

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTEL尼亚

FRAGMENTY KSIĄŻEK ONLINE

Wykłady z informatyki z przykładami w języku C

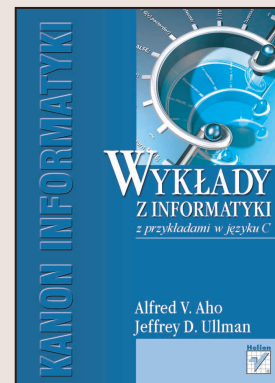
Autorzy: Alfred V. Aho, Jeffrey D. Ullman

Tłumaczenie: Mikołaj Szczepaniak, Bartłomiej Garbacz

ISBN: 83-7361-138-X

Tytuł oryginału: [Foundation of Computer Science in C](#)

Format: B5, stron: 846



Książka Alfreda Aho i Jeffreya Ullmana „Język C” stanowi znaczący postęp w dziedzinie metodyki nauczania podstaw informatyki. Ten nowatorski podręcznik w przystępny sposób prezentuje zagadnienia dotyczące modeli, pojęć i technik z zakresu matematyki dyskretnej i informatyki. Książka stanowi zarówno wprowadzenie do dziedziny informatyki, jak i autorytatywne źródło jej teoretycznych podstaw. Pokazuje, w jaki sposób „matematyczne abstrakcje” przekształca się w działające programy.

Podręcznik dostarcza przyszłym informatikom solidnych podstaw niezbędnych w dalszych studiach oraz w przyszłej pracy zawodowej. Zawiera liczne ćwiczenia, ułatwiające przyswojenie przedstawianej w nim wiedzy i sprawdzenie swoich umiejętności. Autorzy wymagają od czytelnika znajomości języka C.

Zakres tematyczny obejmuje między innymi:

- Iterację, indukcję i rekursję
- Zagadnienia związane z czasem wykonywania programów
- Kombinatorykę i prawdopodobieństwo
- Modele danych oparte na drzewach, listach i zbiorach
- Relacyjny i grafowy model danych
- Wzorce, automaty i wyrażenia regularne, rekurencyjny model wzorców
- Logikę zdań
- Logikę predykatów

Alfred V. Aho jest zastępcą dyrektora ds. informatyki i badań technologicznych w firmie Bellcore. Od 1967 do 1991 roku pracował w Computing Science Research Center w AT&T Bell Laboratories. Wykładał również informatykę na Columbia University, Stanford University oraz Stevens Institute of Technology.



Spis treści

Przedmowa.....	9
----------------	---

1.

Informatyka: mechanizacja abstrakcji	15
1.1. Zagadnienia omawiane w książce	17
1.2. Zagadnienia omawiane w tym rozdziale.....	20
1.3. Modele danych	20
1.4. Model danych języka C.....	27
1.5. Algorytmy i projektowanie programów	35
1.6. Niektóre z wykorzystywanych w tej książce konwencji języka C.....	37
1.7. Podsumowanie rozdziału 1.....	38
1.8. Bibliografia rozdziału 1.....	39

2.

Iteracja, indukcja i rekurencja.....	41
2.1. Zagadnienia poruszane w rozdziale 2.	43
2.2. Iteracje.....	43
2.3. Dowody indukcyjne	51
2.4. Indukcja zupełna	61
2.5. Dowodzenie własności programów	69
2.6. Definicje rekurencyjne	77
2.7. Funkcje rekurencyjne	87
2.8. Sortowanie przez scalanie: rekurencyjny algorytm sortujący.....	93
2.9. Dowodzenie własności programów rekurencyjnych.....	102
2.10. Podsumowanie rozdziału 2.....	105
2.11. Bibliografia rozdziału 2.....	106

3.

Czas działania programów	107
3.1. Zagadnienia poruszane w tym rozdziale	107
3.2. Wybór algorytmu	108
3.3. Pomiar czasu działania programu	109
3.4. Notacja „dużego O” i przybliżony czas działania	114
3.5. Upraszczenie wyrażeń „dużego O”	120
3.6. Analiza czasu działania programu	128
3.7. Reguła rekurencyjna dla szacowania ograniczeń czasów działania	136
3.8. Analizowanie programów zawierających wywołania funkcji	146
3.9. Analizowanie funkcji rekurencyjnych	151
3.10. Analiza sortowania przez scalanie	155
3.11. Rozwiązywanie relacji rekurencyjnych	164
3.12. Podsumowanie rozdziału 3	174
3.13. Bibliografia rozdziału 3	175

4.

Kombinatoryka i prawdopodobieństwo	177
4.1. Zagadnienia poruszane w tym rozdziale	177
4.2. Wariacje z powtórzeniami	178
4.3. Permutacje	182
4.4. Wariacje bez powtórzeń	188
4.5. Kombinacje	191
4.6. Permutacje z powtórzeniami	199
4.7. Rozdzielanie obiektów do koszyków	202
4.8. Łączenie reguł kombinatorycznych	205
4.9. Wprowadzenie do teorii prawdopodobieństwa	209
4.10. Prawdopodobieństwo warunkowe	215
4.11. Rozumowanie probabilistyczne	225
4.12. Oczekiwane wartości obliczeń	235
4.13. Niektóre zastosowania prawdopodobieństwa w programowaniu	238
4.14. Podsumowanie rozdziału 4	244
4.15. Bibliografia rozdziału 4	245

5.

Model danych oparty na drzewach	247
5.1. Zagadnienia poruszane w tym rozdziale	247
5.2. Podstawowa terminologia	248
5.3. Struktury danych dla drzew	256
5.4. Rekurencja w drzewach	263
5.5. Indukcja strukturalna	272
5.6. Drzewa binarne	277
5.7. Drzewa przeszukiwania binarnego	284
5.8. Efektywność operacji na drzewach przeszukiwania binarnego	293
5.9. Kolejki priorytetowe i drzewa częściowo uporządkowane	297
5.10. Sortowanie stogowe — sortowanie za pomocą zrównoważonych drzew częściowo uporządkowanych	306
5.11. Podsumowanie rozdziału 5	311
5.12. Bibliografia rozdziału 5	312

6.

Model danych oparty na listach	313
6.1. Zagadnienia poruszane w tym rozdziale	313
6.2. Podstawowa terminologia	314
6.3. Operacje na listach	318
6.4. Struktura danych — lista jednokierunkowa	320
6.5. Implementacja list oparta na tablicy	329
6.6. Stosy	335
6.7. Wykorzystanie stosu w implementacji wywołań funkcji	341
6.8. Kolejki	347
6.9. Najdłuższy wspólny podciąg	350
6.10. Reprezentowanie ciągów znakowych	357
6.11. Podsumowanie rozdziału 6	364
6.12. Bibliografia rozdziału 6	365

7.

Model danych oparty na zbiorach	367
7.1. Zagadnienia poruszane w tym rozdziale	367
7.2. Podstawowe definicje	367
7.3. Operacje na zbiorach	372
7.4. Implementacja zbiorów oparta na liście	383
7.5. Implementacja zbiorów oparta na wektorze własnym	389
7.6. Mieszanie	392
7.7. Relacje i funkcje	398
7.8. Implementowanie funkcji w formie danych	406
7.9. Implementowanie relacji binarnych	413
7.10. Specyficzne własności relacji binarnych	421
7.11. Zbiory nieskończone	431
7.12. Podsumowanie rozdziału 7	437
7.13. Bibliografia rozdziału 7	437

8.

Relacyjny model danych	439
8.1. Zagadnienia poruszane w tym rozdziale	439
8.2. Relacje	440
8.3. Klucze	447
8.4. Główne struktury przechowywania danych w relacjach	451
8.5. Struktury indeksu drugorzędowego	456
8.6. Poruszanie się wśród wielu relacji	460
8.7. Algebra relacyjna	466
8.8. Implementowanie operacji algebry relacyjnej	473
8.9. Prawa algebraiczne dla relacji	479
8.10. Podsumowanie rozdziału 8	488
8.11. Bibliografia rozdziału 8	489

9.

Grafowy model danych.....	491
9.1. Zagadnienia poruszane w tym rozdziale	491
9.2. Podstawowe pojęcia	492
9.3. Sposoby implementacji grafów	499
9.4. Składowe spójności grafu nieskierowanego	506
9.5. Minimalne drzewa rozpinające	518
9.6. Przeszukiwanie w głąb.....	524
9.7. Zastosowania algorytmu przeszukiwania w głąb.....	536
9.8. Algorytm Dijkstry znajdowania najkrótszych dróg	544
9.9. Algorytm Floyda znajdowania najkrótszych dróg	556
9.10. Wprowadzenie do teorii grafów.....	564
9.11. Podsumowanie rozdziału 9.....	569
9.12. Bibliografia rozdziału 9.....	570

10.

Wzorce, automaty i wyrażenia regularne.....	571
10.1. Zagadnienia poruszane w tym rozdziale	572
10.2. Maszyny stanów i automaty	572
10.3. Automaty deterministyczne i niedeterministyczne	578
10.4. Przechodzenie od niedeterminizmu do determinizmu	588
10.5. Wyrażenia regularne	597
10.6. Rozszerzenia wyrażeń regularnych stosowane w systemie Unix	606
10.7. Prawa algebraiczne wyrażeń regularnych.....	610
10.8. Od wyrażeń regularnych do automatów.....	614
10.9. Od automatów do wyrażeń regularnych.....	624
10.10. Podsumowanie rozdziału 10.....	631
10.11. Bibliografia rozdziału 10.....	631

11.

Rekurencyjny opis wzorców	633
11.1. Zagadnienia poruszane w tym rozdziale	633
11.2. Gramatyki bezkontekstowe	634
11.3. Języki gramatyk.....	641
11.4. Drzewa rozbioru.....	644
11.5. Niejednoznaczność i projektowanie gramatyk.....	652
11.6. Konstruowanie drzew rozbioru	659
11.7. Algorytm analizy składniowej oparty na tabeli.....	667
11.8. Gramatyki a wyrażenia regularne	676
11.9. Podsumowanie rozdziału 11.....	684
11.10. Bibliografia rozdziału 11.....	684

12.

Logika zdań.....	685
12.1. Zagadnienia poruszane w tym rozdziale	685
12.2. Podstawy logiki zdań	686
12.3. Wyrażenia logiczne	688

12.4. Tabele prawdy	692
12.5. Od funkcji boolowskich do wyrażeń logicznych	699
12.6. Określanie wyrażeń logicznych za pomocą tablic Karnaugh'a	704
12.7. Tautologie	712
12.8. Niektóre prawa algebraiczne dla wyrażeń logicznych	717
12.9. Tautologie i metody dowodzenia	726
12.10. Dedukcja	731
12.11. Dowodzenie przez rezolucję	737
12.12. Podsumowanie rozdziału 12	742
12.13. Bibliografia rozdziału 12	743

13.

Wykorzystanie logiki do projektowania komponentów komputerów	745
--	-----

13.1. Zagadnienia poruszane w tym rozdziale	745
13.2. Bramki	746
13.3. Układy	747
13.4. Wyrażenia logiczne i układy	750
13.5. Ograniczenia fizyczne związane z układami	756
13.6. Projekt układu dodawania z zastosowaniem zasady dziel i zwyciężaj	761
13.7. Projekt multiplexera	769
13.8. Elementy pamięciowe	776
13.9. Podsumowanie rozdziału 13	778
13.10. Bibliografia rozdziału 13	778

14.

Logika predykatów	779
-------------------------	-----

14.1. Zagadnienia poruszane w tym rozdziale	779
14.2. Predykaty	780
14.3. Wyrażenia logiczne	782
14.4. Kwantyfikatory	785
14.5. Interpretacje	791
14.6. Tautologie	797
14.7. Tautologie zawierające kwantyfikatory	799
14.8. Dowody w logice predykatów	806
14.9. Dowody na podstawie reguł i faktów	810
14.10. Prawdziwość i możliwość dowodzenia	816
14.11. Podsumowanie rozdziału 14	822
14.12. Bibliografia rozdziału 14	823

Skorowidz	825
-----------------	-----

Iteracja, indukcja i rekurencja

Źródłem potęgi komputerów jest ich zdolność do wielokrotnego wykonywania tego samego zadania lub jego różnych wersji. W informatyce z pojęciem *iteracji* (ang. *iteration*) można się spotkać przy wielu okazjach. Wiele zagadnień związanych z modelami danych, np. listami, opiera się na powtórzeniach typu „lista jest albo pusta, albo składa się z jednego elementu poprzedzającego inny, i kolejny, itd.”. Programy i algorytmy wykorzystują iteracje do wielokrotnego wykonywania określonych zadań bez konieczności definiowania ogromnej liczby pojedynczych kroków, np. w przypadku zadania „wykonaj kolejny krok 1000 razy”. Języki programowania wykorzystują do implementowania algorytmów iteracyjnych pętle, np. `while` czy `for` w języku C.

Zagadnieniem blisko związanym z powtórzeniem jest *rekurencja* (ang. *recursion*) — technika, w której definiuje się pewne pojęcie bezpośrednio lub pośrednio na podstawie tego samego pojęcia. Przykładowo, można by zdefiniować listę stwierdzeniem: „lista albo jest pusta, albo jest elementem poprzedzającym listę”. Rekurencja jest obsługiwana przez wiele języków programowania. W języku C, funkcja *F* może wywoływać samą siebie albo bezpośrednio z poziomu funkcji *F*, albo pośrednio, wywołując inną funkcję, która wywołuje jeszcze inną funkcję itd., aż do pewnej funkcji w tym ciągu, która wywoła funkcję *F*. Innym ważnym zagadnieniem jest *indukcja* (ang. *induction*), która ściśle wiąże się z rekurencją i jest wykorzystywana do dowodzenia wielu twierdzeń matematycznych.

Iteracja, indukcja i rekurencja to podstawowe zagadnienia występujące w wielu formach modeli danych, struktur danych czy algorytmów. Poniższa lista zawiera pewne przykłady wykorzystania tych pojęć, każdy z nich będzie szczegółowo omówiony w tej książce.

- (1) *Techniki iteracyjne*. Najprostszym sposobem wielokrotnego wykonania sekwencji operacji jest wykorzystanie konstrukcji iteracyjnej, jaką jest np. instrukcja `for` w języku C.
- (2) *Programowanie rekurencyjne*. C, podobnie jak wiele innych języków programowania, umożliwia tworzenie funkcji rekurencyjnych, które bezpośrednio lub pośrednio wywołują same siebie. W przypadku początkujących programistów programowanie iteracyjne jest często bezpieczniejsze od programowania rekurencyjnego; jednym z ważniejszych celów niniejszej książki jest jednak przyzwyczajanie Czytelnika do stosowania rekurencji we właściwych momentach. Rekurencyjne programy mogą być łatwiejsze do napisania, analizy i zrozumienia.
- (3) *Dowodzenie indukcyjne*. Ważną techniką dowodzenia prawdziwości twierdzeń jest „dowodzenie indukcyjne”. Zostało ono omówione szczegółowo w podrozdziale 2.3. Poniżej przedstawiono najprostszą formę dowodzenia indukcyjnego. Rozpoczynamy od twierdzenia $S(n)$ zależnego od zmiennej n ; chcemy udowodnić, że $S(n)$ jest prawdziwe. Dowód rozpoczynamy od wykazania podstawy, czyli twierdzenia $S(n)$ dla konkretnego n . Przykładowo, możemy założyć,

Notacja: symbole sumy i iloczynu

Powiększona grecka wielka litera *sigma* jest często wykorzystywana do oznaczenia sumy, np. $\sum_{i=1}^n i$. To konkretne wyrażenie reprezentuje sumę liczb całkowitych od 1 do n , co oznacza, że zastępuje zapis sumy: $1+2+3+\dots+n$. W ogólności, można sumować dowolne funkcje $f(i)$ względem indeksu sumy i (oczywiście indeksem może być symbol inny niż i). Wyrażenie $\sum_{i=a}^b f(i)$ zastępuje:

$$f(a)+f(a+1)+f(a+2)+\dots+f(b)$$

Przykładowo, wyrażenie $\sum_{j=2}^m j^2$ jest równoważne z sumą $4+9+16+\dots+m^2$. W tym przypadku funkcją f jest podnoszenie do kwadratu oraz wprowadzono indeks j zamiast i .

W szczególnych przypadkach, gdy $b < a$, nie ma żadnych składników sumy $\sum_{i=a}^b f(i)$, zatem — zgodnie z konwencją — jej wartość wynosi 0. Jeśli $b = a$, istnieje dokładnie jeden składnik sumy, dla którego $i = a$. Wartością sumy $\sum_{i=a}^a f(i)$ jest więc $f(a)$.

Analogiczną notację wykorzystuje się do reprezentowania iloczynów za pomocą powiększonej wielkiej litery *pi*. Wyrażenie $\prod_{i=a}^b f(i)$ jest równoważne z iloczynem $f(a) \times f(a+1) \times f(a+2) \times \dots \times f(b)$; jeśli $b < a$, iloczyn wynosi 1.

że $n = 0$ i udowodnić prawdziwość twierdzenia $S(0)$. Po drugie, musimy udowodnić krok indukcji, w którym wykazujemy, że twierdzenie S dla jednej wartości jego argumentu wynika z tego samego twierdzenia S dla poprzedniej wartości jego argumentu; $S(n)$ implikuje więc $S(n+1)$ dla wszystkich $n \geq 0$. Przykładowo, $S(n)$ może być prostym wzorem na sumę:

$$\sum_{i=1}^n i = n(n+1)/2, \quad (2.1)$$

który określa, że suma liczb całkowitych od 1 do n jest równa $n(n+1)/2$. Podstawę można określić jako $S(1)$, co oznacza, że w równaniu (2.1) zostaje podstawione 1 w miejsce n . Otrzymana równość ma postać $1 = 1 \times 2/2$. Krok indukcji polega na wykazaniu, że $\sum_{i=1}^n i = n(n+1)/2$ implikuje $\sum_{i=1}^{n+1} i = (n+1)(n+2)/2$ pierwsze równanie to $S(n)$, czyli równanie (2.1); drugie to $S(n+1)$, czyli równanie (2.1) z wyrażeniem $n+1$ podstawionym w miejsce wszystkich wystąpień n . W podrozdziale 2.3 zostanie zaprezentowany sposób konstruowania podobnych dowodów.

- (4) *Dowodzenie poprawności programów.* W informatyce często istnieje konieczność formalnego lub nieformalnego dowiedzenia, że twierdzenie $S(n)$ w danym programie jest prawdziwe. Takie twierdzenie może np. opisywać, co jest prawdą w n -tej iteracji pewnej pętli lub co jest prawdą w n -tym wywołaniu rekurencyjnym pewnej funkcji. Tego typu dowody przeprowadza się zazwyczaj z wykorzystaniem indukcji.
- (5) *Definicje indukcyjne.* Najlepszą formą definiowania wielu ważnych zagadnień w informatyce, szczególnie tych związanych z modelami danych, jest indukcja, w której podstawowa reguła definiuje najprostszy przykład lub przykłady zagadnienia, zaś regułę lub reguły indukcyjne buduje się w celu przedstawienia większych przypadków danego zagadnienia na bazie zdefiniowanych wcześniej mniejszych. Przykładowo, powyżej wspomniano o tym, że listę można zdefiniować

za pomocą podstawowej zasady (lista pusta jest listą) w połączeniu z regułą indukcji (element wraz z następującą po nim listą także tworzą listę).

- (6) *Analiza czasu działania.* Ważnym kryterium oceny jakości algorytmu jest czas potrzebny do wykonania zadania dla danych o różnych rozmiarach. Jeśli algorytm opiera się na rekurencji, wykorzystuje się wzór zwany *równaniem rekurencji* (ang. *recurrence equation*), który jest indukcyjną definicją pozwalającą przewidzieć czas potrzebny danemu algorytmowi na przetworzenie danych wejściowych różnych rozmiarów.

Każde z powyższych zagadnień (poza ostatnim) omówiono w bieżącym rozdziale; czas działania programów jest tematem rozdziału 3.

2.1. Zagadnienia poruszane w rozdziale 2.

W niniejszym rozdziale omówione zostaną następujące podstawowe zagadnienia:

- programowanie iteracyjne (podrozdział 2.2);
- dowody indukcyjne (podrozdziały 2.3 i 2.4);
- definicje indukcyjne (podrozdział 2.6);
- programowanie rekurencyjne (podrozdziały 2.7 i 2.8);
- dowodzenie poprawności programów (podrozdziały 2.5 i 2.9).

Dodatkowo, posługując się przykładami wykorzystania tych pojęć, szczególna uwaga zostanie poświęcona wielu ważnym i interesującym zagadnieniom związanym z informatyką. Będą do nich należeć:

- algorytmy sortujące, w tym sortowanie przez wybieranie (podrozdział 2.2) oraz sortowanie przez scalanie (podrozdział 2.8);
- kontrola parzystości oraz detekcja błędów w danych (podrozdział 2.3);
- wyrażenia arytmetyczne i ich przekształcanie za pomocą praw algebraicznych (podrozdziały 2.4 i 2.6);
- zbilansowane nawiasy zwykłe (podrozdział 2.6).

2.2. Iteracje

Każdy początkujący programista uczy się, jak wykorzystywać iteracje, wprowadzając pewne konstrukcje pętlowe, takie jak instrukcje `for` lub `while` w języku programowania C. W niniejszym podrozdziale zostanie zaprezentowany przykład algorytmu iteracyjnego zwanego „sortowaniem przez wybieranie”. W podrozdziale 2.5 zostanie udowodnione metodą indukcji, że algorytm ten rzeczywiście wykonuje sortowanie, a czas jego działania zostanie przeanalizowany w podrozdziale 3.6. Z kolei w podrozdziale 2.8 zostanie zademonstrowany sposób, w jaki rekurencja może pomóc w opracowaniu efektywnego algorytmu sortującego zgodnie z techniką zwaną „dziel i zwyciężaj”.

Powszechnie stosowane techniki: samodefiniowanie i para podstawowy-indukcyjny

Podczas studiowania bieżącego rozdziału Czytelnik powinien zwrócić uwagę na dwa zagadnienia pojawiające się często w odniesieniu do różnych tematów. Pierwsze to samodefiniowanie, które polega na definiowaniu lub budowaniu pewnego pojęcia odwołując się do niego samego. Przykładowo, wcześniej wspomniano o liście, którą można zdefiniować jako listę pustą lub jako element wraz z następującą po nim listą.

Drugim takim zagadnieniem jest para podstawowy-indukcyjny. Funkcje rekurencyjne zawierają często rodzaj testów dla „podstawowego” przypadku, w którym nie ma wywołania rekurencyjnego, oraz przypadku „indukcyjnego”, który wymaga jednego lub większej liczby wywołań rekurencyjnych. Dobrze znane dowodzenie indukcyjne wymaga kroku podstawowego i indukcyjnego, podobnie jest w przypadku definicji indukcyjnych. Ta para podstawowy-indukcyjny jest tak ważna, że te słowa będą specjalnie podkreślane w tekście, w momencie każdego wystąpienia przypadku podstawowego lub kroku indukcyjnego.

Cykliczność związana z poprawnym zastosowaniem samodefinicji nie jest żadnym paradoksem, ponieważ samodefiniujące się podczęści są zawsze „mniejsze” od definiowanego w ten sposób obiektu. Co więcej, po skończonej liczbie kroków przechodzenia do mniejszych części, dociera się do przypadku podstawowego, w którym kończy się proces samodefiniowania. Przykładowo, lista L jest zbudowana z elementu i listy krótszej o jeden element od L . Kiedy dochodzi się do listy posiadającej zero elementów, otrzymuje się przypadek podstawowy definicji listy: „lista pusta jest listą”.

Jako inny przykład można rozważyć funkcję rekurencyjną, która działa poprawnie, jeśli argumenty jej wywołania są (w pewnym sensie) „mniejsze” od argumentów wywołującej ją innej kopii tej samej funkcji. Co więcej, po pewnej liczbie rekurencyjnych wywołań, należy dojść do argumentów tak „małych”, że funkcja nie będzie już wykonywała żadnych rekurencyjnych wywołań.

Sortowanie

Aby posortować listę n elementów, należy tak permutować jej elementy, by zostały ostatecznie ułożone w porządku niemalejącym.

- **Przykład 2.1.** Przypuśćmy, że mamy listę liczb całkowitych (3, 1, 4, 1, 5, 9, 2, 6, 5). Sortujemy tę listę, permutując ją do postaci sekwencji (1, 1, 2, 3, 4, 5, 5, 6, 9). Należy zauważyć, że sortowanie nie tylko porządkuje wartości tak, że każda jest mniejsza lub równa kolejnej liczbie z listy, ale także zachowuje liczbę wystąpień każdej wartości. Posortowana lista zawiera więc dwie jedynki, dwie piątki i po jednej z każdej z pozostałych liczb znajdujących się na oryginalnej liście. •

Listy elementów dowolnego typu można sortować wówczas, gdy istnieje możliwość zdefiniowania między nimi relacji mniejszości, którą reprezentuje się zazwyczaj znakiem ($<$). Przykładowo, jeśli wartościami do posortowania są liczby całkowite lub zmiennoprzecinkowe, symbol ($<$) oznacza znaną wszystkim relację mniejszości liczb całkowitych lub rzeczywistych. Jeśli natomiast tymi wartościami są ciągi znaków, należy zastosować ich porządek leksykograficzny (patrz ramka „Porządek leksykograficzny”). Niekiedy, gdy dane do posortowania elementy są skomplikowane (np. struktury), do porównania można wykorzystać część każdego z elementów (np. jedno konkretne pole).

Porządek leksykograficzny

Podstawowym sposobem porównywania ciągów znaków jest wykorzystywanie *porządku leksykograficznego* (ang. *lexicographic order*). Niech $c_1c_2\dots c_k$ oraz $d_1d_2\dots d_m$ będą dwoma ciągami, w których kolejne elementy c i d reprezentują pojedyncze znaki. Długości ciągów wynoszą odpowiednio k oraz m i nie muszą być sobie równe. Zakładamy, że istnieje relacja ($<$) porządkująca znaki; przykładowo, w języku C znaki są reprezentowane przez małe liczby całkowite, można więc stałe oraz zmienne znakowe wykorzystać jako liczby całkowite w wyrażeniach arytmetycznych. Stąd można wykorzystać konwencjonalną relację ($<$) dla liczb całkowitych do określania, który z pary rozpatrywanych znaków jest mniejszy. Takie uporządkowanie uwzględnia naturalne wyobrażenie, w którym małe litery znajdujące się wcześniej w alfabecie są „mniejsze” od małych liter znajdujących się w alfabecie później. To samo dotyczy układu wielkich liter.

Można teraz zdefiniować dla ciągów znaków uporządkowanie zwane *leksykograficznym*, *słownikowym* lub *alfabetycznym*. Mówimy, że $c_1c_2\dots c_k < d_1d_2\dots d_m$, jeśli spełniony jest jeden z dwóch poniższych warunków:

- (1) Pierwszy ciąg jest *prefiksem właściwym* (ang. *proper prefix*) drugiego ciągu, co oznacza, że $k < m$ oraz dla $i = 1, 2, \dots, k$ zachodzi $c_i = d_i$. Zgodnie z tą zasadą, $\text{kot} < \text{kotara}$. Szczególnym przypadkiem spełniającym tę zasadę jest sytuacja, w której $k = 0$, czyli gdy pierwszy ciąg nie zawiera żadnych znaków. Do oznaczania *ciagu pustego* (ang. *empty string*), czyli takiego, który nie zawiera żadnych znaków, będzie wykorzystywany znak ε (grecka litera epsilon). Jeśli $k = 0$, pierwsza reguła mówi, że $\varepsilon < s$ dla każdego niepustego ciągu s .
- (2) Dla pewnej wartości $i > 0$, pierwsze $i-1$ znaków dwóch porównywanych ciągów wzajemnie sobie odpowiada, ale i -ty znak pierwszego ciągu jest mniejszy od i -tego znaku drugiego ciągu. Oznacza to, że $c_j = d_j$ dla $j = 1, 2, \dots, i-1$, oraz $c_i < d_i$. Zgodnie z tą zasadą, $\text{kosy} < \text{koty}$, ponieważ te dwa słowa różnią się na pozycji trzeciej — kosy zawiera literę s , która poprzedza literę t znajdującą się na trzeciej pozycji ciągu koty .

Porównanie $a \leq b$ jak zwykle oznacza, że albo $a < b$, albo a i b to te same wartości. O liście $(a_1, a_2, \dots, a_n)$ mówimy, że jest *posortowana* (ang. *sorted*), jeśli $a_1 \leq a_2 \leq \dots \leq a_n$, co oznacza, że wartości są ułożone w porządku niemalejącym. *Sortowanie* (ang. *sorting*) jest operacją, której na wejściu podaje się dowolną listę $(a_1, a_2, \dots, a_n)$, i która zwraca na wyjściu listę $(b_1, b_2, \dots, b_n)$ taką, że:

- (1) Lista $(b_1, b_2, \dots, b_n)$ jest posortowana.
- (2) Lista $(b_1, b_2, \dots, b_n)$ jest *permutacją* (ang. *permutation*) listy oryginalnej. Oznacza to, że każda wartość znajduje się na liście $(a_1, a_2, \dots, a_n)$ dokładnie taką samą liczbę razy, co na wynikowej liście $(b_1, b_2, \dots, b_n)$.

Algorytm sortujący (ang. *sorting algorithm*) pobiera na wejściu dowolną listę i zwraca jako wynik listę posortowaną, która jest permutacją listy wejściowej.

• Przykład 2.2. Rozważmy listę słów:

$\text{koty}, \text{kosy}, \text{las}, \text{kot}, \text{rękawica}, \text{kotara}$

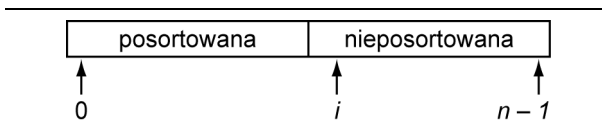
Mając takie dane wejściowe i wykorzystując porządek leksykograficzny, algorytm sortujący zwróci następującą listę wyjściową: $\text{kosy}, \text{kot}, \text{kotara}, \text{koty}, \text{las}, \text{rękawica}$. •

Konwencja związana z nazwami i wartościami

Zmienną można traktować jako skrzynkę zawierającą nazwę i wartość. W przypadku odwoływania się do zmiennej, np. *abc*, do zapisu jej nazwy będzie wykorzystywana czcionka o stałej szerokości znaków. Kiedy jednak odwołanie będzie odnosiło się do wartości zmiennej *abc*, wykorzystywana będzie kursywa, np. *abc*. Podsumowując, *abc* odnosi się do nazwy skrzynki, zaś *abc* do jej zawartości.

Sortowanie przez wybieranie — iteracyjny algorytm sortujący

Przypuśćmy, że mamy tablicę *A* zawierającą *n* liczb całkowitych, którą chcemy posortować w porządku niemalejącym. Można to zrobić wielokrotnie powtarzając krok, w którym wyszukiwany jest najmniejszy¹ element nieposortowanej części tablicy i wymieniany z elementem znajdującym się na pierwszej pozycji nieposortowanej części tablicy. W pierwszej iteracji znajdujemy (wybieramy) najmniejszy element z wszystkich wartości znajdujących się w tablicy *A*[0..*n*-1] i wymieniamy go z elementem *A*[0]². W drugiej iteracji wybieramy najmniejszy element w tablicy *A*[1..*n*-1] i wymieniamy go z elementem *A*[1], następnie wykonujemy kolejne iteracje. Na początku (*i*+1). iteracji, tablica *A*[0..*i*-1] zawiera *i* najmniejszych elementów tablicy *A* ułożonych w porządku niemalejącym, pozostałe elementy tablicy nie są uporządkowane. Stan tablicy *A* bezpośrednio przed (*i*+1). iteracją przedstawiono na rysunku 2.1.



RYSUNEK 2.1.

Stan tablicy bezpośrednio przed (*i*+1). iteracją algorytmu sortowania przez selekcję

W (*i*+1). iteracji znajdujemy najmniejszy element tablicy *A*[*i*..*n*-1] i wymieniamy go z elementem *A*[*i*]. Po tej iteracji tablica *A*[0..*i*] zawiera więc *i*+1 najmniejszych elementów posortowanych w porządku niemalejącym. Po (*i*-1). iteracji posortowana jest cała tablica.

Na listingu 2.1 przedstawiono napisaną w języku C funkcję realizującą sortowanie przez wybieranie. Funkcja, którą nazwano *SelectionSort*, pobiera dwa argumenty: tablicę *A* oraz jej długość *n*.

LISTING 2.1.

Iteracyjne sortowanie przez wybieranie

```
void SelectionSort(int A[], int n)
{
    int i, j, small, temp;
(1) for (i = 0; i < n-1; i++) {
        /* przypisujemy zmiennej small indeks pierwszego */
```

¹ Nie musi to być dokładnie jeden najmniejszy element — w tablicy może istnieć wiele wystąpień najmniejszej wartości. W takim przypadku, wystarczy, że znajdziemy dowolne wystąpienie tej wartości.

² Do opisywania przedziałów elementów w tablicach będzie stosowana konwencja z języka programowania Pascal. Jeśli *A* jest tablicą, to zapis *A*[*i*..*j*] oznacza elementy tej tablicy o indeksach od *i* do *j* włącznie.

```

    /* wystąpienia najmniejszego elementu nieposortowanej*/
    /* części tablicy */
(2)  small = i;
(3)  for (j = i+1; j < n; j++)
(4)      if (A[j] < A[small])
(5)          small = j;
    /* w tym miejscu zmienna small zawiera indeks */
    /* pierwszego najmniejszego elementu tablicy */
    /* A[i..n-1]; wymieniamy więc A[small] z A[i] */
(6)  temp = A[small];
(7)  A[small] = A[i];
(8)  A[i] = temp;
    }
}

```

W wierszach od (2) do (5) zostaje wybrany najmniejszy element nieposortowanej części tablicy, czyli $A[i..n-1]$. Pierwszą operacją jest przypisanie zmiennej indeksowej `small` wartości i w wierszu (2). W pętli `for` w wierszach od (3) do (5) zostają przeanalizowane kolejno wszystkie większe indeksy j i zmiennej `small` zostaje przypisana wartość j , jeśli element $A[j]$ zawiera wartość mniejszą niż wszystkie elementy z przedziału $A[i..j-1]$. Efektem tych działań jest przypisanie zmiennej `small` wartości indeksu pierwszego wystąpienia najmniejszego elementu tablicy $A[i..n-1]$.

Po wybraniu odpowiedniej wartości indeksu dla zmiennej `small` zostaje wymieniony element na wskazywanej przez ten indeks pozycji z elementem $A[i]$ (patrz wiersze od (6) do (8)). Jeśli $small = i$, wymiana jest dokonywana, tyle że nie zmienia ona układu w tablicy. Warto zauważyć, że aby wymienić wartości dwóch elementów w tablicy, trzeba wykorzystać tymczasową zmienną do przechowania wartości jednego z nich. Wartość elementu $A[small]$ zostaje więc przypisana zmiennej `temp` w wierszu (6), wartość elementu $A[i]$ zostaje przypisana elementowi $A[small]$ w wierszu (7) i wreszcie oryginalna wartość elementu $A[small]$ przechowywana w zmiennej `temp` zostaje przypisana do elementu $A[i]$ w wierszu (8).

- **Przykład 2.3.** Poniżej przeanalizowano działanie funkcji `SelectionSort` dla różnych danych wejściowych. Pierwszym przypadkiem jest tablica wejściowa nie zawierająca żadnych elementów. Jeśli $n = 0$, ciało pętli `for` z wiersza (1) nie będzie wykonane ani razu, zatem, zgodnie z oczekiwaniami, funkcja `SelectionSort` nie wykona żadnej operacji.

Kolejnym przypadkiem jest tablica wejściowa zawierająca tylko jeden element. Ponownie, ciało pętli `for` z wiersza (1) nie zostanie wykonane ani razu. Odpowiedź funkcji będzie więc prawidłowa, ponieważ tablica jednoelementowa jest zawsze posortowana. Przypadki, w których n jest równe 0 lub 1 są ważnymi warunkami granicznymi, dla których należy sprawdzać przebieg wszystkich algorytmów czy programów.

Wreszcie, uruchamiamy funkcję `SelectionSort` dla małej tablicy złożonej z czterech elementów od $A[0]$ do $A[3]$:

	0	1	2	3
A	40	30	20	10

Pętla zewnętrzna rozpoczyna działanie przy wartości $i = 0$ i w wierszu (2) zmiennej `small` zostaje przypisana wartość 0. Wiersze od (3) do (5) tworzą pętlę wewnętrzną, w której j przyjmuje kolejno wartości 1, 2 i 3. Dla $j = 1$ spełniony jest warunek z wiersza (4), ponieważ element $A[1]$ o wartości 30

Sortowanie na podstawie kluczy

Kiedy jest przeprowadzany proces sortowania, dla sortowanych wartości stosuje się operację porównania. Porównanie to jest często wykonywane tylko dla określonych części wartości. Taka część używana w operacjach porównywania nosi nazwę *klucza* (ang. *key*).

Przykładowo, wykaz studentów biorących udział w danych zajęciach może być tablicą A zawierającą struktury języka C postaci:

```
struct STUDENT {
    int studentID;
    char *nazwisko;
    char ocena;
} A[MAX];
```

Może zaistnieć potrzeba posortowania tej tablicy na podstawie identyfikatora studenta, nazwiska lub oceny; każdy z tych elementów po kolei stałby się więc kluczem sortowania. Przykładowo, jeśli należałoby posortować struktury na podstawie identyfikatora studenta, wykonywane porównania musiałyby mieć postać:

```
A[j].studentID < A[small].studentID
```

w wierszu (4) funkcji SelectionSort. Typem tablicy A i tymczasowej zmiennej temp (wykorzystywanej do zamiany wartości) byłaby struktura STUDENT, zamiast liczby całkowitej. Należy pamiętać, że wymieniana jest cała struktura, nie tylko pola kluczy.

Ponieważ wymiana całych struktur jest czasochłonna, bardziej wydajnym rozwiązaniem byłoby wykorzystanie drugiej tablicy ze wskaźnikami do struktur STUDENT i sortowanie tylko zawartych w niej wskaźników. Same struktury pozostałyby nienaruszone w pierwszej tablicy. Opracowanie takiej wersji sortowania przez wybieranie autorzy pozostawiają Czytelnikowi jako ćwiczenie.

jest mniejszy od elementu $A[small]$, zawierającego wartość 40. W wierszu (5) zmiennej `small` zostaje więc przypisana wartość 1. W drugiej iteracji pętli zawartej w wierszach od (3) do (5), dla $j = 2$, ponownie spełniony jest warunek z wiersza (4), ponieważ $A[2] < A[1]$. W wierszu (5) następuje więc przypisanie zmiennej `small` wartości 2. W ostatniej iteracji tej pętli, dla $j = 3$, także spełniony jest warunek z wiersza (4), ponieważ $A[3] < A[2]$. W wierszu (5) zmiennej `small` zostaje więc przypisana wartość 3.

W tym momencie następuje przejście z pętli wewnętrznej do wiersza (6). Zmiennej `temp` zostaje przypisana wartość 10, czyli wartość elementu $A[small]$, następnie wartość elementu $A[0]$, 40, zostaje w wierszu (7) przypisana elementowi $A[3]$, po czym elementowi $A[0]$ zostaje przypisana wartość 10 w wierszu (8). Na tym kończy się pierwsza iteracja pętli zewnętrznej; tablica A wygląda w tym momencie następująco:

	0	1	2	3
A	10	30	20	40

W drugiej iteracji pętli zewnętrznej, dla $i = 1$, w wierszu (2) zmiennej `small` zostaje przypisana wartość 1. Pętla wewnętrzna przypisuje początkowo zmiennej `j` wartość 2 i, ponieważ $A[2] < A[1]$,

w wierszu (5) zmiennej *small* zostaje przypisana wartość 2. Dla $j = 3$, warunek z wiersza (4) nie jest spełniony, ponieważ $A[3] \geq A[2]$. Zmienna *small* ma więc wartość 2 w momencie, gdy sterowanie dochodzi do wiersza (6). W wierszach od (6) do (8) zostają zamienione elementy $A[1]$ i $A[2]$, w efekcie czego otrzymuje się tablicę:

	0	1	2	3
A	10	20	30	40

Mimo że tablica A jest już posortowana, musi zostać wykonana ostatnia iteracja pętli zewnętrznej, dla $i = 2$. Zmiennej *small* zostaje więc w wierszu (2) przypisana wartość 2 i uruchomiona pętla wewnętrzna tylko dla przypadku, w którym $j = 3$. Ponieważ warunek w wierszu (4) nie jest spełniony, zmienna *small* nadal przechowuje wartość 2, zatem w wierszach od (6) do (8) następuje „zamiana” elementu $A[2]$ z samym sobą. Czytelnik sam powinien sprawdzić, czy taka wymiana dla *small* = *i* rzeczywiście nie zmienia układu w tablicy. •

Listing 2.2 przedstawia przykładowy sposób wykorzystania funkcji `SelectionSort` w kompletnym programie sortującym ciąg n liczb całkowitych przy założeniu, że $n \leq 100$. W wierszu (1) następuje odczytanie n liczb całkowitych i zapisanie ich w tablicy A. Jeśli liczba danych wejściowych przekroczy MAX, w tablicy zapisywanych jest jedynie pierwsze MAX liczb całkowitych. W tym miejscu przydatny byłby komunikat ostrzegający użytkownika o zbyt dużej liczbie danych, jednak w prezentowanym programie pominięto ten element.

W wierszu (3) następuje wywołanie funkcji `SelectionSort` w celu posortowania tablicy. Wiersze (4) i (5) są odpowiedzialne za wydrukowanie liczb całkowitych w posortowanym porządku.

LISTING 2.2.

Program sortujący elementy tablicy za pomocą algorytmu sortowania przez wybieranie

```
#include <stdio.h>

#define MAX 100
int A[MAX];
void SelectionSort(int A[], int n);

main()
{
    int i, n;
    /* odcytujemy dane wejściowe i zapisujemy je w tablicy A */
(1)   for (n = 0; n < MAX && scanf("%d", &A[n]) != EOF; n++)
(2)       ;
(3)   SelectionSort(A,n); /* sortuje A */
(4)   for (i = 0; i < n; i++)
(5)       printf("%d\n", A[i]); /* wyświetla A */
}

void SelectionSort(int A[], int n)
{
    int i, j, small, temp;
    for (i = 0; i < n-1; i++) {
        small = i;
        for (j = i+1; j < n; j++)
```

```

        if (A[j] < A[small])
            small = j;
        temp = A[small];
        A[small] = A[i];
        A[i] = temp;
    }
}

```

Ćwiczenia

2.2.1. Przeprowadź symulację działania funkcji `SelectionSort` dla tablicy zawierającej następujące elementy:

- (a) 6, 8, 14, 17, 23;
- (b) 17, 23, 14, 6, 8;
- (c) 23, 17, 14, 8, 6.

Ile porównań i zamian będzie wykonywanych dla każdego z powyższych przypadków?

2.2.2.** Jaka jest minimalna i maksymalna liczba (a) porównań i (b) zamian, które funkcja `SelectionSort` musi przeprowadzić do posortowania ciągu n elementów?

2.2.3. Napisz w języku C funkcję, która pobiera jako argumenty dwie listy jednokierunkowe złożone ze znaków i zwraca wartość `TRUE`, jeśli pierwszy ciąg znaków leksykograficznie poprzedza drugi. *Wskazówka:* zaimplementuj opisany w niniejszym rozdziale algorytm porównywania ciągów znaków. Wykorzystaj rekurencję w postaci wywołania funkcji przez nią samą dla reszt ciągów znaków, jeśli zostanie stwierdzone, że pierwsze znaki obu ciągów są takie same. Alternatywnym rozwiązaniem jest opracowanie wykonującego identyczne działania algorytmu iteracyjnego.

2.2.4*. Zmodyfikuj program z ćwiczenia 2.2.3 tak, by ignorował wielkość porównywanych znaków.

2.2.5. Co dzieje się w algorytmie sortowania przez wybieranie, jeśli wszystkie elementy są takie same?

2.2.6. Zmodyfikuj program z listingu 2.2 tak, by program wykonywał sortowanie przez wybieranie dla elementów tablicy nie będących liczbami całkowitymi, lecz strukturami `STUDENT` opisanymi w ramce „Sortowanie na podstawie kluczy”. Zakładamy, że kluczem jest pole `studentID`.

2.2.7*. Jeszcze raz zmodyfikuj program z listingu 2.2 tak, by program sortował elementy dowolnego typu T . Możesz jednak założyć, że istnieje funkcja `key`, która jako argument pobiera element typu T i zwraca klucz dla tego elementu, będący wartością pewnego typu K . Załóż także, że istnieje funkcja `lt`, która jako argumenty pobiera dwa elementy typu K i zwraca wartość `TRUE`, jeśli pierwszy jest „mniejszy” od drugiego; w przeciwnym razie, funkcja powinna zwrócić wartość `FALSE`.

2.2.8. Zamiast wykorzystywać całkowitoliczbowe indeksy w tablicy `A`, możemy do określania pozycji elementów tablicy wykorzystać wskaźniki do liczb całkowitych. Przekształć algorytm sortowania przez wybieranie z listingu 2.2 tak, by wykorzystywał wskaźniki.

2.2.9*. Jak wspomniano w ramce „Sortowanie na podstawie kluczy”, jeśli elementy do posortowania są dużymi strukturami, jak w przypadku struktury `STUDENT`, sensowne jest pozostawienie tych struktur nienaruszonych w ich tablicy i posortowanie wskaźników do nich, znajdujących się w osobnej tablicy. Opracuj także taką wersję sortowania przez wybieranie.

2.2.10. Napisz iteracyjny program wyświetlający odrębne elementy tablicy złożonej z liczb całkowitych.

2.2.11. Wykorzystaj opisane na początku tego rozdziału notacje Σ oraz Π do wyrażenia:

- (a) sumy liczb nieparzystych od 1 do 377;
- (b) sumy kwadratów liczb parzystych od 2 do n (załóż, że n jest liczbą parzystą);
- (c) iloczynu potęg liczby 2 od 8 do 2^k .

2.2.12. Wykaż, że dla $small = i$, wiersze od (6) do (8) na listingu 2.1 (operacja zamiany wartości) nie mają wpływu na układ w tablicy A.

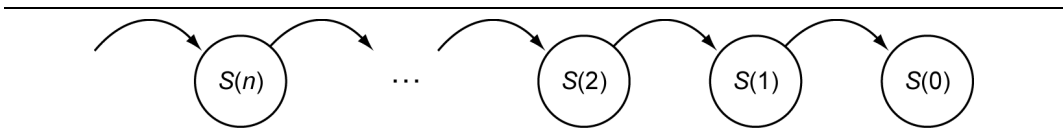
2.3. Dowody indukcyjne

Indukcja matematyczna (ang. *mathematical induction*) jest przydatną techniką dowodzenia, że twierdzenie $S(n)$ jest prawdziwe dla wszystkich nieujemnych liczb całkowitych n lub, uogólniając, dla wszystkich liczb całkowitych większych lub równych danemu ograniczeniu dolnemu. Przykładowo, we wprowadzeniu do niniejszego rozdziału zasugerowano, że w przypadku twierdzenia $\sum_{i=1}^n i = n(n+1)/2$ poprzez indukcję względem n można dowieść, że jest ono prawdziwe dla wszystkich $n \geq 1$.

Niech $S(n)$ będzie dowolnym twierdzeniem dotyczącym liczby całkowitej n . W najprostszej formie dowodu indukcyjnego twierdzenia $S(n)$ dowodzi się dwóch faktów:

- (1) *Przypadku podstawowego* (ang. *basis case*), za który często przyjmuje się twierdzenie $S(0)$. Przypadkiem podstawowym może jednak być równie dobrze $S(k)$ dla dowolnej liczby całkowitej k . Dowodzi się wówczas prawdziwości twierdzenia $S(n)$ dla $n \geq k$.
- (2) *Kroku indukcyjnego* (ang. *inductive step*), gdzie dowodzi się, że dla wszystkich $n \geq 0$ (lub dla wszystkich $n \geq k$, jeśli przypadkiem podstawowym jest $S(k)$), $S(n)$ implikuje $S(n+1)$. W tej części dowodu zakłada się, że twierdzenie $S(n)$ jest prawdziwe. $S(n)$ nazywa się często *hipotezą indukcyjną* (ang. *inductive hypothesis*) — zakładając, że jest prawdziwa, należy udowodnić, że twierdzenie $S(n+1)$ jest prawdziwe.

Rysunek 2.2 ilustruje indukcję rozpoczynaną od wartości 0. Dla każdej liczby całkowitej n , należy udowodnić twierdzenie $S(n)$. Dowód dla twierdzenia $S(1)$ wykorzystuje twierdzenie $S(0)$, dowód dla twierdzenia $S(2)$ wykorzystuje twierdzenie $S(1)$ itd. (kierunek dowodzenia oznaczono strzałkami). Sposób uzależnienia kolejnych twierdzeń od ich poprzedników jest identyczny dla wszystkich n . Oznacza to, że za pomocą jednego dowodu kroku indukcyjnego dowodzi się poprawności wszystkich kroków oznaczonych strzałkami na rysunku 2.2.



RYSUNEK 2.2. W dowodzie indukcyjnym każdy przypadek twierdzenia $S(n)$ jest dowodzony za pomocą twierdzenia dotyczącego kolejnej, mniejszej wartości n

Nazewnictwo parametru indukcji

Niekiedy przydatne jest wyjaśnienie indukcji za pomocą intuicyjnego opisu znaczenia zmiennej n wykorzystywanej w dowodzonym twierdzeniu $S(n)$. Jeśli n nie ma szczególnego znaczenia (patrz przykład 2.4), mówimy po prostu, że „dowodzimy prawdziwości twierdzenia $S(n)$ za pomocą indukcji względem n ”. W przeciwnym razie, jeśli n ma jasne znaczenie (patrz przykład 2.6, w którym n jest liczbą bitów wykorzystywanych w słowach kodu), możemy powiedzieć, że „wykazujemy prawdziwość twierdzenia $S(n)$ względem liczby bitów stosowanych w słowach kodu”.

Przykład 2.4. Aby przybliżyć Czytelnikowi indukcję matematyczną, poniżej zostanie udowodnione następujące

TWIERDZENIE $S(n)$: $\sum_{i=0}^n 2^i = 2^{n+1} - 1$ dla dowolnego $n \geq 0$.

Twierdzenie to określa, że suma potęg liczby 2, od potęgi zerowej do n -tej, jest o 1 mniejsza od $(n+1)$. potęgi liczby 2³. Przykładowo, $1+2+4+8 = 16-1$. Dowód przedstawiono poniżej.

PODSTAWA. Aby udowodnić podstawę, w twierdzeniu $S(n)$ za n należy podstawić 0. Otrzymujemy:

$$\sum_{i=0}^0 2^i = 2^1 - 1 \quad (2.2)$$

Dla $i = 0$, w powyższej sumie istnieje tylko jeden składnik po lewej stronie równania (2.2); lewa strona tego równania jest więc sumą jednego wyrazu 2^0 , czyli 1. Prawa strona równania (2.2), postaci 2^1-1 , czyli $2-1$, również jest równa 1. Stanowi to dowód podstawy $S(n)$, gdyż wykazano prawdziwość tego równania dla $n = 0$.

INDUKCJA. Następnie należy udowodnić krok indukcyjny. Zakładamy, że $S(n)$ jest prawdziwe i dowodzimy prawdziwości tego samego równania dla n zastąpionego przez $n+1$. Należy więc dowieść prawdziwości równania $S(n+1)$:

$$\sum_{i=0}^{n+1} 2^i = 2^{n+2} - 1 \quad (2.3)$$

Aby wykazać, że równanie (2.3) jest prawdziwe, najpierw rozważamy sumę po jego lewej stronie:

$$\sum_{i=0}^{n+1} 2^i$$

³ Prawdziwość twierdzenia $S(n)$ można wykazać bez stosowania indukcji — za pomocą wzoru na sumę wyrazów ciągu geometrycznego. Jednak ten prosty przykład posłuży do zaprezentowania techniki dowodzenia za pomocą indukcji matematycznej. Co więcej, dowody wzorów na sumy wyrazów ciągu geometrycznego i arytmetycznego, z którymi Czytelnik spotkał się zapewne w szkole średniej, są raczej nieformalne, i mówiąc ściśle, właśnie indukcję matematyczną powinno się wykorzystać do dowiedzenia ich prawdziwości.

Powyższa suma jest niemal taka sama, jak poniższa suma po lewej stronie równania $S(n)$:

$$\sum_{i=0}^n 2^i$$

Wyjątkiem jest dodatkowy składnik sumy w równaniu (2.3) dla $i = n+1$, czyli 2^{n+1} .

Ponieważ można założyć, że w dowodzie równania (2.3) hipoteza indukcyjna $S(n)$ jest prawdziwa, należy znaleźć sposób na wykorzystanie $S(n)$ w dalszych rozważaniach. Można to osiągnąć, rozbijając sumę z równania (2.3) na dwie części, z których jedną jest suma znana z równania dla $S(n)$. Oznacza to oddzielenie ostatniego składnika, dla $i = n+1$, i zapisanie:

$$\sum_{i=0}^{n+1} 2^i = \sum_{i=0}^n 2^i + 2^{n+1} \quad (2.4)$$

Teraz można wykorzystać równania $S(n)$, zastępując po prawej stronie równania (2.4) sumę $\sum_{i=0}^n 2^i$ wyrażeniem $2^{n+1}-1$:

$$\sum_{i=0}^{n+1} 2^i = 2^{n+1} - 1 + 2^{n+1} \quad (2.5)$$

Po uproszczeniu prawej strony równania (2.5) otrzymujemy $2 \times 2^{n+1} - 1$, czyli $2^{n+2} - 1$. Widać teraz, że suma po lewej stronie równania (2.5) jest identyczna z sumą po lewej stronie równania (2.3), zaś prawa strona równania (2.5) jest równa prawej stronie równania (2.3). Dowodzi to zatem poprawności równania (2.3) przy wykorzystaniu równania $S(n)$; dowód ten jest właśnie krokiem indukcyjnym. Wniosek jest taki, że twierdzenie $S(n)$ jest prawdziwe dla wszystkich nieujemnych wartości n .

Uzasadnienie poprawności dowodzenia przez indukcję

W dowodzie indukcyjnym najpierw należy dowieść, że twierdzenie $S(0)$ jest prawdziwe. Następnie należy wykazać, że jeśli twierdzenie $S(n)$ jest prawdziwe, to prawdziwe jest także twierdzenie $S(n+1)$. Pojawia się jednak pytanie, dlaczego można przyjąć, że $S(n)$ jest prawdziwe dla wszystkich $n \geq 0$? Poniżej zostaną pokazane dwa „dowody”. Matematyk powiedziałby, że oba prezentowane „dowody” prawidłowości dowodzenia indukcyjnego same wymagają dowodów indukcyjnych, nie są więc żadnymi dowodami. Indukcję należy zaakceptować jako aksjomat. Mimo to, wielu Czytelników powinno uznać poniższe rozważania za pomocne.

Poniżej zakłada się, że wartością podstawową jest $n = 0$. Oznacza to, że wiadomo, iż $S(0)$ jest prawdziwe, oraz że dla wszystkich n większych od 0, jeśli $S(n)$ jest prawdziwe, to prawdziwe jest także $S(n+1)$. Tę samą argumentację można wykorzystać dla wartości podstawowej będącej dowolną inną liczbą całkowitą.

„Dowód” pierwszy: *iteracja kroku indukcyjnego*. Przypuśćmy, że chcemy wykazać, że twierdzenie $S(a)$ jest prawdziwe dla konkretnej nieujemnej liczby całkowitej a . Jeśli $a = 0$, należy po prostu odwołać się do prawdziwości podstawy, $S(0)$. Jeśli $a > 0$, argumentacja przedstawia się następująco.

Zastępowanie zmiennych

Dla wielu osób zastępowanie zmiennych, tak jak n w równaniu $S(n)$, wyrażeniem zawierającym tę samą zmienną stanowi pewien problem. Przykładowo, powyżej za zmienną n podstawiono zmienną $n+1$ w twierdzeniu $S(n)$ z równania (2.3). Aby tego dokonać, należało najpierw oznaczyć wszystkie wystąpienia zmiennej n w S . Metodą ułatwiającą tego typu operację jest zastąpienie wszystkich wystąpień zmiennej n jakąś nową zmienną, np. m , dzięki czemu można całkowicie pozbyć się zmiennej n w twierdzeniu S . Przykładowo, równanie dla $S(n)$ wyglądałoby następująco:

$$\sum_{i=0}^m 2^i = 2^{m+1} - 1$$

Następnie należy zastąpić każde wystąpienie zmiennej m wyrażeniem $n+1$. W efekcie otrzymujemy równanie:

$$\sum_{i=0}^{n+1} 2^i = 2^{(n+1)+1} - 1$$

Po uproszczeniu $(n+1)+1$ do $n+2$ otrzymamy równanie (2.3).

Warto zauważyć, że należy umieszczać wstawiane wyrażenie w nawiasach w celu uniknięcia przypadkowej zmiany kolejności działań. Przykładowo, gdyby zamieniono m na $n+1$ w wyrażeniu $2 \times m$, nie umieszczając wstawianego wyrażenia $n+1$ w nawiasach, otrzymano by $2 \times n+1$, zamiast prawidłowego wyrażenia $2 \times (n+1)$, czyli $2 \times n+2$.

Z podstawy wiadomo, że $S(0)$ jest prawdziwe. Stwierdzenie „ $S(n)$ implikuje $S(n+1)$ ” dla wartości 0 podstawionej w miejsce n przyjmuje formę „ $S(0)$ implikuje $S(1)$ ”. Ponieważ wiadomo, że $S(0)$ jest prawdziwe, wiadomo także, że $S(1)$ jest prawdziwe. Podobnie, jeśli zamieni się n na 1, otrzymując zdanie „ $S(1)$ implikuje $S(2)$ ”, wiadomo, że $S(2)$ również jest prawdziwe. Zamiana n na 2 daje zdanie „ $S(2)$ implikuje $S(3)$ ”, zatem $S(3)$ jest prawdziwe, itd. Nie ma znaczenia, jaka jest wartość a , w końcu dochodzi się do twierdzenia $S(a)$, co kończy dowód.

„Dowód” drugi: *najmniejszy kontrprzykład*. Przypuśćmy, że twierdzenie $S(n)$ nie było prawdziwe dla przynajmniej jednej wartości n . Niech a będzie najmniejszą nieujemną liczbą całkowitą, dla której $S(a)$ jest fałszywe. Jeśli $a = 0$, stanowi to zaprzeczenie podstawy, $S(0)$, i wiadomo, że a musi być większe od 0. Jeśli jednak $a > 0$ i a jest najmniejszą nieujemną liczbą całkowitą, dla której $S(a)$ jest fałszywe, to $S(a-1)$ musi być prawdziwe. Krok indukcyjny, dla n zastąpionego przez $a-1$, mówi teraz, że $S(a-1)$ implikuje $S(a)$. Ponieważ $S(a-1)$ jest prawdziwe, $S(a)$ także musi być prawdziwe, co przynosi kolejną sprzeczność. Ponieważ założono, że istnieją nieujemne wartości n , dla których $S(n)$ jest fałszywe i wywiedziono z tego sprzeczność, $S(n)$ musi być prawdziwe dla dowolnego $n \geq 0$.

Kody detekcji błędów

Poniżej zostanie przedstawiona analiza rozszerzonego przykładu *kodów detekcji błędów*, zagadnienia interesującego samego w sobie i prowadzącego dodatkowo do ciekawego dowodu indukcyjnego. Podczas transmisji danych siecią zwykle koduje się znaki (litery, cyfry, znaki interpunkcyjne itd.) do postaci ciągów bitów, czyli zer i jedynek. Poniżej zostało przyjęte założenie, że znaki są reprezentowane za pomocą siedmiu bitów. Powszechnie stosowaną normą jest przesyłanie większej liczby bitów na znak, ósmy bit może bowiem ułatwić np. wykrywanie prostych błędów w transmisji danych (może się zdarzyć, że jedno z przesyłanych zer lub jedynek ulegnie zmianie na skutek zakłóceń transmisji, co powoduje, że do odbiorcy dotrze przeciwny bit — 0 zamiast 1 lub 1 zamiast 0). Mechanizm używania dodatkowego bitu jest więc bardzo przydatny, ponieważ system komunikacji może przesłać informację o niepożądanym zmianie jednego z ośmiu przesyłanych bitów, co umożliwia zainicjowanie procesu ponownej transmisji.

Aby móc wykrywać zmiany za pomocą jednego bitu, należy mieć pewność, że żadna para znaków nie jest reprezentowana przez sekwencje bitów różniące się od siebie tylko na jednej pozycji. W takim przypadku, gdyby ta pozycja została zmieniona, otrzymany zostałby kod zupełnie innego znaku i nie istniałaby możliwość wykrycia wystąpienia błędu. Przykładowo, jeśli kod pewnego znaku jest sekwencją bitów 01010101, zaś kod innego znaku jest sekwencją 010000101, to zmiana bitu na czwartej pozycji od lewej przekształca pierwszy znak w drugi.

Sposobem zapewnienia, że żadna para znaków nie jest reprezentowana przez kody różniące się tylko na jednej pozycji jest poprzedzenie konwencjonalnego kodu złożonego z siedmiu bitów dodatkowym *bitem parzystości* (ang. *parity bit*). Jeśli łączna liczba jedynek w danej grupie bitów jest nieparzysta, mówimy, że grupa ta jest *nieparzysta* (ang. *odd*). Jeśli łączna liczba jedynek w danej grupie bitów jest parzysta, o grupie tej mówimy, że jest *parzysta* (ang. *even*). Wybrany poniżej schemat kodowania jest reprezentowanie każdego znaku za pomocą kodu ośmiobitowego z parzystością; równie dobrze, można by jednak zdecydować o wykorzystaniu kodu z nieparzystością. Wymuszenie parzystości opiera się na odpowiednim wyborze bitu parzystości w stosowanej reprezentacji.

- **Przykład 2.5.** Konwencjonalny siedmiobitowy kod ASCII (czytany „aski”; pochodzi od skrótu amerykańskiego znormalizowanego kodu wymiany informacji — ang. *American Standard Code for Information Interchange*) dla znaku A to 1000001. Ta sekwencja bitów ma parzystą liczbę jedynek, zatem po dodaniu do niej prefiksu 0 otrzymuje się 01000001. Konwencjonalnym kodem ASCII dla C jest 1000011, który różni się od siedmiobitowego kodu litery A tylko na szóstej pozycji. Kod ten jest jednak nieparzysty i po dodaniu prefiksu 1 otrzymuje się ośmiobitowy kod z parzystością 11000011. Należy zauważyć, że po dodaniu prefiksów będących bitami parzystości dla kodów A i C, otrzymano sekwencje 01000001 oraz 11000011, które różnią się teraz na dwóch pozycjach (pierwszej i siódmej), co widać na rysunku 2.3. •

A: 0 1 0 0 0 0 0 1

C: 1 1 0 0 0 0 1 1

RYSUNEK 2.3.

Początkowy bit parzystości sprawia, że wszystkie ośmiobitowe kody znaków będą parzyste

Zawsze można wybrać taką wartość bitu parzystości dodawanego do kodu siedmiobitowego, by liczba jedynek w kodzie ośmiobitowym była parzysta. Bit 0 jest wybierany, jeśli kod siedmiobitowy znaku już jest parzysty; dla nieparzystego kodu siedmiobitowego, wybierany jest bit 1. W obu przypadkach, liczba jedynek w kodzie ośmiobitowym jest parzysta.

Żadna para parzystych sekwencji bitów nie może różnić się tylko na jednej pozycji. Jeśli dwie sekwencje różnią się na dokładnie jednej pozycji, oznacza to, że jedna z nich ma dokładnie jedną jedynkę więcej niż druga. Jedna sekwencja musi więc być parzysta a druga nieparzysta, co jest sprzeczne z przyjętym założeniem o parzystości obu ciągów. Dopiero dodanie bitu parzystości, który powoduje parzystość liczby jedynek, sprawia, że otrzymane kody znaków umożliwiają detekcję błędów.

Schemat dodawania bitów parzystości jest mechanizmem dosyć „efektywnym” w tym sensie, że umożliwia transmisję wielu różnych znaków. Warto zauważyć, że istnieje 2^n różnych sekwencji złożonych z n bitów, ponieważ można wybrać jedną z dwóch wartości (0 lub 1) na pierwszej pozycji, jedną z dwóch wartości na pozycji drugiej, itd. Razem istnieje $2 \times 2 \times \dots \times 2$ (n czynników) możliwych ciągów. Można by więc oczekiwać, że za pomocą ośmiu bitów będzie można reprezentować $2^8 = 256$ znaków.

W przypadku zastosowania schematu dodawania bitów parzystości można jednak skorzystać tylko z siedmiu pozostałych bitów — ósmy jest zarezerwowany. W takiej sytuacji można reprezentować co najwyżej $2^7 = 128$ znaków z zachowaniem możliwości detekcji pojedynczych błędów. To nadal niezły wynik, można wykorzystać 128 z możliwych 256 prawidłowych ośmiobitowych kodów znakowych i jednocześnie wykrywać błędy polegające na niepożądanym zmianie pojedynczych bitów.

Podobnie, jeśli zostaną wykorzystane sekwencje złożone z n bitów, wybór jednego z nich na bit parzystości spowoduje, że będzie można reprezentować 2^{n-1} znaków za pomocą sekwencji złożonych z $n-1$ bitów poprzedzonych stosownym bitem parzystości, którego wartość określa się za pomocą pozostałych $n-1$ bitów. Ponieważ istnieje 2^n różnych sekwencji złożonych z n bitów, można reprezentować $2^{n-1}/2^n$, czyli połowę z możliwej liczby znaków, i nadal wykrywać błędy związane ze zmianą dowolnych pojedynczych bitów w danych sekwencjach.

Pojawia się pytanie, czy jest możliwe wykrywanie tego typu błędów i jednocześnie wykorzystanie więcej niż połowy możliwych sekwencji bitów do kodowania znaków? Poniższy przykład pokaże, że nie jest to możliwe. W zastosowanym dowodzie indukcyjnym wykorzystano twierdzenie, które nie jest prawdziwe dla zera, i dla którego należy wybrać większą podstawę, a dokładniej 1.

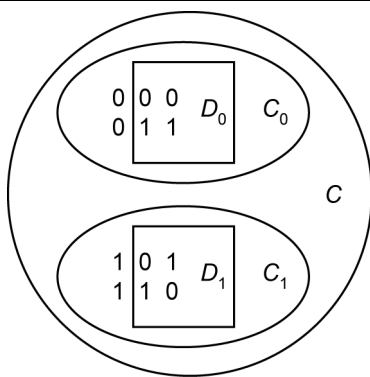
- **Przykład 2.6.** Poniżej zostanie udowodnione następujące twierdzenie za pomocą indukcji względem n .

TWIERDZENIE $S(n)$: jeśli C jest dowolnym zbiorem ciągów bitów o długości n dających możliwość detekcji błędów (tzn. nie istnieje para ciągów różniących się tylko na dokładnie jednej pozycji), to C zawiera co najwyżej 2^{n-1} ciągów.

Powyższa teza nie jest prawdziwa dla $n = 0$. Twierdzenie $S(0)$ oznaczałoby, że dowolny zbiór ciągów umożliwiających detekcję błędów o długości 0 zawiera co najwyżej 2^{-1} ciągów, czyli pół ciągu. Mówiąc ściśle, zbiór C składający się wyłącznie z ciągów pustych (ciągów bez żadnych pozycji) jest zbiorem umożliwiającym detekcję błędów o liczności 0, ponieważ nie mogą w tym w zbiorze istnieć dwa ciągi różniące się tylko na jednej pozycji. Taki zbiór C zawierałby więcej niż pół ciągu — dokładnie jeden ciąg. Twierdzenie $S(0)$ jest więc fałszywe, jednak dla wszystkich wartości $n \geq 1$, $S(n)$ jest prawdziwe, co zostanie wykazane poniżej.

PODSTAWA. Podstawą jest $S(1)$, co oznacza, że dowolny zbiór ciągów o długości 1 umożliwiających detekcję błędów zawiera co najwyżej $2^{1-1} = 2^0 = 1$ ciąg. Istnieją tylko dwa ciągi jednobitowe: 0 i 1. Nie można jednak umieścić ich obu w zbiorze ciągów umożliwiającym detekcję błędów, ponieważ różnią się one na dokładnie jednej pozycji. Zatem każdy taki zbiór dla $n = 1$ musi zawierać co najwyżej jeden ciąg.

INDUKCJA. Niech $n \geq 1$ oraz założmy, że hipoteza indukcyjna — zbiór ciągów o długości n umożliwiających detekcję błędów może zawierać co najwyżej 2^{n-1} ciągów — jest prawdziwa. Należy wykazać, przy tym założeniu, że dowolny zbiór ciągów o długości $n+1$ umożliwiających detekcję błędów może zawierać co najwyżej 2^n ciągów. Zbiór C można podzielić na dwa podzbiory: C_0 będący zbiorem ciągów ze zbioru C rozpoczynających się od 0 oraz C_1 będący zbiorem ciągów ze zbioru C rozpoczynających się od 1. Można na przykład założyć, że $n = 2$, oraz że C zawiera ciągi bitów o długości $n+1 = 3$ (skonstruowanych z uwzględnieniem bitu parzystości). Jak pokazano na rysunku 2.4, C składa się z ciągów 000, 101, 110 oraz 011; C_0 składa się z ciągów 000 i 011, C_1 natomiast zawiera dwa pozostałe ciągi 101 oraz 110.



RYSUNEK 2.4.

Zbiór C zostaje rozbitý na podzbiór C_0 (zbiór ciągów rozpoczynających się od zera) oraz podzbiór C_1 (zbiór ciągów rozpoczynających się od jedynki). Podzbiory D_0 i D_1 są tworzone przez usunięcie odpowiednio pierwszych zer i pierwszych jedynek

Poniżej rozważono zbiór D_0 składający się z ciągów ze zbioru C_0 po usunięciu z nich pierwszych zer. W powyższym przykładzie, D_0 zawiera ciągi 00 i 11. Można stwierdzić, że D_0 nie może zawierać dwóch ciągów różniących się tylko jednym bitem. Jeśli bowiem istniałyby dwa takie ciągi (np. $a_1a_2\dots a_n$ oraz $b_1b_2\dots b_n$), to przywrócenie początkowych zer dałoby takie dwa ciągi w zbiorze C_0 : $0a_1a_2\dots a_n$ oraz $0b_1b_2\dots b_n$, które różniłyby się wyłącznie na jednej pozycji. Ciągi ze zbioru C_0 należą także do zbioru C , o którym wiadomo, że nie zawiera dwóch ciągów różniących się tylko na jednej pozycji. Nie może ich więc zawierać także zbiór D_0 , który jest w tej sytuacji zbiorem ciągów umożliwiających detekcję błędów.

Można teraz zastosować hipotezę indukcyjną do wywnioskowania, że D_0 (zbiór ciągów o długości n umożliwiających detekcję błędów) zawiera co najwyżej 2^{n-1} ciągów. C_0 także zawiera więc co najwyżej 2^{n-1} ciągów.

Podobne wnioskowanie można zastosować w odniesieniu do zbioru C_1 . Niech D_1 będzie zbiorem ciągów ze zbioru C_1 pozbawionych początkowych jedynek. D_1 jest zbiorem n -bitowych ciągów umożliwiających detekcję błędów i — zgodnie z hipotezą indukcyjną — D_1 zawiera co najwyżej 2^{n-1} ciągów. Także C_1 zawiera zatem co najwyżej 2^{n-1} ciągów. Każdy ciąg ze zbioru C należy albo do zbioru C_0 , albo do C_1 . Oznacza to, że zbiór C zawiera co najwyżej $2^{n-1} + 2^{n-1}$, czyli 2^n ciągów.

Dowodzono, że $S(n)$ implikuje $S(n+1)$, można więc wywnioskować, że $S(n)$ jest prawdziwe dla wszystkich $n \geq 1$. Z wnioskowania należy wyłączyć przypadek dla $n = 0$, ponieważ podstawą jest przypadek $n = 1$, nie $n = 0$. Widać teraz, że zbiory umożliwiające detekcję błędów konstruowane z wykorzystaniem testu parzystości są tak duże, jak to tylko możliwe, ponieważ zawierają dokładnie 2^{n-1} n -bitowych ciągów. •

Sposoby tworzenia dowodów indukcyjnych

Nie ma jednego sposobu gwarantującego otrzymanie dowodu indukcyjnego dla dowolnego (prawdziwego) twierdzenia $S(n)$. Znajdowanie dowodów indukcyjnych, podobnie jak odkrywanie każdego innych dowodów, czy też pisanie działających programów, jest zadaniem wymagającym inteligencji. Można więc udzielić jedynie kilku wskazówek. Sprawdzając kroki indukcyjne w przykładach 2.4 i 2.6 można zauważyć, że w obu przypadkach należało przekształcić twierdzenie $S(n+1)$, którego prawdziwości próbowano dowieść, tak by wiązało się z hipotezą indukcyjną, $S(n)$, i zawierało pewne dodatkowe elementy. W przykładzie 2.4 sumę:

$$1+2+4+\dots+2^n+2^{n+1}$$

wyrażono w postaci:

$$1+2+4+\dots+2^n$$

czyli takiej, o której było coś wiadomo z hipotezy indukcyjnej, rozszerzonej o dodatkowy składnik, 2^{n+1} .

W przykładzie 2.6 wyrażono zbiór C (zawierający ciągi o długości $n+1$) za pomocą dwóch zbiorów (nazwanych D_0 i D_1) ciągów o długości n , co pozwoliło na zastosowanie hipotezy indukcyjnej względem tych zbiorów i dalsze wnioskowanie na temat ograniczeń związanych z ich rozmiarami.

Przekształcenie twierdzeniem $S(n+1)$, umożliwiające zastosowanie hipotezy indukcyjnej jest oczywiście jedynie szczególnym przypadkiem bardziej uniwersalnego schematu „wykorzystania tego, co dane”. Największe trudności pojawiają się zawsze, gdy należy wykorzystać część „dodatkową” twierdzenia $S(n+1)$ i dokończyć dowód jego prawdziwości na podstawie $S(n)$. Oto uniwersalna reguła odnosząca się do tego typu dowodów:

- Dowód indukcyjny musi w pewnym momencie zawierać stwierdzenie: „...i na podstawie hipotezy indukcyjnej wiadomo, że...”. Jeśli tego zdania zabraknie, nie będzie to prawdziwy dowód indukcyjny.

Ćwiczenia

2.3.1. Wykaż prawdziwość poniższych równań za pomocą indukcji względem n , rozpoczynając od $n = 1$.

- (a) $\sum_{i=1}^n i = n(n+1)/2$;
- (b) $\sum_{i=1}^n i^2 = n(n+1)(2n+1)/6$;
- (c) $\sum_{i=1}^n i^3 = n^2(n+1)^2/4$;
- (d) $\sum_{i=1}^n 1/i(i+1) = n/(n+1)$.

2.3.2. Liczby w postaci $t_n = n(n+1)/2$ nazywamy *liczbami trójkątnymi*, ponieważ elementy (np. kule), których liczba jest jedną z liczb trójkątnych, można tak rozłożyć, by tworzyły trójkąt równoboczny

o długości boku równej n . Łączna liczba tych elementów jest równa $\sum_{i=1}^n i$ (patrz ćwiczenie 2.3.1(a)), czyli t_n . Przykładowo, kręgle są rozstawione w trójkąt po 4 na każdym boku, mamy więc $t_4 = 4 \times 5 / 2 = 10$ kręgli. Wykaż za pomocą indukcji dla n , że $\sum_{j=1}^n t_j = n(n+1)(n+2)/6$.

2.3.3. Zidentyfikuj parzystość poniższych sekwencji bitów (określ, czy są parzyste, czy nieparzyste):

- (a) 01101;
- (b) 111000111;
- (c) 010101.

2.3.4. Załóżmy, że wykorzystujemy do kodowania symboli trzy cyfry — niech będą to 0, 1 oraz 2. Zbiór ciągów C utworzonych z zer, jedynek i dwójek umożliwia detekcję błędów, jeśli żadna para ciągów w tym zbiorze nie różni się tylko na jednej pozycji. Przykładowo, $\{00, 11, 22\}$ jest zbiorem umożliwiającym detekcję błędów dla ciągów o długości 2 i wykorzystujących cyfry 0, 1 oraz 2. Wykaż, że dla dowolnego $n \geq 1$, zbiór ciągów o długości n umożliwiających detekcję błędów i złożonych z cyfr 0, 1 oraz 2 nie może zawierać więcej niż 3^{n-1} ciągów.

2.3.5*. Wykaż, że dla dowolnego $n \geq 1$ istnieje zbiór ciągów o długości n umożliwiających detekcję błędów i złożonych z cyfr 0, 1 oraz 2, który zawiera 3^{n-1} ciągów.

2.3.6*. Wykaż, że jeśli wykorzystuje się k symboli, dla dowolnego $k \geq 2$, istnieje zbiór ciągów o długości n umożliwiających detekcję błędów i złożonych z k różnych symboli (cyfr), który zawiera k^{n-1} ciągów, ale nie istnieje taki zbiór zawierający więcej niż k^{n-1} ciągów.

2.3.7*. Jeśli $n \geq 1$, liczba ciągów wykorzystujących cyfry 0, 1 oraz 2, niezawierających tej samej cyfry na dwóch kolejnych pozycjach, wynosi $3 \times 2^{n-1}$. Przykładowo, istnieje 12 takich ciągów o długości 3: 010, 012, 020, 021, 101, 102, 120, 121, 201, 202, 210 oraz 212. Udowodnij to twierdzenie za pomocą indukcji względem długości ciągów. Czy przedstawiony wzór jest prawdziwy dla $n = 0$?

2.3.8*. Udowodnij, że omówiony w podrozdziale 1.3 algorytm dodawania kaskadowego zwraca poprawne wyniki. *Wskazówka:* wykaż za pomocą indukcji względem i , że po rozważeniu pierwszych i pozycji od prawego końca, suma reszt o długości i dla dwóch składników sumy jest równa liczbie, której reprezentacją binarną jest bit przeniesienia oraz występujące po nim i bitów wygenerowanego do tej pory wyniku.

2.3.9*. Wzór na sumę n elementów ciągu geometrycznego $a, ar, ar^2, \dots, ar^{n-1}$ ma postać:

$$\sum_{i=0}^{n-1} ar^i = \frac{(ar^n - a)}{(r - 1)}$$

Udowodnij poprawność tego wzoru za pomocą indukcji względem n . Zauważ, że konieczne jest założenie, że $r \neq 1$. W którym miejscu należy wykorzystać to założenie w tworzonym dowodzie?

2.3.10. Wzór na sumę n wyrazów ciągu arytmetycznego o pierwszym elemencie a i różnicy b , czyli $a, (a+b), (a+2b), \dots, (a+(n-1)b)$, ma postać:

$$\sum_{i=0}^{n-1} a + bi = n(2a + (n-1)b) / 2$$

Sumy arytmetyczne i geometryczne

Istnieją dwa wzory znane z algebry przerabianej w szkole średniej, z których będą często wykorzystywane. Z każdym z nich wiąże się ciekawy dowód indukcyjny, o którego przeprowadzenie Czytelnik jest proszony w ćwiczeniach 2.3.9 i 2.3.10.

Ciąg arytmetyczny (ang. *arithmetic series*) jest sekwencją n liczb postaci:

$$a, (a+b), (a+2b), \dots, (a+(n-1)b)$$

Pierwszym wyrazem ciągu jest a i każdy kolejny wyraz jest większy o b od poprzedniego. Suma n wyrazów jest równa średniej liczonej dla pierwszego i ostatniego elementu pomnożonej przez n , czyli:

$$\sum_{i=0}^{n-1} a + bi = n(2a + (n-1)b) / 2$$

Przykładowo, można rozważyć sumę $3+5+7+9+11$. Zawiera ona $n = 5$ wyrazów, pierwszy z nich to 3, a ostatni to 11. Ich suma wynosi więc $5 \times (3+11)/2 = 5 \times 7 = 35$. Poprawność otrzymanego wyniku można sprawdzić, dodając do siebie pięć wypisanych powyżej wyrazów.

Ciąg geometryczny (ang. *geometric series*) jest sekwencją n liczb postaci:

$$a, ar, ar^2, ar^3, \dots, ar^{n-1}$$

co oznacza, że pierwszym wyrazem jest a , zaś każdy kolejny wyraz jest równy iloczynowi wyrazu poprzedniego i liczby r . Wzór na sumę n wyrazów ciągu geometrycznego wygląda następująco:

$$\sum_{i=0}^{n-1} ar^i = \frac{(ar^n - a)}{(r-1)}$$

Jak widać, r może być większe lub mniejsze od 1. Jeśli $r = 1$, powyższy wzór nie jest prawdziwy, ale wszystkie wyrazy są równe a , zatem oczywiste jest, że ich suma wynosi an .

Jako przykład sumy ciągu geometrycznego można rozważyć $1+2+4+8+16$. W tym przypadku $n = 5$, pierwszym wyrazem a jest 1, natomiast iloraz ciągu r wynosi 2. Oto suma:

$$(1 \times 2^5 - 1) / (2 - 1) = (32 - 1) / 1 = 31$$

Otrzymaną wartość można oczywiście łatwo sprawdzić. Jako inny przykład warto rozważyć $1+1/2+1/4+1/8+1/16$. Ponownie $n = 5$ i $a = 1$, ale tym razem $r = 1/2$. Suma wynosi więc:

$$(1 \cdot (\frac{1}{2})^5 - 1) / (\frac{1}{2} - 1) = (-31/32) / (-1/2) = 1 \frac{15}{16}$$

- udowodnij prawdziwość tego wzoru za pomocą indukcji względem n ;
- wykaż, że ćwiczenie 2.3.1(a) jest przykładem zastosowania tego wzoru.

Szablon dla prostych indukcji

Podsumowaniem podrozdziału 2.3 może być opis szablonu, do którego pasują proste dowody indukcyjne omówione w niniejszym podrozdziale. W podrozdziale 2.4 zostanie przedstawiony bardziej ogólny szablon.

- (1) Określ twierdzenie $S(n)$ do udowodnienia. Przykładowo chcemy dowieść prawdziwości $S(n)$, stosując indukcję względem n , dla wszystkich $n \geq i_0$. W takim przypadku i_0 jest stałą dla podstawy, zazwyczaj równą 0 lub 1, może to jednak być dowolna liczba całkowita. Wyjaśnij własnymi słowami, czym jest n (np. długością słów kodu).
- (2) Określ przypadek podstawowy, $S(i_0)$.
- (3) Udowodnij przypadek podstawowy, tzn. wyjaśnij, dlaczego $S(i_0)$ jest prawdziwe.
- (4) Określ krok indukcyjny, stwierdzając, że zakładasz prawdziwość $S(n)$ dla pewnego $n \geq i_0$, co jest „hipotezą indukcyjną”. Wyraż $S(n+1)$, zastępując wystąpienia n we wzorze $S(n)$ wyrażeniem $n+1$.
- (5) Udowodnij prawdziwość $S(n+1)$, zakładając, że hipoteza indukcyjna $S(n)$ jest prawdziwa.
- (6) Zakończ dowód wnioskiem, że $S(n)$ jest prawdziwe dla wszystkich $n \geq i_0$ (ale niekoniecznie dla mniejszych n).

2.3.11. Podaj dwa nieformalne dowody poprawności indukcji rozpoczętej od liczby 1, mimo fałszywości twierdzenia $S(0)$.

2.3.12. Dowiedz za pomocą indukcji względem długości ciągów, że kod składający się z ciągów nieparzystych umożliwia detekcję błędów.

2.3.13.** Jeśli żadna para ciągów w kodzie nie różni się na mniej niż trzech pozycjach, to istnieje możliwość korekcji pojedynczego błędu, poprzez znalezienie unikatowego ciągu w kodzie, który różni się od otrzymanego tylko na jednej pozycji. Okazuje się, że istnieje kod złożony z szesnastu ciągów 7-bitowych, który umożliwia korekcję pojedynczych błędów. Znajdź taki kod. *Wskazówka:* wnioskowanie jest prawdopodobnie najlepszym sposobem na rozwiązanie tego zadania. Jeśli jednak utkniesz, spróbuj napisać program wyszukujący taki kod.

2.3.14*. Czy kod uwzględniający parzystość umożliwia detekcję „podwójnych błędów”, czyli zmian wartości dwóch różnych bitów? Czy może poprawiać dowolne pojedyncze błędy?

2.4. Indukcja zupełna

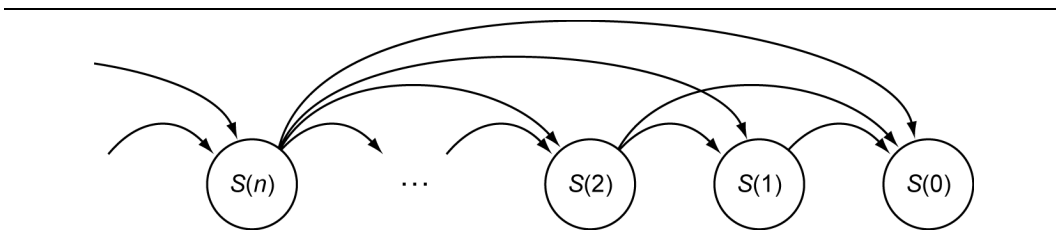
W przeanalizowanych do tej pory przykładach, dowodzono prawdziwości twierdzenia $S(n+1)$ wyłącznie na podstawie twierdzenia $S(n)$, czyli hipotezy indukcyjnej. Ponieważ jednak dowodzi się, że twierdzenie S jest prawdziwe dla wartości swojego parametru, rozpoczynając od wartości podstawowej i przechodząc do kolejnych rosnących liczb całkowitych, można wykorzystać $S(i)$ dla wszystkich wartości i , od podstawy aż do n . Ta forma indukcji nosi nazwę *zupełnej* (ang. *complete*; czasem zwanej także *całkowitą* lub *silną*). Prosta forma indukcji, stosowana w podrozdziale 2.3, w której wykorzystywano wyłącznie $S(n)$ do wykazania prawdziwości $S(n+1)$ jest czasami nazywana indukcją *słabą* (ang. *weak*).

Zacznijmy od rozważenia metody realizacji indukcji zupełnej, rozpoczynając od podstawy $n = 0$. Należy dowieść, że $S(n)$ jest prawdziwe dla wszystkich $n \geq 0$ w dwóch krokach:

- (1) Najpierw należy wykazać prawdziwość podstawy, $S(0)$.
- (2) Jako hipotezę indukcyjną przyjmuje się, że wszystkie twierdzenia $S(0), S(1), \dots, S(n)$ są. Na podstawie tych twierdzeń należy udowodnić, że także twierdzenie $S(n+1)$ jest prawdziwe.

Podobnie jak w przypadku indukcji słabej, opisanej w poprzednim podrozdziale, można wybrać jakąś inną niż 0 wartość podstawy a . Po dokonaniu wyboru należy dowieść, że $S(a)$ jest prawdziwe, zaś w kroku indukcyjnym można założyć, że tylko twierdzenia $S(a), S(a+1), \dots, S(n)$ są prawdziwe. Należy zauważyć, że indukcja słaba jest szczególnym przypadkiem indukcji zupełnej, w którym określa się, że nie będzie się stosować żadnego z poprzednich twierdzeń poza $S(n)$ dla udowodnienia $S(n+1)$.

Rysunek 2.5 demonstruje działanie indukcji zupełnej. Każdy przypadek twierdzenia $S(n)$ może (opcjonalnie) wykorzystywać do swojego dowodu dowolny z przypadków niżej indeksowanych (po swojej prawej stronie).



RYSUNEK 2.5. Indukcja zupełna umożliwia wykorzystanie na potrzeby dowodu każdego przypadku jednego, kilku lub wszystkich przypadków wcześniejszych

Indukcje z większą liczbą przypadków podstawowych

Podczas przeprowadzania indukcji zupełnej, niekiedy przydatne jest wykorzystanie więcej niż jednego przypadku podstawowego. Jeśli należy dowieść prawdziwości twierdzenia $S(n)$ dla wszystkich $n \geq i_0$, można jako przypadek podstawowy potraktować nie tylko wspomniane i_0 , ale także pewną liczbę kolejnych liczb całkowitych: $i_0, i_0+1, i_0+2, \dots, j_0$. Należy wówczas wykonać następujące dwa kroki:

- (1) Udowodnić wszystkie przypadki podstawowe, czyli twierdzenia $S(i_0), S(i_0+1), \dots, S(j_0)$.
- (2) Jako hipotezę indukcyjną należy założyć, że wszystkie twierdzenia $S(i_0), S(i_0+1), \dots, S(n)$ są prawdziwe, dla pewnego $n \geq j_0$, i udowodnić twierdzenie $S(n+1)$.

- **Przykład 2.7.** Pierwszy prezentowany, prosty przykład indukcji zupełnej wykorzystuje kilka przypadków podstawowych. Jak się okaże, analizowany przykład jest „zupełny” tylko w pewnym ograniczonym znaczeniu. Do dowiedzenia prawdziwości twierdzenia $S(n+1)$ nie jest wykorzystywane twierdzenie $S(n)$, lecz $S(n-1)$. Bardziej ogólny schemat indukcji zupełnej przewiduje wykorzystanie twierdzeń $S(n), S(n-1)$ i wielu innych przypadków twierdzenia S .

Poniżej zostanie dowiedzione za pomocą indukcji względem n , że następujące twierdzenie jest prawdziwe dla wszystkich $n \geq 0$ ⁴.

TWIERDZENIE $S(n)$: istnieją liczby całkowite a i b (dodatnie, ujemne lub równe 0) takie, że:
 $n = 2a + 3b$.

PODSTAWA. Jako przypadki podstawowe zostanie wykorzystane zarówno 0, jak i 1:

- (a) dla $n = 0$ można wybrać $a = 0$ oraz $b = 0$, co prowadzi do oczywistego rozwiązania: $0 = 2 \times 0 + 3 \times 0$.
- (b) dla $n = 1$ można wybrać $a = -1$ oraz $b = 1$, wówczas: $1 = 2 \times (-1) + 3 \times 1$.

INDUKCJA. Można teraz założyć, że twierdzenie $S(n)$ jest prawdziwe i udowodnić, że prawdziwe jest także twierdzenie $S(n+1)$ dla dowolnego $n \geq 1$. Warto zauważyć, że można założyć, iż n jest co najmniej tak duże, jak największa spośród kolejnych wartości, dla których udowodniono podstawę — w omawianym przypadku $n \geq 1$. Twierdzenie $S(n+1)$ mówi, że $n+1 = 2a+3b$ dla pewnych liczb całkowitych a i b .

Hipoteza indukcyjna określa, że wszystkie twierdzenia $S(0), S(1), \dots, S(n)$ są prawdziwe. Należy zauważyć, że sekwencja rozpoczyna się od 0, ponieważ tyle wynosił najmniejszy rozważany przypadek podstawowy. Ponieważ możemy założyć, że $n \geq 1$ oraz wiadomo, że $n-1 \geq 0$, zatem $S(n-1)$ jest prawdziwe. To twierdzenie mówi, że istnieją takie liczby całkowite a i b , że $n-1 = 2a+3b$.

Ponieważ w twierdzeniu $S(n+1)$ potrzebna jest wartość a , można ponownie zapisać twierdzenie $S(n-1)$ z innymi nazwami liczb całkowitych — niech to będą liczby a' i b' , takie że:

$$n-1 = 2a' + 3b' \quad (2.6)$$

Jeśli do obu stron równania (2.6) dodamy 2, otrzymujemy równanie $n+1 = 2(a'+1) + 3b'$. Jeśli podstawimy teraz a za $a'+1$ oraz b za b' , otrzymamy równanie $n+1 = 2a+3b$ dla pewnych liczb całkowitych a i b . Jest to twierdzenie $S(n+1)$, co kończy dowód indukcyjny. Jak widać, w dowodzie tym nie wykorzystywano twierdzenia $S(n)$, lecz $S(n-1)$. •

Zasadność indukcji zupełnej

Podobnie jak w przypadku omówionej w podrozdziale 2.3 typowej, „słabej” indukcji, zasadność indukcji zupełnej można wykazać intuicyjnie jako technikę dowodzenia opartą na argumentach „najmniejszego kontrprzykładu”. Niech przypadkami podstawowymi będą $S(i_0), S(i_0+1), \dots, S(j_0)$; zakłada się, że wcześniej wykazano już, iż dla każdego $n \geq j_0$, twierdzenia $S(i_0), S(i_0+1), \dots, S(j_0)$ razem implikują $S(n+1)$. Załóżmy teraz, że twierdzenie $S(n)$ nie było prawdziwe dla przynajmniej jednej wartości $n \geq i_0$ oraz niech b będzie najmniejszą liczbą całkowitą większą lub równą i_0 , dla której $S(b)$ jest fałszywe. Liczba b nie może w takim przypadku mieścić się w przedziale od i_0 do j_0 — podstawa byłaby wówczas sprzeczna. Ponadto b nie może być większe od j_0 — gdyby tak było, wszystkie twierdzenia $S(i_0), S(i_0+1), \dots, S(b-1)$ byłyby prawdziwe. Jednak wówczas krok indukcyjny pozwalałby stwierdzić, że $S(b)$ jest prawdziwe, co prowadziłoby do kolejnej sprzeczności.

⁴ Przedstawiony wzór jest w rzeczywistości prawdziwy dla wszystkich n , dodatnich i ujemnych, jednak dowód dla ujemnych n wymaga przeprowadzenia drugiego dowodu indukcyjnego, który będzie zadaniem czytelnika w jednym z ćwiczeń.

Postacie normalne wyrażeń arytmetycznych

Poniżej zostanie omówiony rozbudowany przykład związany z przekształcaniem wyrażeń arytmetycznych do postaci równoważnych. Przykład ten jest ilustracją indukcji zupełnej, w której zostaje w pełni wykorzystany fakt, że można zakładać prawdziwość dowodzonego twierdzenia S dla wszystkich argumentów mniejszych od n .

Kompilator języka programowania może korzystać z algebraicznych własności operatorów arytmetycznych w celu przedstawiania kolejności obliczania operandów wyrażeń arytmetycznych. Celem takiego przedstawiania jest znalezienie sposobu obliczenia wartości danego wyrażenia w czasie krótszym niż wymagany do obliczenia pierwotnej, oczywistej wersji.

W niniejszym podrozdziale zostaną omówione wyrażenia arytmetyczne zawierające pojedyncze łączne i przemienne operatory, np. (+), oraz zostaną przeanalizowane istniejące możliwości przekształceń operandów. Zostanie dowiedzione, że w przypadku dowolnego wyrażenia zawierającego wyłącznie operatory (+), jego wartość jest równa wartości dowolnego innego wyrażenia zawierającego operatory (+) zastosowane względem tych samych operandów, ułożonych i (lub) pogrupowanych w dowolny inny sposób. Przykładowo:

$$(a_3 + (a_4 + a_1)) + (a_2 + a_5) = a_1 + (a_2 + (a_3 + (a_4 + a_5)))$$

Postawiona teza zostanie udowodniona za pomocą dwóch oddzielnych indukcji, z których pierwsza będzie indukcją zupełną.

- **Przykład 2.8.** Poniżej zostanie dowiedzione za pomocą indukcji zupełnej względem n (liczby operandów w wyrażeniu) następujące twierdzenie.

TWIERDZENIE $S(n)$: jeżeli E jest wyrażeniem zawierającym operator (+) i n operandów oraz a jest jednym z tych operandów, to E może zostać przekształcone zgodnie z prawami łączności i przemienności do wyrażenia postaci $a+F$, gdzie F jest wyrażeniem zawierającym wszystkie operandy wyrażenia E oprócz a , pogrupowane w pewnym porządku za pomocą operatora (+).

Twierdzenie $S(n)$ jest prawdziwe tylko dla $n \geq 2$, ponieważ w wyrażeniu E musi istnieć przynajmniej jedno wystąpienie operatora (+). Tak więc podstawą będzie $n = 2$.

PODSTAWA. Niech $n = 2$; wówczas E może być albo wyrażeniem postaci $a+b$, albo $b+a$, dla pewnego operandu b różnego od a . W pierwszym przypadku wystarczy, że F będzie wyrażeniem b . W drugim przypadku można stwierdzić, że zgodnie z prawem przemienności, wyrażenie $b+a$ można przekształcić do postaci $a+b$, co ponownie pozwala na wykorzystanie podstawienia $F = b$.

INDUKCJA. Niech E zawiera $n+1$ operandów, oraz niech twierdzenie $S(i)$ będzie prawdziwe dla $i = 2, 3, \dots, n$. Należy udowodnić krok indukcyjny dla $n \geq 2$, tak więc można założyć, że E zawiera co najmniej trzy operandy, co oznacza przynajmniej dwa wystąpienia operatora (+). Wyrażenie E można zapisać w postaci E_1+E_2 dla pewnych wyrażeń E_1 i E_2 . Ponieważ E zawiera dokładnie $n+1$ operandów oraz wyrażenia E_1 i E_2 muszą zawierać przynajmniej po jednym z tych operandów, ani wyrażenie E_1 , ani E_2 nie może zawierać więcej niż n operandów. W tej sytuacji hipoteza indukcyjna ma zastosowanie względem wyrażeń E_1 i E_2 wówczas, gdy każde z nich zawiera więcej niż jeden operand (ponieważ dowód rozpoczęto dla podstawy $n = 2$). Istnieją cztery przypadki, które należy rozważyć, w zależności od tego, czy operand a występuje w wyrażeniu E_1 , czy E_2 oraz czy a jest lub nie jest jedynym operandem w wyrażeniu E_1 lub E_2 .

Łączność i przemienność

Jak Czytelnik zapewne pamięta, *prawo łączności* (ang. *associative law*) dodawania mówi o tym, że sumowanie trzech wartości można wykonać najpierw przez dodanie do siebie dwóch pierwszych i później dodanie do otrzymanej wartości trzeciej, albo przez dodanie pierwszej do wartości otrzymanej w wyniku dodania do siebie drugiej i trzeciej; wynik będzie ten sam. Oto formalny zapis:

$$(E_1 + E_2) + E_3 = E_1 + (E_2 + E_3)$$

gdzie E_1 , E_2 i E_3 są dowolnymi wyrażeniami arytmetycznymi. Przykładowo:

$$(1 + 2) + 3 = 1 + (2 + 3)$$

W tym przypadku $E_1 = 1$, $E_2 = 2$ oraz $E_3 = 3$. Prawdziwe jest także równanie:

$$((xy) + (3z - 2)) + (y + z) = xy + ((3z - 2) + (y + z))$$

gdzie $E_1 = xy$, $E_2 = 3z - 2$ oraz $E_3 = y + z$.

Z kolei *prawo przemienności* (ang. *commutative law*) dodawania mówi o tym, że możemy dwa wyrażenia można sumować w dowolnej kolejności. Oto formalny zapis:

$$E_1 + E_2 = E_2 + E_1$$

Przykładowo, $1 + 2 = 2 + 1$ oraz $xy + (3z - 2) = (3z - 2) + xy$.

- (a) E_1 to a . Przykładem takiej sytuacji jest wyrażenie E postaci $a + (b + c)$; wówczas wyrażenie E_1 ma postać a , zaś wyrażenie E_2 to $b + c$. W takim przypadku, E_2 pełni rolę F , co oznacza, że E znajduje się już w postaci $a + F$;
- (b) E_1 zawiera więcej niż jeden operand, wśród których jest a . Przykładowo:

$$E = (c + (d + a)) + (b + e)$$

gdzie $E_1 = c + (d + a)$ zaś $E_2 = b + e$. W takim przypadku, ponieważ E_1 zawiera nie więcej niż n i nie mniej niż dwa operandy, można zastosować hipotezę indukcyjną, która pozwoli określić, że wyrażenie E_1 można przekształcić zgodnie z prawami łączności i przemienności do postaci $a + E_3$. Wyrażenie E można więc przekształcić do postaci $(a + E_3) + E_2$. Stosując prawo łączności można stwierdzić, że wyrażenie E da się przekształcić dalej do postaci $a + (E_3 + E_2)$. Pozwala to określić, że F zastępuje $E_3 + E_2$, co jest dowodem kroku indukcyjnego dla tego przypadku. W przedstawionym powyżej przykładowym wyrażeniu E można by założyć, że wyrażenie $E_1 = c + (d + a)$ jest przekształcane na mocy hipotezy indukcyjnej do postaci $a + (c + d)$. Wówczas wyrażenie E można przegrupować do postaci $a + ((c + d) + (b + e))$;

- (c) E_2 to a . Przykładowo, $E = (b + c) + a$. W takim przypadku zastosowanie prawa przemienności pozwala przekształcić wyrażenie E do postaci $a + E_1$, które ma odpowiednią formę, jeżeli za E_1 podstawimy F ;

- (d) E_2 zawiera więcej niż jeden operand, z a wyłącznie. Przykładowo, $E = b+(a+c)$. Zastosowanie prawa przemienności pozwala przekształcić wyrażenie E do postaci E_2+E_1 i postąpić zgodnie ze schematem przedstawionym w przypadku b). Jeśli $E = b+(a+c)$, wyrażenie E zostaje przekształcone najpierw do postaci $(a+c)+b$. Hipoteza indukcyjna umożliwia wyrażenia przedstawienie $a+c$ w wymaganej formie, w której w rzeczywistości już się znajduje. Zastosowanie prawa łączności pozwala następnie przekształcić wyrażenie E do postaci $a+(c+b)$.

We wszystkich czterech przypadkach udało się przekształcić wyrażenie E do odpowiedniej postaci. Krok indukcyjny został więc udowodniony, co pozwala stwierdzić, że twierdzenie $S(n)$ jest prawdziwe dla wszystkich $n \geq 2$. •

- **Przykład 2.9.** Dowód indukcyjny z przykładu 2.8 prowadzi bezpośrednio do algorytmu przekształcania wyrażenia do pożądanej formy. Przykładowo, można rozważyć następujące wyrażenie:

$$E = (x + (z + v)) + (w + y)$$

i założyć, że v jest operandem, który należy „wyciągnąć”, czyli umieścić go w roli operandu a z przekształceniem z przykładu 2.8. Początkowo przykład spełnia założenia przypadku (b), gdzie $E_1 = x+(z+v)$, oraz $E_2 = w+y$.

Następnie należy przekształcić wyrażenie E_1 i „wyciągnąć” v . E_1 jest przykładem zgodnym z opisanym powyżej przypadkiem (d), można więc zastosować najpierw prawo przemienności w celu przekształcenia wyrażenia E_1 do postaci $(z+v)+x$. Zgodnie z procedurą opisaną dla przypadku (b), należy przekształcić wyrażenie $z+v$, które spełnia założenia przypadku (c). Stosując prawo przemienności można je przekształcić do postaci $v+z$.

Wyrażenie E_1 zostało już przekształcone do postaci $(v+z)+x$ i kolejne zastosowanie prawa łączności prowadzi do postaci $v+(z+x)$. To z kolei powoduje otrzymanie wyrażenia E w postaci $(v+(z+x))+(w+y)$. Zastosowanie prawa łączności umożliwia przekształcenie wyrażenia E do postaci $v+((z+x)+(w+y))$. Wyrażenie E ma więc postać $v+F$, gdzie F jest wyrażeniem $(z+x)+(w+y)$. Całą sekwencję przekształceń przedstawiono na rysunku 2.6. •

$$\begin{aligned} &(x + (z + v)) + (w + y) \\ &((z + v) + x) + (w + y) \\ &((v + z) + x) + (w + y) \\ &(v + (z + x)) + (w + y) \\ &(v + ((z + x) + (w + y))) \end{aligned}$$

RYSUNEK 2.6.
Stosując prawa łączności i przemienności, można „wyciągnąć” dowolny operand, np. v

Można teraz wykorzystać twierdzenie udowodnione w przykładzie 2.8 do dowiedzenia oryginalnego twierdzenia, że dowolne dwa wyrażenia zawierające wyłącznie operator $(+)$ i tę samą listę różnych operandów, można przekształcić jedno w drugie, stosując prawa łączności i przemienności. W poniższym dowodzie wykorzystywana jest indukcja słaba (zamiast indukcji pełnej), której poświęcono podrozdział 2.3.

- **Przykład 2.10.** Poniżej zostanie dowiedziona prawdziwość następującego twierdzenia za pomocą indukcji względem n (liczby operandów w wyrażeniu).

TWIERDZENIE $T(n)$: jeżeli E i F są wyrażeniami zawierającymi operator $(+)$ i ten sam zbiór n różnych operandów, to możliwe jest przekształcenie E w F za pomocą sekwencji zastosowań praw łączności i przemienności.

PODSTAWA. Jeśli $n = 1$, to oba wyrażenia muszą być pojedynczym operandem a . Ponieważ są tym samym wyrażeniem, E w oczywisty sposób można przekształcić w F .

INDUKCJA. Załóżmy, że $T(n)$ jest prawdziwe dla pewnego $n \geq 1$. Należy dowieść teraz prawdziwości twierdzenia $T(n+1)$. Niech E i F będą wyrażeniami zawierającymi ten sam zbiór $n+1$ różnych operandów oraz niech a będzie jednym z tych operandów. Ponieważ $n+1 \geq 2$, $S(n+1)$, czyli twierdzenie z przykładu 2.8, musi być prawdziwe. Można więc przekształcić wyrażenie E do postaci $a+E_1$ dla pewnego wyrażenia E_1 zawierającego pozostałe n operandów wyrażenia E . Podobnie, można przekształcić wyrażenie F do postaci $a+F_1$ dla pewnego wyrażenia F_1 zawierającego te same n operandów co E_1 . Co ważniejsze, w tym przypadku można także wykonać przekształcenie w przeciwnym kierunku, z $a+F_1$ do postaci F za pomocą praw łączności i przemienności.

Teraz można odwołać się do hipotezy indukcyjnej $T(n)$ dla wyrażen E_1 i F_1 . Każde z nich zawiera te same n operandów, zatem postawiona hipoteza indukcyjna ma zastosowanie. Mówi ona o tym, że można przekształcić wyrażenie E_1 w F_1 , a stąd także przekształcić wyrażenie $a+E_1$ w $a+F_1$. Możliwe są zatem następujące przekształcenia:

$$\begin{array}{ll} E \rightarrow \dots \rightarrow a+E_1 & \text{Stosując } S(n+1) \\ \rightarrow \dots \rightarrow a+F_1 & \text{Stosując } T(n) \\ \rightarrow \dots \rightarrow F & \text{Odwrotnie stosując } S(n+1) \end{array}$$

do otrzymania F z E . •

- **Przykład 2.11.** Poniżej przekształcono wyrażenie $E = (x+y)+(w+z)$ do postaci $F = ((w+z)+y)+x$. Najpierw należy wybrać operand do „wyciągnięcia”, niech będzie nim w . Jeśli sprawdzi się przykładki z przykładu 2.8, widać, że należy wykonać dla wyrażenia E następującą sekwencję przekształceń:

$$(x+y)+(w+z) \rightarrow (w+z)+(x+y) \rightarrow w+(z+(x+y)) \quad (2.7)$$

Wyrażenie F należy przekształcać w sposób następujący:

$$((w+z)+y)+x \rightarrow (w+(z+y))+x \rightarrow w+((z+y)+x) \quad (2.8)$$

teraz tym momencie pojawia się podproblem przekształcenia wyrażenia $z+(x+y)$ do postaci $(z+y)+x$. Jego rozwiązaniem jest „wyciągnięcie” operandu x . Sekwencja przekształceń wygląda więc następująco:

$$z+(x+y) \rightarrow (x+y)+z \rightarrow x+(y+z) \quad (2.9)$$

oraz

$$(z+y)+x \rightarrow x+(z+y) \quad (2.10)$$

To z kolei stwarza podproblem przekształcania wyrażenia $y+z$ do postaci $z+y$. Stosujemy zatem prawo przemienności, czyli — mówiąc dokładniej — dla obu wyrażeń wykorzystujemy przedstawioną w przykładzie 2.8 technikę „wyciągania” y , otrzymując w ten sposób wyrażenie $y+z$. Przypadek podstawowy z przykładu 2.10 określa, że wyrażenie z można teraz „przekształcić” do niego samego.

W tym momencie możliwe jest przekształcanie wyrażenia $z+(x+y)$ do postaci $(z+y)+x$ stosując kroki przedstawione w wierszu (2.9), stosując prawo przemienności względem podwyrażenia $y+z$ i ostatecznie dokonując przekształcenia odwrotnego do pokazanego w wierszu (2.10). Przekształcenia te stanowią pośredni etap przekształceń wyrażenia $(x+y)+(w+z)$ do postaci $((w+z)+y)+z$. Najpierw stosujemy przekształcenie z wiersza (2.7), następnie właśnie omówione z wyrażenia $z+(x+y)$ do postaci $(z+y)+x$, i wreszcie przekształcenie przeciwne do zawartego w wierszu (2.8). Całą sekwencję przekształceń przedstawiono na rysunku 2.7. •

$(x + y) + (w + z)$	Wyrażenie E	RYSUNEK 2.7. Przekształcanie jednego wyrażenia w drugie za pomocą praw łączności i przemienności
$(w + z) + (x + y)$	Środek wiersza (2.7)	
$w + (z + (x + y))$	Koniec wiersza (2.7)	
$w + ((x + y) + z)$	Środek wiersza (2.9)	
$w + (x + (y + z))$	Koniec wiersza (2.9)	
$w + (x + (z + y))$	Prawo przemienności	
$w + ((z + y) + x)$	Przeciwieństwo wiersza (2.10)	
$(w + (z + y)) + x$	Przeciwieństwo środka wiersza (2.8)	
$((w + z) + y) + x$	Wyrażenie F , przeciwieństwo końca wiersza (2.8)	

Ćwiczenia

2.4.1. „Wyciągnij” kolejno wszystkie operandy z wyrażenia $E = (u+v)+((w+(x+y))+z)$. Należy rozpocząć od wyrażenia E i dla każdego z jego sześciu operandów, stosując techniki opisane w przykładzie 2.8, przekształcić je do postaci $u+E_1$, $v+E_2$, itd.

2.4.2. Wykorzystaj technikę z przykładu 2.11 do przekształcania:

- (a) $w + (x + (y + z))$ do postaci $((w + x) + y) + z$
- (b) $(v + w) + ((x + y) + z)$ do postaci $((y + w) + (v + z)) + x$

2.4.3*. Niech E będzie wyrażeniem zawierającym operatory $+$, $-$, $*$ oraz $/$; każdy z nich jest jedynie binarny, co oznacza, że działa na dwóch operandach. Wykaż, stosując indukcję zupełną dla liczby wystąpień operatorów w wyrażeniu E , że jeśli E zawiera n wystąpień operatorów, to zawiera $n+1$ operandów.

2.4.4. Podaj przykład przemiennego operatora dwuargumentowego, który nie jest łączny.

2.4.5. Podaj przykład łącznego operatora dwuargumentowego, który nie jest przemienny.

2.4.6*. Rozważ wyrażenie E , którego wszystkie operatory są dwuargumentowe. *Długością* wyrażenia E jest liczba zawartych w nim symboli, licząc jako jeden symbol każdy operator oraz lewy lub prawy nawias, a także każdy operand, np. 123 lub abc . Udowodnij, że E musi mieć nieparzystą długość. *Wskazówka:* wykorzystaj indukcję zupełną względem długości wyrażenia E .

2.4.7. Wykaż, że wszystkie liczby ujemne można zapisać w postaci $2a+3b$ dla pewnych (niekoniecznie dodatnich) liczb całkowitych a i b .

Szablon ogólny dla indukcji

Poniższy schemat przeprowadzania dowodów indukcyjnych dotyczy indukcji zupełnej z wieloma przypadkami podstawowymi. Jako przypadek szczególny, zawiera on także omówioną w podrozdziale 2.3 indukcję słabą oraz często spotykaną sytuację, gdzie istnieje tylko jeden przypadek podstawowy.

- (1) Określ twierdzenie $S(n)$ do udowodnienia. Załóżmy, że należy udowodnić twierdzenie $S(n)$ za pomocą indukcji względem n , dla $n \geq i_0$. Określ wartość i_0 (najczęściej jest to 0 lub 1, może to jednak być dowolna liczba całkowita). Wyjaśnij intuicyjnie, jaką wielkość reprezentuje n .
- (2) Przedstaw przypadek podstawowy (może ich być więcej niż jeden). Będą to wszystkie liczby całkowite od i_0 do pewnej liczby j_0 . Często $i_0 = j_0$, jednak j_0 może być większe.
- (3) Udowodnij prawdziwość wszystkich przypadków podstawowych $S(i_0), S(i_0+1), \dots, S(j_0)$.
- (4) Wyznacz krok indukcyjny, stwierdzając, że zakładasz prawdziwość

$$S(i_0), S(i_0+1), \dots, S(j_0)$$

(czyli „hipotezy indukcyjnej”), oraz że chcesz udowodnić prawdziwość twierdzenia $S(n+1)$. Załóż, że $n \geq j_0$, co oznacza, że n jest co najmniej tak duże, jak największy przypadek podstawowy. Wyraż $S(n+1)$, zastępując n wyrażeniem $n+1$ w twierdzeniu $S(n)$.

- (5) Udowodnij prawdziwość $S(n+1)$ przy założeniach wspomnianych w punkcie 4. Jeśli przeprowadzasz indukcję słabą, zamiast zupełnej, w dowodzie wykorzystane będzie tylko twierdzenie $S(n)$, masz jednak swobodę w stosowaniu dowolnych twierdzeń z hipotezy indukcyjnej.
- (6) Wywnioskuj, że $S(n)$ jest prawdziwe dla wszystkich $n \geq i_0$ (ale niekoniecznie dla mniejszych n).

2.4.8*. Wykaż, że każdą liczbę całkowitą (dodatnią lub ujemną) można zapisać w postaci $5a+7b$ dla pewnych (niekoniecznie dodatnich) liczb całkowitych a i b .

2.4.9*. Czy każdy dowód, w którym wykorzystuje się indukcję słabą (jak w podrozdziale 2.3) jest także dowodem przez indukcji zupełnej? Czy każdy dowód, w którym wykorzystuje się indukcję zupełną jest także dowodem przez indukcję słabą?

2.4.10*. W niniejszym rozdziale przedstawiono uzasadnienie poprawności indukcji zupełnej za pomocą argumentu najmniejszego kontrprzykładu. Uzasadnij teraz poprawność indukcji zupełnej iteracyjnie.

2.5. Dowodzenie własności programów

W niniejszym podrozdziale zostaną omówione zagadnieniami, w których dowody indukcyjne mają zasadnicze znaczenie — dowodzeniem, że program robi to, co powinien. Zostanie przedstawiona technika wyjaśniania działania programu iteracyjnego w czasie wykonywania przebiegu pętli. Jeśli rozumie się działanie pętli, posiada się ogólne zrozumienie tego, co należy wiedzieć o działaniu programu iteracyjnego. W podrozdziale 2.9 zostaną rozważone zagadnienia niezbędne do dowodzenia własności programów rekurencyjnych.

Prawda w reklamach

Dowodzenie poprawności programów wiąże się z wieloma trudnościami, zarówno teoretycznymi, jak i praktycznymi. Oczywiście pytanie brzmi: „Co oznacza „poprawność” programu?” W rozdziale 1. wspomniano o tym, że w praktyce większość programów jest tworzona w celu wypełnienia jakiejś nieformalnej specyfikacji, która sama w sobie może być niekompletna lub niespójna. Nawet jeśli istnieje precyzyjna specyfikacja formalna, można wykazać, że nie istnieje algorytm dowodzący, iż dowolny program dokładnie odpowiada danej specyfikacji.

Wbrew tym trudnościom, bardzo korzystne jest jednak przedstawienie i udowodnienie określonych twierdzeń odnośnie do programów. Niezmienniki pętli w programie są często najbardziej przydatne w krótkich wyjaśnieniach opisujących działanie programu. Co więcej, programista powinien mieć na uwadze niezmienniki pętli już podczas pisania fragmentów kodu. Oznacza to, że musi być powód, dla którego program poprawnie działa i powód ten często wiąże się z hipotezą indukcyjną, która jest prawdziwa za każdym razem, gdy program działa w pętli lub gdy wykonuje wywołanie rekurencyjne. Programista powinien być w stanie wyobrazić sobie odpowiedni dowód, nawet jeśli jego zapisanie wiersz po wierszu byłoby niepraktyczne.

Niezmienniki pętli

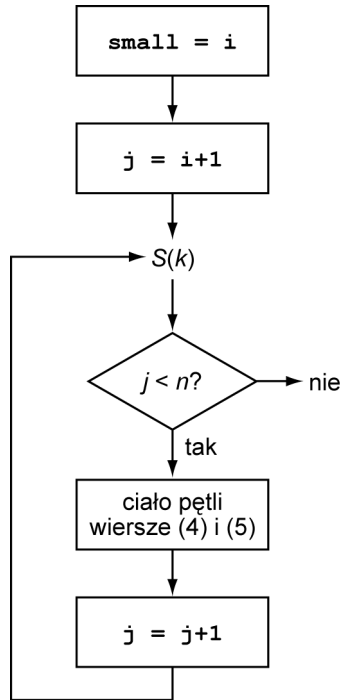
Kluczem do dowodzenia własności pętli w programie jest wybranie *niezmiennika pętli* (ang. *loop invariant*) lub *twierdzenia indukcyjnego* (ang. *inductive assertion*), które jest twierdzeniem S prawdziwym za każdym razem, gdy sterowanie dochodzi do konkretnego miejsca w pętli. Twierdzenie S jest następnie dowodzone za pomocą indukcji względem parametru, który w pewien sposób mierzy liczbę przebiegów pętli. Przykładowo, parametrem może być liczba dojsć do wybranego testu w pętli `while`, wartość indeksu pętli `for` lub pewne wyrażenie zawierające zmienne programowe, o których wiadomo, że zwiększa się o jeden dla każdego przebiegu pętli.

- **Przykład 2.12.** Jako pierwszy przykład zostanie rozważona pętla wewnętrzna funkcji `SelectionSort` z podrozdziału 2.2. Oto fragment kodu z oryginalną numeracją wierszy z listingu 2.1:

```
(2)  small = i;
(3)  for (j = i+1; j < n; j++)
(4)    if (A[j] < A[small])
(5)      small = j;
```

Jak Czytelnik zapewne pamięta, celem tych instrukcji było przypisanie zmiennej `small` indeksu najmniejszego elementu tablicy $A[i..n-1]$. Aby sprawdzić poprawność tego rozwiązania, wystarczy przeanalizować schemat blokowy z rysunku 2.8. Schemat pokazuje pięć kroków niezbędnych do wykonania programu:

- (1) Najpierw następuje inicjalizacja zmiennej `small` (jest jej przypisywana wartość i) — patrz wiersz (2).
- (2) Na początku pętli `for` (w wierszu (3)), następuje inicjalizacja zmiennej `j` wartością $i+1$.
- (3) Następnie następuje sprawdzenie czy $j < n$.
- (4) Jeśli tak, wykonywane jest ciało pętli, które składa się z wierszy (4) i (5).
- (5) Na końcu ciała pętli następuje zwiększenie wartości zmiennej `j` i powrót do instrukcji warunkowej.



RYSUNEK 2.8.
Schemat blokowy
dla pętli wewnętrznej
funkcji SelectionSort

Na rysunku 2.8 zaznaczono punkt znajdujący się bezpośrednio przed instrukcją warunkową, który oznaczono za pomocą niezmiennika pętli nazwanego $S(k)$; za moment okaże się, jaką postać powinno mieć to twierdzenie. W momencie dojścia do instrukcji warunkowej po raz pierwszy zmienna j ma wartość $i+1$, zmienna `small` przechowuje natomiast wartość i . Za drugim razem zmienna j ma wartość $i+2$, gdyż została już raz zwiększona. Ponieważ w ciele pętli (wiersze (4) i (5)) zmiennej `small` jest przypisywana wartość $i+1$, jeśli $A[i+1]$ jest mniejsze od $A[i]$, wiadomo, że zmienna `small` przechowuje indeks mniejszego z pary elementów $A[i]$ oraz $A[i+1]$ ⁵.

Podobnie, gdy po raz trzeci dochodzi się do instrukcji warunkowej, zmienna j przyjmuje wartość $i+3$, a zmiennej `small` zostaje przypisana wartość indeksu najmniejszego elementu tablicy $A[i..i+2]$. Można zatem spróbować udowodnić poniższe twierdzenie, które wydaje się być ogólną regułą.

TWIERDZENIE $S(k)$: jeśli dochodzi się do testu $j < n$ w instrukcji `for` w wierszu (3), gdzie k jest wartością indeksu pętli j , to wartością zmiennej `small` jest indeks najmniejszego elementu tablicy $A[i..k-1]$.

Należy zauważyć, że litera k określa jedną z wartości przyjmowanych przez zmienną j w trakcie przechodzenia pętli. Takie rozwiązanie jest mniej skomplikowane niż próba wykorzystania j jako wartości zmiennej j , ponieważ niekiedy zachodzi konieczność zachowania niezmienniczej wartości k , podczas gdy wartość zmiennej j będzie podlegać zmianom. Należy również zauważyć, że $S(k)$ ma postać „jeśli dochodzimy do ...”, ponieważ dla niektórych wartości k może okazać się, że dojście

⁵ W przypadku ich równości, zmienna `small` przyjmuje wartość i . W ogólności, przyjmujemy, że takie równości nie występują i używamy sformułowania „najmniejszy element”, gdy w rzeczywistości chodzi nam o „pierwsze wystąpienie najmniejszego elementu”.

do warunku pętli jest niemożliwe, na przykład jeśli przerwie się jej działanie dla jakiejś mniejszej wartości indeksu pętli j . Jeśli k jest jedną z tych wartości, to $S(k)$ na pewno jest prawdziwe, ponieważ dowolne twierdzenie postaci „jeśli A to B ” jest prawdziwe, gdy A jest fałszywe.

PODSTAWA. Przypadek podstawowy ma postać: $k = i+1$, gdzie i jest wartością zmiennej i z wiersza (3)⁶. Na początku pętli $j = i+1$, co oznacza, że właśnie została wykonana instrukcja z wiersza (2), która powoduje przypisanie zmiennej `small` wartości i , oraz została zainicjalizowana zmienna j wartością $i+1$, rozpoczynając tym samym działanie pętli. Twierdzenie $S(i+1)$ określa, że `small` jest indeksem najmniejszego elementu tablicy $A[i..i]$, co oznacza, że wartością zmiennej `small` musi być wartość i . Z technicznego punktu widzenia, należy także wykazać, że zmienna j nie może nigdy przyjąć wartości $i+1$, poza momentem dojścia do pierwszego testu. Można to wyjaśnić intuicyjnie — za każdym razem, gdy wykonywana jest iteracja pętli, zostaje zwiększona wartość zmiennej j , zatem jej wartość nigdy nie wróci do najniższego poziomu, czyli $i+1$ (dla pełnej jasności, należałoby przedstawić odpowiedni dowód indukcyjny dla założenia, że $j > i+1$ z wyjątkiem pierwszego przejścia przez test). Wykazano więc, że podstawa, $S(i+1)$, jest prawdziwa.

INDUKCJA. Jako hipoteza indukcyjna zostaje przyjęte założenie, że $S(k)$ jest prawdziwe dla pewnego $k \geq i+1$, a następnie dowiedziona prawdziwość twierdzenia $S(k+1)$. Po pierwsze, jeśli $k \geq n$, następuje przerwanie działania pętli nie później niż w momencie, gdy zmienna j osiąga wartość k ; daje to pewność, że nigdy nie dochodzi się do warunku pętli z wartością zmiennej j równą $k+1$. W takim przypadku, $S(k+1)$ na pewno jest prawdziwe.

Załóżmy więc, że $k < n$, tak aby móc sprawdzić twierdzenie dla zmiennej j równej $k+1$. $S(k)$ określa, że zmienna `small` indeksuje najmniejszy element tablicy $A[i..k-1]$; $S(k+1)$ określa, że zmienna `small` indeksuje najmniejszy element tablicy $A[i..k]$. Należy rozważyć, co dzieje się w ciele pętli (wiersze (4) i (5)), gdy zmienna j ma wartość k — istnieją wówczas dwa przypadki uzależnione od prawdziwości warunku z wiersza (4):

- (1) Jeśli $A[k]$ jest nie mniejsze od najmniejszego elementu $A[i..k-1]$, to wartość zmiennej `small` nie ulega zmianie. Jednak w takim przypadku zmienna `small` indeksuje także najmniejszy element tablicy $A[i..k]$, ponieważ $A[k]$ nie jest tym najmniejszym elementem. Można zatem wywnioskować, że w tym przypadku twierdzenie $S(k+1)$ jest prawdziwe.
- (2) Jeśli wartość $A[k]$ jest mniejsza od najmniejszego spośród elementów od $A[i]$ do $A[k-1]$, to zmiennej `small` zostaje przypisana wartość k . Ponownie można wywnioskować, że twierdzenie $S(k+1)$ jest prawdziwe, ponieważ k jest indeksem najmniejszego elementu tablicy $A[i..k]$.

W obu przypadkach zmienna `small` jest indeksem najmniejszego elementu tablicy $A[i..k]$. Przebieg pętli powoduje zwiększenie wartości zmiennej j . Zatem bezpośrednio przed sprawdzeniem warunkiem pętli, gdy j ma wartość $k+1$, twierdzenie $S(k+1)$ jest prawdziwe. W ten sposób wykazano, że $S(k)$ implikuje $S(k+1)$. Stanowi to ostatni etap indukcji i można wywnioskować, że $S(k)$ jest prawdziwe dla wszystkich wartości $k \geq i+1$.

Następnie twierdzenie $S(k)$ zostanie zastosowane w celu wyciągnięcia wniosków na temat pętli wewnętrznej w wierszach od (3) do (5). Wyjście z pętli następuje wówczas, gdy wartość zmiennej j osiąga n . Ponieważ $S(n)$ określa, że zmienna `small` indeksuje najmniejszy element tablicy $A[i..n-1]$, można wyciągnąć ciekawy wniosek na temat działania pętli wewnętrznej. Zagadnieniu temu poświęcony został kolejny przykład. •

⁶ Jak długo rozważamy tylko wiersze od (3) do (5), tak długo wartość zmiennej i nie jest modyfikowana. Wartość $i+1$ jest więc stałą, którą bez przeszkód można wykorzystać w charakterze wartości podstawowej.

- **Przykład 2.13.** Poniżej zostanie rozważona cała funkcja `SelectionSort`, której trzon przypominano na listingu 2.3. Schemat blokowy dla tego kodu przedstawiono na rysunku 2.9, na którym wyrażenie „ciało pętli” odnosi się do wierszy od 2. do 8. z listingu 2.3. Rozpatrywane twierdzenie indukcyjne, które jest określane jako $T(m)$, ponownie dotyczy tego, co jest prawdą bezpośrednio przed sprawdzeniem warunku zakończenia wykonywania pętli. Mówiąc nieformalnie, w momencie, gdy zmienna i ma wartość m , wybrano już m najmniejszych elementów tablicy i posortowano je na jej początku. Ścisłej rzecz ujmując, należy dowieść poniższe twierdzenie $T(m)$ za pomocą indukcji względem m .

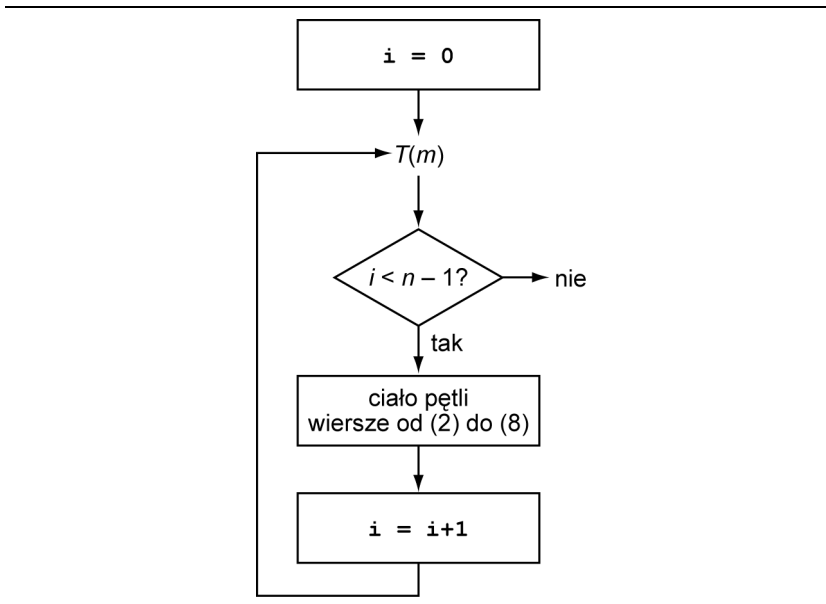
LISTING 2.3.

Ciało funkcji `SelectionSort`

```

(1)   for (i = 0; i < n-1; i++) {
(2)       small = i;
(3)       for (j = i+1; j < n; j++)
(4)           if (A[j] < A[small])
(5)               small = j;
(6)       temp = A[small];
(7)       A[small] = A[i];
(8)       A[i] = temp;
        }

```



RYSUNEK 2.9.
Schemat blokowy
dla całej funkcji
sortowania przez
wybieranie

TWIERDZENIE $T(m)$: jeśli dochodzimy do warunku pętli $i < n-1$ w wierszu (1) z wartością zmiennej i równą m , to:

- elementy tablicy $A[0..m-1]$ są już posortowane, co oznacza, że $A[0] \leq A[1] \leq \dots \leq A[m-1]$.
- wszystkie elementy tablicy $A[m..n-1]$ są co najmniej tak duże, jak wszystkie elementy tablicy $A[0..m-1]$.

PODSTAWA. Przypadkiem podstawowym jest $m = 0$. Z oczywistych powodów podstawa jest prawdziwa. Jeśli przyjrzymy się twierdzeniu $T(0)$, punkt (a) określa, że elementy tablicy $A[0 \dots -1]$ są posortowane. Przedział $A[0], \dots, A[-1]$ nie zawiera jednak żadnych elementów, zatem punkt (a) musi być prawdziwy. Podobnie, punkt (b) twierdzenia $T(0)$ określa, że wszystkie elementy tablicy $A[0 \dots n-1]$ są co najmniej tak duże, jak wszystkie elementy tablicy $A[0 \dots -1]$. Ponieważ drugi przedział nie zawiera żadnych elementów, punkt (b) także jest prawdziwy.

INDUKCJA. Na potrzeby kroku indukcyjnego zakładamy, że twierdzenie $T(m)$ jest prawdziwe dla pewnego $m \geq 0$ i wykazujemy, że prawdziwe jest twierdzenie $T(m+1)$. Podobnie jak w przykładzie 2.12, należy spróbować dowieść prawdziwości twierdzenia w postaci „jeżeli A , to B ”, które jest prawdziwe zawsze wtedy, gdy A jest fałszywe. $T(m+1)$ jest więc prawdziwe, jeśli założenie, że sprawdzenie warunku pętli `for` jest osiągane z wartością i równą $m+1$ jest fałszywe. Można więc założyć, że faktycznie sprawdzenie to jest osiągane ze zmienną i o wartość $m+1$, co oznacza, że można założyć, iż $m < n-1$.

Kiedy zmienna i ma wartość m , ciało pętli znajduje najmniejszy element tablicy $A[m \dots n-1]$ (dowodzono tego dla twierdzenia $S(m)$ w przykładzie 2.12). Znalezione element jest wymieniany z elementem $A[m]$ w wierszach od (6) do (8). Punkt (b) hipotezy indukcyjnej dla $T(m)$ stwierdza, że wybierany element musi być co najmniej tak duży, jak wszystkie elementy tablicy $A[0 \dots m-1]$. Co więcej, te elementy muszą być już posortowane, zatem wszystkie elementy tablicy $A[0 \dots m]$ są już posortowane. To dowodzi słuszności punktu (a) dla twierdzenia $T(m+1)$.

Aby udowodnić prawdziwość punktu (b) dla twierdzenia $T(m+1)$, można pokazać, że element $A[m]$ został wybrany jako element nie większy niż wszystkie elementy tablicy $A[m+1 \dots n-1]$. Punkt (b) dla twierdzenia $T(m)$ określa, że elementy tablicy $A[0 \dots m-1]$ są nie większe niż wszystkie elementy tablicy $A[m+1 \dots n-1]$. Po wykonaniu ciała pętli z wierszy od (2) do (8) i zwiększeniu i wiadomo, że wszystkie elementy tablicy $A[m+1 \dots n-1]$ są co najmniej tak duże, jak wszystkie elementy tablicy $A[0 \dots m]$. Ponieważ wartością zmiennej i jest teraz $m+1$, dowodzi to prawdziwości twierdzenia $T(m+1)$, a tym samym stanowi dowód kroku indukcyjnego.

Niech $m = n-1$. Wiemy, że wyjście z pętli zewnętrznej następuje w momencie, gdy zmienna i ma wartość $n-1$, zatem twierdzenie $T(n-1)$ będzie prawdziwe po zakończeniu pętli. Punkt (a) dla twierdzenia $T(n-1)$ określa, że wszystkie elementy tablicy $A[0 \dots n-2]$ są posortowane; punkt (b) określa natomiast, że element $A[n-1]$ jest co najmniej tak duży, jak dowolny inny element. Po zakończeniu działania programu elementy w tablicy A są więc ułożone w porządku niemalejącym, czyli są posortowane. •

Niezmienniki w pętlach `while`

Jeśli mamy do czynienia z pętlą w postaci:

```
while (<warunek>)  
    <ciało>
```

sensowne jest zazwyczaj znalezienie odpowiedniego niezmiennika pętli dla punktu bezpośrednio poprzedzającego sprawdzenie warunku. W ogólności, należy spróbować udowodnić prawdziwość niezmiennika pętli za pomocą indukcji względem liczby przebiegów pętli. Następnie, gdy warunek okaże się fałszywy, można wykorzystać niezmiennik pętli razem z informacją o fałszywości warunku do wywnioskowania czegoś przydatnego o tym, co jest prawdziwe po zakończeniu działania pętli `while`.

Jednak w przeciwieństwie do pętli `for`, może nie istnieć zmienna, której wartość odpowiadałaby łącznej liczbie przebiegów pętli `while`. Co gorsza, podczas gdy pętla `for` gwarantuje określoną maksymalną liczbę iteracji (przykładowo, do $n-1$ w przypadku pętli wewnętrznej funkcji `SelectionSort`), nie istnieją żadne przesłanki do tego, by wierzyć, że warunek pętli `while` stanie się fałszywy. Zatem częścią dowodu poprawności pętli `while` jest dowód jej ostatecznego zakończenia. Zazwyczaj tego typu dowody są oparte na identyfikacji pewnego wyrażenia E , zawierającego zmienne wykorzystywane w programie. Przykładowo:

- (1) Wartość wyrażenia E zmniejsza się o co najmniej 1 podczas każdego przebiegu pętli.
- (2) Warunek pętli jest fałszywy, jeśli wartość wyrażenia E jest nie większa od pewnej określonej stałej, np. 0.

- **Przykład 2.14.** Funkcję silni, zapisywaną jako $n!$, definiuje się jako iloczyn liczb całkowitych $1 \times 2 \times \dots \times n$. Przykładowo, $1! = 1$, $2! = 1 \times 2 = 2$, natomiast

$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

Na listingu 2.4 przedstawiono fragment prostego programu obliczającego $n!$ dla liczb całkowitych $n \geq 1$.

LISTING 2.4.

Fragment programu obliczającego silnię

```
(1)    scanf("%d", &n);
(2)    i = 2;
(3)    fact = 1;
(4)    while (i <= n) {
(5)        fact = fact*i;
(6)        i++;
(7)    }
(7)    printf("%d\n", fact);
```

Najpierw zostanie wykazane, że pętla `while` zapisana w wierszach od (4) do (6) na listingu 2.4 musi się zakończyć. Jako E zostaje określone wyrażenie $n-i$. Należy zauważyć, że w każdym przebiegu pętli wartość zmiennej i jest zwiększana o 1 w wierszu (6), zaś n pozostaje niezmienione. Zatem wartość wyrażenia E zmniejsza się z każdym przebiegiem pętli. Co więcej, jeśli E jest mniejsze lub równe -1 , zachodzi $n-i \leq -1$, czyli $i \geq n+1$. Jeśli więc E ma wartość ujemną, warunek pętli $i \leq n$ jest fałszywy i jej działanie kończy się. Nie wiadomo, jak dużą wartość ma początkowo E , ponieważ nie jest znana odczytywana przez program wartość zmiennej n . Jednak niezależnie od niej, E ostatecznie dojdzie do wartości mniejszej lub równej -1 i pętla się zakończy.

Należy teraz udowodnić, że program z listingu 2.4 działa zgodnie ze swoim przeznaczeniem. Poniżej przedstawiono odpowiednie twierdzenie o niezmienniku pętli, którego prawdziwość zostanie wykazana za pomocą indukcji względem wartości zmiennej i .

TWIERDZENIE $S(j)$: jeśli dochodzimy do warunku pętli $i \leq n$ ze zmienną i o wartości j , to wartością zmiennej `fact` jest $(j-1)!$.

PODSTAWA. Podstawą jest $S(2)$. Do warunku pętli można dojść ze zmienną i o wartości 2 tylko wówczas, gdy wchodzi się do pętli po raz pierwszy. Wcześniej, w wierszach (2) i (3) na listingu 2.4, następuje przypisanie zmiennej `fact` wartości 1 oraz zmiennej i wartości 2. Ponieważ $1 = (2-1)!$, podstawa jest prawdziwa.

INDUKCJA. Załóżmy, że $S(j)$ jest prawdziwe i wykażmy, że prawdziwe jest także $S(j+1)$. Jeśli $j > n$, przerywamy pętlę `while` (zmienna i jest wtedy mniejsza lub równa j), nigdy więc nie dochodzimy do warunku pętli ze zmienną i o wartości $j+1$. W takim przypadku twierdzenie $S(j+1)$ jest oczywiście prawdziwe, ponieważ sformułowano je w postaci „jeśli dochodzimy do warunku...”.

Zakładamy zatem, że $j \leq n$ i rozważamy działanie programu w momencie, gdy wykonywane będzie ciało pętli `while` ze zmienną i o wartości j . Z hipotezy indukcyjnej wiadomo, że zanim zostanie wykonana instrukcja z wiersza (5), zmienna `fact` będzie mieć wartość $(j-1)!$ oraz zmienna i będzie mieć wartość j . Po wykonaniu tej instrukcji zmienna `fact` przyjmie wartość $j \times (j-1)!$, czyli $j!$.

W wierszu (6) i jest zwiększane o 1, osiąga tym samym wartość $j+1$. Zatem w momencie dotarcia do warunku pętli, przy wartości zmiennej i wynoszącej $j+1$, zmienna `fact` zawiera wartość $j!$. Twierdzenie $S(j+1)$ określa, że w momencie, gdy i jest równe $j+1$, zmienna `fact` jest równa $((j+1)-1)!$, czyli $j!$. Dowodzi to tym samym prawdziwości twierdzenia $S(j+1)$ i kończy krok indukcyjny.

Wykazano już, że omawiana pętla `while` zakończy swoje działanie. Nie ma wątpliwości, że zakończy się w momencie, gdy zmiennej i zostanie przypisana po raz pierwszy wartość większą od n . Ponieważ zmienna i jest typu całkowitoliczbowego i jest zwiększana w każdej iteracji o 1, kiedy pętla kończy swoje działanie, zmienna i ma wartość $n+1$. Zatem w momencie dotarcia do wiersza (7), twierdzenie $S(n+1)$ musi być prawdziwe. Twierdzenie to mówi jednak, że zmienna `fact` ma wartość $n!$. Program wyświetla więc wartość $n!$, czego należało dowiedzieć.

Ze względów praktycznych należy podkreślić, że niezależnie od używanego komputera, przedstawiony na listingu 2.4 program liczący silnię wyświetli wartość $n!$ tylko dla bardzo małych wartości n . Problem polega na tym, że funkcja silni rośnie tak gwałtownie, że rozmiar odpowiedzi szybko przekracza maksymalną wielkość liczb całkowitych reprezentowanych na powszechnie używanych komputerach. •

Ćwiczenia

2.5.1. Jaki jest właściwy niezmiennik pętli `for` w następującym fragmencie programu, który przypisuje zmiennej `sum` sumę liczb całkowitych od 1 do n ?

```
scanf("%d", &n);
sum = 0;
for (i = 1; i <= n; i++)
    sum = sum+i;
```

Udowodnij poprawność wybranego niezmiennika pętli za pomocą indukcji względem i oraz wykorzystaj go do wykazania, że program działa zgodnie z oczekiwaniami.

2.5.2. Poniższy fragment kodu oblicza sumę liczb całkowitych umieszczonych w tablicy `A[0..n-1]`:

```
sum = 0;
for (i = 0; i < n; i++)
    sum = sum+A[i];
```

Jaki jest prawidłowy niezmiennik pętli? Wykorzystaj go do wykazania, że powyższy fragment działa zgodnie z oczekiwaniami.

2.5.3*. Rozważ poniższy fragment kodu:

```
scanf("%d", &n);
x = 2;
for (i = 1; i <= n; i++)
    x = x * x;
```

Właściwym niezmiennikiem pętli dla punktu bezpośrednio poprzedzającego sprawdzenie warunku $i \leq n$ jest kryterium: jeśli dochodzimy do tego punktu ze zmienną i o wartości k , to $x = 2^{2^{k-1}}$. Udowodnij za pomocą indukcji względem k , że opisany niezmiennik jest poprawny. Jaka jest wartość zmiennej x po zakończeniu działania pętli?

2.5.4*. We fragmencie kodu przedstawionym na listingu 2.5 odczytujemy liczby całkowite do momentu, w którym napotkamy ujemną liczbę całkowitą — wyświetlamy wówczas łączną sumę. Jaki jest właściwy niezmiennik pętli dla punktu bezpośrednio poprzedzającego warunek pętli? Wykorzystaj go do wykazania, że fragment kodu działa zgodnie z oczekiwaniami.

LISTING 2.5.

Sumowanie listy liczb całkowitych do momentu napotkania liczby ujemnej

```
sum = 0;
scanf("%d", &x);
while (x >= 0) {
    sum = sum+x;
    scanf("%d", &x);
}
```

2.5.5. Znajdź największą wartość n , dla której program z listingu 2.4 działa poprawnie na Twoim komputerze. Jaki wpływ na dowodzenie poprawności programu ma ograniczona długość reprezentowanych liczb całkowitych?

2.5.6. Wykaż za pomocą indukcji względem liczby przebiegów pętli ze schematu blokowego przedstawionego na rysunku 2.8, że po pierwszym przejściu $j > i+1$.

2.6. Definicje rekurencyjne

W *definicji rekursywnej* (ang. *recursive definition* — zwanej też *indukcyjną*) definiuje się jedną lub więcej klas reprezentujących ściśle powiązane ze sobą obiekty (lub fakty) na bazie tych samych obiektów. Taka definicja nie może być niezrozumiała, jak „dany element jest elementem mającym pewien kolor”, ani paradoksalna, jak „coś jest przyrządem wtedy i tylko wtedy, gdy nie jest przyrządem”. Definicja rekurencyjna powinna raczej zawierać:

- (1) Jedną lub więcej *reguł podstawowych* (ang. *basis rules*), z których niektóre definiują pewne obiekty proste.
- (2) Jedną lub więcej *reguł indukcyjnych* (ang. *inductive rules*), za pomocą których definiuje się większe obiekty na bazie mniejszych z tego samego zbioru.

- **Przykład 2.15.** W poprzednim podrozdziale zdefiniowano funkcję silni za pomocą algorytmu iteracyjnego: mnożąc $1 \times 2 \times \dots \times n$ otrzymywano $n!$. Można jednak zdefiniować wartość $n!$ także rekurencyjnie, co przedstawiono poniżej.

PODSTAWA. $1! = 1$.

INDUKCJA. $n! = n \times (n-1)!$.

Przykładowo, podstawa określa, że $1! = 1$. Fakt ten można wykorzystać w kroku indukcyjnym dla $n = 2$ do odkrycia, że:

$$2! = 2 \times 1! = 2 \times 1 = 2$$

Dla $n = 3, 4$ i 5 otrzymujemy:

$$3! = 3 \times 2! = 3 \times 2 = 6$$

$$4! = 4 \times 3! = 4 \times 6 = 24$$

$$5! = 5 \times 4! = 5 \times 24 = 120$$

itd. Mimo że na pierwszy rzut oka pojęcie silni zdefiniowano na podstawie samej silni, w praktyce można otrzymać wartość $n!$ dla rosnących wartości parametru n , opierając się wyłącznie na wartościach silni obliczonych dla mniejszych wartości n . Nowa definicja silni jest więc sensowna.

Ścisłe rzecz ujmując, należy udowodnić, że przedstawiona rekurencyjna definicja $n!$ daje te same wyniki, co przedstawiona wcześniej definicja oryginalna:

$$n! = 1 \times 2 \times \dots \times n$$

Aby tego dokonać, należy udowodnić następujące twierdzenie:

TWIERDZENIE $S(n)$: zdefiniowana rekurencyjnie powyżej funkcja $n!$ jest równa $1 \times 2 \times \dots \times n$.

Dowód będzie stanowił indukcję względem n .

PODSTAWA. $S(1)$ jest oczywiście prawdziwe. Podstawa definicji rekurencyjnej określa, że $1! = 1$; iloczyn $1 \times \dots \times 1$ (czyli iloczyn liczb całkowitych „od 1 do 1”), jest oczywiście równy 1.

INDUKCJA. Załóżmy, że $S(n)$ jest prawdziwe, co oznacza, że wartość $n!$ opisana powyższą definicją rekurencyjną jest równa $1 \times 2 \times \dots \times n$. Definicja rekurencyjna określa, że:

$$(n+1)! = (n+1) \times n!$$

Wykorzystując prawo przemienności mnożenia można zauważyć, że:

$$(n+1)! = n \times (n+1) \tag{2.11}$$

Z hipotezy indukcyjnej wiadomo, że:

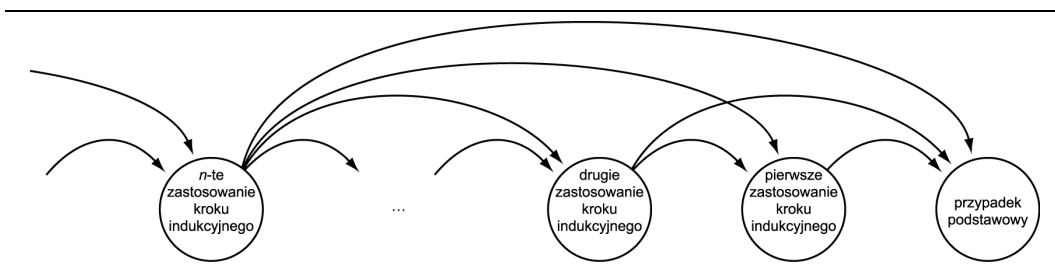
$$n! = 1 \times 2 \times \dots \times n$$

Można więc wyrażenie $1 \times 2 \times \dots \times n$ podstawić w równaniu (2.11) zamiast $n!$. Otrzymuje się wówczas:

$$(n+1)! = 1 \times 2 \times \dots \times n \times (n+1)$$

czyli twierdzenie $S(n+1)$. Udowodniono więc hipotezę indukcyjną i wykazano, że podana definicja rekurencyjna funkcji $n!$ jest równoważna z definicją iteracyjną. •

Rysunek 2.10 sugeruje ogólny charakter definicji rekurencyjnej. Jej struktura jest podobna do indukcji zupełnej, w której istnieje nieskończona sekwencja przypadków, z których każdy może zależeć od dowolnego lub od wszystkich wcześniejszych przypadków. W pierwszej kolejności należy zastosować regułę lub reguły podstawowe. W drugim etapie stosuje się regułę lub reguły indukcyjne względem już posiadanych elementów, w celu skonstruowania nowych faktów lub obiektów. W kolejnym przebiegu ponownie należy zastosować reguły indukcyjne względem posiadanych elementów, otrzymując nowe fakty lub obiekty, itd.



RYSUNEK 2.10. W definicji rekurencyjnej, obiekty są konstruowane w kolejnych przebiegach (obiekty skonstruowane w jednym etapie mogą zależeć od obiektów skonstruowanych we wszystkich wcześniejszych etapach)

W przykładzie 2.15, w którym zdefiniowano funkcję silni, wartość $1!$ odkryto na podstawie przypadku podstawowego, $2!$ po jednym zastosowaniu kroku indukcyjnego, $3!$ po dwóch takich zastosowaniach, itd. Zaprezentowana indukcja miała więc „typową” postać, w której na każdym etapie wykorzystywano wyłącznie to, co odkryto w poprzednim etapie.

- **Przykład 2.16.** W podrozdziale 2.2 zdefiniowano pojęcie porządku leksykograficznego dla ciągów i przedstawiona definicja była, zgodnie ze swoim charakterem, iteracyjna. Z grubsza rzecz biorąc, należy sprawdzić czy ciąg $c_1 \dots c_n$ poprzedza ciąg $d_1 \dots d_m$, porównując od lewej strony odpowiednie symbole c_i i d_i , aż do napotkania i , dla którego $c_i \neq d_i$ lub osiągnięcia końca jednego z porównywanych ciągów. Poniższa definicja rekurencyjna definiuje taką parę ciągów w i x , że w poprzedza x w porządku leksykograficznym. Zgodnie z intuicją, indukcja jest stosowana względem liczby par identycznych znaków rozpoczynających oba porównywane ciągi.

PODSTAWA. Podstawa opisuje takie pary ciągów, że można natychmiast odpowiedzieć na pytanie, który jest pierwszym w kolejności leksykograficznej. Podstawa składa się z dwóch części:

- (1) $\varepsilon < w$ dla dowolnego ciągu w różnego od ε . Jak Czytelnik zapewne pamięta, ε jest ciągiem pustym, czyli takim, który nie zawiera żadnych znaków.
- (2) Jeśli $c < d$, gdzie c i d są znakami, to dla dowolnych ciągów w i x zachodzi $cw < dx$.

INDUKCJA. Jeśli $w < x$ dla ciągów w i x , to dla dowolnego znaku c zachodzi $cw < cx$.

Przykładowo, można wykorzystać powyższą definicję do wykazania, że $\text{base} < \text{batter}$. Na podstawie drugiej reguły podstawy, gdzie $c = s$, $d = t$, $w = e$ oraz $x = \text{ter}$, mamy $se < tter$. Po jednokrotnym zastosowaniu reguły rekurencyjnej, dla $c = a$, $w = se$ i $x = tter$, można wywnioskować,

że $ase < atter$. Wreszcie, po zastosowaniu reguły rekurencyjnej po raz drugi dla $c = b$, $w = ase$ i $x = atter$, okazuje się, że $base < batter$, co oznacza, że podstawa i kroki indukcyjne przebiegały następująco:

se	<	tter
ase	<	atter
base	<	batter

Można także wykazać, że $bat < batter$, jak poniżej. Część 1. podstawy określa, że $\varepsilon < ter$. Jeśli trzykrotnie zastosuje się regułę rekurencyjną, dla c kolejno równego t , a oraz b , otrzymuje się następującą sekwencję wniosków:

ε	<	ter
t	<	tter
at	<	atter
bat	<	batter

Należy teraz dowieść za pomocą indukcji względem liczby identycznych znaków na początku obu ciągów, że jeden ciąg poprzedza drugi zgodnie z definicją z podrozdziału 2.2, wtedy i tylko wtedy, gdy poprzedza go zgodnie z przedstawioną właśnie definicją rekurencyjną. Oba dowody indukcyjne pozostawiono Czytelnikowi jako ćwiczenia. •

W przykładzie 2.16 zasugerowany na rysunku 2.10 zbiór faktów jest całkiem duży. Przypadek podstawowy opisuje wszystkie fakty $w < x$, dla których albo $w = \varepsilon$, albo w i x rozpoczynają się od różnych liter. Jedno zastosowanie kroku indukcyjnego prowadzi do wszystkich faktów $w < x$, gdzie w i x mają dokładnie jedną wspólną literę początkową. Drugie zastosowanie obejmuje przypadki, w których w i x mają dokładnie dwie wspólne litery początkowe, itd.

Wyrażenia

Wszystkie rodzaje wyrażeń arytmetycznych można w naturalny sposób zdefiniować rekurencyjnie. W podstawie definicji należy określić dopuszczalne operandy niepodzielne. Przykładowo, w języku C operandami niepodzielnymi mogą być zmienne lub stałe. Następnie indukcja określa, jakie operatory można stosować oraz na ilu operandach każdy z nich działa. Na przykład, w języku C operator $<$ można stosować względem dwóch operandów, symbol operatora - można stosować względem jednego lub dwóch operandów, natomiast operator wywołania funkcji (reprezentowany przez parę nawiasów oraz dowolną liczbę potrzebnych przecinków) można stosować dla jednego lub większej liczby operandów: $f(a_1, \dots, a_n)$.

- **Przykład 2.17.** Następujący zbiór wyrażeń często określa się jako „wyrażenia arytmetyczne”.

PODSTAWA. Poniższe typy operandów niepodzielnych są wyrażeniami arytmetycznymi:

- (1) Zmienne.
- (2) Liczby całkowite.
- (3) Liczby rzeczywiste.

INDUKCJA. Jeśli E_1 i E_2 są wyrażeniami arytmetycznymi, to następujące zdania także są wyrażeniami arytmetycznymi:

- (1) $(E_1 + E_2)$
- (2) $(E_1 - E_2)$
- (3) $(E_1 \times E_2)$
- (4) (E_1 / E_2)

O operatorach $+$, $-$, $\times$ oraz $/$ mówi się, że są operatorami *dwuargumentowymi* (ang. *binary*), ponieważ działają na dwóch argumentach. Mówi się także, że są to operatory *wrostkowe* (ang. *infix*), ponieważ są wstawiane pomiędzy swoimi dwoma argumentami.

Dodatkowo, znak minusa, poza odejmowaniem, może oznaczać liczbę ujemną (zmianę znaku). Tę możliwość uwzględnia piąta, ostatnia reguła rekurencyjna:

- (5) Jeśli E jest wyrażeniem arytmetycznym, to takim wyrażeniem jest także $(-E)$.

O operatorze takim jak $(-)$ w regule piątej, który działa tylko na jednym operandzie, mówi się, że jest operatorem *jednoargumentowym* (ang. *unary*). Mówi się także, że jest to operator *przedrostkowy* (ang. *prefix*), ponieważ stawia się go przed jego argumentem.

Na rysunku 2.11 przedstawiono pewne wyrażenia arytmetyczne i wyjaśniono dlaczego są one wyrażeniami. Warto zauważyć, że nawiasy są niekiedy zbędne i można je pominąć w zapisie wyrażenia. W ostatnim wyrażeniu (vi) nawias zewnętrzny można pominąć i zapisać całe wyrażenie w postaci $y \times (-(x+10))$. Nie można natomiast pominąć nawiasu zawierającego wyrażenie $x+10$, ponieważ wyrażenie postaci $y \times -x+10$ byłoby interpretowane jako $(y \times -x)+10$, które nie jest wyrażeniem równoważnym (wystarczy sprawdzić to np. dla $y = 1$ oraz $x = 0$)⁷. •

i) x	Reguła podstawowa (1)	RYSUNEK 2.11. Kilka przykładowych wyrażeń arytmetycznych
ii) 10	Reguła podstawowa (2)	
iii) $(x + 10)$	Reguła rekurencyjna (1) na podstawie punktów (i) oraz (ii)	
iv) $(-(x + 10))$	Reguła rekurencyjna (5) na podstawie punktu (iii)	
v) y	Reguła podstawowa (1)	
vi) $(y \times (-(x + 10)))$	Reguła rekurencyjna (3) na podstawie punktów (v) oraz (iv)	

Nawiasy zbilansowane

Spotykane w wyrażeniach ciągi nawiasów nazywamy *nawiasami zbilansowanymi* (ang. *balanced parentheses*). Przykładowo, w wyrażeniu (vi) z rysunku 2.11 występuje wzorzec $((()))$, natomiast wyrażenie:

$$((a+b)((c+d)-e))$$

⁷ Nawiasy są niepotrzebne, jeśli ich znaczenie pokrywa się z konwencjonalną kolejnością działań (najpierw jednoargumentowy minus, potem mnożenie i dzielenie, na końcu dodawanie i odejmowanie) oraz zasadą „lewej łączności”, która mówi o tym, że operatory na tym samym poziomie pierwszeństwa działań (np. ciąg plusów i minusów) grupuje się od lewej strony. Wspomniane konwencje powinny być znane zarówno z języka C, jak i zwykłej arytmetyki.

Więcej terminologii związanej z operatorami

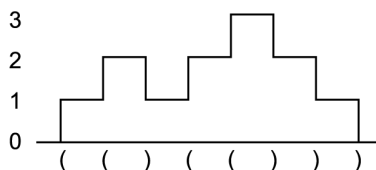
Operator jednoargumentowy zapisywany po argumencie, jak w przypadku operatora silni (!) w wyrażeniach typu $n!$, określa się jako operator *przyrostkowy* (ang. *postfix*). Operatory działające na więcej niż jednym operandzie mogą być zarówno operatorami przedrostkowymi, jak i przyrostkowymi, jeśli wstawia się je odpowiednio przed i po ich argumentach. Ani w języku programowania C, ani w zwykłej arytmetyce nie ma przykładów tego typu operatorów, jednak w podrozdziale 5.4 zostaną omówione notacje, w których wszystkie operatory są przedrostkowe lub przyrostkowe.

Operator działający na trzech argumentach nosi nazwę operatora trzyargumentowego (ang. *ternary*). W języku C takim operatorem jest $?:$, wyrażenie $c?x:y$ oznacza bowiem: „jeśli c , to x ; w przeciwnym razie, y ”. W ogólności, jeśli operator działa na k argumentach, mówi się o nim, że jest operatorem k -argumentowym.

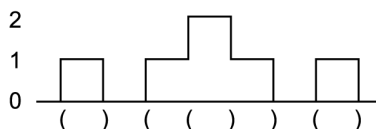
zawiera wzorzec $(((())))$. Ciąg pusty (ϵ) także jest ciągiem zbilansowanych nawiasów, jest to na przykład wzorzec wyrażenia x . W ogólności, warunkiem istnienia ciągu zbilansowanych nawiasów jest możliwość dopasowania każdego lewego nawiasu do prawego, występującego gdzieś na prawo od niego. Definicja „ciągu zbilansowanych nawiasów” składa się więc z dwóch reguł:

- (1) Ciąg zbilansowanych nawiasów zawiera taką samą liczbę lewych i prawych nawiasów.
- (2) Gdy analizuje się ciąg od strony lewej do prawej, profil ciągu nigdy nie jest ujemny, gdzie *profil* oznacza liczbę znalezionych do danego momentu lewych nawiasów pomniejszoną o liczbę znalezionych prawych nawiasów.

Warto zauważyć, że na początku i na końcu analizy profil musi mieć wartość 0. Przykładowo, rysunek 2.12(a) przedstawia profil dla ciągu $(((())))$, zaś rysunek 2.12(b) dla ciągu $((()) ())$.



(a) Profil łańcucha $(((())))$



(b) Profil łańcucha $((()) ())$

RYSUNEK 2.12.
Profile dla dwóch
ciągów nawiasów

Istnieje wiele rekurencyjnych definicji dla notacji „zbilansowanych nawiasów”. Poniższa definicja jest sprytnie sformułowana, jednak zostanie pokazane, że jest równoważna z wcześniejszą, nierekurencyjną definicją profilu.

PODSTAWA. Ciąg pusty jest ciągiem zbilansowanych nawiasów.

INDUKCJA. Jeśli x i y są ciągami zbilansowanych nawiasów, to także $(x)y$ jest ciągiem zbilansowanych nawiasów.

- **Przykład 2.18.** Zgodnie z podstawą, ε jest ciągiem zbilansowanych nawiasów. Jeśli przedstawiona reguła rekurencyjną zostanie zastosowana dla x i y równych ε , dochodzi się do stwierdzenia, że ciąg $()$ jest zbilansowany. Warto zauważyć, że jeśli podstawimy ciąg pusty za zmienną, np. x lub y , w procesie wnioskowania ta zmienna „zniknie”. Można wówczas zastosować regułę rekurencyjną dla wartości:

- (1) $x = ()$ oraz $y = \varepsilon$ do odkrycia, że $(())$ jest ciągiem zbilansowanym.
- (2) $x = \varepsilon$ oraz $y = ()$ do stwierdzenia, że $(())$ jest ciągiem zbilansowanym.
- (3) $x = y = ()$ do wywnioskowania, że $((()))$ jest ciągiem zbilansowanym.

I ostatni przykład: ponieważ wiadomo, że $(())$ oraz $(())$ są ciągami zbilansowanymi, można przypisać je odpowiednio do x i y i, za pomocą reguły rekurencyjnej, wykazać, że $((()))$ także jest ciągiem zbilansowanym. •

Można dowieść, że te dwie definicje „zbilansowanych nawiasów” opisują te same zbiory ciągów. Aby uprościć rozważania, ciągi zbilansowane zgodnie z definicją rekurencyjną będą nazywane *zbilansowanymi*, zaś ciągi zbilansowane zgodnie z definicją nierekurencyjną będą określane jako posiadające *zbilansowany profil*. Oznacza to, że ciągi ze zbilansowanym profilem to takie, których profil na końcu analizy wynosi 0 i nigdy nie jest liczbą ujemną. Należy udowodnić dwie rzeczy:

- (1) Każdy ciąg zbilansowany ma zbilansowany profil.
- (2) Każdy ciąg o zbilansowanym profilu jest zbilansowany.

Są to cele dowodów indukcyjnych przeprowadzonych w dwóch kolejnych przykładach.

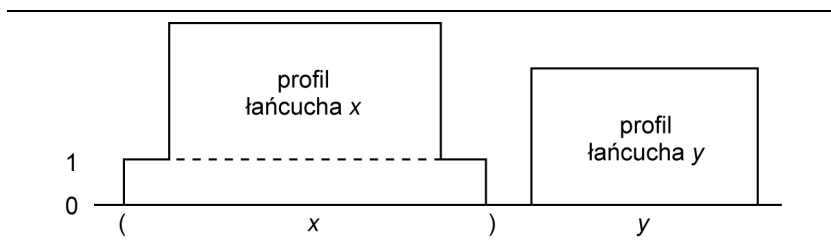
- **Przykład 2.19.** Najpierw zostanie dowiedziona prawdziwość punktu 1., który mówi o tym, że każdy ciąg zbilansowany ma zbilansowany profil. Dowodem jest indukcja zupełna odzwierciedlająca indukcję wykorzystaną do zdefiniowania klasy ciągów zbilansowanych. Oto dowodzone twierdzenie:

TWIERDZENIE $S(n)$: jeśli ciąg w jest zdefiniowany jako zbilansowany poprzez n zastosowań reguły rekurencyjnej, to w ma zbilansowany profil.

PODSTAWA. Podstawą jest $n = 0$. Jedynym ciągiem, co do którego można wykazać, że jest zbilansowany, bez stosowania reguły rekurencyjnej, jest ciąg pusty ε — zbilansowany zgodnie z regułą podstawową. Profil ciągu pustego, co oczywiste, ma na końcu wartość 0 i nigdy nie ma wartości ujemnej, ε ma zatem zbilansowany profil.

INDUKCJA. Załóżmy, że $S(i)$ jest prawdziwe dla $i = 0, 1, \dots, n$ i rozważmy przypadek $S(n+1)$, czyli ciąg w , którego dowód zbilansowania wymaga $n+1$ zastosowań reguły rekurencyjnej. Rozważmy ostatnie takie zastosowanie — analizujemy dwa ciągi x i y , o których już wiemy, że są zbilansowane, tworzące ciąg w postaci $(x)y$. Do stworzenia w wykorzystano regułę rekurencyjną $n+1$ razy, w tym raz w ostatnim kroku, który nie był związany z postacią ani x , ani y . Zatem ani x , ani y nie wymaga więcej niż n zastosowań reguły rekurencyjnej. Hipoteza indukcyjna ma więc zastosowanie zarówno do x , jak i do y , co pozwala na wyciągnięcie wniosku, że x i y mają zbilansowane profile.

Przykładowy profil ciągu w przedstawiono na rysunku 2.13. Najpierw profil podnosi się o jeden poziom w odpowiedzi na pierwszy lewy nawias. Następnie następuje profil ciągu x , który dalej podnosi profil na kolejny poziom ponad dotychczasowy (oznaczony przerywaną linią). Wykorzystano hipotezę indukcyjną do wywnioskowania, że ciąg x ma zbilansowany profil, czyli taki, który na końcu analizy ma wartość 0 i nigdy nie ma wartości ujemnej. Część profilu ciągu w związana z ciągiem x podnosi się o jeden poziom (patrz rysunek 2.13), zatem rozpoczyna się od poziomu 1 i nigdy poniżej tego poziomu nie schodzi.



RYСУNEK 2.13.
Konstruowanie
profilu ciągu $w = (x)y$

Jawnie użyty prawy nawias pomiędzy x i y obniża profil ciągu w do poziomu 0. Następnie rozpoczyna się profil ciągu y . Zgodnie z hipotezą indukcyjną, ciąg y ma zbilansowany profil, zatem część profilu ciągu w związana z ciągiem y nie schodzi poniżej poziomu 0 i kończy profil ciągu w na poziomie 0.

Skonstruowano właśnie profil ciągu w i przekonano się, że spełnia on warunki ciągu o zbilansowanym profilu, czyli takiego, który rozpoczyna się i kończy na poziomie 0 i nigdy nie przyjmuje wartości ujemnych. Udowodniono więc, że jeśli ciąg jest zbilansowany, to ma zbilansowany profil. •

Teraz należy skupić się na drugim kierunku równoważności pomiędzy dwiema definicjami „nawiasów zbilansowanych”. W kolejnym przykładzie zostanie pokazane, że ciąg o zbilansowanym profilu jest zbilansowany.

- **Przykład 2.20.** Należy dowieść prawdziwości drugiej części definicji, zgodnie z którą „ciąg o zbilansowanym profilu” implikuje „ciąg zbilansowany”. Zostanie wykorzystana indukcja zupełna względem długości ciągu nawiasów. Formalny zapis twierdzenia przedstawiono poniżej.

TWIERDZENIE $S(n)$: jeśli ciąg w o długości n ma zbilansowany profil, to jest zbilansowany.

PODSTAWA. Jeśli $n = 0$, ciąg musi być pusty (ε). Z reguły podstawowej definicji rekurencyjnej wiadomo, że ciąg ε jest zbilansowany.

INDUKCJA. Przypuśćmy, że ciągi o zbilansowanym profilu, o długości równej lub mniejszej od n , są zbilansowane. Należy dowieść twierdzenie $S(n+1)$, czyli, że ciągi o zbilansowanym profilu, o długości $n+1$ także są zbilansowane⁸. Rozważmy taki ciąg w . Ponieważ ciąg w ma zbilansowany profil, nie może się rozpoczynać od prawego nawiasu — jego profil byłby wówczas już na samym początku analizy ujemny. Ciąg w rozpoczyna się więc od lewego nawiasu.

⁸ Warto zauważyć, że wszystkie ciągi o zbilansowanym profilu mają parzystą długość, nieparzystość $n+1$ nie jest więc problemem. Kryterium parzystości n nie jest bowiem brane pod uwagę podczas dowodu.

Dowody dla definicji rekurencyjnych

Warto zauważyć, że przykład 2.19 dowodzi prawdziwości założenia dotyczącego klasy rekurencyjnie zdefiniowanych obiektów (zbilansowanych ciągów nawiasów) za pomocą indukcji względem liczby zastosowań reguły rekurencyjnej prowadzących do stwierdzenia, że obiekt należy do zdefiniowanej klasy. Jest to powszechny sposób postępowania z pojęciami definio-
wanymi rekurencyjnie; faktycznie jest to także jedna z przyczyn użyteczności takich definicji. Wystarczy przypomnieć przykład 2.15, w którym udowodniono własność zdefiniowanej rekurencyjnie funkcji silnia ($n!$ jest iloczynem liczb całkowitych od 1 do n) za pomocą indukcji względem n . Jednak n to także 1 plus liczba zastosowań reguły rekurencyjnej w definicji $n!$, zatem dowód można było również potraktować jako indukcję względem liczby zastosowań reguły rekurencyjnej.

Podzielmy teraz ciąg w na dwie części. Pierwsza część rozpoczyna się z początkiem ciągu w i kończy w miejscu, gdzie jego profil wraca do poziomu 0. Druga część to reszta ciągu w . Przykładowo, profil na rysunku 2.12(a) pierwszy raz wraca do poziomu 0 na końcu ciągu, zatem jeśli $w = (() (()))$, pierwszą częścią jest cały ciąg w , drugą jest ciąg pusty ε . Rysunek 2.12(b) przedstawia ciąg $w = () (()) ()$ — pierwszą częścią jest $()$, drugą $(()) ()$.

Pierwsza część nigdy nie może kończyć się lewym nawiasem, ponieważ jej profil na pozycji bezpośrednio poprzedzającej ten nawias byłby wówczas ujemny. Pierwsza część zaczyna się więc od lewego nawiasu i kończy prawym. Można zatem zapisać ciąg w w postaci $(x)y$, gdzie (x) jest pierwszą częścią, a y drugą. Zarówno x , jak i y są krótsze od w , jeśli więc można wykazać, że mają zbilansowane profile, będzie można wykorzystać hipotezę indukcyjną do wywnioskowania, że są także zbilansowane. Następnie będzie można wykorzystać regułę rekurencyjną z definicji zbilansowanego ciągu do wykazania, że także ciąg $w = (x)y$ jest zbilansowany.

Łatwo zauważyć, że y ma zbilansowany profil. Relacje pomiędzy profilami ciągów w , x i y ilustruje rysunek 2.13. Widać, że profil ciągu y , który rozpoczyna się i kończy na poziomie 0, jest końcówką profilu ciągu w . Ponieważ w ma zbilansowany profil, można wywnioskować, że tak samo jest w przypadku ciągu y . Wykazanie, że ciąg x ma zbilansowany profil przebiega niemal identycznie. Profil x jest częścią profilu w ; rozpoczyna się i kończy na poziomie 1 profilu ciągu w , można jednak obniżyć ten poziom o jeden w celu otrzymania profilu samego ciągu x . Wiadomo, że profil ciągu w nigdy nie spada do poziomu 0 w części zajmowanej przez ciąg x , ponieważ wybrano (x) jako najkrótszy prefiks ciągu w , kończący się w miejscu, w którym profil ciągu w spada właśnie do poziomu 0. Zatem profil ciągu x wewnątrz ciągu w nigdy nie osiąga poziomu 0 i nigdy nie przyjmuje wartości ujemnej.

Wykazano właśnie, że zarówno ciąg x , jak i y mają zbilansowany profil. Ponieważ oba ciągi są krótsze od w , można zastosować względem nich hipotezę indukcyjną i stwierdzić, że są zbilansowane. Reguła rekurencyjna zbilansowanych ciągów mówi, że jeśli x i y są zbilansowane, to zbilansowany jest także ciąg $(x)y$. Wiadomo, że $w = (x)y$, zatem ciąg w jest zbilansowany. Tym samym zakończono krok indukcji i wykazano, że twierdzenie $S(n)$ jest prawdziwe dla wszystkich $n \geq 0$. •

Ćwiczenia

2.6.1*. Udowodnij, że definicje porządku leksykograficznego przedstawione w przykładzie 2.16 i podrozdziale 2.2 są równoważne. *Wskazówka:* dowód powinien składać się z dwóch części, z których każda powinna być dowodem indukcyjnym. W pierwszej zakłada się, że $w < x$ zgodnie z definicją

z przykładu 2.16. Należy dowieść prawdziwości twierdzenia $S(i)$ za pomocą indukcji względem i : „jeśli konieczne jest zastosowane reguły rekurencyjnej i razy do wykazania, że $w < x$, to w poprzedza x zgodnie z definicją porządku leksykograficznego z podrozdziału 2.2”. Podstawą jest przypadek $i = 0$. Drugą częścią dowodu jest wykazanie, że jeśli w poprzedza x w porządku leksykograficznym zgodnie z definicją z podrozdziału 2.2, to $w < x$ zgodnie z definicją z przykładu 2.16. Tym razem należy wykorzystać indukcję względem liczby takich samych początkowych liter w ciągach w i x .

2.6.2. Narysuj profile następujących ciągów nawiasów:

- (a) (() ());
- (b) () ()) (();
- (c) ((() ()) ());
- (d) (() (() ()))).

Które mają zbilansowane profile? Dla nich zastosuj definicję rekurencyjną z podrozdziału 2.6, by wykazać, że są zbilansowane.

2.6.3*. Wykaż, że każdy ciąg zbilansowanych nawiasów (zgodnie z definicją rekurencyjną z podrozdziału 2.6) jest ciągiem nawiasów dla pewnego wyrażenia arytmetycznego (definicję wyrażenia arytmetycznego można znaleźć w przykładzie 2.17). *Wskazówka:* wykorzystaj dowód indukcyjny względem liczby zastosowań reguły rekurencyjnej z definicji „zbilansowanych nawiasów” niezbędnych do skonstruowania danego ciągu zbilansowanych nawiasów.

2.6.4. Określ, czy każdy z poniższych operatorów języka C jest operatorem przedrostkowym, przyrostkowym, czy wrostkowym. Określ także, czy przedstawione operatory są jedno-, dwu-, czy też k -argumentowe dla pewnego $k > 2$:

- (a) $<$;
- (b) $\&$;
- (c) $\%$.

2.6.5. Jeśli znasz system plików systemu operacyjnego Unix lub podobnego, podaj definicję rekurencyjną możliwych struktur plików i katalogów.

2.6.6*. Pewien zbiór liczb całkowitych S jest rekurencyjnie zdefiniowany za pomocą poniższych reguł.

PODSTAWA. 0 należy do S .

INDUKCJA. Jeśli i należy do S , to $i+5$ oraz $i+7$ także należą do S .

- (a) jaka jest największa liczba całkowita, która *nie* należy do S ?
- (b) niech j będzie odpowiedzią udzieloną na poprzednie pytanie. Udowodnij, że wszystkie liczby całkowite większe lub równe $j+1$ należą do S . *Wskazówka:* zwróć uwagę na podobieństwo do ćwiczenia 2.4.8 (mimo że tutaj mamy do czynienia wyłącznie z nieujemnymi liczbami całkowitymi).

2.6.7*. Zdefiniuj rekurencyjnie zbiór parzystych ciągów za pomocą indukcji względem długości ciągu. *Wskazówka:* pomocne będzie jednocześnie zdefiniowanie dwóch pojęć — zarówno ciągów parzystych, jak i nieparzystych.

2.6.8*. Posortowaną listę liczb całkowitych można zdefiniować w następujący sposób.

PODSTAWA. Lista składająca się z jednej liczby całkowitej jest posortowana.

INDUKCJA. Jeśli L jest listą posortowaną, w której ostatnim elementem jest a , oraz jeśli $b \geq a$, to L wraz z występującym po nim b także tworzą listą posortowaną.

Udowodnij, że powyższa rekurencyjna definicja „listy posortowanej” jest równoważna z oryginalną, nierekurencyjną definicją, która mówi, że lista składa się z liczb całkowitych:

$$a_1 \leq a_2 \leq \dots \leq a_n$$

Pamiętaj, że dowód będzie się musiał składać z dwóch części: (a) jeśli lista jest posortowana zgodnie z definicją rekurencyjną, to jest także posortowana zgodnie z definicją nierekurencyjną; oraz (b) jeśli lista jest posortowana zgodnie z definicją nierekurencyjną, to jest także posortowana zgodnie z definicją rekurencyjną. W części (a) można wykorzystać indukcję względem liczby zastosowań reguły rekurencyjnej, a w części (b) — indukcję względem długości listy.

2.6.9.** Jak zasugerowano na rysunku 2.10, za każdym razem, gdy mowa o definicji rekurencyjnej, można sklasyfikować obiekty zdefiniowane zgodnie z opisem „etapu” ich generowania, czyli liczbę zastosowań kroku indukcyjnego w celu uzyskania danego obiektu. W przykładach 2.15 i 2.16 opisanie wyników wygenerowanych w każdym z etapów było dosyć łatwe. Niekiedy jest to dużo trudniejsze. Jak scharakteryzujesz obiekty wygenerowane zgodnie z poniższymi definicjami w n -tym etapie?

- (a) definicja wyrażeń arytmetycznych, takich jak opisane w przykładzie 2.17. *Wskazówka:* jeśli nieobca jest Ci problematyka drzew, które zostaną omówione w rozdziale 5., możesz rozważyć drzewiastą reprezentację wyrażeń;
- (b) definicja ciągów zbilansowanych. Należy zauważyć, że „liczba zastosowań”, omówiona w przykładzie 2.19, nie jest tożsama z liczbą etapów, w których są odkrywane ciągi. Przykładowo, dla ciągu $(\)(\)(\)$ należy wykorzystać regułę trzykrotnie, ale ciąg jest odkrywany już w drugim etapie.

2.7. Funkcje rekurencyjne

Funkcja rekurencyjna to taka, która jest wywoływana z poziomu własnej treści. Takie wywołanie określa się jako *bezpośrednie* (ang. *direct*); przykładowo, funkcja F zawiera we własnej treści wywołanie funkcji F . Czasem jednak wywołanie jest *pośrednie* (ang. *indirect*): pewna funkcja F_1 wywołuje bezpośrednio funkcję F_2 , która wywołuje bezpośrednio funkcję F_3 , itd., aż w końcu należąca do tej sekwencji pewna funkcja F_k wywołuje funkcję F_1 .

Istnieje powszechne przekonanie o tym, że nauczanie się programowania iteracyjnego czy też stosowania nierekurencyjnych wywołań funkcji jest łatwiejsze niż nauczanie się programowania rekurencyjnego. Mimo że nie można kategorycznie zaprzeczyć tego typu poglądom, autorzy wierzą, że po zdobyciu odpowiedniego doświadczenia programowanie rekurencyjne jest równie łatwe. Programy rekurencyjne są często mniejsze i łatwiejsze do zrozumienia od ich iteracyjnych odpowiedników. I, co ważniejsze, niektóre problemy są znacznie łatwiejsze do rozwiązania za pomocą programów rekurencyjnych niż programów iteracyjnych⁹.

⁹ Do takich problemów należą często różnego rodzaju problemy wyszukiwania. Przykładowo, w rozdziale 5. do przeszukiwania drzew zostaną wykorzystane pewne algorytmy rekurencyjne, które nie mają odpowiedników iteracyjnych (mimo że istnieją równoważne algorytmy wykorzystujące stosy).

Więcej prawdy w reklamach

Potencjalną wadą stosowania rekurencji jest długi czas związany z wywołaniami funkcji na niektórych maszynach — program rekurencyjny może być bardziej kosztowny czasowo od iteracyjnej wersji tego samego programu. Jednak na wielu współczesnych maszynach, wywołania funkcji są efektywne, zatem ten argument przeciwko wykorzystywaniu programów rekurencyjnych traci na znaczeniu.

Nawet na maszynach z powolnym mechanizmem wywoływania funkcji, można określić profil wykonania programu, by dowiedzieć się, ile czasu traci się na wykonanie poszczególnych jego części. Następnie można ponownie opracować te części programu, które wymagają największych nakładów czasowych, zastępując (w razie konieczności) rekurencję rozwiązaniami iteracyjnymi. W ten sposób można wykorzystać zalety rekurencji w większości programu, z wyjątkiem małych fragmentów kodu, w których szybkość przetwarzania ma podstawowe znaczenie.

Często można opracowywać algorytmy rekurencyjne, naśladując definicje rekurencyjne zawarte w specyfikacji programu, który jest implementowany. Funkcja rekurencyjna, implementująca definicję rekurencyjną składa się z części podstawowej i indukcyjnej. W części podstawowej sprawdza się często, czy nie jest rozpatrywany prosty typ danych wejściowych, dzięki czemu można by rozwiązać zadanie opierając się na podstawie definicji, bez konieczności stosowania wywołań rekurencyjnych. Część indukcyjna funkcji wymaga jednego lub większej liczby wywołań rekurencyjnych siebie samej i jest implementacją indukcyjnej części definicji. Dobrą ilustracją tego schematu są przedstawione poniżej przykłady.

- **Przykład 2.21.** Listing 2.6 zawiera przykład funkcji rekurencyjnej, obliczającej $n!$ dla danej dodatniej liczby całkowitej n . Funkcja ta jest bezpośrednim przekształceniem definicji rekurencyjnej dla funkcji silnia z przykładu 2.15. W wierszu (1) listingu 2.6 następuje rozróżnienie przypadku podstawowego od indukcyjnego. Zakłada się, że $n \geq 1$, zatem warunek w wierszu (1) sprawdza w rzeczywistości, czy $n = 1$. Jeśli tak, w wierszu (2) stosowana jest reguła podstawowa, $1! = 1$. Jeśli $n > 1$, w wierszu (3) stosowana jest reguła indukcyjna, $n! = n \times (n-1)!$.

LISTING 2.6.

Funkcja rekurencyjna obliczająca $n!$ dla $n \geq 1$

```

        int fact(int n)
        {
(1)      if (n <= 1)
(2)          return 1; /* podstawa */
        else
(3)          return n*fact(n-1); /* indukcja */
        }

```

Przykładowo, jeśli zostanie wywołana funkcja `fact(4)`, efektem będzie wywołanie funkcji `fact(3)`, która wywoła funkcję `fact(2)`, a ta z kolei wywoła funkcję `fact(1)`. W tym momencie funkcja `fact(1)` zastosuje regułę podstawową, ponieważ $n \leq 1$, i zwróci do funkcji `fact(2)` wartość 1. To wywołanie funkcji `fact` wykona wiersz (3) i zwróci 2 do funkcji `fact(3)`, która z kolei zwróci 6 do `fact(4)`, a ta zakończy przetwarzanie na wierszu (3) i jako wynik zwróci wartość 24. Rysunek 2.14 ilustruje schemat wywołań funkcji i zwracanych wartości. •

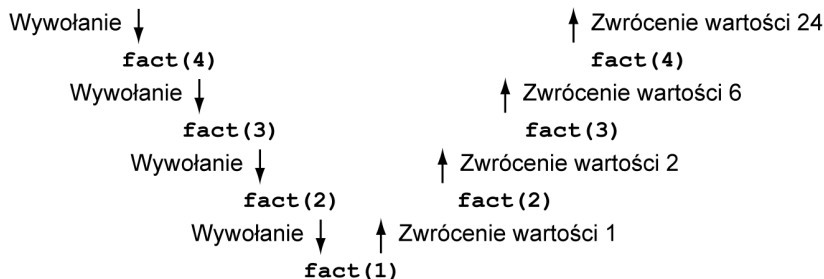
Programowanie defensywne

Program pokazany na listingu 2.6 ilustruje ważne zagadnienie dotyczące pisania programów rekurencyjnych w taki sposób, by nie zapęłtiły się w nieskończonej liczbie wywołań. Założono tam, że funkcja `fact` nigdy nie będzie wywoływana z argumentem mniejszym od 1. Najlepszym rozwiązaniem byłoby oczywiście rozpocząć tę funkcję od sprawdzenia czy $n \geq 1$ i, w przypadku niespełnienia tego warunku, wyświetlić stosowny komunikat o błędzie oraz zwrócić jakąś wartość, np. 0. Jednak nawet jeśli jest się całkowicie pewnym, że funkcja `fact` nigdy nie będzie wywoływana z wartością $n < 1$, należy być na tyle rozważnym, by umieścić w przypadku podstawowym także wszystkie „błędne przypadki”. Wówczas funkcja `fact` wywołana z błędnymi danymi wejściowymi zwróci po prostu wartość 1, która oczywiście nie będzie prawidłowym wynikiem, nie powoduje jednak krytycznych błędów w działaniu całego programu (w rzeczywistości, wartość 1 jest poprawna nawet dla wartości $n = 0$, ponieważ, zgodnie z konwencją, definiuje się $0!$ jako 1).

Przypuśćmy jednak, że zignorowano błędne przypadki i zapisano wiersz (1) z listingu 2.6 jako:

```
if (n == 1)
```

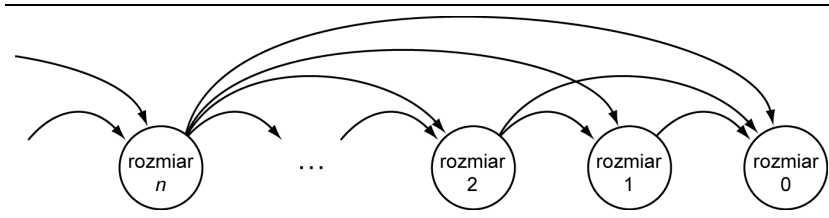
Jeśli nastąpi wówczas wywołanie `fact(0)`, dla funkcji będzie to przykład zgodny z przypadkiem indukcyjnym, co spowoduje wywołanie funkcji `fact(-1)`, następnie `fact(-2)`, itd. Będzie to trwało do momentu przerwania działania programu na skutek błędu związanego z wyczerpaniem się przestrzeni pamięciowej komputera wykorzystywanej do wywołań rekurencyjnych.



RYSUNEK 2.14.
Wywołania funkcji
i zwrócenia
ich wyników
w efekcie wywołania
funkcji `fact(4)`

Zagadnienie rekurencji można zobrazować tak samo, jak miało to wcześniej miejsce w odniesieniu do dowodów indukcyjnych czy definicji. Na rysunku 2.15 założono, że istnieje pojęcie „rozmiaru” argumentów funkcji rekurencyjnej. Przykładowo, w przypadku funkcji `fact` z przykładu 2.21, wartość argumentu n już jest odpowiednim rozmiarem. Zagadnienia związane z rozmiarami argumentów zostaną przeanalizowane dokładniej w podrozdziale 2.9. Na tym etapie należy jednak stwierdzić, że istotne znaczenie ma rekurencyjne wywoływanie funkcji dla argumentów o coraz mniejszych rozmiarach. Należy przecież osiągnąć ostatecznie przypadek podstawowy — czyli taki, który kończy rekurencję — w momencie, gdy dojdzie się do określonego rozmiaru argumentu (na rysunku 2.15 tym rozmiarem jest 0).

W przypadku funkcji `fact`, stosowane wywołania nie są tak ogólne, jak to zasugerowano na rysunku 2.15. Wywołanie funkcji `fact(n)` powoduje bezpośrednie wywołanie funkcji `fact(n-1)`, jednak sama funkcja `fact(n)` nie może bezpośrednio wywołać funkcji `fact` z dowolnym mniejszym argumentem.



RYSUNEK 2.15.
Funkcja rekurencyjna
wywołująca samą
siebie z argumentami
o coraz mniejszym
rozmiarze

- **Przykład 2.22.** Funkcję `SelectionSort` przedstawioną na listingu 2.1 można przekształcić do postaci funkcji rekurencyjnej `recSS`, jeśli wykorzystany algorytm sortowania zostanie wyrażony w następujący sposób. Zakładamy, że dane do posortowania są umieszczone w tablicy $A[0..n-1]$.

- (1) Wybieramy najmniejszy element z reszty tablicy A (czyli z $A[i..n-1]$).
- (2) Wymieniamy wybrany w poprzednim kroku element z elementem $A[i]$.
- (3) Sortujemy resztę tablicy, czyli $A[i+1..n-1]$.

Sortowanie przez wybieranie można wyrazić za pomocą następującego algorytmu rekurencyjnego.

PODSTAWA. Jeśli $i = n-1$, to pozostaje do posortowania jedynie ostatni element tablicy. Ponieważ pojedynczy element jest zawsze „posortowany”, nie trzeba podejmować dalej żadnych działań.

INDUKCJA. Jeśli $i < n-1$, to należy znaleźć najmniejszy element w tablicy $A[i..n-1]$, wymienić go z elementem $A[i]$ i rekurencyjnie posortować tablicę $A[i+1..n-1]$.

Kompletny algorytm realizujący powyższą rekurencję rozpoczyna się od wartości $i = 0$.

Jeśli i zostanie potraktowane jako parametr omówionej wcześniej indukcji, będzie to przypadek tzw. *indukcji wstecznej* (ang. *backward induction*), w której rozpoczyna się od dużej podstawy i — za pomocą reguły indukcyjnej — oblicza kolejno przypadki z mniejszymi wartościami parametru na podstawie przypadków z wartościami większymi. Jest to równie dobry sposób wykorzystania indukcji, chociaż nie wspomniano wcześniej o takiej możliwości. Można jednak taką formę indukcji postrzegać jako normalną indukcję dla parametru $k = n-i$, który reprezentuje liczbę elementów w czekającej na posortowanie reszcie tablicy.

Na listingu 2.7 przedstawiono kod funkcji `recSS(A, i, n)`. Drugi parametr, i , jest indeksem pierwszego elementu nieposortowanej reszty tablicy A . Trzeci parametr, n , jest łączną liczbą elementów tablicy A , które należy posortować. Przypuszczalnie, n jest mniejsze lub równe maksymalnemu rozmiarowi tablicy A . Wywołanie funkcji `recSS(A, 0, n)` spowoduje więc posortowanie całej tablicy $A[0..n-1]$.

LISTING 2.7.

Rekurencyjne sortowanie przez wybieranie

```
void recSS(int A[], int i, int n)
{
    int j, small, temp;
(1)    if (i < n-1) { /* przypadek podstawowy, dla i = n - 1, w którym */
        /* funkcja konczy dzialanie nie zmieniajac tablicy A */
        /* w przeciwnym przypadku wykonujemy indukcje */
(2)        small = i;
(3)        for (j = i+1; j < n; j++)
```

```

(4)         if (A[j] < A[small])
(5)             small = j;
(6)         temp = A[small];
(7)         A[small] = A[i];
(8)         A[i] = temp;
(9)         recSS(A, i+1, n);
    }
}

```

Na podstawie rysunku 2.15 można stwierdzić, że $s = n - i$ jest właściwym pojęciem „rozmiaru” argumentów w przypadku funkcji `recSS`. Podstawą jest $s = 1$, czyli sortowanie pojedynczego elementu — w takim przypadku nie stosuje się żadnych wywołań rekurencyjnych. Krok indukcyjny określa sposób sortowania s elementów przez wybór i wymianę najmniejszego elementu i dalsze sortowanie pozostałych $s-1$ elementów.

W wierszu (1) następuje sprawdzenie przypadku podstawowego, w którym do posortowania jest tylko jeden element (znowu zastosowano podejście defensywne, jeśli więc zdarzy się wywołanie dla $i \geq n$, nie spowoduje to wykonania nieskończonego ciągu wywołań). Dla przypadku podstawowego nie są wykonywane żadne działania i należy zakończyć działanie funkcji.

Pozostałą częścią funkcji jest realizacja przypadku indukcyjnego. Wiersze od (2) do (8) zostały bezpośrednio skopiowane z iteracyjnej wersji sortowania przez wybieranie. Podobnie jak w tamtym programie, w tej części programu zmiennej `small` zostaje przypisana wartość indeksu najmniejszego elementu tablicy `A[i..n-1]` i jego wymiana z elementem `A[i]`. Wreszcie, w wierszu (9) następuje wywołanie rekurencyjne, które sortuje pozostałą część tablicy. •

Ćwiczenia

2.7.1. Istnieje możliwość rekurencyjnego zdefiniowania n^2 w następujący sposób.

PODSTAWA. Dla $n = 1$, $1^2 = 1$.

INDUKCJA. Jeśli $n^2 = m$, to $(n+1)^2 = m+2n+1$.

- napisz rekurencyjną funkcję w języku C, implementującą powyższą definicję;
- wykaż za pomocą indukcji względem n , że powyższa definicja poprawnie oblicza n^2 .

2.7.2. Załóżmy, że mamy daną tablicę `A[0..4]`, która zawiera elementy 10, 13, 4, 7, 11 w przedstawionym porządku. Jaka będzie zawartość tablicy `A` bezpośrednio przed każdym rekurencyjnym wywołaniem funkcji `recSS`, którą przedstawiono na listingu 2.7?

2.7.3. Przypuśćmy, że definiujemy komórki listy jednokierunkowej zawierającej liczby całkowite (patrz podrozdział 1.3) za pomocą makra `DefCell(int, CELL, LIST)` z podrozdziału 1.6. Gwoli przypomnienia, wspomniane makro jest rozwijane do następującej definicji typu:

```

typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};

```

Dziel i zwyciężaj

Jednym ze sposobów rozwiązywania postawionego problemu jest próba podzielenia go na podproblemy, rozwiązania ich i połączenia otrzymanych wyników w celu otrzymania rozwiązania całego problemu. Określenie *dziel i zwyciężaj* jest stosowane do nazwania takiej właśnie techniki rozwiązywania problemów. Jeśli podproblemy są podobne do większego, oryginalnego problemu, może nawet istnieć możliwość wykorzystania tej samej funkcji do ich rekurencyjnego rozwiązywania.

Poprawne funkcjonowanie tej techniki wymaga spełnienia dwóch wymagań. Pierwsze z nich określa, że podproblemy muszą być prostsze od oryginalnego problemu. Drugie mówi o tym, że po skończonej liczbie podziałów musi być zapewnione dojście do podproblemu, który można łatwo rozwiązać. Gdy te kryteria nie są spełnione, algorytm rekurencyjny kontynuuje dzielenie problemu w nieskończoność, nie dochodząc do żadnego rozwiązania.

Należy zauważyć, że rekurencyjna funkcja `recSS` z listingu 2.7 spełnia oba założenia. Za każdym razem, gdy jest wywoływana, działa na podtablicy zawierającej o jeden mniejszą liczbę elementów i w momencie, gdy zostaje wywołana dla podtablicy zawierającej pojedynczy element, kończy działanie i nie kontynuuje wywołań rekurencyjnych. Podobnie, funkcja obliczająca silnię przedstawiona na listingu 2.6 opiera się na kolejnych wywołaniach rekurencyjnych z mniejszymi liczbami całkowitymi. W tym przypadku rekurencja kończy się w momencie, gdy argument wywołania osiąga wartość 1. W podrozdziale 2.8 zostanie omówione znacznie korzystniejsze zastosowanie techniki *dziel i zwyciężaj*, zwane sortowaniem przez scalanie. Rozmiar sortowanych tablic zmniejsza się bardzo gwałtownie, ponieważ sortowanie przez scalanie opiera się na dzieleniu (w każdym wywołaniu rekurencyjnym) rozmiaru przez 2, a nie odejmowania 1 od rozmiaru.

Napisz rekurencyjną funkcję `find`, która pobiera argument typu `LIST` i zwraca wartość `TRUE`, jeśli jedna z komórek listy zawiera liczbę całkowitą 1698 jako swój element, oraz `FALSE` w przeciwnym razie.

2.7.4. Napisz rekurencyjną funkcję `add`, która pobiera argument typu `LIST` (taki jak zdefiniowany w ćwiczeniu 2.7.3) i zwraca sumę wszystkich elementów danej listy.

2.7.5. Napisz taką wersję rekurencyjnej funkcji sortowania przez wybieranie, która pobiera jako argument listę liczb całkowitych wykorzystującą komórki zdefiniowane w ćwiczeniu 2.7.3.

2.7.6. W ćwiczeniu 2.2.8 zasugerowano, że można uogólnić mechanizm sortowania przez wybieranie w taki sposób, by wykorzystywać dowolny klucz (`key`) i funkcję `lt` do porównywania elementów. Napisz ponownie funkcję realizującą rekurencyjny algorytm sortowania przez wybieranie w wersji realizującej te ogólne założenia.

2.7.7*. Podaj rekurencyjny algorytm, który pobiera liczbę całkowitą i , po czym zwraca binarną reprezentację tej liczby w formie sekwencji zer i jedynek, począwszy od bitu najmniej znaczącego.

2.7.8*. *Największy wspólny dzielnik* (NWD; ang. *greatest common divisor*, GCD) dla dwóch liczb całkowitych i oraz j jest największą liczbą całkowitą, która bez reszty dzieli zarówno i , jak i j . Przykładowo, $nwd(24, 30) = 6$, zaś $nwd(24, 35) = 1$. Napisz rekurencyjną funkcję pobierającą dwie liczby całkowite i oraz j , gdzie $i > j$, i zwracającą $nwd(i, j)$. *Wskazówka:* możesz wykorzystać poniższą rekurencyjną definicję nwd . Zakłada się w niej, że $i > j$.

PODSTAWA. Jeśli j dzieli i bez reszty, to j jest NWD liczb i oraz j .

INDUKCJA. Jeśli j nie dzieli bez reszty i , niech k będzie resztą z dzielenia i przez j . $nwd(i, j)$ jest wówczas równe $nwd(j, k)$.

2.7.9.** Udowodnij, że podana w ćwiczeniu 2.7.8 rekurencyjna definicja NWD daje te same wyniki, co definicja nierekurencyjna (największa liczba całkowita, która bez reszty dzieli zarówno i , jak i j).

2.7.10. Definicje rekurencyjne można często niemal bezpośrednio zamieniać na algorytmy. Przykładowo można rozważyć podaną w przykładzie 2.16 definicję rekurencyjną relacji „mniejszości” dla ciągów. Napisz funkcję rekurencyjną sprawdzającą, czy pierwszy z dwóch danych ciągów jest „mniejszy” od drugiego. Załóż, że ciągi są reprezentowane przez jednokierunkowe listy znaków.

2.7.11*. Na podstawie podanej w ćwiczeniu 2.6.8 rekurencyjnej definicji posortowanej listy, opracuj rekurencyjny algorytm sortowania. Porównaj ten algorytm z rekurencyjnym sortowaniem przez wybieranie z przykładu 2.22.

2.8. Sortowanie przez scalanie: rekurencyjny algorytm sortujący

Poniżej zostanie rozważony kolejny algorytm sortujący, zwany *sortowaniem przez scalanie* (ang. *merge sort*), który znacznie różni się od sortowania przez wybieranie. Najlepszy opis sortowania przez scalanie opiera się na rekurencji i ilustruje jednocześnie bardzo korzystne zastosowanie techniki *dziel i zwyciężaj*, gdzie sortuje się listę $(a_1, a_2, \dots, a_n)$, dzieląc problem na dwa podobne problemy o dwukrotnie mniejszych rozmiarach. W ogólności, można by rozpocząć od podzielenia listy na dwie przypadkowo wskazane listy o równych rozmiarach, jednak w opracowywanym programie zostanie stworzona jedna lista elementów o indeksach nieparzystych, $(a_1, a_3, a_5, \dots)$ oraz drugą o indeksach parzystych, $(a_2, a_4, a_6, \dots)$. Następnie obie listy są sortowane osobno. Aby zakończyć proces sortowania oryginalnej listy n elementów, dwie posortowane połowy zostają scalone za pomocą algorytmu, któremu został poświęcony kolejny przykład.

W następnym rozdziale okaże się, że czas potrzebny do posortowania metodą sortowania przez scalanie rośnie znacznie wolniej w funkcji długości n sortowanej listy, niż ma to miejsce w przypadku sortowania przez wybieranie. Zatem, nawet jeśli wywołania rekurencyjne wymagają dodatkowych nakładów czasowych, sortowanie przez scalanie jest znacznie lepszym rozwiązaniem od sortowania przez wybieranie dla dużych wartości n . W rozdziale 3. zostanie przeanalizowana i porównana wydajność obu algorytmów sortujących.

Scalanie

Przez „scalanie” należy rozumieć utworzenie na podstawie dwóch posortowanych list jednej posortowanej listy, zawierającej wszystkie elementy dwóch danych list i żadne inne. Przykładowo, gdy istnieją listy $(1, 2, 7, 7, 9)$ i $(2, 4, 7, 8)$, ich scaleniem jest lista $(1, 2, 2, 4, 7, 7, 7, 8, 9)$. Warto zauważyć, że nie ma sensu rozważanie scalania list, które nie są jeszcze posortowane.

Prostym sposobem scalania dwóch list jest analiza od ich początków. W każdym kroku należy znaleźć mniejszy z dwóch elementów będących aktualnie na czele list, wybrać go jako kolejny element łączonej listy i usunąć z jego listy, wskazując nowy „pierwszy” element tej listy. W przypadku równych elementów rozpoczynających obie listy można tak naprawdę postępować w sposób dowolny, poniżej przyjęto jednak założenie, że do łączonej listy wstawiany jest element z pierwszej listy.

- **Przykład 2.23.** Rozważmy scalanie dwóch poniższych list:

$$L_1 = (1, 2, 7, 7, 9) \text{ oraz } L_2 = (2, 4, 7, 8)$$

Pierwsze elementy list to odpowiednio 1 i 2. Ponieważ element 1 jest liczbą mniejszą, zostaje wybrany jako pierwszy wstawiany do scalanej listy M i usunięty z listy L_1 . Nową listą L_1 jest więc $(2, 7, 7, 9)$. W tym momencie, pierwszym elementem list L_1 i L_2 jest 2. Można więc wybrać dowolny z nich. Zakładamy, że w przypadku takiej równości, zawsze wybierany będzie element z listy L_1 . Scalana lista M ma więc postać $(1, 2)$, lista L_1 $(7, 7, 9)$, zaś na liście L_2 pozostają wszystkie elementy $(2, 4, 7, 8)$. Tabela 2.1 pokazuje kolejne kroki scalania do momentu wyczerpania elementów list L_1 i L_2 . •

TABELA 2.1.
Przykład scalania

L_1	L_2	M
1, 2, 7, 7, 9	2, 4, 7, 8	pusta
2, 7, 7, 9	2, 4, 7, 8	1
7, 7, 9	2, 4, 7, 8	1, 2
7, 7, 9	4, 7, 8	1, 2, 2
7, 7, 9	7, 8	1, 2, 2, 4
7, 9	7, 8	1, 2, 2, 4, 7
9	7, 8	1, 2, 2, 4, 7, 7
9	8	1, 2, 2, 4, 7, 7, 7
9	pusta	1, 2, 2, 4, 7, 7, 7, 8
pusta	pusta	1, 2, 2, 4, 7, 7, 7, 8, 9

Jak się okaże, projektowanie rekurencyjnych algorytmów scalających jest łatwiejsze, jeśli listy jednokierunkowe są reprezentowane w formie przedstawionej w podrozdziale 1.3. Listy jednokierunkowe zostaną omówione bardziej szczegółowo w rozdziale 6. Poniżej zakłada się, że elementami listy są liczby całkowite. Każdy element może więc być reprezentowany jako „komórka”, czyli struktura typu `CELL`. Listę elementów reprezentuje typ `LIST`, który jest wskaźnikiem do struktury `CELL`. Wykorzystywane definicje dostarcza makro `DefCell(int, CELL, LIST)`, które omówiono w podrozdziale 1.6. Zastosowanie makra `DefCell` zostaje rozwinięte do postaci:

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};
```

Pole `element` w każdej komórce zawiera liczbę całkowitą, pole `next` zawiera natomiast wskaźnik do kolejnej komórki listy. Jeśli aktualny element jest ostatnim na liście, jego pole `next` zawiera wartość `NULL`, która reprezentuje wskaźnik pusty. Lista liczb całkowitych jest więc reprezentowana przez wskaźnik do pierwszej komórki na liście, czyli przez zmienną typu `LIST`. Lista pusta jest reprezentowana przez zmienną o wartości `NULL`, zamiast wskaźnika do pierwszego elementu.

Na listingu 2.8 przedstawiono implementację w języku C rekurencyjnego algorytmu scalającego. Funkcja `merge` pobiera jako argumenty dwie listy i zwraca jedną, scaloną listę. Parametry formalne `list1` i `list2` są wskaźnikami do dwóch danych list, natomiast zwracaną wartością jest wskaźnik do scalonej listy. Przedstawiony algorytm rekurencyjny można opisać za pomocą poniższej definicji rekurencyjnej.

LISTING 2.8.

Rekurencyjne scalanie

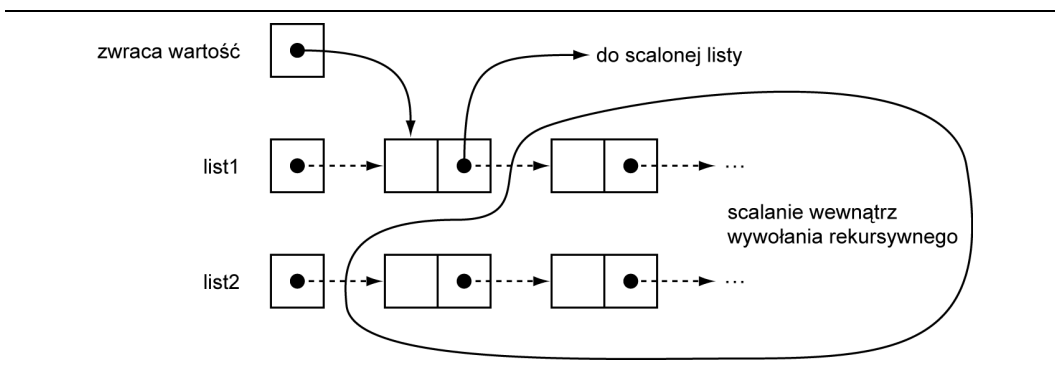
```

LIST merge(LIST list1, LIST list2)
{
(1)   if (list1 == NULL) return list2;
(2)   else if (list2 == NULL) return list1;
(3)   else if (list1->element <= list2->element) {
        /* Wiemy, że żadna z dwóch list wejściowych nie jest
           pusta, pierwsza lista ma mniejszy pierwszy element.
           Wynikiem jest pierwszy element pierwszej listy
           poprzedzający scalenie pozostałych elementów */
(4)   list1->next = merge(list1->next, list2);
(5)   return list1;
      }
      else { /* list2 ma mniejszy pierwszy element */
(6)   list2->next = merge(list1, list2->next);
(7)   return list2;
      }
}

```

PODSTAWA. Jeśli jedna z list wejściowych jest pusta, druga lista jest gotowym wynikiem. Tę regułę implementują wiersze (1) i (2) na listingu 2.8. Należy zauważyć, że jeśli obie listy są puste, zwracana jest lista `list2`. Jest to działanie poprawne, ponieważ wartością wskaźnika `list2` jest w takim przypadku `NULL`, a efektem scalania dwóch pustych list powinna być właśnie lista pusta.

INDUKCJA. Jeśli żadna z list wejściowych nie jest pusta, każda ma swój pierwszy element. Można więc się do nich odwoływać za pomocą konstrukcji `list1->element` oraz `list2->element`, czyli do pól element komórek wskazywanych odpowiednio przez zmienne `list1` i `list2`. Na rysunku 2.16 przedstawiono graficznie wykorzystywaną strukturę danych. Zwracana lista rozpoczyna się od komórki zawierającej najmniejszy element. Reszta tej listy jest tworzona przez scalanie wszystkich elementów z wyjątkiem wspomnianego.



RYSUNEK 2.16. Krok indukcyjny algorytmu scalającego

Przykładowo, w wierszach (4) i (5) zostaje obsługany przypadek, w którym pierwszy element listy `list1` jest najmniejszy. Wiersz (4) zawiera rekurencyjne wywołanie funkcji `merge`. Pierwszym argumentem tego wywołania jest `list1->next`, czyli wskaźnik do drugiego elementu pierwszej listy

(lub NULL, jeśli pierwsza lista zawiera tylko jeden element). Wywołanie rekurencyjne zostaje przekazane do listy składającej się z wszystkich elementów pierwszej listy poza pierwszym. Drugim argumentem jest cała druga lista. Efektem tych operacji jest zwrócenie przez wywołanie rekurencyjne z wiersza (4) wskaźnika do scalonej listy składającej się z wszystkich pozostałych elementów i zachowanie tego wskaźnika w polu next pierwszej komórki pierwszej listy. W wierszu (5) zostaje zwrócony wskaźnik do tej komórki, która jest w tym momencie pierwszą komórką scalonej listy zawierającej wszystkie elementy.

Rysunek 2.16 ilustruje opisane zmiany. Kropkowane strzałki reprezentują wywołania funkcji merge. Strzałki ciągle reprezentują wartości (wskaźniki) tworzone i zwracane przez wywołania funkcji merge. W szczególności, wartością zwracaną przez funkcję merge jest wskaźnik do komórki zawierającej najmniejszy element, pole next tej komórki wskazuje na listę zwróconą przez rekurencyjne wywołanie funkcji merge z wiersza (4).

Wiersze (6) i (7) obsługują przypadek, w którym to druga lista zawiera najmniejszy element. Zachowanie algorytmu jest takie samo, jak w wierszach (4) i (5), zamienione są jedynie role dwóch list wejściowych.

- **Przykład 2.24.** Załóżmy, że wywołano funkcję merge dla list (1, 2, 7, 7, 9) i (2, 4, 7, 8) z przykładu 2.23. Tabela 2.2 zawiera sekwencję kolejnych rekurencyjnych wywołań tej funkcji (czytając pierwszą kolumnę od góry). Pominęto przecinki oddzielające elementy list, ale rozdzielają one argumenty funkcji merge.

TABELA 2.2.
Rekurencyjne wywołania funkcji merge

Wywołanie	Zwracana wartość
merge(12779, 2478)	122477789
merge(2779, 2478)	22477789
merge(779, 2478)	2477789
merge(779, 478)	477789
merge(779, 78)	77789
merge(79, 78)	7789
merge(9, 78)	789
merge(9, 8)	89
merge(9, NULL)	9

Przykładowo, ponieważ pierwszy element pierwszej listy jest mniejszy od pierwszego elementu drugiej listy, w wierszu (4) listingu 2.8 następuje rekurencyjne wywołanie funkcji, powodując scale nie wszystkich elementów poza pierwszym z pierwszej listy. Oznacza to, że pierwszym argumentem tego wywołania jest reszta tej listy, czyli (2, 7, 7, 9); drugim jest pełna druga lista, czyli (2, 4, 7, 8). Pierwsze elementy na obu listach są teraz identyczne. Ponieważ test z wiersza (3) listingu 2.8 faworyzuje pierwszą listę, zostaje usunięty element 2 z tej listy i wywołana funkcja merge, tym razem z pierwszym argumentem (7, 7, 9) i drugim (2, 4, 7, 8).

Kolejno zwracane listy zawarto w drugiej kolumnie tabeli (czytając od dołu). Należy zauważyć, że w przeciwieństwie do iteracyjnego opisu scalania zasugerowanego w tabeli 2.1, algorytm rekurencyjny scala listy od końca, podczas gdy w wersji iteracyjnej scalanie listy odbywało się od początku. •

Dzielenie list

Innym ważnym zadaniem niezbędnym w procesie sortowania przez scalanie jest dzielenie list na dwie równe części lub dwie części, których długości różnią się o 1, jeśli oryginalna lista zawiera nieparzystą liczbę elementów. Jednym ze sposobów wykonania tego zadania jest zliczenie elementów listy, podzielenie otrzymanego wyniku przez dwa i rozdzielenie listy w jej środku. Zamiast tego, można wykorzystać prostą funkcję rekurencyjną `split`, która dzieli elementy między dwie listy, jedną złożoną z elementu pierwszego, trzeciego, piątego, itd.; drugą złożoną z elementów występujących na parzystych pozycjach na liście oryginalnej. Ściśle mówiąc, funkcja `split` usuwa elementy występujące na parzystych pozycjach na liście pobranej jako argument i zwraca nową listę złożoną właśnie z tych elementów.

Napisany w języku C kod funkcji `split` przedstawiono na listingu 2.9. Argumentem funkcji jest zdefiniowana wcześniej na potrzeby funkcji `merge` lista typu `LIST`. Należy zauważyć, że zmienna lokalna `pSecondCell` także została zdefiniowana jako lista typu `LIST`. W rzeczywistości zmienna `pSecondCell` jest wykorzystywana jako wskaźnik do drugiej komórki listy, nie do samej listy, choć typ `LIST` faktycznie jest właśnie wskaźnikiem do komórki.

LISTING 2.9.

Dzielenie listy na dwie równe części

```
LIST split(LIST list)
{
    LIST pSecondCell;

(1)    if (list == NULL) return NULL;
(2)    else if (list->next == NULL) return NULL;
        else { /* istnieją co najmniej dwie komórki */
(3)        pSecondCell = list->next;
(4)        list->next = pSecondCell->next;
(5)        pSecondCell->next = split(pSecondCell->next);
(6)        return pSecondCell;
    }
}
```

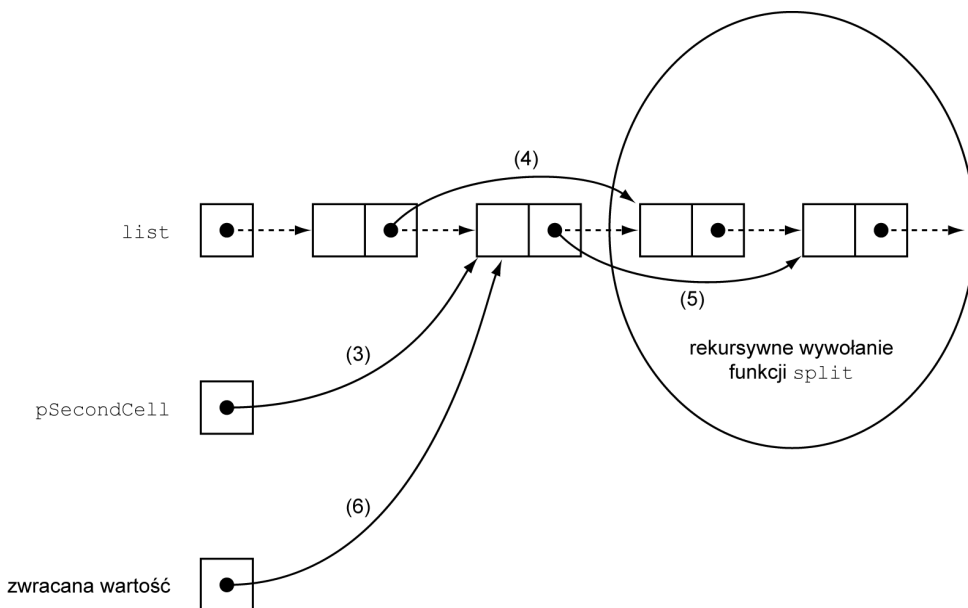
Istotną rzeczą jest zauważenie, że działanie funkcji `split` wiąże się z pewnym efektem ubocznym. Usuwa ona komórki z parzystych pozycji listy pobranej jako argument i dołącza je do nowej listy, która jest ostatecznie zwracaną wartością funkcji.

Algorytm podziału można, jak poniżej, opisać intuicyjnie. Wykorzystuje się tu indukcję względem długości listy; indukcja ta posiada wiele przypadków podstawowych.

PODSTAWA. Jeśli dana lista ma długość 0 lub 1, nie trzeba nic robić. Lista pusta jest już „podzielona” na dwie puste listy, zaś lista zawierająca pojedynczy element zostaje podzielona przez zostawienie tego elementu na danej liście i zwrócenie pustej listy elementów na pozycjach parzystych, których na danej liście nie ma. Przypadki podstawowe są obsługiwane w wierszach (1) i (2) listingu 2.9. Wiersz (1) obsługuje przypadek, w którym dana na wejściu lista jest pusta; wiersz (2) odpowiada za obsługę przypadku, w którym lista jest jednoelementowa. Warto zauważyć, że zachowując ostrożność w wierszu (2) nie sprawdza się wartości `list->next`, chyba że w wierszu (1) określono, że zmienna `list` nie jest równa `NULL`.

INDUKCJA. Krok indukcyjny zostaje zastosowany w sytuacji, gdy na liście wejściowej istnieją co najmniej dwa elementy. W wierszu (3) zostaje zachowany wskaźnik do drugiej komórki z listy w zmiennej lokalnej `pSecondCell`. W wierszu (4) polu `next` pierwszej komórki zostaje przypisana wartość wskaźnika do komórki trzeciej (z pominięciem drugiej) — oczywiście, jeśli lista zawiera tylko dwie komórki, wspomniane pole `next` przyjmuje wartość `NULL`. W wierszu (5) następuje rekurencyjne wywołanie funkcji `split` dla listy składającej się z wszystkich elementów oryginalnej listy z wyjątkiem dwóch pierwszych. Wartością zwracaną przez to wywołanie jest wskaźnik do czwartego elementu (lub `NULL`, jeśli lista jest krótsza niż czteroelementowa). Wskaźnikiem tym zastępuje się wartość pola `next` z drugiej komórki, uzupełniając w ten sposób łączenie elementów występujących na parzystych pozycjach. Wskaźnik do drugiej komórki zwracany jest przez funkcję `split` w wierszu (6); wskaźnik ten umożliwia uzyskanie dostępu do jednokierunkowej listy złożonej z wszystkich elementów położonych na parzystych pozycjach na oryginalnej liście.

Zmiany wprowadzane przez funkcję `split` zilustrowano na rysunku 2.17. Kropkowane strzałki reprezentują oryginalne wskaźniki; nowe wskaźniki oznaczono normalnymi strzałkami. Oznaczono także numery wierszy kodu, w których są tworzone poszczególne wskaźniki.



RYSUNEK 2.17. Działanie funkcji `split`

Algorytm sortujący

Rekurencyjny algorytm sortujący przedstawiono na listingu 2.10. Zaprezentowany algorytm można opisać za pomocą odpowiedniej podstawy i kroku indukcyjnego.

PODSTAWA. Jeśli lista do posortowania jest pusta lub jednoelementowa, zostaje zwrócona sama lista — jest ona już posortowana. Za obsługę przypadków podstawowych odpowiadają instrukcje w wierszach (1) i (2) listingu 2.10.

LISTING 2.10.

Algorytm sortowania przez scalanie

```

LIST MergeSort(LIST list)
{
    LIST SecondList;

(1)    if (list == NULL) return NULL;
(2)    else if (list->next == NULL) return list;
        else {
            /* co najmniej dwa elementy na liście */
(3)    SecondList = split(list);
            /* warto zauważyć, że efektem ubocznym jest usunięcie
               połowy elementów z oryginalnej listy wejściowej */
(4)    return merge(MergeSort(list), MergeSort(SecondList));
        }
}

```

INDUKCJA. Jeśli lista ma długość nie mniejszą niż 2, w wierszu (3) zostaje wykorzystana funkcja `split` do usunięcia elementów na parzystych pozycjach listy wejściowej, służących do utworzenia nowej listy, wskazywanej przez zmienną lokalną `SecondList`. W wierszu (4) następuje rekurencyjne posortowanie obu połówek listy i zwrócenie scalonych rezultatów tych operacji.

- **Przykład 2.25.** Poniżej zostanie posortowana przy użyciu metody sortowania przez scalanie następująca lista złożona z liczb jednocyfrowych:

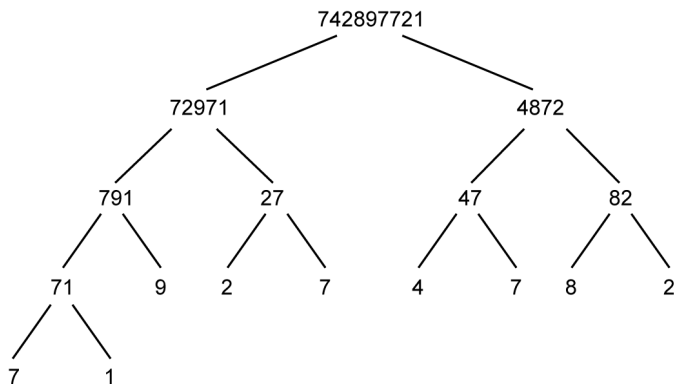
742897721

Dla zwięzłości zapisu ponownie pominięto przecinki oddzielające cyfry na liście. Najpierw lista zostaje podzielona na dwie części za pomocą funkcji `split` wywoływanej w wierszu (3) funkcji `MergeSort`. Jedną z powstałych w ten sposób list zawiera elementy z pozycji nieparzystych; druga składa się z cyfr leżących na pozycjach parzystych (`list = 72971`, `SecondList = 4872`). Listy zostają posortowane w wierszu (4), co w efekcie daje listy 12779 i 2478, następnie scalane w celu otrzymania ostatecznego wyniku: 122477789.

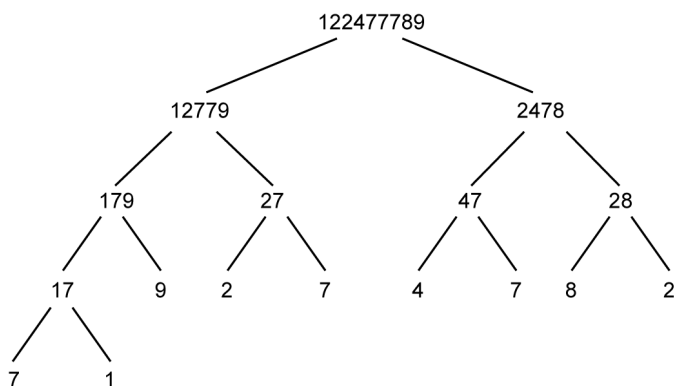
Wspomniane sortowanie dwóch list nie przebiega oczywiście w jakiś magiczny sposób, a raczej polega na konsekwentnym stosowaniu algorytmu rekurencyjnego. Początkowo funkcja `MergeSort` dzieli listę, dla której została wywołana, jeśli zawiera ona więcej niż 1 element. Rysunek 2.18(a) ilustruje mechanizm rekurencyjnego dzielenia list do momentu, w którym wszystkie osiągną długość 1. Następnie listy są parami scalane, od liści do korzenia drzewa, do czasu posortowania całej listy. Ten proces przedstawiono na rysunku 2.18(b). Warto zauważyć, że podziały i scalania nie są wykonywane w tej ustalonej kolejności — nie wszystkie scalania następują po wszystkich podziałach. Przykładowo, pierwsza połowa listy, 72971, zostaje całkowicie podzielona i scalona zanim rozpocznie się przetwarzanie drugiej połowy listy, 4872. •

Kompletny program

Listing 2.11 zawiera kompletny program realizujący sortowanie przez scalanie. Jest on analogiczny do programu z listingu 2.2, który opierał się na mechanizmie sortowania przez wybieranie. Funkcja



(a) Dzielenie



(b) Scalanie

RYSUNEK 2.18.
Rekurencyjne
dzielenie i scalanie

MakeList w wierszu (1) odczytuje liczby całkowite z wejścia programu i umieszcza je na liście jednokierunkowej za pomocą prostego algorytmu rekurencyjnego, który zostanie szczegółowo opisany w kolejnym podrozdziale. W wierszu (2) w funkcji main umieszczono wywołanie funkcji MergeSort, która zwraca posortowaną listę do funkcji PrintList. Ta ostatnia przechodzi przez całą listę i wyświetla każdy z jej elementów.

LISTING 2.11.

Program sortujący listę za pomocą algorytmu sortowania przez scalanie

```

#include <stdio.h>
#include <stdlib.h>

typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};

LIST merge(LIST list1, LIST list2);

```

```
LIST split(LIST list);
LIST MergeSort(LIST list);
LIST MakeList();
void PrintList(LIST list);

main()
{
    LIST list;

(1)    list = MakeList();
(2)    PrintList(MergeSort(list));
}

LIST MakeList()
{
    int x;
    LIST pNewCell;

(3)    if (scanf("%d", &x) == EOF) return NULL;
    else {
(4)        pNewCell = (LIST) malloc(sizeof(struct CELL));
(5)        pNewCell->next = MakeList();
(6)        pNewCell->element = x;
(7)        return pNewCell;
    }
}

void PrintList(LIST list)
{
(8)    while (list != NULL) {
(9)        printf("%d\n", list->element);
(10)       list = list->next;
    }
}

LIST MergeSort(LIST list)
{
    LIST SecondList;

    if (list == NULL) return NULL;
    else if (list->next == NULL) return list;
    else {
        SecondList = split(list);
        return merge(MergeSort(list), MergeSort(SecondList));
    }
}

LIST merge(LIST list1, LIST list2)
{
    if (list1 == NULL) return list2;
    else if (list2 == NULL) return list1;
    else if (list1->element <= list2->element) {
```

```

    list1->next = merge(list1->next, list2);
    return list1;
}
else {
    list2->next = merge(list1, list2->next);
    return list2;
}

LIST split(LIST list)
{
    LIST pSecondCell;

    if (list == NULL) return NULL;
    else if (list->next == NULL) return NULL;
    else {
        pSecondCell = list->next;
        list->next = pSecondCell->next;
        pSecondCell->next = split(pSecondCell->next);
        return pSecondCell;
    }
}

```

Ćwiczenia

2.8.1. Przedstaw wynik zastosowania funkcji `merge` dla list (1, 2, 3, 4, 5) oraz (2, 4, 6, 8, 10).

2.8.2. Załóżmy, że rozpoczynamy działania od listy (8, 7, 6, 5, 4, 3, 2, 1). Przedstaw sekwencję wywołań funkcji `merge`, `split` i `MergeSort`.

2.8.3*. Sortowanie przez *wielodrożne* (ang. *multiway*) scalanie polega na podzieleniu listy na k części o równych (lub równych w pewnym przybliżeniu) rozmiarach, ich rekurencyjnym posortowaniu i scaleniu wszystkich k list poprzez porównywanie ich pierwszych elementów i wybieranie najmniejszego. Opisane w niniejszym podrozdziale sortowanie przez scalanie dotyczy przypadku $k = 2$. Zmodyfikuj program z listingu 2.11 tak, by realizował sortowanie przez wielodrożne scalanie dla przypadku $k = 3$.

2.8.4*. Napisz ponownie program sortujący przez scalanie w taki sposób, by wykorzystywał funkcje `lt` i `key` (patrz ćwiczenie 2.2.8) do porównywania elementów dowolnych typów.

2.8.5. Spróbuj powiązać funkcję (a) `merge`, (b) `split` oraz (c) `MakeList` ze schematem z rysunku 2.15. Jakie jest właściwe pojęcie rozmiaru dla każdej z tych funkcji?

2.9. Dowodzenie własności programów rekurencyjnych

Jeśli należy dowieść prawdziwości pewnej własności funkcji rekurencyjnej, ogólnie rzecz biorąc trzeba wykazać prawdziwość twierdzenia opisującego efekt pojedynczego wywołania tej funkcji. Przykładowo, takim efektem może być związek pomiędzy argumentem i zwracaną wartością, np.

„funkcja wywołana dla argumentu i zwraca wartość $i!$ ”. Często definiuje się także pojęcie „rozmiaru” argumentów funkcji i przeprowadza odpowiedni dowód za pomocą indukcji względem tego rozmiaru. Oto niektóre z wielu możliwych sposobów definiowania rozmiarów argumentów:

- (1) Wartość jednego z argumentów. Przykładowo, dla rekurencyjnego programu liczącego silnie z listingu 2.6 odpowiednim rozmiarem jest wartość argumentu n .
- (2) Długość listy wskazywanej przez jeden z argumentów. Rekurencyjna funkcja `split` z listingu 2.9 jest przykładem, w którym długość listy jest właściwym rozmiarem.
- (3) Pewna funkcja argumentów. Przykładowo, wspomniano, że rekurencyjny algorytm sortowania przez wybieranie z wydruku 2.7 wykonuje indukcję względem liczby elementów w pozostającej do posortowania tablicy. Korzystając z wartości argumentów n oraz i funkcję tę można zapisać jako $n-i$. Innym przykładem jest suma długości list wskazywanych przez dwa argumenty funkcji `merge` z listingu 2.8.

Niezależnie od wybranej definicji pojęcia rozmiaru, istotne jest, by funkcja wywoływana z argumentem o rozmiarze s zawierała wyłącznie wywołania z argumentami o rozmiarze $s-1$ lub mniejszym. Spełnienie tego wymagania umożliwia wykorzystanie indukcji względem rozmiaru w dowodzie własności programu. Co więcej, w momencie gdy rozmiar spada do pewnej stałej wartości (np. 0), funkcja nie może wykonywać dalszych wywołań rekurencyjnych. Spełnienie tego wymagania umożliwia rozpoczęcie indukcji od przypadku podstawowego.

- **Przykład 2.26.** Poniżej rozważono przedstawiony na listingu 2.6 w podrozdziale 2.7 program obliczający silnie. Twierdzenie dowodzone za pomocą indukcji względem i , dla $i \geq 1$, ma następującą postać.

TWIERDZENIE $S(i)$: jeśli funkcja `fact` jest wywoływana z wartością i argumentu n , zwraca wartość $i!$.

PODSTAWA. Dla $i = 1$, warunek w wierszu (1) listingu 2.6 powoduje obsługę przypadku podstawowego w wierszu (2). Efektem jest zwrócenie wyniku o wartości 1, czyli $1!$.

INDUKCJA. Załóżmy, że $S(i)$ jest prawdziwe, co oznacza, że wywołanie funkcji `fact` z pewnym argumentem $i \geq 1$ powoduje zwrócenie wartości $i!$. Rozważmy teraz, co się stanie, jeśli funkcja `fact` zostanie wywołana z wartością $i+1$ przekazaną za pomocą zmiennej n . Jeśli $i \geq 1$, wyrażenie $i+1$ ma wartość nie mniejszą niż 2, zatem zastosowanie ma przypadek indukcyjny obsługiwany w wierszu (3). Zwracana wartość wynosi więc $n \times \text{fact}(n-1)$, czyli (ponieważ zmienna n ma wartość $i+1$) zwracany jest wynik $(i+1) \times \text{fact}(i)$. Zgodnie z hipotezą indukcyjną, $\text{fact}(i)$ zwraca $i!$. Ponieważ $(i+1) \times i! = (i+1)!$, stanowi to wykazanie poprawności kroku indukcyjnego określającego, że funkcja `fact` dla argumentu $i+1$ zwraca wartość $(i+1)!$. •

- **Przykład 2.27.** Poniżej zostanie przeanalizowana funkcja `MakeList`, będąca jedną z funkcji pomocniczych na listingu 2.11 zaprezentowanym w podrozdziale 2.8. Funkcja ta tworzy listę jednokierunkową złożoną z elementów wejściowych i zwracającą wskaźnik do tej listy. Zostanie udowodnione poniższe twierdzenie za pomocą indukcji względem $n \geq 0$, gdzie n jest liczbą elementów w sekwencji wejściowej.

TWIERDZENIE $S(n)$: jeśli $x_1, x_2, \dots, x_n$ jest sekwencją elementów wejściowych, funkcja `MakeList` tworzy listę jednokierunkową zawierającą elementy $x_1, x_2, \dots, x_n$ i zwracającą wskaźnik do tej listy.

PODSTAWA. Podstawą jest przypadek, w którym $n = 0$, czyli pusta sekwencja wejściowa. Sprawdzany w wierszu (3) funkcji `MakeList` warunek końca pliku (EOF) powoduje, że zwracaną wartością jest w takim przypadku `NULL`. Wynikiem funkcji `MakeList` jest więc poprawna lista pusta.

INDUKCJA. Załóżmy, że $S(n)$ jest prawdziwe dla $n \geq 0$ i rozważmy, co stanie się w momencie, gdy funkcja `MakeList` zostanie wywołana dla wejściowej sekwencji $n+1$ elementów. Przypuśćmy także, że odczytano właśnie pierwszy element x_1 .

W wierszu (4) funkcji `MakeList` tworzony jest wskaźnik do nowej komórki c . W wierszu (5) następuje rekurencyjne wywołanie funkcji `MakeList` w celu utworzenia, na podstawie hipotezy indukcyjnej, wskaźnika do jednokierunkowej listy pozostałych $n-1$ elementów, $x_2, x_3, \dots, x_n$. Nowy wskaźnik zostaje w wierszu (5) umieszczony w polu `next` komórki c . Następnie, w wierszu (6) wartość x_1 zostaje umieszczona w polu `element` komórki c . W ostatnim wierszu (7) funkcji zostaje zwrócony wskaźnik utworzony w wierszu (4), który wskazuje na listę jednokierunkową zawierającą n elementów, $x_1, x_2, \dots, x_n$.

Udowodniono właśnie krok indukcyjny i można wywnioskować, że funkcja `MakeList` działa poprawnie dla wszystkich danych wejściowych. •

- **Przykład 2.28.** W ostatnim przykładzie zostanie wykazana poprawność programu sortującego przez scalanie z listingu 2.10. Zakłada się, że funkcje `split` i `merge` wykonują swoje zadania poprawnie. Zostanie wykorzystana indukcja względem długości listy danej jako argument funkcji `MergeSort`. Będzie to indukcja zupełna względem $n \geq 0$, służąca do dowiedzenia prawdziwości twierdzenia:

TWIERDZENIE $S(n)$: jeśli w momencie wywołania funkcji `MergeSort`, `list` jest listą długości n , funkcja zwraca posortowaną listę złożoną z tych samych elementów.

PODSTAWA. Podstawą jest zarówno $S(0)$, jak i $S(1)$. Jeśli lista `list` ma długość 0, jej wartością jest `NULL`, warunek z wiersza (1) listingu 2.10 jest więc prawdziwy i cała funkcja zwraca wartość `NULL`. Jeśli `list` ma długość 1, spełniony jest warunek z wiersza (2) i funkcja zwraca `list`. Funkcja `MergeSort` zwraca daną listę `list`, jeśli n jest równe 0 lub 1. To spostrzeżenie potwierdza prawdziwość twierdzeń $S(0)$ i $S(1)$, ponieważ listy o długości 0 i 1 są już posortowane.

INDUKCJA. Załóżmy, że $n \geq 1$ i twierdzenie $S(i)$ jest prawdziwe dla wszystkich $i = 0, 1, \dots, n$. Należy dowieść prawdziwości twierdzenia $S(n+1)$, zatem trzeba rozważyć listę o długości $n+1$. Ponieważ $n \geq 1$, lista ma długość nie mniejszą niż 2, dochodzi się więc do wiersza (3) listingu 2.10. Funkcja `split` dzieli tam daną listę na dwie listy o długości $(n+1)/2$, jeśli $n+1$ jest parzyste, lub $n/2+1$ i $n/2$, jeśli $n+1$ jest nieparzyste. Ponieważ $n \geq 1$, żadna z tych list nie może mieć długości dochodzącej do $n+1$. Oznacza to, że dla obu przypadków ma zastosowanie hipoteza indukcyjna i można wywnioskować, że otrzymane połowy list są poprawnie posortowane za pomocą rekurencyjnych wywołań funkcji `MergeSort` z wiersza (4). Na końcu posortowane listy zostają scalone w jedną listę, która jest zwracana jako wynik funkcji. Założono, że funkcja `merge` działa poprawnie, zatem zwracana lista wynikowa jest posortowana. •

Ćwiczenia

2.9.1. Udowodnij, że funkcja `PrintList` z listingu 2.11 wyświetla elementy znajdujące się na liście przekazanej jako argument. Jakiego twierdzenia $S(i)$ należy dowieść indukcyjnie? Jaka jest wartość podstawowa dla i ?

2.9.2. Funkcja `sum` z listingu 2.12 oblicza sumę elementów znajdujących się na danej liście (której komórki należą do znanego nam typu zdefiniowanego za pomocą makra `DefCell` w podrozdziale 1.6 i wykorzystanego w programie sortowania przez scalanie w podrozdziale 2.8), dodając pierwszy element do sumy pozostałych elementów, która jest obliczana za pomocą wywołania rekurencyjnego dla reszty listy. Udowodnij, że funkcja `sum` poprawnie oblicza sumę elementów. Jakiego twierdzenia $S(i)$ należy dowieść indukcyjnie? Jaka jest wartość podstawowa dla i ?

LISTING 2.12.

Dwie funkcje rekursywne, `sum` i `find0`

```
DefCell(int, CELL, LIST);

int sum(LIST L)
{
    if (L == NULL) return 0;
    else return L->element+sum(L->next);
}

int find0(LIST L)
{
    if (L == NULL) return FALSE;
    else if (L->element == 0) return TRUE;
    else return find0(L->next);
}
```

2.9.3. Funkcja `find0` z listingu 2.12 zwraca `TRUE`, jeśli co najmniej jeden z elementów danej listy to 0; w przeciwnym razie, funkcja zwraca `FALSE`. Funkcja zwraca `FALSE`, jeśli lista jest pusta; `TRUE`, jeśli pierwszym elementem jest 0; jeśli żaden z tych warunków nie jest spełniony, wykonywane jest wywołanie rekurencyjne dla reszty listy i zwracana jest odpowiedź otrzymana dla tej reszty. Udowodnij, że funkcja `find0` poprawnie określa, czy dana lista zawiera 0. Jakiego twierdzenia $S(i)$ należy dowieść indukcyjnie? Jaka jest wartość podstawowa dla i ?

2.9.4*. Udowodnij, że funkcja `merge` z listingu 2.8 i funkcja `split` z listingu 2.9 wykonują zadania założone w podrozdziale 2.8.

2.9.5. Podaj intuicyjny dowód poprawności oparty na „najmniejszym kontrprzykładzie” dla indukcji rozpoczynającej się od dwóch przypadków podstawowych 0 i 1.

2.9.6.** Udowodnij poprawność rekurencyjnego algorytmu NWD (w implementacji Czytelnika w języku C) z ćwiczenia 2.7.8.

2.10. Podsumowanie rozdziału 2.

Oto najważniejsze zagadnienia, które Czytelnik powinien zapamiętać po zapoznaniu się z niniejszym rozdziałem:

- ścisłe powiązanie pojęć dowodów indukcyjnych, definicji rekurencyjnych oraz programów rekurencyjnych. Każde opiera się na podstawie i „działaniu” kroku indukcyjnego;

- w „zwykłych”, „słabych” indukcjach kolejne kroki zależą wyłącznie od kroków poprzednich. Często zachodzi konieczność przeprowadzania dowodów za pomocą indukcji zupełnej, w której każdy krok może zależeć od wszystkich wcześniejszych;
- istnieje wiele różnych sposobów sortowania. Sortowanie przez wybieranie jest łatwym, ale też wolnym algorytmem sortującym; sortowanie przez scalanie jest algorytmem szybszym, ale i bardziej skomplikowanym;
- indukcja ma zasadnicze znaczenie w dowodzeniu poprawności działania programów lub ich fragmentów;
- dziel i zwyciężaj jest przydatną techniką projektowania dobrych algorytmów, takich jak sortowanie przez scalanie. Opiera się na dzieleniu problemu na niezależne podproblemy i scalaniu otrzymanych wyników;
- wyrażenia definiuje się zgodnie z ich naturą, w sposób rekurencyjny, opierając się na zawartych w nich operandach i operatorach. Operatory można klasyfikować na podstawie liczby argumentów, na których działają: jednoargumentowe, dwuargumentowe oraz k -argumentowe). Istnieje także dodatkowy podział operatorów dwuargumentowych oparty na ich umiejscowieniu: operatory wrostkowe leżą między operandami, przedrostkowe poprzedzają operandy oraz przyrostkowe, które są położone za operandami, na których działają.

2.11. Bibliografia rozdziału 2.

Doskonałej analizy rekurencji dokonał Roberts [1986]. Szczegółowy opis algorytmów sortujących można znaleźć w wykorzystywanym powszechnie podręczniku Knutha [2003]. Berlekamp [1968] poświęcił swoją publikację technikom detekcji (przedstawiony w podrozdziale 2.3 schemat detekcji błędów jest wśród nich najprostszy) i korekcji błędów w strumieniach bitów.

Berlekamp E. R., *Algebraic Coding Theory*, McGraw-Hill, Nowy Jork, 1968.

Knuth D. E., *Sztuka programowania*, Tom III: *Sortowanie i przeszukiwanie*, WNT, Warszawa, 2003.

Roberts E., *Thinking Recursively*, Wiley, Nowy Jork, 1986.