c*. Kompresor buduje dłuższy napis, który w końcu wstawi do słownika. W przeciwnym razie *s* jest w słowniku, lecz nie ma w nim *s c*. W takim przypadku kompresor wyprowadza słownikowy indeks *s*, wstawia *s c* na następne wolne miejsce w słowniku i nadaje *s* wartość znaku wejściowego *c*. Przez wstawienie *s c* do słownika kompresor dodał napis, który wydłuża *s* o jeden znak, a po nadaniu *s* wartości *c* wznawia proces budowania napisu do poszukiwania w słowniku. Ponieważ *c* w słowniku jest napisem jednoznakowym, kompresor utrzymuje niezmiennik polegający na tym, że *s* występuje gdzieś w słowniku. Po wyczerpaniu danych wejściowych kompresor wyprowadza indeks napisu pozostałego w *s*.

Procedurę KOMPRESOR-LZW zamieszczono na następnej stronie. Przeanalizujmy przykład, dokonując kompresji tekstu TATAGATCTTAATATA. (Wystąpi tu ciąg TAG, który widzieliśmy na poprzedniej stronie). W poniższej tabeli przedstawiono, co się dzieje w każdej iteracji pętli w kroku 3. Uwidocznione wartości napisu *s* dotyczą początku iteracji.

Iteracja	*s*	*c*	Wyjście	Nowy napis w słowniku
1	T	A	84 (T)	256: TA
2	A	T	65 (A)	257: AT
3	T	A		
4	TA	G	256 (TA)	258: TAG
5	G	A	71 (G)	259: GA
6	A	T		
7	AT	C	257: (AT)	260: ATC
8	C	T	67 (C)	261: CT
9	T	T	84 (T)	262: TT
10	T	A		
11	TA	A	256 (TA)	263: TAA
12	A	T		
13	AT	A	257 (AT)	264: ATA
14	A	T		
15	AT	A		
krok 4	ATA		264 (ATA)	

Procedura KOMPRESOR-LZW(*tekst*)

Dane wejściowe: ciąg znaków ze zbioru stanowiącego kod ASCII.

Wynik: ciąg indeksów do słownika.

1. Dla każdego znaku *c* w zbiorze znaków ASCII:
   A. Wstaw *c* do słownika pod indeksem równym numerycznej wartości jego kodu ASCII.
2. Podstaw pod *s* pierwszy znak z *tekstu*.
3. Dopóki *tekst* nie jest wyczerpany, wykonuj, co następuje:
   A. Pobierz następny znak z *tekstu* i przypisz go do *c*.
   B. Jeśli *s c* jest w słowniku, to nadaj *s* wartość *s c*.
   C. W przeciwnym razie (*s c* nie ma jeszcze w słowniku) wykonaj, co następuje:
      i. Wyprowadź indeks przyporządkowany *s* w słowniku.
      ii. Wstaw *s c* na następne wolne miejsce w słowniku.
      iii. Nadaj *s* wartość jednoznakowego napisu *c*.
4. Wyprowadź indeks przyporządkowany *s* w słowniku.

Po kroku 1 słownik zawiera napisy jednoznakowe dla każdego z 256 znaków ASCII, znajdują się one na pozycjach od 0 do 255. W kroku drugim napis *s* przyjmie wartość pierwszego znaku z wejścia, czyli `T`. W pierwszej iteracji pętli głównej z kroku 3 *c* jest następnym znakiem wejściowym, tzn. `A`. Złączenie *s c* stanowi napis `TA`, którego jeszcze nie ma w słowniku, więc zaczyna się krok 3C. Ponieważ napis *s* zawiera tylko `T`, a kod ASCII `T` wynosi 84, w kroku 3Ci jest wyprowadzany indeks 84. W kroku 3Cii napis `TA` jest wstawiany na następne wolne miejsce w słowniku, znajdujące się pod indeksem 256, a w kroku 3Ciii następuje wznowienie budowania *s*, zainicjowane podstawieniem pod *s* znaku `A`. W drugiej iteracji pętli z kroku 3 *c* jest następnym znakiem z wejścia, czyli `T`. Napisu *s c* = `AT` nie ma w słowniku, więc w kroku 3C wyprowadza się indeks 65 (kod ASCII litery `A`), wstawia napis `AT` na pozycję 257 i podstawia do *s* znak `T`.

Przy okazji dwóch następnych iteracji pętli w kroku 3 dostrzegamy zaletę słownika. W trzeciej iteracji *c* przyjmuje wartość następnego znaku z wejścia, tj. `A`. Napis *s c* = `TA` jest już teraz obecny w słowniku, więc procedura nie wyprowadza niczego. Zamiast tego w kroku 3B dołącza znak z wejścia na koniec *s*, tworząc z *s* napis `TA`. W czwartej iteracji *c* staje się równe `G`. Napisu *s c* = `TAG` nie ma w słowniku, więc w kroku 3Ci jest wyprowadzany indeks słownikowy 256 napisu *s*. Jedna wyprowadzana liczba stanowi nie jeden, lecz dwa znaki: `TA`.

Nie każdy indeks słownikowy jest wyprowadzany do czasu zakończenia działania KOMPRESORA-LZW, a niektóre mogą być wyprowadzone więcej niż raz. Jeśli połączysz wszystkie znaki w nawiasach podane w kolumnie Wyjście, to otrzymasz tekst oryginalny: `TATAGATCTTAATATA`.

Ten przykład jest trochę za mały, aby ukazać prawdziwą korzyść wynikającą z kompresji LZW. Dane wejściowe zajmują 16 bajtów, a wyjście składa się z 10 indeksów słownikowych. Każdy indeks wymaga więcej niż jednego bajta. Nawet gdybyśmy użyli dwóch bajtów na indeks w wynikach, to zajęłoby to 20 bajtów. Gdyby każdy indeks zajmował 4 bajty, wyniki miałyby rozmiar 40 bajtów.

Dłuższe teksty wykazują lepsze rezultaty. Kompresja LZW zmniejsza rozmiar *Moby Dicka* z 1 193 826 do 919 012 bajtów. W tym wypadku słownik składa się z 230 007 wpisów, indeksy muszą więc być przynajmniej czterobajtowe[11]. Wyjście składa się z 229 753 indeksów, czyli 919 012 bajtów. Jest to wynik słabszy od uzyskanego za pomocą kodowania Huffmana (673 579 bajtów), lecz niedługo zapoznamy się z pewnymi pomysłami poprawienia tej kompresji.

Kompresja LZW jest przydatna tylko wówczas, gdy potrafimy również dekompresować. Na szczęście słownik nie musi być pamiętany wraz ze skompresowanymi informacjami. (Gdyby tak było, wynik kompresji LZA wraz ze słownikiem byłby większy od oryginalnego tekstu, chyba że tekst oryginalny zawierałby mnóstwo powtarzających się napisów). Jak już wspomniano, dekompresja LZW odbudowuje słownik wprost z informacji skompresowanych.

Oto jak działa dekompresja LZW. Na podobieństwo kompresora dekompresor przesiewa słownik za pomocą 256 ciągów jednoznakowych odpowiadających zbiorowi znaków ASCII. Czyta on z wejścia ciąg indeksów do słownika i wykonuje czynności będące lustrzanym odbiciem wykonywanych przez kompresor podczas budowania słownika. Ilekroć produkuje coś na wyjściu, pochodzi to z napisu, który dodał do słownika.

W większości wypadków następny indeks słownikowy na wejściu odnosi się do wpisu już występującego w słowniku (zaraz zobaczymy, co się dzieje w pozostałych przypadkach), toteż dekompresor LZW znajduje napis pod indeksem w słowniku i wyprowadza go. Jak jednak buduje słownik? Zastanówmy się przez chwilę, jak działa kompresor. Gdy wyprowadza indeks w ramach kroku 3C, zauważa, że choć napis *s* jest w słowniku, napisu *s c* tam nie ma. Wyprowadza indeks dotyczący *s* ze słownika, wstawia do słownika *s c* i zaczyna budować nowy napis do zapamiętania, zaczynając od *c*. Dekompresor ma iść jego śladem. Dla każdego indeksu, który pobiera z wejścia, wyprowadza napis *s* spod tego indeksu ze słownika. Wie również, że gdy kompresor wyprowadzał indeks dotyczący *s*, nie miał w słowniku napisu *s c*, gdzie przez *c* rozumie się znak występujący bezpośrednio po *s*. Dekompresor wie, że kompresor wstawił napis *s c* do słownika, pozostaje mu więc, koniec końców, zrobić to samo. Nie może jeszcze wstawić *s c*, ponieważ nie widział znaku *c*. Pojawi się on jako pierwszy znak następnego napisu, który dekompresor wyprowadzi. Ale dekompresor nie ma jeszcze następnego napisu. Dlatego musi śledzić dwa kolejne napisy, które wyprowadza. Jeśli dekompresor wyprowadza kolejno napisy *X* i *Y*, to łączy pierwszy znak *Y* z *X*, po czym wstawia wynikowy napis do słownika.

[11] Zakładam, że reprezentujemy liczby całkowite w standardowy dla komputerów sposób, gdzie zajmują one jeden, dwa, cztery lub osiem bajtów. Teoretycznie moglibyśmy reprezentować indeksy sięgające wartości 230 007, używając trzech bajtów, toteż wyniki zajęłyby wówczas 689 259 bajtów.

Spójrzmy na przykład odnoszący się do pokazanej wcześniej tabeli, który ilustruje działanie kompresora na napisie TATAGATCTTAATATA. W iteracji 11 kompresor wyprowadza indeks 256 napisu TA i wstawia do słownika napis TAA. Jest tak, ponieważ w danym momencie kompresor miał już w słowniku *s* = TA, lecz nie *s c* = TAA. To ostatnie A zaczyna następny napis wyprowadzany przez kompresor (AT, indeks 257) w iteracji 13. Dlatego gdy dekompresor widzi indeksy 256 i 257, powinien wyprowadzić TA, a także powinien zapamiętać ten napis, aby podczas wyprowadzania AT mógł dołączyć A pobrane z AT do TA i wstawić wynikowy napis TAA do słownika.

W rzadkich wypadkach następny indeks słownikowy na wejściu dekompresora dotyczy wpisu jeszcze nieobecnego w słowniku. Ta sytuacja powstaje tak rzadko, że podczas dekompresji *Moby Dicka* wystąpiła raptem piętnastokrotnie na 229 753 indeksy. Dochodzi do niej, gdy indeks wyprowadzony przez kompresor dotyczy napisu ostatnio wstawionego do słownika. Sytuacja taka występuje tylko wówczas, gdy napis spod tego indeksu zaczyna się i kończy tym samym znakiem. Dlaczego? Przypomnijmy, że kompresor wyprowadza indeks napisu *s* tylko wtedy, kiedy znajdzie *s* w słowniku, przy jednoczesnej nieobecności w nim napisu *s c*; wówczas wstawia *s c* do słownika, powiedzmy pod indeksem *i*, i zaczyna tworzyć nowy napis *s*, poczynając od *c*. Jeśli następny indeks wyprowadzony przez kompresor okazuje się równy *i*, to napis pod indeksem *i* w słowniku musi się zaczynać od *c*, lecz właśnie widzieliśmy, że napisem tym jest *s c*. Jeśli więc następny indeks słownikowy na wejściu dekompresora dotyczy wpisu jeszcze nieobecnego w słowniku, to dekompresor może wyprowadzić napis, który ostatnio wstawił do słownika, złączyć z pierwszym znakiem tego napisu i wstawić ten nowy napis do słownika.

Ponieważ te sytuacje są rzadkie, przykład jest cokolwiek naciągany. Napis TATATAT powoduje wystąpienie sytuacji tego rodzaju. Kompresor robi, co następuje: wyprowadza indeks 84 (T) i wstawia TA pod indeksem 256; wyprowadza indeks 65 (A) i wstawia AT pod indeksem 257; wyprowadza indeks 256 (TA) i wstawia TAT pod indeksem 258; na koniec wyprowadza indeks 258 (TAT — napis już wstawiony). Dekompresor po przeczytaniu indeksu 258 pobiera napis, który ostatnio wyprowadził, czyli TA, łączy go z pierwszym znakiem tego napisu (T), wyprowadza napis wynikowy TAT i wstawia ten napis do słownika.

Choć ta rzadka sytuacja występuje tylko wówczas, gdy napis zaczyna się i kończy tym samym znakiem, nie pojawia się ona przy takiej okazji za każdym razem. Na przykład podczas kompresowania *Moby Dicka* napis, którego indeks wyprowadzono, miał taki sam znak na początku i na końcu 11 376 razy (nieco poniżej 5% przypadków), ale nie był napisem ostatnio wstawionym do słownika.

Procedura DEKOMPRESOR-LZW na następnej stronie uściśla te wszystkie czynności. W tabeli poniżej uwidoczniono, co się dzieje w każdej iteracji pętli z kroku 4 dla zadanych na wejściu indeksów z kolumny Wyjście z poprzedniej tabeli. Napisy indeksowane w słowniku przez *poprzedni* i *następny* są wyprowadzane w kolejnych iteracjach, a pokazane wartości *poprzedni* i *następny* odnoszą się w każdej iteracji do chwili po zakończeniu kroku 4B.

Iteracja	*poprzedni*	*bieżący*	Wyjście (*s*)	Nowy napis w słowniku
kroki 2, 3		84	`T`	
1	84	65	`A`	256: `TA`
2	65	256	`TA`	257: `AT`
3	256	71	`G`	258: `TAG`
4	71	257	`AT`	259: `GA`
5	257	67	`C`	260: `ATC`
6	67	84	`T`	261: `CT`
7	84	256	`TA`	262: `TT`
8	256	257	`AT`	263: `TAA`
9	257	264	`ATA`	264: `ATA`

Z wyjątkiem ostatniej iteracji indeks wejściowy jest już w słowniku, toteż krok 4D jest wykonywany tylko w ostatniej iteracji. Zauważmy, że słownik budowany przez DEKOMPRESOR-LZW jest zgodny ze słownikiem budowanym przez KOMPRESOR-LZW.

Procedura DEKOMPRESOR-LZW(*indeksy*)

Dane wejściowe: indeksy — ciąg indeksów do słownika, utworzonych przez KOMPRESOR-LZW.

Wynik: tekst, który KOMPRESOR-LZW pobierał na wejściu.

1. Dla każdego znaku *c* w zbiorze znaków ASCII:
   A. Wstaw *c* do słownika pod indeksem równym numerycznej wartości jego kodu ASCII.
2. Podstaw pod *bieżący* pierwszy indeks z *indeksów*.
3. Wyprowadź napis ze słownika spod indeksu *bieżący*.
4. Dopóki *indeksy* się nie wyczerpią, wykonuj, co następuje:
   A. Podstaw *bieżący* do *poprzedni*.
   B. Pobierz następną liczbę z *indeksów* i przypisz ją do *bieżący*.
   C. Jeśli słownik zawiera wpis o indeksie *bieżący*, to wykonaj, co następuje:
      i. Podstaw pod *s* napis ze słownika, z pozycji indeksowanej przez *bieżący*.
      ii. Wyprowadź napis *s*.
      iii. Wstaw na następne wolne miejsce w słowniku napis występujący w słowniku na pozycji indeksowanej przez *poprzedni* złączony z pierwszym znakiem napisu *s*.
   D. W przeciwnym razie (słownik nie zawiera jeszcze wpisu indeksowanego przez *bieżący*) wykonaj, co następuje:
      i. Podstaw pod *s* napis ze słownika, z pozycji indeksowanej przez *poprzedni*, złączony z pierwszym znakiem tej pozycji słownika.
      ii. Wyprowadź napis *s*.
      iii. Wstaw napis *s* na następną wolną pozycję w słowniku.

Nie odniosłem się do tego, jak poszukiwać informacji w słowniku w procedurach KOMPRESOR-LZW i DEKOMPRESOR-LZW. To drugie jest łatwe: wystarczy śledzić ostatnio używany indeks słownikowy i jeśli indeks w *bieżący* jest mniejszy lub równy ostatnio używanemu indeksowi, to napis jest w słowniku. Procedura KOMPRESOR-LZW ma trudniejsze zadanie: dla danego napisu trzeba określić, czy występuje on w słowniku, i jeśli tak, to pod jakim indeksem. Oczywiście moglibyśmy po prostu przeszukać słownik liniowo, lecz jeśli słownik zawiera n elementów, to każde wyszukiwanie liniowe zabiera $O(n)$ czasu. Postąpimy lepiej, używając dowolnej z paru struktur danych. Nie będę jednak wchodził tutaj w szczegóły. Jedna z nich zwie się **trie** (ang. *trie*)[12] i przypomina drzewo binarne, które budowaliśmy dla kodów Huffmana, z tym że każdy węzeł może mieć wielu potomków, a nie tylko dwóch, a każda krawędź jest etykietowana znakiem ASCII. Inną strukturą danych jest **tablica z haszowaniem** (ang. *hash table*), umożliwia ona proste i przeciętnie szybko działające wyszukiwanie napisów w katalogu.

Ulepszenia LZW

Jak wspomniałem, nie byłem zachwycony jakością metody kompresji LZW zastosowanej do tekstu *Moby Dicka*. Część trudności wynika z wielkiego słownika. Przy 230 007 wpisach każdy indeks wymaga przynajmniej czterech bajtów, toteż z wyprowadzonymi 229 753 indeksami wersja skompresowana wymaga czterokrotnie więcej miejsca, czyli 919 012 bajtów. Tu również możemy zaobserwować kilka specyficznych cech indeksów produkowanych przez kompresor LZW. Po pierwsze, wiele z nich to małe liczby, a to oznacza, że w reprezentacji 32-bitowej mają wiele początkowych zer. Po drugie, niektóre indeksy występują znacznie częściej niż inne.

Jeśli występują obie te cechy, kodowanie Huffmana może dawać dobre rezultaty. Zmodyfikowałem program kodowania Huffmana, aby działał na czterobajtowych liczbach zamiast na znakach, i wykonałem go dla danych pochodzących z kompresora LZW przetwarzającego *Moby Dicka*. Wynikowy plik zajmuje tylko 460 971 bajtów, czyli 31,61% rozmiaru oryginalnego (1 193 826 bajtów), co przewyższa samo kodowanie Huffmana. Zauważmy jednak, że nie uwzględniłem w tym obrazie rozmiaru kodowania Huffmana. I podobnie jak w przypadku kompresji, na którą złożyły się dwa etapy — kompresowanie tekstu za pomocą LZW i kompresowanie wynikowych indeksów za pomocą kodowania Huffmana — dekompresja byłaby dwuetapowa: najpierw zdekompresowanie za pomocą kodowania Huffmana, a potem dekompresja metodą LZW.

Inne podejścia do kompresji LZW skupiają się na redukowaniu liczby bitów niezbędnych do przechowywania indeksów wyprowadzanych przez kompresor. Ponieważ wiele indeksów to małe liczby, jedno z podejść polega na użyciu mniejszej liczby bitów dla małych liczb, jednak z drugiej strony — dajmy na to — dwa pierwsze bity muszą wskazywać, ile bitów wymaga dana liczba. Oto jeden ze schematów:

[12] Wym. ⟨traj⟩; nieprzetłumaczalny neologizm angielski, nazwa pochodzi od ang. *re**trie**val*: odzyskanie, wyszukiwanie — *przyp. tłum.*

- Jeśli pierwsze dwa bity są równe 00, to indeks jest w przedziale od 0 do 63 (2^6 – 1) i wymaga kolejnych sześciu bitów, więc jest w sumie 1-bajtowy.
- Jeśli pierwsze dwa bity są równe 01, to indeks jest w przedziale od 64 (2^6) do 16 383 (2^{14} – 1) i wymaga dodatkowych 14 bitów, więc w sumie dwóch bajtów.
- Jeśli pierwsze dwa bity są równe 10, to indeks jest w przedziale od 16 384 (2^{14}) do 4 194 303 (2^{22} – 1), co wymaga kolejnych 22 bitów, czyli razem trzech bajtów.
- Wreszcie, jeśli pierwsze dwa bity są równe 11, to indeks jest w przedziale od 4 194 304 (2^{22}) do 1 073 741 823 (2^{30} – 1), na co trzeba jeszcze 30 bitów, czyli razem cztery bajty.

W dwóch innych metodach indeksy wyprowadzane przez kompresor mają jednakowe rozmiary, ponieważ kompresor ogranicza rozmiar słownika. W jednym z podejść po osiągnięciu przez słownik rozmiaru maksymalnego nie wstawia się już innych wpisów. W drugim podejściu z chwilą osiągnięcia przez słownik maksymalnego rozmiaru opróżnia się go (z wyjątkiem pierwszych 256 wpisów) i proces zapełniania słownika jest wznawiany od miejsca w tekście, w którym słownik się zapełnił. We wszystkich tych metodach dekompresor musi odzwierciedlać działania kompresora.

Co czytać dalej

Książka Salomona [Sal08] jest szczególnie przejrzysta i zwarta, a przy tym ujmuje szeroki wybór technik kompresji. Książka Storera [Sto88], opublikowana 20 lat przed książką Salomona, stanowi klasyczny tekst z tej dziedziny. Podrozdział 16.3 CLRS [CLRS09] wnika w kody Huffmana z pewnymi szczegółami, chociaż nie dowodzi się w nim, że są one najlepszymi możliwymi kodami bezprzedrostkowymi.

10 Trudne (?) problemy

Gdy kupuję towary w Internecie, sprzedawca musi mi je dostarczyć do domu. Z reguły korzysta on z firmy przewozowej. Nie zdradzę, który z kurierów najczęściej dostarcza mi zakupione produkty, powiem tylko, że co jakiś czas można zobaczyć na moim podjeździe brązowe furgonetki.

Brązowe furgonetki

Owa firma przewozowa dysponuje ponad 91 000 takich brązowych furgonetek w USA, a także w wielu innych krajach świata. Co najmniej przez pięć dni w tygodniu każda furgonetka wyrusza z konkretnej bazy, do której potem wraca, i rozwozi paczki do odbiorców indywidualnych i firm handlowych. Przedsiębiorstwo przewozowe jest żywotnie zainteresowane minimalizacją kosztów powodowanych przez każdą furgonetkę, jako że zatrzymuje się ona wiele razy każdego dnia. Na przykład w jednym ze źródeł online, w którym zasięgałem języka, twierdzono, że wytyczenie przez firmę tras dla kierowców, mających zredukować liczbę skrętów w lewo, pozwoliło w ciągu 18 miesięcy zmniejszyć ogólną długość przebiegów jej pojazdów o 464 000 mil, skutkując zaoszczędzeniem 51 000 galonów paliwa[1] i dodatkową korzyścią w postaci zmniejszenia emisji dwutlenku węgla o 506 ton.

W jaki sposób przedsiębiorstwo może zminimalizować codzienne koszty ekspedycji każdej furgonetki? Załóżmy, że dana furgonetka musi dostarczyć paczki do n miejsc konkretnego dnia. Z uwzględnieniem bazy furgonetka musi dotrzeć do $n + 1$ miejsc. Dla każdego z tych $n + 1$ miejsc przedsiębiorstwo może obliczyć koszty wyekspediowania furgonetki z danego miejsca do każdego z n pozostałych, tak więc przedsiębiorstwo dysponuje tablicą $(n + 1) \times (n + 1)$ kosztów przewozu z jednego miejsca do drugiego, przy czym pozycje na przekątnej są bez znaczenia, ponieważ i-ty wiersz oraz i-ta kolumna odpowiadają temu samemu miejscu. Przedsiębiorstwo chce wyznaczyć trasę, która zaczyna się i kończy w danej bazie, wzdłuż której n miejsc jest odwiedzanych tylko raz, i taką, aby jej sumaryczny koszt był możliwie najmniejszy.

Można napisać program komputerowy, który rozwiąże ten problem. W końcu jeśli uwzględnimy konkretną trasę i znamy kolejność przystanków na trasie, to jest to wyłącznie kwestia odnalezienia w tablicy kosztów przejazdu z jednego miejsca do drugiego i ich zsumowania. Potem musimy tylko uwzględnić wszystkie możliwe trasy i zobaczyć, która ma najmniejszy koszt. Liczba możliwych tras jest skończona, więc program zakończy w którymś momencie działanie i poda odpowiedź. Nie wygląda na to, aby taki program był trudny do napisania, czyż nie?

Rzeczywiście, ten program nie jest trudny do napisania.

[1] W przeliczeniu: 747 576 km i 139 035 l — *przyp. tłum.*

Jest trudny do wykonania.

Sęk w tym, że liczba możliwych tras odwiedzających *n* miejsc jest ogromna, wynosi *n*! (*n* silnia). Dlaczego? Furgonetka wyrusza z bazy. Pierwszy przystanek może być w każdym z pozostałych *n* miejsc. Z pierwszego przystanku następny może być w każdym z pozostałych *n* – 1 miejsc, dla pierwszych dwóch przystanków istnieje więc, wymieniając kolejno, $n \cdot (n-1)$ możliwych kombinacji. Kiedy już zdecydujemy się na pierwsze dwa przystanki, trzecim przystankiem mogłoby zostać każde z *n* – 2 miejsc, co daje $n \cdot (n-1) \cdot (n-2)$ możliwych uporządkowań pierwszych trzech przystanków. Rozszerzając to rozumowanie na *n* punktów dostawy, otrzymujemy jako liczbę ich możliwych uporządkowań $n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$, czyli *n*!.

Przypomnijmy, że *n*! rośnie szybciej niż funkcja wykładnicza — jest ponadwykładnicza. W rozdziale 8 zauważyłem, że 10! równa się 3 628 800. Dla komputera nie jest to aż tak duża liczba. Jednak brązowe furgonetki dostarczają paczki pod znacznie więcej niż 10 adresów w ciągu dnia. Załóżmy, że furgonetka rozwozi towary w 20 miejsc dziennie. (Dane przedsiębiorstwo na terenie USA przydziela średnio 170 paczek na samochód, dopuszczając więc nawet, że ileś tam paczek trafia w jedno miejsce, 20 przystanków dziennie nie wydaje się wartością zawyżoną). Dla 20 przystanków program komputerowy musiałby wyliczyć 20! możliwych uporządkowań, a 20! równa się 2 432 902 008 176 640 000. Gdyby komputery w firmie mogły sporządzać i oceniać bilion uporządkowań na sekundę, uporanie się ze wszystkimi zajęłoby ponad 28 dni. A dotyczy to tylko wartości dostaw z jednego dnia, wykonywanych przez jedną z 91 000 furgonetek.

Gdyby, używając takiego rozwiązania, firma miała pozyskać moc obliczeniową niezbędną do znalezienia najtańszych tras dla wszystkich furgonetek każdego dnia, to koszt obliczeń mógłby łatwo przekreślić zyski z bardziej efektywnych tras. Nie, pomysł z wyliczeniem wszystkich możliwych tras i wyborem najlepszych — choć matematycznie przejrzysty — jest po prostu niepraktyczny. Czy istnieje lepszy sposób znalezienia najtańszej trasy dla każdej furgonetki?

Tego nie wie nikt. (A jeśli nawet wie, nie chce powiedzieć). Nikt nie znalazł lepszego sposobu, nikt również nie udowodnił, że lepszy sposób nie może istnieć. Czy to nie frustrujące?!

Jest to bardziej irytujące, niż możesz sobie wyobrazić. Problem znalezienia najtańszych tras dla brązowych furgonetek jest lepiej znany pod nazwą **problemu komiwojażera** (am. ang. *traveling-salesman problem*). Nazwę tę zawdzięcza pierwotnemu sformułowaniu, w którym komiwojażer[2] ma odwiedzić *n* miast, zaczynając i kończąc w tym samym mieście, i ma to zrobić, podróżując możliwie najkrótszą trasą. Dla problemu komiwojażera nigdy nie wynaleziono algorytmu działającego w czasie $O(n^c)$, dla żadnej stałej *c*. Nie znamy algorytmu, który dla zadanych odległości między *n* miastami znajduje najlepszą możliwą kolejność odwiedzania *n* miast w czasie $O(n^{100})$, $O(n^{1000})$, czy choćby $O(n^{1\,000\,000})$.

[2] Przepraszam za seksistowski język. Nazwa jest historyczna, gdyby więc problem pojawił się dzisiaj, mam nadzieję, że byłby znany jako „problem osoby trudniącej się przedstawicielstwem handlowym".

Jest jeszcze gorzej. Wiele problemów — są ich tysiące — wykazuje tę samą charakterystykę: dla danych wejściowych rozmiaru n nie znamy żadnego algorytmu, który działa w czasie $O(n^c)$, dla jakiejkolwiek stałej c, a jednocześnie nikt nie udowodnił, że taki algorytm nie może istnieć. Te problemy wywodzą się z rozlicznych dziedzin: logiki, grafów, arytmetyki, w tym również planowania.

Aby tej irytacji nadać zupełnie nowy ciężar gatunkowy, przytaczamy najbardziej zabawny fakt: *gdyby istniał algorytm, który działa w czasie $O(n^c)$ dla któregokolwiek z tych problemów, gdzie c jest pewną stałą, to dla każdego z tych problemów istniałby algorytm działający w czasie $O(n^c)$*. Problemy te nazywamy **NP-zupełnymi** (ang. *NP-complete*). Algorytm, który działa w czasie $O(n^c)$ na danych rozmiaru n, gdzie c jest stałą, określa się jako **algorytm o czasie wielomianowym** (algorytm wielomianowy, ang. *polynomial-time algorithm*), nazywany tak, ponieważ n^c z pewnym współczynnikiem byłoby najbardziej znaczącym składnikiem w jego czasie działania. Dla żadnego problemu NP-zupełnego nie znamy jakiegokolwiek algorytmu wielomianowego, lecz nikt nie udowodnił, że rozwiązanie jakiegoś problemu NP-zupełnego w czasie wielomianowym jest niemożliwe.

Frustracja jest jeszcze większa: wiele problemów NP-zupełnych wygląda prawie tak samo jak problemy, które potrafimy rozwiązać w czasie wielomianowym. Wystarczy niewielka zmiana, aby je rozdzielić. Przypomnijmy sobie na przykład z rozdziału 6, że algorytm Bellmana-Forda znajduje najkrótsze ścieżki z jednego źródła w grafie skierowanym w czasie $\Theta(n\,m)$ — nawet wówczas, gdy graf ma ujemne wagi krawędzi; n i m oznaczają, odpowiednio, liczbę wierzchołków i krawędzi grafu. Jeśli graf jest zadany w postaci list sąsiedztwa, to rozmiar wejścia wynosi $\Theta(n + m)$. Załóżmy, że $m \geq n$, wtedy dane wejściowe są rozmiaru $\Theta(m)$ i $n\,m \leq m^2$, więc czas działania algorytmu Bellmana-Forda jest wielomianowy względem rozmiaru danych wejściowych. (Ten sam wynik możesz otrzymać, jeśli $n > m$). Wobec tego znalezienie *najkrótszej* ścieżki jest łatwe. Może Cię jednak zaskoczyć wiadomość, że znalezienie *najdłuższej* ścieżki prostej (tzn. najdłuższej ścieżki bez cykli) między dwoma wierzchołkami jest NP-zupełne. W rzeczywistości już samo określenie, czy graf zawiera ścieżkę bez cykli, mającą przynajmniej zadaną liczbę krawędzi, jest NP-zupełne.

Jako inny przykład problemów pokrewnych, z których jeden jest łatwy, a drugi NP-zupełny, rozważmy cykle Eulera i cykle hamiltonowskie. W obu tych problemach mamy do czynienia ze znajdowaniem ścieżek w spójnym grafie nieskierowanym. W **grafie nieskierowanym** (ang. *undirected graph*) krawędzie nie mają kierunku, zatem (u, v) i (v, u) oznaczają tę samą krawędź; mówimy, że krawędź (u, v) jest **incydentna** (ang. *incident*) z wierzchołkami u i v.[3] **Graf spójny** (ang. *connected graph*) ma ścieżkę między każdą parą wierzchołków. **Cykl Eulera** (droga Eulera, ang. *Euler tour*)[4] zaczyna się i kończy w tym samym wierzchołku i przechodzi przez

[3] Tzn. rozpięta na wierzchołkach, padająca na wierzchołki — *przyp. tłum.*

[4] Nazywany tak dlatego, że matematyk Leonhard Euler udowodnił w 1736 r., że nie da się przejść przez Królewiec (miasto w Prusach), idąc każdym z jego siedmiu mostów tylko raz i kończąc spacer w miejscu, z którego się go rozpoczęło.

każdą krawędź tylko jeden raz. **Cykl Hamiltona** (cykl hamiltonowski, ang. *hamiltonian cycle*)[5] zaczyna się i kończy w tym samym wierzchołku i przechodzi przez każdy wierzchołek tylko jeden raz (oczywiście z wyjątkiem wierzchołka, w którym się zaczyna i kończy). Jeśli pytamy, czy spójny graf nieskierowany ma cykl Eulera, to algorytm jest wyraźnie łatwy: określ **stopień** (ang. *degree*) każdego wierzchołka, to znaczy liczbę krawędzi z nim incydentnych. Graf ma cykl Eulera wtedy i tylko wtedy, gdy stopień każdego wierzchołka jest parzysty. Jeśli jednak spytamy, czy spójny graf nieskierowany ma cykl Hamiltona, mamy do czynienia z problemem NP-zupełnym. Zauważmy, że nie pytano „Jakie jest uporządkowanie wierzchołków cyklu Hamiltona w tym grafie?", lecz o coś bardziej podstawowego: „Tak czy nie — czy jest możliwe zbudowanie cyklu hamiltonowskiego w tym grafie?".

Problemy NP-zupełne pojawiają się zaskakująco często — to właśnie powoduje, że włączam dotyczący ich materiał do tej książki. Jeśli próbujesz znaleźć algorytm wielomianowy dla problemu, który okazuje się NP-zupełny, to prawdopodobnie czeka Cię niemałe rozczarowanie. (Zobacz jednak końcowy podrozdział dotyczący perspektyw radzenia sobie w podobnych sytuacjach). Koncepcja problemów NP-zupełnych datuje się na wczesne lata 70. XX wieku, a próby rozwiązania problemów, które okazały się NP-zupełne (jak problem komiwojażera), podjęto znacznie wcześniej. Do dziś nie wiemy, czy istnieje algorytm wielomianowy dla któregokolwiek z problemów NP-zupełnych, nie wiemy też, czy taki algorytm nie może istnieć. Wielu znakomitych informatyków głowiło się nad tym pytaniem przez lata, nie uzyskując odpowiedzi. Nie mówię, że *Ty* nie zdołasz znaleźć algorytmu o czasie wielomianowym jakiegoś problemu NP-zupełnego, lecz trzeba Ci będzie zmierzyć się z wieloma przeciwnościami, gdybyś się na to zdecydował.

Klasy P i NP oraz NP-zupełność

W poprzednich rozdziałach zajmowałem się różnicami w czasach działania, porównując na przykład $O(n^2)$ z $O(n \lg n)$. W tym rozdziale wszakże będziemy się czuli szczęśliwi, jeśli algorytm zadziała w czasie wielomianowym, tak więc różnice między $O(n^2)$ i $O(n \lg n)$ są nieistotne. Informatycy na ogół uważają problemy rozwiązywalne w czasie wielomianowym za „podatne"[6], co oznacza „takie, z którymi łatwo się uporać". Jeśli dla danego problemu istnieje algorytm o czasie wielomianowym, to mówimy, że problem jest w **klasie P**.[7]

---

[5] Nazwa upamiętnia W.R. Hamiltona, który w 1856 r. opisał grę matematyczną na grafie, znaną jako dwunastościan, polegającą na tym, że jeden z graczy wbija pięć szpilek w dowolnych kolejnych pięciu wierzchołkach, a drugi gracz musi dokończyć ścieżkę tak, aby utworzyła cykl zawierający wszystkie wierzchołki.

[6] W teorii złożoności występują dwa angielskie określenia: *tractable* i *decidable*; ze względu na stałe przypisanie temu drugiemu znaczenia „rozstrzygalny", temu pierwszemu, spośród znaczeń takich jak: podatny, łatwy w obróbce, możliwy do rozwiązania, rozstrzygalny, przypisujemy znaczenie „podatny" — *przyp. tłum.*

[7] Litera **P** pochodzi od angielskiej wersji słowa wielomian: *polynomial* — *przyp. tłum.*

W tym miejscu możesz się zdziwić: jak można uważać problem wymagający $\Theta(n^{100})$ czasu za podatny? Wziąwszy dane rozmiaru $n = 10$, czyż liczba n^{100} nie jest przytłaczająco wielka? Tak, zgoda, liczba n^{100} to w istocie googol (od czego bierze początek nazwa „Google"). Na szczęście nie będziemy się rozglądać za algorytmami działającymi w czasie $\Theta(n^{100})$. Problemy w P, które napotykamy w praktyce, wymagają znacznie mniej czasu. Rzadko spotykałem algorytmy wielomianowe, których czas działania był gorszy niż, powiedzmy, $O(n^5)$. Co więcej, gdy już ktoś znajdzie algorytm wielomianowy jakiegoś problemu, inni często pospieszają z jeszcze efektywniejszymi algorytmami. Gdyby więc komuś udało się obmyślić dla jakiegoś problemu pierwszy algorytm wielomianowy, lecz działający w czasie $\Theta(n^{100})$, byłaby to dobra okazja dla innych, aby mogli dostarczyć całą gamę szybszych algorytmów.

Przypuśćmy więc, że przedłożono Ci propozycję rozwiązania problemu i chcesz zweryfikować poprawność tego rozwiązania. Na przykład w problemie cyklu Hamiltona proponowanym rozwiązaniem mógłby być ciąg wierzchołków. Aby sprawdzić, czy rozwiązanie to jest poprawne, trzeba by zbadać, czy każdy wierzchołek pojawia się w ciągu tylko raz, z wyjątkiem wierzchołka pierwszego i ostatniego, bo to musi być ten sam wierzchołek, i jeśli ciąg ma postać $\langle v_1, v_2, v_3,..., v_n, v_1\rangle$, to graf musi zawierać krawędzie (v_1, v_2), (v_2, v_3), (v_3, v_4),..., (v_{n-1}, v_n) i powracającą (v_n, v_1). Łatwo możesz sprawdzić w czasie wielomianowym, że to rozwiązanie jest poprawne. Jeśli proponowane rozwiązanie problemu można zweryfikować w czasie wielomianowym względem rozmiaru danych wejściowych problemu, to mówimy, że problem jest w **klasie NP**.[8] Proponowane rozwiązanie nazywamy **certyfikatem** (świadectwem, ang. *certificate*). Aby problem mieścił się w NP, czas weryfikacji certyfikatu musi być wielomianowy względem rozmiaru danych wejściowych problemu i rozmiaru certyfikatu.

Jeśli potrafisz rozwiązać problem w czasie wielomianowym, to z pewnością potrafisz zweryfikować certyfikat tego problemu w czasie wielomianowym. Innymi słowy, każdy problem w P jest automatycznie w NP. Kwestia odwrotna — czy każdy problem w NP jest w P? — wprawia w zakłopotanie informatyków teoretyków od lat. Często określamy to jako problem „P = NP?".

Problemy „NP-zupełne" są „najtrudniejszymi" w NP. Ujmując kwestię nieformalnie, problem jest **NP-zupełny** (ang. *NP-complete*), jeśli spełnia dwa warunki: (1) jest w NP i (2) jeśli istnieje dla niego algorytm wielomianowy, to istnieje możliwość zamiany *każdego* problemu w NP na ten problem w sposób równoznaczny z rozwiązaniem wszystkich tych problemów w czasie wielomianowym. Jeśli istnieje algorytm o czasie wielomianowym dla jakiegokolwiek problemu NP-zupełnego, tzn. jeśli którykolwiek z problemów NP-zupełnych jest w P, to P = NP. Ponieważ problemy NP-zupełne są najtrudniejszymi w NP, jeśli okaże się, że jakikolwiek problem w NP nie jest rozwiązywalny w czasie wielomianowym, to będzie to dotyczyło wszystkich

[8] Zapewne domyślasz się, że nazwa P pochodzi od *polynomial time* (z ang. czas wielomianowy). Jeśli zastanawiasz się, od czego pochodzi nazwa NP, to już odpowiadam: bierze się ona od *nondeterministic polynomial time* (z ang. niedeterministyczny czas wielomianowy). Jest to równoważny, choć nie całkiem intuicyjny, sposób widzenia tej klasy problemów.

problemów NP-zupełnych. Problem jest **NP-trudny** (ang. *NP-hard*), jeśli spełnia drugi warunek NP-zupełności, lecz niekoniecznie należy do NP.

Oto podręczna lista niezbędnych definicji:

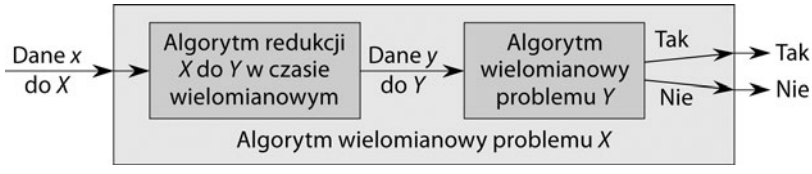
- **P** — problemy rozwiązywalne w czasie wielomianowym, tzn. możemy rozwiązać dany problem w czasie wielomianowym względem rozmiaru danych wejściowych problemu.
- **Certyfikat** (świadectwo) — proponowane rozwiązanie problemu.
- **NP** — problemy weryfikowalne w czasie wielomianowym, tzn. mając certyfikat, potrafimy sprawdzić w czasie wielomianowym względem rozmiaru danych wejściowych problemu i rozmiaru certyfikatu, że certyfikat jest rozwiązaniem problemu.
- **NP-trudny** — taki problem, że jeśli istnieje działający w czasie wielomianowym algorytm rozwiązania tego problemu, to potrafimy zamienić każdy problem w NP na ten problem, co oznacza możliwość rozwiązania każdego problemu w NP w czasie wielomianowym.
- **NP-zupełny** — problem, który jest NP-trudny i jest również w NP.

Problemy decyzyjne i redukcje

Kiedy mówimy o klasach P i NP lub o pojęciu NP-zupełności, ograniczamy się do **problemów decyzyjnych** (ang. *decision problems*) — ich wyjściem jest jeden bit symbolizujący „tak" lub „nie". Potraktowałem w ten sposób problemy cyklu Eulera i cyklu Hamiltona: Czy graf ma cykl Eulera? Czy graf ma cykl Hamiltona?

Niektóre problemy nie mają jednak natury decyzyjnej, lecz są problemami optymalizacyjnymi, kiedy to chcemy znaleźć najlepsze możliwe rozwiązania. Na szczęście często możemy przerzucić most ponad częścią tej luki, przeformułowując problem optymalizacyjny na problem decyzyjny. Rozważmy na przykład problem najkrótszej ścieżki. Do znajdywania najkrótszych ścieżek używaliśmy tam algorytmu Bellmana-Forda. W jaki sposób moglibyśmy obrócić problem najkrótszych ścieżek w problem tak-nie? Możemy spytać: „Czy graf zawiera ścieżkę między dwoma konkretnymi węzłami, której waga wynosi nie więcej niż zadana wartość k?" Nie pytamy o wierzchołki lub krawędzie ścieżki, lecz tylko o to, czy taka ścieżka istnieje. Zakładając, że wagi ścieżki są liczbami całkowitymi, możemy znaleźć rzeczywistą wagę najkrótszej ścieżki między dwoma wierzchołkami, zadając pytania tak-nie. Jak? Postawmy pytanie dla $k = 1$. Jeśli odpowiedź brzmi: nie, to spróbujmy z $k = 2$. Jeśli odpowiedź jest negatywna, spróbujmy z $k = 4$. Podwajamy wartość k, aż otrzymamy odpowiedź: tak. Gdyby ostatnią wartością k było k', to odpowiedź mieści się gdzieś między $k'/2$ a k'. Wtedy znajdujemy właściwą odpowiedź, stosując wyszukiwanie binarne z początkowym przedziałem między $k'/2$ a k'. Za pomocą tej metody nie dowiemy się, które wierzchołki i krawędzie zawiera najkrótsza ścieżka, lecz przynajmniej poznamy jej wagę.

Drugi warunek, który musi być spełniony, aby problem był NP-zupełny, to wymaganie, żeby w wypadku istnienia wielomianowego tego algorytmu problemu istniał sposób przekształcenia każdego problemu w NP w ten problem, umożliwiający rozwiązanie ich wszystkich w czasie wielomianowym. Pozostając przy problemach decyzyjnych, zobaczmy, na czym polega ogólna idea zamiany problemu decyzyjnego *X* na inny problem decyzyjny *Y* na takich zasadach, że jeśli istnieje algorytm wielomianowy dla *Y*, to istnieje algorytm wielomianowy dla *X*. Konwersję taką nazywamy **redukcją** (ang. *reduction*), ponieważ „redukujemy" rozwiązywanie problemu *X* do rozwiązywania problemu *Y*. Oto, na czym polega ten pomysł:

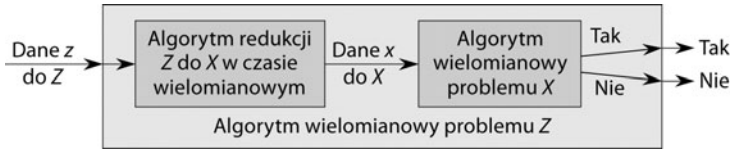


Mamy pewne dane wejściowe rozmiaru *n* do problemu *X*. Przekształcamy je w dane wejściowe *y* do problemu *Y* i robimy to w czasie wielomianowym względem *n*, powiedzmy $O(n^c)$ z pewną stałą *c*. W tym, jak przekształcamy dane wejściowe *x* w dane wejściowe *y*, musi być przestrzegana ważna zasada: jeśli algorytm *Y* rozstrzyga „tak" dla danych *y*, to algorytm *X* powinien dla danych *x* rozstrzygać „tak", a jeśli *Y* rozstrzyga „nie" dla *y*, to *X* powinien rozstrzygać „nie" dla *x*. Przekształcenie to nazywamy **algorytmem redukcji w czasie wielomianowym** (ang. *polynomial-time reduction algorithm*). Zobaczmy, jak długo trwa cały algorytm dla problemu *X*. Algorytm redukcji zużywa $O(n^c)$ czasu, a jego wyniki nie mogą być dłuższe niż zużywany przez niego czas, wobec tego rozmiar danych wyjściowych algorytmu redukcji wynosi $O(n^c)$. Lecz te wyniki są danymi wejściowymi *y* do algorytmu problemu *Y*. Ponieważ algorytm dla *Y* jest algorytmem o czasie wielomianowym, z wejściem rozmiaru *m*, działa w czasie $O(m^d)$, gdzie *d* jest pewną stałą. Tutaj *m* jest rzędu $O(n^c)$, zatem algorytm dla *Y* zużywa $O((n^c)^d)$ czasu, czyli $O(n^{cd})$. Ponieważ zarówno *c*, jak i *d* są stałymi, więc również *cd* jest stałą, widzimy więc, że algorytm *Y* jest algorytmem o czasie wielomianowym. Sumaryczny czas algorytmu problemu *X* wynosi $O(n^c + n^{cd})$, co czyni go również algorytmem wielomianowym.

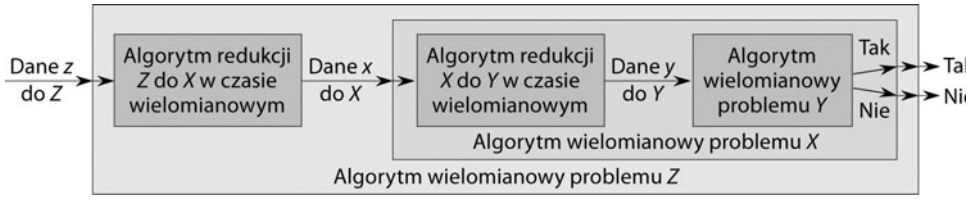
To podejście wykazuje, że jeśli problem *Y* jest „łatwy" (rozwiązywalny w czasie wielomianowym), to problem *X* — również. Redukcjami w czasie wielomianowym posłużymy się jednak nie do wykazywania, że problemy są łatwe, lecz że są trudne:

> Jeżeli problem *X* jest NP-trudny i potrafimy zredukować go do problemu *Y* w czasie wielomianowym, to problem *Y* też jest NP-trudny.

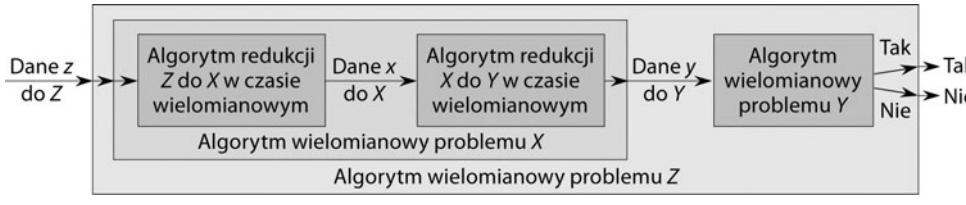
Dlaczego to stwierdzenie jest prawdziwe? Załóżmy, że problem *X* jest NP-trudny i istnieje algorytm redukcji w czasie wielomianowym zamiany danych wejściowych problemu *X* na dane wejściowe problemu *Y*. Ponieważ *X* jest NP-trudny, istnieje taki sposób zamiany dowolnego problemu w NP — powiedzmy *Z* — na *X*, że jeśli *X* ma algorytm o czasie wielomianowym, to *Z* też. W tej chwili już wiesz, jak ta zamiana przebiega — za pomocą redukcji w czasie wielomianowym:



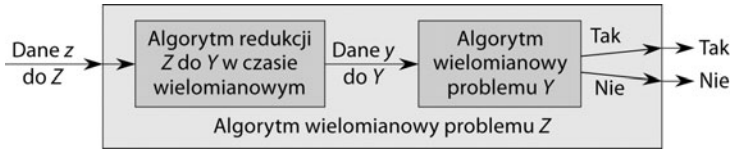
Ponieważ możemy zamienić dane wejściowe do X na dane wejściowe do Y za pomocą redukcji w czasie wielomianowym, możemy rozszerzyć X, jak zrobiliśmy to wcześniej:



Zamiast zestawiania ze sobą redukcji w czasie wielomianowym X do Y i algorytmu dla Y, połączmy razem dwie redukcje w czasie wielomianowym:



Zauważamy teraz, że jeśli natychmiast po redukcji w czasie wielomianowym Z do X wykonamy redukcję w czasie wielomianowym z X do Y, to otrzymamy redukcję w czasie wielomianowym z Z do Y:



Żeby się upewnić, że te dwie kolejne redukcje w czasie wielomianowym tworzą jedną redukcję w czasie wielomianowym, posłużymy się analizą podobną do wykonanej poprzednio. Załóżmy, że dane wejściowe z do problemu Z mają rozmiar n, redukcja z Z do X zajmuje czas $O(n^c)$, a redukcja z X do Y na danych wejściowych rozmiaru m zajmuje czas $O(m^d)$, gdzie c oraz d są stałymi. Wynik redukcji z Z do X nie może być dłuższy niż czas potrzebny do jego wyprodukowania i dlatego ten wynik, stanowiący również dane wejściowe x redukcji z X do Y, ma rozmiar $O(n^c)$. Wiemy teraz, że rozmiar m danych wejściowych redukcji z X do Y wynosi $m = O(n^c)$, więc czas zużywany przez redukcję z X do Y wynosi $O((n^c)^d)$, co jest równe $O(n^{cd})$. Ponieważ c oraz d są stałymi, ta druga redukcja zużywa czas wielomianowy względem n.

Ponadto czas zużywany w ostatnim etapie, tj. przez algorytm wielomianowy dla Y, jest również wielomianowy względem n. Załóżmy, że algorytm dla Y na danych wejściowych rozmiaru p zużywa czas $O(p^b)$, gdzie b jest stałą. Wynik redukcji, jak poprzednio, nie może przekroczyć czasu potrzebnego do jego wyprodukowania, więc $p = O(n^{cd})$, co oznacza, że algorytm dla Y zużywa czas $O(n^{cd})^b)$, czyli $O(n^{bcd})$. Ponieważ b, c, i d są stałymi, algorytm dla Y zużywa czas wielomianowy na oryginalnych danych rozmiaru n. Uwzględniając to wszystko, algorytm dla Z zużywa czas $O(n^c + n^{cd} + n^{bcd})$, który jest wielomianowy względem n.

Co zobaczyliśmy? Wykazaliśmy, że jeśli problem X jest NP-trudny i istnieje algorytm redukcji w czasie wielomianowym, który przekształca dane wejściowe x problemu X w dane wejściowe y do problemu Y, to Y jest także NP-trudny. Ponieważ NP-trudność X oznacza, że każdy problem w NP redukuje się do niego w czasie wielomianowym, wybraliśmy w NP dowolny problem Z redukowalny do X w czasie wielomianowym, i wykazaliśmy, że redukuje się on również do Y w czasie wielomianowym.

Naszym ostatecznym celem jest wykazanie, że problemy są NP-zupełne. Żeby więc wykazać, że problem Y jest NP-zupełny, pozostaje nam:

- dowieść, że jest on w NP, czemu możemy sprostać, pokazując, że istnieje sposób zweryfikowania certyfikatu Y w czasie wielomianowym;
- oraz wziąć jakiś inny problem X, o którym wiemy, że jest NP-trudny, i podać redukcję w czasie wielomianowym z X do Y.

Jest jeszcze jeden drobny szczegół, który do tej pory pominąłem: tzw. problem matka. Musimy rozpocząć od jakiegoś [pierwszego] problemu NP-zupełnego M (**problemu matki**, ang. *Mother Problem*), redukowalnego do *każdego* problemu w NP w czasie wielomianowym. Wówczas możemy zredukować M w czasie wielomianowym do pewnego innego problemu, aby wykazać, że ten inny problem jest NP-trudny, zredukować ów inny problem do jeszcze innego problemu, żeby pokazać, iż ten kolejny jest NP-trudny itd. Miej również na uwadze, że nie ma ograniczeń co do tego, do ilu problemów zredukujemy pojedynczy problem, tak więc drzewo genealogiczne problemów NP-zupełnych zaczyna się od problemu matki i potem się rozgałęzia.

Problem matka

Różne książki podają różne problemy matki. Nie ma w tym nic złego, bo jeśli zredukujesz jeden problem matkę do jakiegoś innego problemu, ten inny problem może posłużyć jako problem matka. Jednym z często spotykanych problemów matek jest boolowska (logiczna) formuła spełnialności. Opiszę pokrótce ten problem, lecz nie będę dowodził, że każdy problem w NP redukuje się do niego w czasie wielomianowym. Dowód jest długi i — mówię to z przykrością — zawiły.

Na początek: „boolowski" jest w żargonie matematycznym określeniem prostej logiki, w której zmienne mogą przyjmować tylko dwie wartości: 0 lub 1 (nazywane

wartościami boolowskimi)[9], a operatory przyjmują jedną lub dwie wartości boolowskie i produkują wartość boolowską. W rozdziale 8 zapoznaliśmy się już z alternatywą wykluczającą (XOR). Do typowych operatorów boolowskich należą: AND, OR, NOT, IMPLIES i IFF:[10]

- *x* AND *y* równa się 1 tylko wtedy, gdy zarówno *x*, jak i *y* są równe 1; w przeciwnym razie (któryś z argumentów lub oba są równe 0) *x* AND *y* równa się 0.
- *x* OR *y* równa się 0 tylko wtedy, gdy zarówno *x*, jak i *y* są równe 0; w przeciwnym razie (któryś z argumentów lub oba są równe 1) *x* OR *y* równa się 1.
- NOT *x* jest zaprzeczeniem *x*: jest równe 0, jeśli *x* równa się 1, i jest równe 1, jeśli *x* równa się 0.
- *x* IMPLIES *y* równa się 0 tylko wtedy, gdy *x* równa się 1 i *y* równa się 0; w przeciwnym razie (*x* równa się 0 albo *x* i *y* są równe 1) *x* IMPLIES *y* równa się 1.
- *x* IFF *y* oznacza „*x* wtedy i tylko wtedy, gdy *y*" i jest równe 1 tylko wtedy, kiedy *x* oraz *y* są sobie równe (oba są równe 0 lub oba są równe 1); jeśli *x* różni się od *y* (jedno z nich równa się 0, a drugie równa się 1), to *x* IFF *y* równa się 0.

Istnieje 16 dwuargumentowych operatorów boolowskich, lecz te są najpopularniejsze[11]. **Formuła boolowska** (formuła logiczna, ang. *boolean formula*) składa się ze zmiennych o wartościach boolowskich, operatorów boolowskich i nawiasów używanych do grupowania.

W **problemie spełnialności formuły boolowskiej** (ang. *boolean formula satisfiability problem*) dane wejściowe są formułą boolowską i pytamy, czy istnieje możliwość takiego przypisania wartości 0 i 1 zmiennym w formule, aby jej obliczona wartość równała się 1. Jeśli taka możliwość istnieje, to mówimy, że formuła jest **spełnialna** (ang. *satisfiable*). Na przykład formuła boolowska

((*w* IMPLIES *x*) OR NOT (((NOT *w*) IFF *y*) OR *z*)) AND (NOT *x*)

jest spełnialna: niech *w* = 0, *x* = 0, *y* = 1 i *z* = 1. Wówczas, obliczając tę formułę, otrzymujemy:

((0 IMPLIES 0) OR NOT (((NOT 0) IFF 1) OR 1) AND (NOT 0)
= (1 OR NOT ((1 IFF 1) OR 1)) AND 1
= (1 OR NOT (1 OR 1) AND 1
= (1 OR 0) AND 1
= 1 AND 1
= 1.

---

[9] W polskiej mowie matematyków również „logicznymi", tu jednak dochowujemy wierności własnym wyborom terminologicznym; przymiotnik boolowski pochodzi od nazwiska George'a Boola, twórcy algebry, której zastosowania w komputerach są ewidentne — *przyp. tłum.*

[10] Odpowiednie nazwy polskie tych operatorów to: „i", „lub", „nie", „implikuje" i „wtedy i tylko wtedy, gdy" — *przyp. tłum.*

[11] Niektóre z tych 16 dwuargumentowych operatorów boolowskich nie są specjalnie interesujące, na przykład operator, który niezależnie od wartości argumentów daje w wyniku 0.

A oto prosty przykład formuły niespełnialnej:

x AND (NOT x).

Jeśli $x = 0$, to ta formuła da w wyniku 0 AND 1, co jest równe 0; natomiast jeśli $x = 1$, to formuła przybiera postać 1 AND 0, a to też jest równe 0.

Próbnik problemów NP-zupełnych

Posługując się spełnialnością formuł boolowskich jako naszym problemem matką, przyjrzyjmy się kilku problemom, których NP-zupełność możemy wykazać, stosując redukcje w czasie wielomianowym. Oto drzewo genealogiczne redukcji, które obejrzymy:

Nie pokażę wszystkich redukcji w tym drzewie genealogicznym, ponieważ niektóre z nich są dość długie i skomplikowane. Zobaczymy jednak kilka takich, które są ciekawe, ponieważ pokazują, jak redukować problem z jednej dziedziny do innej, na przykład logiczny (spełnialność 3-CNF) do grafów (problem kliki).

Spełnialność 3-CNF

Ponieważ formuły boolowskie mogą zawierać dowolne z 16 dwuargumentowych operatorów boolowskich, a także dlatego, że mogą zawierać nawiasy w dowolnej liczbie konfiguracji, bezpośrednie redukowanie z problemu spełnialności formuły boolowskiej — problemu matki — jest trudne. W zamian zdefiniujemy problem pokrewny, który także dotyczy spełnialności formuł boolowskich, lecz ma pewne ograniczenia co do struktury formuły stanowiącej dane wejściowe problemu. Redukowanie z tego ograniczonego problemu będzie znacznie łatwiejsze. Narzucamy wymaganie, aby formuła była koniunkcją (wyrażeniem połączonym operatorami AND) **klauzul** (ang. *clauses*), przy czym każda klauzula jest alternatywą trzech składników (wyrażeniem połączonym operatorami OR), a każdy składnik jest **literałem** (ang. *literal*): zmienną albo negacją zmiennej (np. NOT x). Tego rodzaju formuła boolowska ma **postać 3-koniunkcyjną normalną** (ang. *3-conjunctive normal form*), czyli **3-CNF**. Na przykład formuła boolowska

(*w* OR (NOT *w*) OR (NOT *x*)) AND (*y* OR *x* OR *z*)
AND ((NOT *w*) OR (NOT *y*) OR (NOT *z*))

jest postaci 3-CNF. Jej pierwszą klauzulą jest (*w* OR (NOT *w*) OR (NOT *x*)).

Rozstrzygnięcie, czy formuła boolowska w postaci 3-CNF ma spełniające przypisanie [wartości] do jej zmiennych — **problem spełnialności 3-CNF** — jest NP-zupełne. Certyfikatem jest proponowane przypisanie wartości 0 i 1 do zmiennych[12]. Sprawdzenie certyfikatu jest łatwe — wystarczy podstawić proponowane wartości pod zmienne i sprawdzić, czy wyrażenie daje w wyniku 1. Aby pokazać, że spełnialność 3-CNF jest NP-trudna, redukujemy z (nieograniczonej) spełnialności formuły boolowskiej. Tu również nie będę wchodził w (nie aż tak bardzo ciekawe) szczegóły. W zamian proponuję coś ciekawszego: dokonamy redukcji problemu z jednej dziedziny do problemu z innej dziedziny.

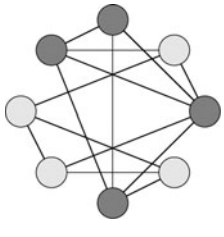
Oto frustrujący aspekt spełnialności 3-CNF: podczas gdy ona jest NP-zupełna, istnieje algorytm wielomianowy sprawdzenia, czy formuła 2-CNF jest spełnialna. Formuła 2-CNF jest podobna do formuły 3-CNF, z tym że w każdej klauzuli ma dwa literały, a nie trzy. Taka mała zmiana sprawia, że problem trudny jak najtrudniejsze w NP staje się łatwy!

Klika

Zmierzamy obecnie do ciekawej redukcji dotyczącej problemów z różnych dziedzin: ze spełnialności 3-CNF do problemu odnoszącego się do grafów nieskierowanych. **Klika** (ang. *klika*) w nieskierowanym grafie *G* jest podzbiorem *S* takich wierzchołków, że graf ma krawędź między każdą parą wierzchołków należących do *S*. **Rozmiarem kliki** jest liczba zawartych w niej wierzchołków.

Jak możesz przypuszczać, kliki mają znaczenie w teorii sieci społecznych. Modelując każdego osobnika w postaci wierzchołka, a powiązania między osobnikami jako nieskierowane krawędzie, klika reprezentuje grupę osobników, wśród których każdy ma powiązania z każdym. Kliki mają również zastosowania w bioinformatyce, inżynierii i chemii.

W **problemie kliki** danymi wejściowymi są: graf *G* i dodatnia liczba całkowita *k*, a pytamy, czy *G* ma klikę o rozmiarze *k*. Przykładowo, graf na następnej stronie ma klikę rozmiaru 4, pokazaną za pomocą mocno przyciemnionych wierzchołków, i nie ma żadnej innej kliki o rozmiarze 4 lub większym.



[12] Zbiór wartości boolowskich (logicznych) stanowiących to przypisanie jest nazywany **wartościowaniem**, nie będziemy jednak używali tego słowa, pamiętając, że znaczy ono przede wszystkim ocenianie wartości czegoś — *przyp. tłum.*

Sprawdzenie certyfikatu jest łatwe. Certyfikatem jest *k* wierzchołków, o których stwierdza się, że tworzą klikę, więc musimy tylko sprawdzić, że każdy z *k* wierzchołków ma wspólną krawędź z pozostałymi *k* – 1. To sprawdzenie można łatwo wykonać w czasie wielomianowym względem rozmiaru grafu. Dzięki temu wiemy, że problem kliki jest w NP.

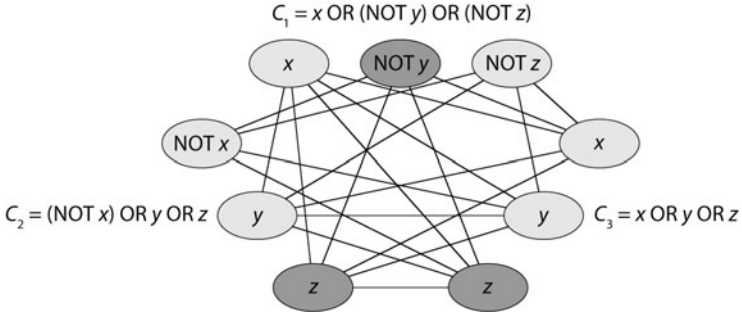
Jak zredukować problem spełnialności formuł boolowskich do problemu grafowego? Zaczynamy od formuły boolowskiej w problemie 3-CNF. Załóżmy, że jest to formuła C_1 AND C_2 AND C_3 AND ... AND C_k, gdzie C_r jest jedną z *k* klauzul. Na podstawie tej formuły skonstruujemy graf w czasie wielomianowym, który będzie miał *k*-klikę (klikę o rozmiarze *k*) wtedy i tylko wtedy, gdy formuła 3-CNF jest spełnialna. Potrzebne są nam trzy rzeczy: konstrukcja, uzasadnienie, że konstruowanie działa w czasie wielomianowym względem rozmiaru formuły 3-CNF, i dowód, że graf ma *k*-klikę wtedy i tylko wtedy, gdy istnieje taki sposób przypisania wartości zmiennym formuły 3-CNF, że przybiera ona wartość 1.

Aby zbudować graf na podstawie formuły 3-CNF, skupmy się na *r*-tej klauzuli: C_r. Ma ona trzy literały; nazwijmy je l_1^r, l_2^r i l_3^r, zatem $C_r = l_2^r$ OR l_2^r OR l_3^r. Każdy literał jest zmienną lub negacją zmiennej. Tworzymy po jednym wierzchołku dla każdego literału, a więc dla klauzuli C_r tworzymy trójkę złożoną z wierzchołków v_1^r, v_2^r i v_3^r. Między wierzchołkami v_i^r i v_j^s dodajemy krawędzie, jeżeli są spełnione dwa warunki:

- v_i^r i v_j^s należą do różnych trójek, tzn. *r* i *s* są numerami różnych klauzul,
- odpowiadające im literały nie są wzajemnymi negacjami (zaprzeczeniami).

Na przykład graf na następnej stronie odpowiada formule 3-CNF:

(*x* OR (NOT *y*) OR (NOT *z*)) AND ((NOT *x*) OR *y* OR *z*) AND (*x* OR *y* OR *z*).



Nietrudno się przekonać, że tę redukcję można wykonać w czasie wielomianowym. Jeśli formuła 3-CNF ma *k* klauzul, to ma 3*k* literałów, wobec tego graf ma 3*k* wierzchołków. Z każdego wierzchołka wychodzi krawędź do najwyżej 3*k* – 1 pozostałych krawędzi, więc liczba krawędzi wynosi nie więcej niż 3*k*(3*k* – 1), co równa się $9k^2 – 3k$. Rozmiar skonstruowanego grafu jest wielomianowy względem rozmiaru danych wejściowych problemu 3-CNF i łatwo określić, które krawędzie należą do grafu.

Na koniec musimy wykazać, że skonstruowany graf ma *k*-klikę wtedy i tylko wtedy, gdy formuła 3-CNF jest spełnialna. Zaczniemy od założenia, że formuła jest spełnialna, i pokażemy, że graf ma *k*-klikę. Jeśli istnieje przypisanie spełniające, to każda klauzula C_r zawiera przynajmniej jeden literał l_i^r dający w wyniku 1, i każdy taki literał odpowiada wierzchołkowi v_i^r w grafie. Jeżeli wybierzemy jeden taki literał z każdej z *k* klauzul, to otrzymamy odpowiedni zbiór *S* mający *k* wierzchołków. Twierdzę, że *S* jest *k*-kliką. Rozważmy dowolne dwa wierzchołki w *S*. Odpowiadają one literałom w różnych klauzulach, mającym wartość 1 po obliczeniu dla spełniającego je przypisania. Te literały nie mogą być wzajemnymi negacjami, bo gdyby były, to jeden z nich dawałby w wyniku 1, a drugi 0. Skoro literały te nie są wzajemnymi negacjami, podczas konstruowania grafu utworzyliśmy między danymi dwoma wierzchołkami krawędź. Ponieważ jako tę parę możemy obrać dowolne dwa wierzchołki z *S*, widzimy, że krawędzie istnieją między wszystkimi parami wierzchołków w *S*. Stąd *S* — zbiór *k* wierzchołków — jest *k*-kliką.

Teraz musimy przeprowadzić dowód w drugą stronę: jeśli graf ma *k*-klikę *S*, to formuła 3-CNF jest spełnialna. Żadna z krawędzi w grafie nie łączy wierzchołków w tej samej trójce, zatem zbiór *S* zawiera dokładnie po jednym wierzchołku na trójkę. Dla każdego wierzchołka v_i^r należącego do *S* przypisujemy odpowiedniemu literałowi l_i^r w formule 3-CNF wartość 1. Nie musimy się martwić o przypisanie 1 zarówno literałowi, jak i jego negacji, gdyż *k*-klika nie może zawierać wierzchołków odpowiadających literałowi i jego negacji. Ponieważ każda klauzula ma literał dający po obliczeniu wynik 1, każda klauzula jest spełnialna, więc cała formuła 3-CNF jest spełnialna. Gdyby którymś zmiennym nie odpowiadały wierzchołki w klice, przypisz im dowolne wartości — nie mają wpływu na spełnialność formuły.

W powyższym przykładzie, w spełniającym przypisaniu jest $y = 0$ i $z = 1$; nie ma znaczenia, co podstawimy pod x. Odpowiednia 3-klika składa się z mocno zaciemnionych wierzchołków, które odpowiadają literałowi NOT y w klauzuli C_i i literałowi z w klauzulach C_2 i C_3.

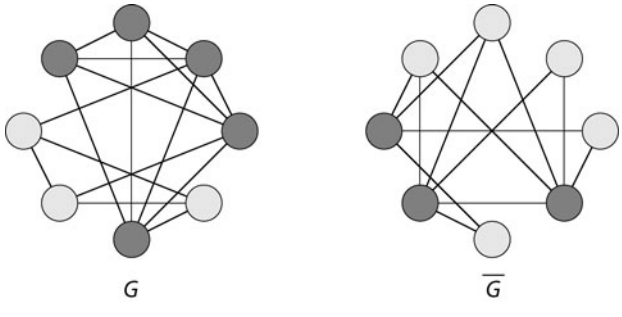
W ten sposób wykazaliśmy, że istnieje redukcja w czasie wielomianowym NP-zupełnego problemu spełnialności 3-CNF do problemu znalezienia *k*-kliki. Gdyby Ci dano formułę boolowską problemu 3-CNF z *k* klauzulami i miałbyś znaleźć jej spełniające przypisanie, to mógłbyś się posłużyć przedstawioną właśnie konstrukcją do zamiany w czasie wielomianowym formuły na graf nieskierowany, po czym określić, czy ten graf ma *k*-klikę. Gdybyś potrafił rozsądzić w czasie wielomianowym, czy graf ma *k*-klikę, to poradziłbyś sobie z ustaleniem w czasie wielomianowym, czy formuła 3-CNF ma spełniające przypisanie. Ponieważ spełnialność 3-CNF jest NP-zupełna, więc ustalenie, czy graf zawiera *k*-klikę, również jest takim problemem. Na dodatek gdybyś umiał określić nie tylko, czy graf ma *k*-klikę, lecz także wierzchołki, które ją tworzą, to mógłbyś użyć tych informacji do znalezienia wartości podstawianych pod zmienne formuły 3-CNF w spełniającym ją przypisaniu.

Pokrycie wierzchołkowe

Pokrycie wierzchołkowe (ang. *vertex cover*) grafu nieskierowanego G jest takim podzbiorem S wierzchołków, że każda krawędź G jest incydentna z co najmniej jednym wierzchołkiem S. Mówimy, że każdy wierzchołek S „pokrywa" incydentne z nim krawędzie. **Rozmiar pokrycia wierzchołkowego** jest liczbą zawieranych przez nie wierzchołków. Podobnie jak w problemie kliki, dane do **problemu pokrycia wierzchołkowego** (ang. *vertex-cover problem*) stanowią: graf nieskierowany G i dodatnia liczba całkowita m. Pytamy, czy G ma pokrycie wierzchołkowe rozmiaru m. Podobnie jak problem kliki, problem pokrycia wierzchołkowego ma zastosowanie w bioinformatyce. Jako inne zastosowanie wyobraź sobie budynek, w którym na skrzyżowaniach korytarzy zainstalowano kamery o kącie widzenia do 360 stopni, i chcesz ustalić, czy m kamer umożliwi wgląd we wszystkie korytarze. Tutaj krawędzie modelują korytarze, a wierzchołki — ich skrzyżowania. W jeszcze innym zastosowaniu znajdowanie pokryć wierzchołków pomaga w projektowaniu strategii udaremniania ataków robaków w sieciach komputerowych.

Certyfikatem w problemie pokrycia wierzchołkowego — co nie zaskakuje — jest proponowane pokrycie wierzchołkowe. Łatwo jest sprawdzić w czasie wielomianowym względem rozmiaru grafu, czy proponowane pokrycie wierzchołkowe ma rozmiar m i naprawdę pokrywa wszystkie wierzchołki, widzimy więc, że ten problem jest w NP.

W drzewie genealogicznym zaznaczono, że do problemu pokrycia wierzchołkowego redukujemy problem kliki. Załóżmy, że danymi wejściowymi do problemu kliki są: nieskierowany graf G mający n wierzchołków i dodatnia liczba całkowita k. W czasie wielomianowym utworzymy graf $\overline{G}$, stanowiący dane wejściowe problemu pokrycia wierzchołkowego, taki że G ma klikę rozmiaru k wtedy i tylko wtedy, gdy $\overline{G}$ ma pokrycie wierzchołkowe rozmiaru $n - k$. Jest to naprawdę łatwa redukcja. Graf $\overline{G}$ ma te same wierzchołki co G, a jego krawędzie dopełniają te, które występują w G. Mówiąc inaczej, krawędź (u, v) należy do $\overline{G}$ wtedy i tylko wtedy, gdy (u, v) *nie* należy do G. Zgadujesz być może, że pokrycie wierzchołkowe o rozmiarze $n - k$ w grafie $\overline{G}$ składa się z wierzchołków *nie* należących do k-wierzchołkowej kliki w G — i masz rację! Oto przykłady grafów G i $\overline{G}$ z ośmioma wierzchołkami. Pięć wierzchołków tworzących klikę w G i pozostałe trzy wierzchołki tworzące pokrycie wierzchołkowe w $\overline{G}$ mocno zaciemniono.



Zauważmy, że każda krawędź w $\overline{G}$ jest incydentna z co najmniej jednym mocno zaciemnionym wierzchołkiem.

Musimy wykazać, że G ma *k*-klikę wtedy i tylko wtedy, gdy $\overline{G}$ ma pokrycie wierzchołkowe rozmiaru $n - k$. Zaczniemy od założenia, że *G* ma *k*-klikę *C*. Niech *S* składa się z $n - k$ wierzchołków nie należących do *C*. Twierdzę, że każda krawędź $\overline{G}$ jest incydentna z przynajmniej jednym wierzchołkiem w *S*. Niech (*u*, *v*) będzie dowolną krawędzią w $\overline{G}$. Jest ona w $\overline{G}$, ponieważ nie należała do *G*. Skoro (*u*, *v*) nie ma w *G*, to przynajmniej jeden z wierzchołków: *u* lub *v*, nie należy do kliki *C* w *G*, ponieważ każdą parę wierzchołków w *C* łączy jakaś krawędź. Zważywszy, że co najmniej jeden z wierzchołków: *u* lub *v*, nie należy do *C*, przynajmniej jeden z nich należy do *S*, co oznacza, że krawędź (*u*, *v*) jest incydentna z przynajmniej jednym z wierzchołków *S*. Ponieważ wybraliśmy (*u*, *v*) jako dowolną krawędź w $\overline{G}$, dochodzimy do wniosku, że *S* jest pokryciem wierzchołkowym $\overline{G}$.

Idziemy teraz w drugą stronę. Załóżmy, że $\overline{G}$ ma pokrycie wierzchołkowe *S* zawierające $n - k$ wierzchołków, i niech *C* składa się z *k* krawędzi nie należących do *S*. Każda krawędź $\overline{G}$ jest incydentna z pewnym wierzchołkiem *S*. Innymi słowy, jeśli (*u*, *v*) jest krawędzią w *G*, to przynajmniej jeden z wierzchołków: *u* lub *v*, jest w *S*. Jeżeli przypomnisz sobie definicję antytezy, zauważysz, że antytezą tej implikacji jest, że jeśli ani *u*, ani *v* nie należą do *S*, to (*u*, *v*) nie należy do $\overline{G}$ i dlatego (*u*, *v*) należy do *G*. Innymi słowy, jeżeli zarówno *u*, jak i *v* należą do *C*, to krawędź (*u*, *v*) występuje w *G*. Ponieważ *u* oraz *v* są dowolną parą wierzchołków w *C*, wnioskujemy, że w *G* istnieje krawędź między każdą parą wierzchołków należących do *C*. A to oznacza, że *C* jest *k*-kliką.

Wykazaliśmy zatem, że istnieje redukcja w czasie wielomianowym NP-zupełnego problemu ustalenia, czy nieskierowany graf zawiera *k*-klikę do problemu ustalenia, czy nieskierowany graf zawiera pokrycie wierzchołkowe rozmiaru $n - k$. Gdyby Ci dano nieskierowany graf *G* i chciałbyś się przekonać, czy zawiera on *k*-klikę, to mógłbyś skorzystać z przedstawionej tu konstrukcji do zamiany *G* w czasie wielomianowym na $\overline{G}$ i ustalić, czy $\overline{G}$ zawiera pokrycie wierzchołkowe mające $n - k$ wierzchołków. A gdybyś potrafił w czasie wielomianowym ustalić, czy $\overline{G}$ ma pokrycie wierzchołkowe rozmiaru $n - k$, to poradziłbyś sobie ze sprawdzeniem w wielomianowym czasie, czy *G* ma *k*-klikę. Ponieważ problem kliki jest NP-zupełny, więc problem pokrycia wierzchołkowego — również. Dodatkowo, jeśli potrafiłbyś określić nie tylko, czy $\overline{G}$ ma pokrycie wierzchołkowe $n - k$ wierzchołkami, lecz również które wierzchołki je tworzą, to mógłbyś skorzystać z tych informacji do znalezienia wierzchołków w *k*-klice.

Cykl Hamiltona i ścieżka Hamiltona

Poznaliśmy już problem cyklu Hamiltona: czy nieskierowany graf spójny zawiera cykl Hamiltona (ścieżkę zaczynającą się i kończącą w tym samym wierzchołku i przechodzącą przez wszystkie inne wierzchołki tylko jeden raz)? Zastosowania tego problemu są cokolwiek trudne do przeniknięcia, lecz na podstawie drzewa genealogicznego NP-zupełności możesz się przekonać, że używamy go do wykazania,

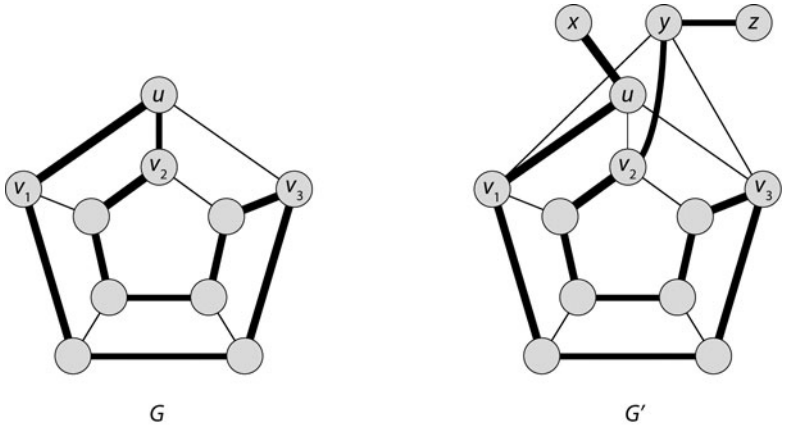
iż problem komiwojażera jest NP-zupełny, a praktyczne zastosowania problemu komiwojażera zdążyliśmy już poznać.

Blisko spokrewnionym problemem jest **problem ścieżki Hamiltona** (ang. *hamiltonian-path problem*), w którym pytamy, czy graf zawiera ścieżkę wiodącą przez każdy wierzchołek tylko raz, nie wymagamy jednak, aby ścieżka tworzyła cykl. Ten problem także jest NP-zupełny i nieco dalej skorzystamy z niego do wykazania, że problem najdłuższej ścieżki prostej jest NP-zupełny.

Certyfikaty obu problemów hamiltonowskich są oczywiste — stanowi je uporządkowanie wierzchołków w cyklu lub ścieżce Hamiltona. (W wypadku cyklu Hamiltona nie powtarzaj pierwszego wierzchołka na końcu). Mając certyfikat, potrzebujemy tylko sprawdzić, czy każdy wierzchołek występuje jeden raz na liście i czy graf zawiera krawędź między każdą parą kolejnych wierzchołków w podanym uporządkowaniu. W odniesieniu do problemu cyklu Hamiltona musimy też sprawdzić, czy istnieje krawędź między pierwszym i ostatnim wierzchołkiem.

Nie będę wnikał w szczegóły redukcji w czasie wielomianowym z problemu pokrycia wierzchołkowego do problemu cyklu Hamiltona, która wykazuje, że ten drugi jest NP-trudny. Jest ona dość skomplikowana i opiera się na **gadżecie** — fragmencie grafu wymuszającym pewne cechy. Gadżet używany w redukcji ma tę własność, że dowolny cykl Hamiltona w grafie skonstruowanym za pomocą redukcji może prowadzić przez gadżet tylko jednym z trzech sposobów.

Żeby zredukować problem cyklu Hamiltona do problemu ścieżki Hamiltona, rozpoczynamy od spójnego, nieskierowanego grafu G o n wierzchołkach i na jego podstawie budujemy nowy nieskierowany graf spójny G' mający $n + 3$ wierzchołki. Wybieramy z G dowolny wierzchołek u i zakładamy, że sąsiaduje on z wierzchołkami v_1, v_2,..., v_k. Aby zbudować G', dodajemy trzy nowe wierzchołki: x, y i z oraz krawędzie (u, x) i (y, z), a także krawędzie (v_1, y), (v_2, y),...,(v_k, y) między y i każdym z wierzchołków sąsiadujących z u. Oto przykład:



G *G'*

Grube krawędzie wskazują cykl Hamiltona w G i odpowiednią ścieżkę Hamiltona w G'. Redukcja zajmuje czas wielomianowy, ponieważ G' zawiera tylko trzy wierzchołki więcej niż G i najwyżej $n + 1$ dodatkowych krawędzi.

Jak zwykle musimy wykazać, że redukcja działa poprawnie, tzn. że G ma cykl Hamiltona wtedy i tylko wtedy, gdy G' ma ścieżkę Hamiltona. Załóżmy, że G ma cykl Hamiltona. Musi on zawierać krawędź (u, v_i) dla pewnego wierzchołka v_i, sąsiadującego z u, więc również sąsiadującego z y w G'. Aby skonstruować ścieżkę Hamiltona w G, prowadzącą z x do z, bierzemy wszystkie krawędzie cyklu Hamiltona z wyjątkiem (u, v_i) i dodajemy krawędzie (u, x), (v_i, y) i (y, z). W powyższym przykładzie v_i jest wierzchołkiem v_2, więc ścieżka Hamiltona omija krawędź (v_2, u) i dodaje krawędzie (u, x), (v_2, y) i (y, z).

Załóżmy teraz, że G' ma ścieżkę Hamiltona. Ponieważ wierzchołki x oraz z mają tylko po jednej krawędzi incydentnej, ścieżka Hamiltona musi prowadzić z x do z i musi zawierać krawędź (v_i, y) dla pewnego wierzchołka sąsiadującego z y, a więc sąsiadującego z u. Żeby znaleźć w G cykl Hamiltona, usuwamy x, y oraz z, a także wszystkie krawędzie z nimi incydentne, i używamy tych wszystkich krawędzi w ścieżce Hamiltona, w G', razem z (v_i, u).

Stosuje się tu rozwiązanie podobne do używanego w naszych poprzednich redukcjach. Istnieje redukcja w czasie wielomianowym problemu NP-zupełnego polegającego na ustaleniu, czy spójny graf nieskierowany zawiera cykl Hamiltona, do problemu ustalenia, czy spójny graf nieskierowany zawiera ścieżkę Hamiltona. Ponieważ pierwszy problem jest NP-zupełny, drugi jest również takim. Ponadto znajomość krawędzi ścieżki Hamiltona umożliwia określenie krawędzi w cyklu Hamiltona.

Komiwojażer

W decyzyjnej wersji **problemu komiwojażera** mamy zadany nieskierowany graf pełny z nieujemną, całkowitą wagą każdej krawędzi i nieujemną liczbę całkowitą k. **Graf pełny** (ang. *complete graph*) ma krawędź między każdą parą wierzchołków, jeżeli więc graf pełny ma n wierzchołków, to ma $n(n – 1)$ krawędzi. Pytamy, czy graf ma cykl zawierający wszystkie wierzchołki, którego sumaryczna waga wynosi najwyżej k.

Łatwo pokazać, że ten problem jest w NP. Certyfikat składa się z wierzchołków cyklu, wymienionych po kolei. Łatwo możemy sprawdzić w czasie wielomianowym, czy krawędziami tego cyklu można dotrzeć do wszystkich wierzchołków i czy ich łączna waga wynosi k lub mniej.

Aby pokazać, że problem komiwojażera jest NP-trudny, dokonujemy redukcji z problemu cyklu Hamiltona i jest to kolejna prosta redukcja. Mając dany na wejściu problemu cyklu Hamiltona graf G, konstruujemy graf pełny G' o tych samych wierzchołkach co G. Nadajmy krawędzi (u, v) w G' wagę 0, jeśli (u, v) jest krawędzią w G, lub wagę 1, jeśli w G nie ma krawędzi (u, v). Nadajmy k wartość 0. Redukcja zużywa czas wielomianowy względem rozmiaru G, ponieważ jest w niej dodawane najwyżej $n(n – 1)$ krawędzi.

Aby udowodnić, że redukcja działa poprawnie, musimy pokazać, że G ma cykl Hamiltona wtedy i tylko wtedy, gdy G' ma cykl o wadze 0, zawierający wszystkie krawędzie. I znowu dowód jest łatwy. Załóżmy, że G ma cykl Hamiltona. Wówczas każda krawędź w cyklu jest w G, wobec tego każda z tych krawędzi otrzymuje wagę 0 w G'. Tym samym G' ma cykl zawierający wszystkie krawędzie i sumaryczna

waga tego cyklu wynosi 0. Załóżmy teraz na odwrót, że G' ma cykl zawierający wszystkie wierzchołki, mający łączną wagę 0. Wtedy każda krawędź tego cyklu musi również znajdować się w G, wobec czego G ma cykl Hamiltona.

Chyba nie muszę powtarzać znanego już postępowania?

Najdłuższa ścieżka prosta

W decyzyjnej wersji **problemu najdłuższej ścieżki prostej** (najdłuższej ścieżki acyklicznej, ang. *longest-acyclic-path problem*) mamy dany graf nieskierowany G i liczbę całkowitą k, i pytamy, czy G zawiera dwa wierzchołki połączone ścieżką prostą mającą co najmniej k krawędzi.

Po raz kolejny certyfikat problemu najdłuższej ścieżki prostej jest łatwy do zweryfikowania. Składa się on z wymienionych kolejno wierzchołków proponowanej ścieżki. Możemy sprawdzić w czasie wielomianowym, czy lista zawiera przynajmniej $k + 1$ wierzchołków ($k + 1$, ponieważ ścieżka mająca k krawędzi zawiera $k + 1$ wierzchołków), z których żaden nie jest powtórzony i między każdą parą kolejnych wierzchołków na liście istnieje krawędź.

Jeszcze jedna prosta redukcja wykazuje, że ten problem jest NP-trudny. Redukujemy z problemu ścieżki Hamiltona. Przyjąwszy, że danymi wejściowymi do problemu ścieżki Hamiltona jest graf G o n wierzchołkach, danymi wejściowymi do problemu najdłuższej ścieżki prostej staje się graf G w postaci niezmienionej i liczba całkowita $k = n - 1$. Jeśli to nie jest redukcja w czasie wielomianowym, to nie wiem, co by nią miało być.

Poprawności tej redukcji dowodzimy przez pokazanie, że G ma ścieżkę Hamiltona wtedy i tylko wtedy, gdy ma ścieżkę prostą złożoną z co najmniej $n - 1$ krawędzi. Lecz ścieżka Hamiltona *jest* ścieżką prostą zawierającą $n - 1$ krawędzi, mamy więc robotę z głowy!

Suma podzbioru

W problemie **sumy podzbioru** (ang. *subset-sum problem*) dane wejściowe stanowią: zbiór S nieuporządkowanych dodatnich liczb całkowitych i docelowa liczba t, również całkowita i dodatnia. Pytamy, czy istnieje podzbiór S' zbioru S, którego elementy dają w sumie t. Na przykład, jeśli S jest zbiorem {1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993}, a t = 138457, to rozwiązaniem jest podzbiór S' = {1, 2, 98, 343, 686, 2409, 17206, 117705}. Certyfikatem jest oczywiście podzbiór zbioru S, który weryfikujemy po prostu przez zsumowanie jego elementów i wykazanie, że ich suma równa się t.

Jak możesz sprawdzić w drzewie genealogicznym NP-zupełności, NP-trudność problemu sumy podzbioru wykazujemy za pomocą redukcji ze spełnialności 3-CNF. To kolejna redukcja przekraczająca granicę dziedzin problemów, przekształcająca problem z dziedziny logiki w problem arytmetyczny. Przekonasz się, że przekształcenie jest sprytne, lecz w ostatecznym rozrachunku — całkiem oczywiste.

Zaczniemy od formuły boolowskiej F problemu 3-CNF, mającej n zmiennych i k klauzul. Nazwijmy zmienne v_1, v_2, v_3,...,v_n, a klauzule C_1, C_2, C_3,..., C_k. Każda klauzula zawiera dokładnie trzy literały (pamiętamy, że każdy literał jest postaci v_i

lub NOT v_i) połączone za pomocą operatorów OR, a cała formuła F ma postać C_1 AND C_2 AND C_3 AND ... AND C_k. Ujmując inaczej, dla danego przypisania 0 lub 1 do każdej ze zmiennych, każda z klauzul jest spełniona, jeśli dowolny z jej literałów daje w wyniku 1, a cała formuła F jest spełniona tylko wtedy, kiedy są spełnione jej wszystkie klauzule.

Nim skonstruujemy zbiór S dla problemu sumy podzbioru, zbudujmy docelową liczbę t na podstawie formuły F problemu 3-CNF. Skonstruujemy ją jako dodatnią liczbę całkowitą o $n + k$ cyfrach. Najmniej znaczące cyfry t (k cyfr z prawej strony) odpowiadają k klauzulom F i każda z nich jest równa 4. Najbardziej znaczące n cyfr t odpowiada n zmiennym F i każda z nich równa się 1. Jeśli formuła F ma — załóżmy — trzy zmienne i cztery klauzule, to t wyniesie 1114444. Jak zobaczymy, jeśli istnieje podzbiór S sumujący się do t, to cyfry t, które odpowiadają zmiennym (jedynki), zapewnią, że przypiszemy wartość do każdej zmiennej F, a cyfry t, które odpowiadają klauzulom (czwórki), zapewnią, że każda klauzula F będzie spełniona.

Zbiór S będzie się składał z $2n + 2k$ liczb całkowitych. Zawiera on liczby całkowite x_i i x_i' dla każdej z n zmiennych v_i w formule F problemu 3-CNF i zawiera liczby całkowite q_j i q_j' dla każdej z k klauzul C_j w F. Każdą liczbę całkowitą zbioru S konstruujemy jako dziesiętną, cyfra po cyfrze. Zobaczmy przykład z $n = 3$ zmiennymi i $k = 4$ klauzulami, tzn. formuła F ma postać $F = C_1$ AND C_2 AND C_3 AND C_4, i niech klauzulami będą:

$C_1 = v_1$ OR (NOT v_2) OR (NOT v_3),
C_2 = (NOT v_1) OR (NOT v_2) OR (NOT v_3),
C_3 = (NOT v_1) OR (NOT v_2) OR v_3,
$C_4 = v_1$ OR v_2 OR v_3.

Oto odpowiedni zbiór S i liczba docelowa t:

		v_1	v_2	v_3	v_4	c_1	c_2	c_3
x_1	=	1	0	0	1	0	0	1
x'_1	=	1	0	0	0	1	1	0
x_2	=	0	1	0	0	0	0	1
x'_2	=	0	1	0	1	1	1	0
x_3	=	0	0	1	0	0	1	1
x'_3	=	0	0	1	1	1	0	0
q_1	=	0	0	0	1	0	0	0
q'_1	=	0	0	0	2	0	0	0
q_2	=	0	0	0	0	1	0	0
q'_2	=	0	0	0	0	2	0	0
q_3	=	0	0	0	0	0	1	0
q'_3	=	0	0	0	0	0	2	0
q_4	=	0	0	0	0	0	0	1
q'_4	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

Zauważmy, że zacieniowane elementy *S* — 1000110, 101110, 10011, 1000, 2000, 200, 10, 1 i 2 — dają w sumie 1114444. Niedługo zobaczymy, czemu te elementy odpowiadają w formule *F* problemu 3-CNF.

Budujemy liczby całkowite w *S* w ten sposób, aby — cyfra po cyfrze — każda kolumna w powyższym diagramie sumowała się albo do 2 (skrajne *n* kolumn z lewej), albo do 6 (skrajne *k* kolumn z prawej). Zauważmy, że jeśli dodajemy elementy *S*, to na żadnej pozycji nie może wystąpić przeniesienie, możemy więc pracować na liczbach cyfra po cyfrze.

Każdy wiersz diagramu jest zaetykietowany elementem zbioru *S*. Pierwsze 2*n* wierszy odpowiada *n* zmiennym formuły 3-CNF, a ostatnie 2*k* wierszy jest „luźnych" — z ich przeznaczeniem zapoznamy się później. Wiersze z etykietami x_i i x_i' odpowiadają, stosownie, wystąpieniom w *F* literałów v_i i NOT v_i. Powiemy, że te wiersze „są" literałami, rozumiejąc przez to, że odpowiadają one literałom. Dążymy do włączenia do podzbioru *S'* dokładnie *n* z pierwszych 2*n* wierszy — w istocie po jednym z każdej pary x_i, x_i' — co będzie odpowiadało spełniającemu przypisaniu w formule *F* problemu 3-CNF. Ponieważ wymagamy, żeby wiersze, które wybieramy spośród literałów, sumowały się do 1 w każdej z *n* kolumn po lewej stronie, zapewniamy, że dla każdej zmiennej v_i formuły 3-CNF, zaliczamy do *S'* jeden wiersz: x_i lub x_i', lecz nie oba. Skrajne prawe *k* kolumn zapewnia, że wiersze, które zaliczamy do *S'*, są literałami spełniającymi każdą klauzulę w formule 3-CNF.

Skoncentrujmy się przez chwilę na *n* kolumnach z lewej strony, zaetykietowanych zmiennymi v_1, v_2,..., v_n. Dla danej zmiennej v_i zarówno x_i, jak i x_i' mają 1 w pozycji cyfry odpowiadającej v_i i 0 na pozostałych pozycjach cyfr, odpowiadających innym zmiennym. Na przykład trzy skrajne cyfry z lewej odnoszące się do zarówno do x_2, jak i do x_2' to 010. W ostatnich 2*k* wierszach i w *n* kolumnach z lewej występują cyfry 0. Ponieważ cel *t* ma 1 na pozycji każdej zmiennej, x_i albo x_i' (dokładnie jedna z nich) musi być w podzbiorze *S'*, aby współtworzyć sumę. Należenie x_i do *S'* odpowiada nadaniu v_i wartości 1, a należenie x_i' do *S'* odpowiada nadaniu v_i wartości 0.

Kierujemy teraz uwagę na *k* kolumn z prawej strony. Odpowiadają one klauzulom. Te kolumny zapewniają, że każda klauzula jest spełniona, o czym przekonamy się niżej. Jeśli literał v_i występuje w klauzuli C_j, to x_i ma jedynkę w kolumnie dotyczącej C_j; jeśli w klauzuli C_j występuje literał NOT v_i, to x_i' ma jedynkę w kolumnie dotyczącej C_j. Ponieważ każda klauzula w formule 3-CNF zawiera dokładnie trzy różne literały, kolumna dotycząca każdej klauzuli musi zawierać dokładnie trzy jedynki we wszystkich wierszach x_i i x_i'. Dla danej klauzuli C_j wiersze spośród pierwszych 2*n* dołączonych do *S'* odpowiadają spełnieniu 0, 1, 2 lub 3 literałów w C_j, toteż te wiersze wnoszą 0, 1, 2 lub 3 do sumy dotyczącej kolumny C_j.

Jednak docelową cyfrą każdej klauzuli jest 4, i tu przydają się „luźne" elementy q_j i q_j' dla $j = 1, 2, 3,..., k$. Zapewniają one, że dla każdej klauzuli podzbiór *S'* zawiera pewien literał w klauzuli (któryś z x_i lub x_i' mający 1 w kolumnie tej klauzuli).

Wiersz dotyczący q_j ma 1w kolumnie klauzuli C_j i 0 we wszystkich innych miejscach, a wiersz dotyczący q_j' jest taki sam, z tym że ma 2. Aby otrzymać docelową cyfrę 4, możemy dorzucić te wiersze, lecz tylko wtedy, kiedy podzbiór *S'* zawiera choć jeden literał z C_j. To, który z luźnych wierszy ma być dodany, zależy od tego, ile literałów z klauzuli C_j włączono do *S'*. Jeśli *S'* zawiera tylko jeden literał, to są potrzebne oba luźne wiersze, ponieważ suma w kolumnach wynosi 1 z literału plus 1 z q_j plus 2 z q_j'. Jeśli *S'* zawiera dwa literały, to wystarczy q_j', ponieważ na sumę w kolumnach składa się 2 z dwóch literałów plus 2 z q_j'. Jeśli *S'* zawiera trzy literały, to potrzebny jest tylko q_j, ponieważ suma kolumn wyniesie 3 z trzech literałów plus 1 z q_j. Gdy jednak do *S'* nie włączono żadnych literałów z klauzuli C_j, to $q_j + q_j' = 3$ nie wystarczy, aby osiągnąć docelową cyfrę 4. Dlatego docelową cyfrę 4 możemy osiągnąć dla każdej klauzuli tylko wtedy, kiedy któryś z literałów klauzuli został zaliczony do *S'*.

Skoro już obejrzeliśmy redukcję, możemy się przekonać, że zabiera ona czas wielomianowy. Tworzymy $2n + 2k + 1$ liczb całkowitych (licząc wraz z docelową *t*), każda o $n + k$ cyfrach. Patrząc na diagram, możesz zaobserwować, że wśród skonstruowanych liczb żadne dwie nie są równe, więc *S* jest naprawdę zbiorem. (Definicja zbioru nie dopuszcza powtórzonych elementów).

Aby dowieść, że redukcja działa poprawnie, musimy wykazać, że formuła *F* problemu 3-CNF ma przypisanie spełniające wtedy i tylko wtedy, gdy istnieje podzbiór *S'* zbioru *S*, który sumuje się dokładnie do *t*. W tym miejscu pomysł jest już Ci znany, zróbmy jednak małą rekapitulację. Po pierwsze, załóżmy, że *F* ma przypisanie spełniające. Jeśli w tym przypisaniu v_i otrzymuje wartość 1, to włączamy x_i do *S'*; w przeciwnym razie włączamy x_i'. Ponieważ tylko jedno: x_i albo x_i', należy do *S*, kolumna dotycząca v_i musi sumować się do 1, pasując do odpowiedniej cyfry w *t*. Ponieważ przypisanie spełnia każdą klauzulę C_j, wiersze x_i i x_i' muszą wnosić 1, 2 lub 3 (liczba literałów w C_j równych 1) do sumy w kolumnie klauzuli C_j. Dołączając do *S'* niezbędne luźne wiersze q_j i (lub) q_j', osiągamy docelową cyfrę 4.

Załóżmy na odwrót, że *S* ma podzbiór *S'*, którego suma jest równa *t*. Aby liczba *t* miała 1 na *n* pozycjach od lewej strony, *S'* musi zawierać tylko jedną liczbę: x_i albo x_i' dla każdej zmiennej v_i. Jeśli zawiera x_i, to nadajemy v_i wartość 1; jeżeli zawiera x_i', to ustawiamy v_i na 0. Ponieważ w wyniku zsumowania luźnych wierszy q_j i q_j' nie można uzyskać docelowej cyfry 4 w kolumnie klauzuli C_j, podzbiór *S'* musi również zawierać przynajmniej jeden wiersz, x_i lub x_i', z jedynką w kolumnie dotyczącej C_j, i klauzula jest spełniona. Jeśli *S'* zawiera x_i', to w klauzuli występuje literał NOT v_i i klauzula jest spełniona. Tym samym każda klauzula jest spełniona, istnieje więc przypisanie spełniające formułę *F* problemu 3-CNF.

W ten sposób zauważamy, że gdybyśmy rozwiązali w czasie wielomianowym problem sumy podzbioru, to potrafilibyśmy również określić, czy formuła 3-CNF jest spełnialna w czasie wielomianowym. Ponieważ spełnialność 3-CNF jest NP-

zupełna, więc dotyczy to także problemu sumy podzbioru. Co więcej, jeśli wiemy, które liczby całkowite w skonstruowanym zbiorze S dają w sumie wartość docelową t, to potrafimy określić, jak podstawić zmienne w formule 3-CNF, aby w wyniku jej obliczenia uzyskać 1.

Z redukcją, którą się posłużyłem, wiąże się jeszcze jedna uwaga: cyfry nie muszą być dziesiętne. Istotne jest, aby podczas dodawania liczb całkowitych nie następowały przeniesienia z jednej pozycji na drugą. Skoro żadna z sum w kolumnach nie może przekroczyć 6, dopuszczalne byłoby interpretowanie liczb z użyciem dowolnej podstawy 7 lub większej. Przykład, który podałem na początku tego omówienia, pochodzi w istocie z liczb w diagramie, lecz interpretowanych przy podstawie 7.

Podział

Problem podziału (ang. *partition problem*) jest blisko spokrewniony z problemem sumy podzbioru. W gruncie rzeczy jest to specjalny przypadek problemu sumy podzbioru: jeśli z równa się sumie wszystkich liczb całkowitych w zbiorze S, to cel t wynosi dokładnie $z/2$. Inaczej mówiąc, cel polega na określeniu, czy istnieje podział zbioru S na dwa rozłączne zbiory S' i S'', takie że każda liczba całkowita z S należy albo do S', albo do S'', lecz nie do obu naraz (to właśnie oznacza, że S' i S'' tworzą podział S), a suma liczb całkowitych w S' równa się sumie liczb całkowitych w S''. Podobnie jak w problemie sumy podzbioru, certyfikatem jest podzbiór zbioru S.

Aby wykazać, że problem podziału jest NP-trudny, zredukujemy problem sumy podzbioru. (Nie jest to wielkim zaskoczeniem). Mając jako dane wejściowe problemu sumy podzbioru zbiór R dodatnich liczb całkowitych i cel t określony dodatnią liczbą całkowitą, konstruujemy zbiór S jako dane wejściowe problemu podziału. Najpierw obliczamy z jako sumę wszystkich liczb całkowitych w R. Zakładamy, że z nie jest równe $2t$, bo gdyby było, to problem od razu stałby się problemem podziału. (Jeśli $z = 2t$, to $t = z/2$, próbujemy więc znaleźć podzbiór R, który daje tę samą sumę co suma liczb nie należących do podzbioru). Następnie wybieramy dowolną liczbę całkowitą y większą od $t + z$ i od $2z$. Definiujemy zbiór S jako zawierający wszystkie liczby z R i dwie dodatkowe liczby całkowite: $y - t$ oraz $y - z + t$. Ponieważ y jest większa od $t + z$ i $2z$, wiemy, że zarówno $y - t$, jak i $y - z + t$ są większe niż z (suma liczb całkowitych w R), dlatego te dwie liczby nie mogą należeć do R. (Pamiętamy, że skoro S jest zbiorem, wszystkie jego elementy muszą być niepowtarzalne. Wiemy też — biorąc pod uwagę, że z nie jest równe $2t$ — że musimy mieć $y - t \neq y - z + t$, zatem dwie nowe liczby całkowite są niepowtarzalne). Zauważmy, że suma wszystkich liczb całkowitych w S równa się $z + (y - t) + (y - z + t)$, czyli $2y$. Stąd jeśli S jest podzielony na dwa rozłączne podzbiory, to suma elementów w każdym podzbiorze musi wynosić y.

Aby wykazać poprawność redukcji, musimy udowodnić, że istnieje podzbiór R' zbioru R, którego liczby dają w sumie t wtedy i tylko wtedy, gdy istnieje taki podział S na S' i S'' , że sumy liczb w S' i liczb w S'' są jednakowe. Najpierw załóżmy, że pewien podzbiór R' zbioru R ma liczby całkowite, których suma wynosi t. Wtedy liczby całkowite w R, które nie należą do R', muszą dawać w sumie $z - t$. Zdefiniujmy S' jako zbiór wszystkich liczb całkowitych w R' wraz z $y - t$ (wobec tego

S'' ma $y - z + t$ wraz ze wszystkimi liczbami w R, które nie należą do R'). Musimy tylko wykazać, że suma liczb w S' wynosi y. To wszakże jest łatwe: suma liczb w R' wynosi t, a dodanie $y - t$ daje w sumie y.

Załóżmy na odwrót, że istnieje podział zbioru S na zbiory S' i S'', przy czym w każdym suma elementów równa się y. Twierdzę, że obie liczby całkowite, które dodaliśmy do R, tworząc S ($y - t$ oraz $y - z + t$), nie mogą jednocześnie należeć do S' (i to samo zastrzeżenie dotyczy S''). Dlaczego? Gdyby były w tym samym zbiorze, to suma elementów tego zbioru musiałaby wynosić przynajmniej $(y - t) + (y - z + t)$, a to równa się $2y - z$. Pamiętajmy jednak, że y jest większa niż z (w rzeczywistości większa niż $2z$), zatem $2y - z$ jest większe niż y. Gdyby więc $y - t$ i $y - z + t$ były w tym samym zbiorze, to suma tego zbioru byłaby większa niż y. Wiemy zatem, że jedna z liczb: $y - t$ lub $y - z + t$, należy do S', a druga — do S''. Nie ma znaczenia, o którym zbiorze powiemy, że zawiera $y - t$, powiedzmy więc, że jest ona w S'. Wiemy też, że liczby całkowite w S' dają w sumie y, co oznacza, że liczby w S', inne niż $y - t$, muszą sumować się do $y - (y - t)$, czyli t. Ponieważ $y - z + t$ nie może być również w S', wiemy, że wszystkie inne liczby w S' pochodzą z R. Stąd istnieje podzbiór zbioru R, który sumuje się do t.

Plecak

W **problemie plecakowym** (ang. *knapsack problem*) mamy dany zbiór n przedmiotów o znanej wadze i wartości i pytamy, czy istnieje podzbiór przedmiotów, których łączna waga wynosi najwyżej tyle, co zadana waga W, i których łączna wartość jest przynajmniej taka jak zadana wartość V. Ten problem jest decyzyjną wersją problemu optymalizacyjnego, w którym chcemy zapakować do plecaka podzbiór najcenniejszych przedmiotów, nie przekraczając limitu wagowego. Ów problem optymalizacyjny ma oczywiste zastosowania, jak rozstrzyganie, które przedmioty zabrać na wędrówkę z plecakiem lub co powinien włamywacz zgarnąć w łupie podczas kradzieży.

Problem podziału jest w istocie tylko specjalnym przypadkiem problemu plecakowego, w którym wartość każdego przedmiotu równa się jego wadze i zarówno W, jak i V są równe połowie wagi całkowitej. Gdybyśmy potrafili rozwiązać problem plecakowy w czasie wielomianowym, to potrafilibyśmy rozwiązać problem podziału w czasie wielomianowym. Dlatego problem plecakowy jest co najmniej tak trudny jak problem podziału, i nawet nie musimy się silić na przechodzenie całego procesu redukcji, aby wykazać, że problem plecakowy jest NP-zupełny.

Ogólne strategie

Jak zapewne sobie w tej chwili uświadamiasz, nie ma dobrej na wszystko metody redukowania jednego problemu do innego, aby udowodnić NP-trudność. Niektóre redukcje są całkiem proste, na przykład redukowanie problemu cyklu Hamiltona do problemu komiwojażera, inne są skrajnie skomplikowane. Oto kilka spraw do zapamiętania i niektóre strategie, które często okazują się pomocne.

Przechodź od ogółu do szczegółu

Redukując problem *X* do problemu *Y*, zawsze musisz zacząć od dowolnych danych wejściowych problemu *X*. Dane wejściowe problemu *Y* możesz jednak ograniczać do woli. Na przykład, dane wejściowe redukcji spełnialności 3-CNF do problemu sumy podzbioru winny być *dowolną* formułą 3-CNF, natomiast produkowane przez nią dane wejściowe sumy podzbioru miały specyficzną strukturę: $2n + 2k$ liczb całkowitych w zbiorze, a każda liczba była tworzona w specjalny sposób. W redukcji nie można było wyprodukować *wszelkich* możliwych danych do problemu sumy podzbioru, lecz było to w porządku. Chodzi o to, że potrafimy rozwiązać problem spełnialności 3-CNF za pomocą transformacji [jego] danych wejściowych na dane wejściowe problemu sumy podzbioru, używając potem odpowiedzi dotyczącej problemu sumy podzbioru jako odpowiedzi dotyczącej problemu spełnialności 3-CNF.

Zauważmy jednak, że każda redukcja musi mieć tę postać: przekształć *dowolne* dane wejściowe problemu *X* ma *pewne* danych wejściowe problemu *Y* — także wówczas, gdy redukcje są łączone w łańcuch. Jeśli chcesz zredukować problem *X* do problemu *Y* oraz problem *Y* do problemu *Z*, to pierwsza redukcja musi przekształcać *dowolne* dane wejściowe problemu *X* na *pewne* dane wejściowe problemu *Y*, a *druga* redukcja musi przekształcać *dowolne* dane wejściowe problemu *Y* na *pewne* dane wejściowe problemu *Z*. Nie wystarczy w drugiej redukcji przekształcić tylko tych typów danych wejściowych do *Y*, które są produkowane przez redukcję z *X*.

Skorzystaj z ograniczeń problemu, który redukujesz

Ogólnie biorąc, podczas redukowania problemu *X* do problemu *Y* możesz wybrać problem *X* w ten sposób, aby wymusić więcej ograniczeń w jego danych. Na przykład prawie zawsze jest łatwiej redukować ze spełnialności 3-CNF niż redukować z problemu matki, czyli spełnialności formuł boolowskich. Formuły boolowskie (logiczne) mogą być dowolnie złożone, widziałeś wszakże, jak możemy wykorzystać podczas redukowania strukturę formuł 3-CNF.

Analogicznie, zazwyczaj łatwiej jest redukować z problemu cyklu Hamiltona niż z problemu komiwojażera, mimo że są podobne. Wynika to z tego, że w problemie komiwojażera wagi krawędzi mogą być dowolnymi dodatnimi liczbami całkowitymi, a nie tylko 0 lub 1, czego wymagaliśmy, redukując do niego. Problem cyklu Hamiltona jest bardziej ograniczony, gdyż każda krawędź ma tylko jedną z dwu „wartości": obecna lub nieobecna.

Poszukuj przypadków specjalnych

Kilka problemów NP-zupełnych jest po prostu specjalnymi przypadkami innych problemów NP-zupełnych, tak jak problem podziału jest specjalnym przypadkiem problemu plecakowego. Jeżeli wiesz, że problem *X* jest NP-zupełny i jest specjalnym przypadkiem problemu *Y*, to problem *Y* musi być również NP-zupełny. Jest tak dlatego, że — jak widzieliśmy na przykładzie problemu plecakowego — wielomianowe rozwiązanie problemu *Y* dawałoby automatycznie wielomianowe rozwiązanie problemu *X*. Podpowiada nam to intuicja: problem *Y*, ogólniejszy niż problem *X*, jest co najmniej tak samo trudny [jak *X*].

Wybierz odpowiedni problem do redukcji

Dobrą strategią jest często redukowanie problemu z tej samej lub zbliżonej dziedziny co problem, którego NP-zupełność próbujesz udowodnić. Pokazaliśmy na przykład, że problem pokrycia wierzchołkowego — problem grafowy — był NP-zupełny, redukując z problemu kliki — również problemu grafowego. Na tej zasadzie w drzewie genealogicznym NP-zupełności uwidoczniono, że redukowaliśmy do problemów cyklu Hamiltona, ścieżki Hamiltona, komiwojażera i najdłuższej ścieżki prostej, a wszystkie one dotyczą grafów.

Niekiedy jednak lepiej jest przeskoczyć z jednej dziedziny do drugiej, tak jak wtedy, kiedy redukowaliśmy ze spełnialności 3-CNF do problemu kliki lub do problemu sumy podzbioru. Spełnialność 3-CNF często okazuje się dobrym wyborem do redukcji z przekraczaniem granic dziedzin.

Pozostając w kręgu problemów grafowych, jeśli musisz wybrać fragment grafu, nie zważając na uporządkowanie, to problem pokrycia wierzchołkowego jest niejednokrotnie dobrym miejscem, od którego można zaczynać. Jeśli uporządkowanie ma znaczenie, to rozważ rozpoczęcie od problemu cyklu Hamiltona lub ścieżki Hamiltona.

Ustanawiaj duże nagrody i kary

Kiedy przekształcaliśmy dane wejściowe grafu G związane z problemem cyklu Hamiltona na ważony graf G' jako dane wejściowe do problemu komiwojażera, bardzo nam zależało na zachęcaniu do używania krawędzi występujących w G podczas wybierania krawędzi do marszruty komiwojażera. W tym celu nadaliśmy tym krawędziom bardzo małą wagę: 0. Innymi słowy, mocno nagradzaliśmy używanie tych krawędzi.

Moglibyśmy też nadać krawędziom w G skończoną wagę, a krawędziom nie należącym do G wagę nieskończoną, egzekwując w ten sposób ogromną karę za używanie krawędzi nie należących do G. Gdybyśmy przyjęli to podejście i nadali każdej krawędzi w G wagę W, to moglibyśmy stanąć wobec konieczności określenia docelowej wagi k całej marszruty komiwojażera jako $n\ W$.

Projektuj gadżety

Nie wnikałem w projektowanie gadżetów, ponieważ gadżety mogą być skomplikowane. Gadżety przydają się do wymuszania pewnych cech. Książki przywołane w podrozdziale „Co czytać dalej” dostarczają przykładów konstruowania i użytkowania gadżetów w redukcjach.

Perspektywy

Trzeba przyznać, że nakreśliłem tutaj dość ponury obraz. Wyobraź sobie taki scenariusz: starasz się dojść do algorytmu o czasie wielomianowym, aby rozwiązać jakiś problem, i choć robisz, co się da, nie potrafisz sprostać zadaniu. Po jakimś czasie uradowałoby Cię znalezienie choćby algorytmu o czasie $O(n^5)$, mimo że n^5 rośnie gwałtownie. Być może ten problem jest zbliżony do takiego, o którym wiesz, że jest

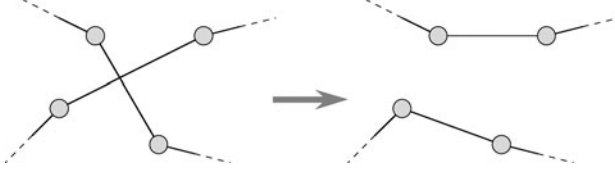
łatwo rozwiązywalny w czasie wielomianowym (jak np. spełnialność 2-CNF w przeciwieństwie do 3-CNF czy cykl Eulera w opozycji do cyklu Hamiltona). Poczujesz więc niesamowitą frustrację, nie potrafiąc dorobić algorytmu wielomianowego do swojego problemu. Zaczniesz w końcu podejrzewać, że być może — tylko być może! — walisz głową w mur, chcąc rozwiązać problem NP-zupełny. I oto widzisz, że zdołasz zredukować do swojego problemu któryś ze znanych problemów NP-zupełnych; wtedy już wiesz, że jest on NP-trudny.

Czy na tym koniec? Nie ma nadziei, że uda Ci się rozwiązać ten problem w jakimkolwiek rozsądnym czasie?

Nie do końca. Kiedy problem jest NP-zupełny, oznacza to, że *pewne* dane wejściowe będą sprawiać kłopoty, lecz niekoniecznie *wszystkie* dane wejściowe będą tak złe. Na przykład znalezienie najdłuższej ścieżki prostej w grafie skierowanym jest NP-zupełne, lecz jeśli wiesz, że graf jest acykliczny, to możesz znaleźć najdłuższą ścieżkę prostą nie tylko w jakimś tam czasie wielomianowym, lecz w czasie $O(n + m)$ (w grafie, który ma n wierzchołków i m krawędzi). Przypomnijmy, że dokonaliśmy tego, poszukując ścieżki krytycznej w diagramie PERT w rozdziale 5. A oto inny przykład: jeśli próbujesz rozwiązać problem podziału i suma liczb całkowitych w zbiorze jest liczbą nieparzystą, to wiesz, że nie ma sposobu na taki podział zbioru, aby obie części miały jednakowe sumy.

Dobre nowiny wychodzą poza takie patologiczne przypadki specjalne. Poczynając od tego miejsca, skupmy się na problemach optymalizacyjnych, których warianty decyzyjne są NP-zupełne, takich jak problem komiwojażera. Niektóre szybkie metody dają dobre, a niekiedy bardzo dobre wyniki. W metodzie **podziału i ograniczeń** (ang. *branch and bound*) poszukiwanie optymalnego rozwiązania jest zorganizowane w strukturę drzewiastą, przy czym następuje tu odcinanie kawałków drzewa, co eliminuje duże porcje przestrzeni poszukiwań, a jest oparte na prostym pomyśle, że jeśli można określić, iż wszystkie rozwiązania biorące początek w jednym węźle drzewa przeszukiwań nie mogą być lepsze od dotychczas znalezionego rozwiązania najlepszego, to nie warto zawracać sobie głowy sprawdzaniem rozwiązań w przestrzeni reprezentowanej przez ten węzeł i to, co znajduje się pod nim.

Inną pomocną techniką jest **przeszukiwanie sąsiedztwa** (ang. *neighborhood search*), w której po przyjęciu jakiegoś rozwiązania są stosowane lokalne operacje mające na celu jego ulepszanie, dopóki nie wyczerpią się możliwości ulepszeń. Rozważmy problem komiwojażera, w którym wszystkie wierzchołki są punktami na płaszczyźnie, a waga każdego wierzchołka jest planarną odległością między punktami. Nawet z tym ograniczeniem problem jest NP-zupełny. W technice **2-opt** ilekroć jakieś krawędzie się krzyżują, zostaną przełączone, czego wynikiem jest skrócenie cyklu:



Ponadto cała chmara **algorytmów aproksymacyjnych** (algorytmów przybliżających, ang. *approximation algorithms*) daje wyniki z gwarancją pozostawania ich w przedziale otaczającym wartość optymalną z określonym współczynnikiem. Jeśli na przykład dane wejściowe problemu komiwojażera spełniają **nierówność trójkąta** (ang. *triangle inequality*) — dla każdego z wierzchołków u, v i x waga krawędzi (u, v) nie przekracza sumy wag krawędzi (u, x) i (x, v) — to istnieje prosty algorytm aproksymacyjny zawsze znajdujący marszrutę komiwojażera, której łączna waga najwyżej dwukrotnie przewyższa najmniejszą, i algorytm ten działa w czasie liniowym względem rozmiaru danych. W danej sytuacji istnieje jeszcze lepsza aproksymacja w czasie wielomianowym, podająca marszrutę o sumarycznej wadze wynoszącej najwyżej 3/2 najmniejszej.

O dziwo, jeśli dwa problemy NP-zupełne są blisko spokrewnione, rozwiązanie tworzone przez dobry algorytm aproksymacyjny dla jednego z nich może okazać się marne w odniesieniu do drugiego. To znaczy rozwiązanie, które jest prawie optymalne dla jednego problemu, niekoniecznie odzworowuje się na rozwiązanie, które jest wszędzie takie dla drugiego problemu.

Niemniej w wielu sytuacjach spotykanych w rzeczywistym świecie rozwiązanie prawie optymalne jest wystarczająco dobre. Nawiązując do spraw dotyczących firmy przewożącej paczki brązowymi furgonetkami: zainteresowani cieszą się w niej ze znajdowanych rozwiązań prawie optymalnych dla swoich furgonetek, nawet jeśli te trasy nie są najlepszymi z możliwych. Każdy dolar, który udaje się zaoszczędzić dzięki planowaniu efektywnych tras, poprawia ich ostateczne wyniki.

Problemy nierozstrzygalne

A teraz, jeśli jesteś pod wrażeniem tego, że problemy NP-zupełne są najtrudniejszymi w świecie algorytmów, czeka Cię mała niespodzianka. Informatycy teoretycy zdefiniowali wielką hierarchię klas złożoności na podstawie ilości czasu i innych zasobów niezbędnych do rozwiązania problemu. Niektóre problemy wymagają takich ilości czasu, które są dowodnie wykładnicze względem rozmiaru danych wejściowych.

Jest nawet gorzej. W odniesieniu do niektórych problemów nie są możliwe żadne algorytmy. To znaczy, że istnieją problemy, dla których w sposób możliwy do udowodnienia nie da się zbudować algorytmu dającego zawsze poprawną odpowiedź. Takie problemy nazywamy **nierozstrzygalnymi** (ang. *undecidable*), a najlepiej znanym spośród nich jest **problem stopu** (problem zatrzymania, ang. *halting problem*), którego nierozstrzygalność udowodnił matematyk Alan Turing w 1937 r. W problemie stopu danymi wejściowymi są: program komputerowy A i dane wejściowe x do programu A. Celem jest ustalenie, czy program A, wykonywany dla danych x, kiedyś się zatrzyma. To znaczy czy działanie A na danych x dobiegnie końca.

Być może sądzisz, że potrafiłbyś napisać program — nazwijmy go programem B — który czyta program A, czyta dane x i symuluje działanie A na danych x. To się sprawdza, o ile program A, działając na danych x, rzeczywiście kończy działanie. A co, jeśli nie kończy? W jaki sposób program B ma poznać, kiedy oznajmić, że A się nigdy nie zatrzyma? Czy sprawdzanie A przez B nie popadłoby w jakąś nie-

skończoną pętlę? Odpowiedź jest taka, że choć mógłbyś napisać *B* w celu sprawdzenia niektórych przypadków, w których *A* się nie zatrzyma, napisanie *takiego* programu *B*, który by zawsze się zatrzymywał i podawał poprawną odpowiedź, czy *A*, działając na danych *x*, się zatrzyma, jest dowodnie niemożliwe.

Ponieważ nie da się napisać programu określającego, czy inny program działający na konkretnych danych wejściowych kiedyś się zatrzyma, nie jest również możliwe napisanie programu, który ustala, czy inny program spełnia warunki swojej specyfikacji. Jak może program określić, czy inny program podaje poprawną odpowiedź, skoro nie może nawet określić, czy ten program się zatrzyma? Nie za wiele jak na doskonałe automatyczne testowanie oprogramowania!

Żeby Ci nie przyszło do głowy, że jedyne nierozstrzygalne problemy dotyczą badania właściwości programów komputerowych, **problem odpowiedniości Posta** (ang. *Post's correspondence problem*, **PCP**) dotyczy napisów — takich, z którymi mieliśmy do czynienia w rozdziale 7. Załóżmy, że mamy co najmniej dwa znaki i dwa wykazy *A* i *B* liczące po *n* napisów utworzonych z tych znaków. Niech w skład *A* wchodzą napisy A_1, A_2, A_3,..., A_n, a w skład *B* — napisy B_1, B_2, B_3,..., B_n. Problem polega na określeniu, czy istnieje taki ciąg indeksów i_1, i_2, i_3,..., i_m, że $A_{i_1} A_{i_2} A_{i_3} \cdots A_{i_m}$ (tzn. połączone ze sobą napisy $A_{i_1}, A_{i_2}, A_{i_3}, ..., A_{i_m}$) daje ten sam napis co $B_{i_1} B_{i_2} B_{i_3} \cdots B_{i_m}$. Załóżmy na przykład, że znakami są litery `e`, `h`, `m`, `n`, `o`, `r` i `y`, $n = 5$ oraz że

A_1 = `ey`, B_1 = `ym`,
A_2 = `er`, B_2 = `r`,
A_3 = `mo`, B_3 = `oon`,
A_4 = `on`, B_4 = `e`,
A_5 = `h`, B_5 = `hon`.

Wówczas jednym z rozwiązań jest ciąg indeksów {5, 4, 1, 3, 4, 2}, ponieważ zarówno $A_5A_4A_1A_3A_4A_2$, jak i $B_5B_4B_1B_3B_4B_2$ tworzą `honeymooner`[13]. Oczywiście, jeśli istnieje jedno rozwiązanie, to istnieje nieskończona liczba rozwiązań, jako że możesz powtarzać dany ciąg indeksów rozwiązania (otrzymując `honeymoonerhoneymooner` itd.). Żeby problem PCP stał się nierozstrzygalny, musimy dopuścić, by napisy w *A* i *B* mogły występować wielokrotnie, w przeciwnym razie można by po prostu wyliczyć wszystkie możliwe kombinacje napisów.

Aczkolwiek sam problem odpowiedniości Posta nie wydaje się zbyt ciekawy, możemy go redukować do innych problemów, aby wykazać, że również one są nierozstrzygalne. U podstaw tego leży ten sam pomysł, z którego korzystaliśmy, żeby wykazać NP-trudność problemu: mając egzemplarz PCP, przekształć go w egzemplarz pewnego problemu *Q*, tak aby odpowiedź dotycząca egzemplarza *Q* stanowiła odpowiedź dla egzemplarza PCP. Gdybyśmy potrafili rozstrzygnąć *Q*, to potrafilibyśmy rozstrzygnąć PCP. Skoro jednak wiemy, że nie możemy rozstrzygnąć PCP, to również *Q* musi być nierozstrzygalny.

[13] *Honeymooner*, z ang. nowożeniec — *przyp. tłum.*

Pośród problemów nierozstrzygalnych, do których umiemy zredukować PCP, znajduje się kilka dotyczących **gramatyk bezkontekstowych** (ang. *contex-free grammars*, **CGF**), opisujących składnię większości języków programowania. Gramatyka bezkontekstowa (CFG) jest zbiorem reguł generowania **języka formalnego** (ang. *formal language*), który to termin umożliwia w wykwintny sposób wyrażenie, że chodzi o „zbiór napisów". Redukując z problemu PCP, możemy udowodnić, że nie jest rozstrzygalne określenie, czy dwie gramatyki bezkontekstowe generują ten sam język formalny, czy generują jakieś wspólne napisy lub czy dana gramatyka CFG jest **niejednoznaczna** (ang. *ambiguous*), tzn. czy istnieją dwa sposoby wygenerowania tego samego napisu z użyciem jej reguł.

Podsumowanie

Obejrzeliśmy spory wachlarz algorytmów z dość różnorodnych dziedzin. Poznaliśmy algorytm działający w czasie podliniowym — było nim sortowanie binarne. Widzieliśmy algorytmy zużywające czas liniowy: wyszukiwanie liniowe, sortowanie przez zliczanie, sortowanie pozycyjne, sortowanie topologiczne i znajdowanie najkrótszych ścieżek w grafie skierowanym. Zapoznaliśmy się z algorytmami, które zużywają czas $O(n \lg n)$ — było to sortowanie przez scalanie i sortowanie szybkie. Przyjrzeliśmy się algorytmom działającym w czasie $O(n^2)$: sortowaniu przez wybieranie, sortowaniu przez wstawianie i sortowaniu szybkiemu (w przypadku najgorszym). Oglądaliśmy algorytmy grafowe, które zużywają czas opisany przez pewną nieliniową kombinację liczby wierzchołków n i liczby krawędzi m: algorytm Dijkstry i algorytm Bellmana-Forda. Mieliśmy wgląd w algorytm grafowy wymagający $\Theta(n^3)$ czasu: algorytm Floyda-Warshalla. Teraz dowiedzieliśmy się, że co do niektórych problemów nie mamy pojęcia, czy w ogóle są dla nich możliwe algorytmy wielomianowe. I wreszcie dowiedzieliśmy się nawet, że dla pewnych problemów nie są możliwe żadne algorytmy, niezależnie od czasu działania.

Nawet na podstawie tego stosunkowo krótkiego wprowadzenia w świat algorytmów komputerowych[14] możesz się zorientować, że jest to dziedzina bardzo rozległa, a ta książka zawiera zaledwie maleńki jej skrawek. W dodatku ograniczyłem nasze badania do konkretnego modelu obliczeniowego, w którym operacje wykonuje tylko jeden procesor, a czas wykonania każdej operacji jest mniej więcej taki sam, niezależnie od tego, gdzie w pamięci komputera rezydują dane. W minionych latach zaproponowano wiele alternatywnych modeli obliczeniowych, takich jak modele z wieloma procesorami, modele, w których czas wykonania operacji zależy od umiejscowienia danych, modele, w których dane napływają niepowtarzalnym strumieniem, i modele, w których komputer jest urządzeniem kwantowym.

Widzisz więc, że w dziedzinie algorytmów komputerowych jest mnóstwo pytań, na które jeszcze nie ma odpowiedzi, a także takich, których jeszcze nie postawiono. Zapisz się na wykłady z algorytmów — możesz to nawet zrobić online — i pomóż nam!

[14] Porównaj rozmiar tej książki z objętością CLRS, która w trzecim wydaniu ma 1292 strony.

Co czytać dalej

Książką *o* NP-zupełności jest dzieło Gareya i Johnsona [GJ79]. Jeśli masz ochotę pogłębić ten temat, to ją przeczytaj. W CLRS [CLRS09] jest rozdział dotyczący NP-zupełności, wchodzący dalej w techniczne detale, niż zrobiłem to tutaj. Jest tam również rozdział o algorytmach aproksymacyjnych. Polecam też książkę Sipsera [Sip06], w której podano więcej materiału o obliczalności i złożoności, a także bardzo ładny, krótki i zrozumiały dowód nierozstrzygalności problemu stopu.

Literatura

[AHU74] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. *(Algorytmy i struktury danych*, Helion, 2003)

[AMOT90] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, Robert E. Tarjan: *Faster algorithms for the shortest path problem*. „Journal of the ACM", 1990, 37, 2, s. 213 – 223.

[CLR90] Thomas H. Cormen, Karol E. Leiserson, Ronald L. Rivest: *Introduction to Algorithms*. The MIT Press, first edition, 1990.

[CLRS09] Thomas H. Cormen, Karol E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[DH76] Whitfield Diffie and Martin E. Hellman: *New directions in cryptography*. „IEEE Transactions on Information Theory", 1976, IT-22, 6, s. 644 – 654.

[FIP11] Annex C: Approved random number generators for FIPS PUB 140-2. *Security requirements for cryptographic modules*. *http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexc.pdf*, July 2011. Wersja robocza.

[GJ79] Michael R. Garey, David S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[Gri81] David Gries: *The Science of Programming*. Springer, 1981.

[KL08] Jonathan Katz, Yehuda Lindell: *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2008.

[Knu97] Donald E. Knuth: *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, third edition, 1997. (*Sztuka programowania. Tom I: Algorytmy podstawowe*, WNT, 2002)

[Knu98a] Donald E. Knuth: *The Art of Computer Programming, Volume 2: Seminumeral Algorithms*. Addison-Wesley, third edition, 1998. (*Sztuka programowania. Tom II: Algorytmy seminumeryczne*, WNT, 2002)

[Knu98b] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, second edition, 1998. (*Sztuka programowania. Tom III: Sortowanie i wyszukiwanie*, WNT, 2002)

[Knu11] Donald E. Knuth: *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part I*. Addison-Wesley, 2011.

[Mac12] John MacCormick: *Nine Algorithms That Changed the Future: The Ingenious Ideas That Drive Today's Computers*. Princeton University Press, 2012.

[Mit96] John C. Mitchell: *Foundations for Programming Languages*. The MIT Press, 1996.

[MvOV96] Alfred Menezes, Paul van Oorschot, Scott Vanstone: *Handbook of Applied Cryptography*. CRC Press, 1996.

[RSA78] Ronald L. Rivest, Adi Shamir, Lewnard M. Adleman: *A method for obtaining digital signatures and public-key cryptosystems*. „Communications of the ACM", 1978, 21, 2, s. 120 – 126. Zob. też U.S. Patent 4 405 829.

[Sal08] David Salomon: *A Concise Introduction to Data Compression*. Springer, 2008.

[Sip06] Michael Sipser: *Introduction to the Theory of Computation*. Course Technology, second edition, 2006.

[SM08] Sean Smith, John Marchesini: *The Craft of System Security*. Addison-Wesley, 2008.

[Sto88] James A. Storer: *Data Compression: Methods and Theory*. Computer Science Press, 1988.

[TC11] Greg Taylor, George Cox: *Digital randomness*. „IEEE Spectrum", 2011, 48, 9, s. 32 – 58.

Skorowidz

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Z