Wprowadzenie do algorytmów

Rozwiązania zadań i problemów

Krzysztof Wojtas

wersja 0.4.99
zawiera rozwiązania z części I, II, III i VIII

21 czerwca 2017

Najnowsza wersja tego opracowania znajduje się pod adresem:
`https://github.com/wojtask/CormenSol`

Wstęp

Niniejsze opracowanie zawiera rozwiązania zadań i problemów z drugiego wydania monografii *Introduction to Algorithms* [3] autorstwa Thomasa H. Cormena, Charlesa E. Leisersona, Ronalda L. Rivesta i Clifforda Steina. Przy opracowywaniu rozwiązań korzystałem równolegle z polskiego tłumaczenia pt. *Wprowadzenie do algorytmów* [4] w wydaniu szóstym. Na chwilę obecną opracowanie pokrywa rozdziały należące do części pierwszej (rozdz. 1–5), drugiej (rozdz. 6–9) i trzeciej (rozdz. 10–14) oraz dodatki wypełniające część ósmą książki. Kolejne wersje będą sukcesywnie uzupełniane o rozwiązania zadań z kolejnych części, wprowadzając jednocześnie poprawki znalezionych błędów i ewentualnie doskonaląc poprzednie rozwiązania.

Moim celem było stworzenie kompletnego zbioru rozwiązań, który pomaga w przyswajaniu materiału z *Wprowadzenia do algorytmów* (określanego dalej jako „Podręcznik"), będąc czymś w rodzaju wzorcowego „klucza odpowiedzi", z którym student porównuje uzyskany przez siebie rezultat. Oczywiście gorąco zachęcam do uprzedniego zmierzenia się z zadaniami samodzielnie, zamiast natychmiastowego zaglądania do odpowiedzi – z pewnością więcej można się w ten sposób nauczyć, a samo rozwiązywanie problemów może dostarczyć mnóstwa satysfakcji. Wyznaczony przeze mnie cel rzetelności przedstawionych tu treści zaowocował zastosowaniem języka formalnego i w wielu miejscach niewątpliwie trudnego, jednak należy mieć na uwadze to, że dokładność i precyzja powinny być nieodłącznymi cechami tekstów ścisłych.

Zwracam uwagę na to, że sami Autorowie Podręcznika dostarczają rozwiązania niektórych zadań i problemów – można je pobrać ze strony WWW książki: `http://mitpress.mit.edu/algorithms`. Stanowią one jednak niewielki ułamek wszystkich rozwiązań – niemal połowa rozdziałów Podręcznika jest w całości pominięta, a pozostałe są opracowane tylko częściowo. Ponadto wiele rozwiązań jest napisana zbyt drobiazgowo. Niektóre posłużyły mi jako podstawa dla moich własnych rozwiązań, zawsze jednak dążyłem do przeformułowania ich treści do bardziej skondensowanych (i niejednokrotnie bardziej precyzyjnych) form.

Wspomnę teraz o kilku kwestiach technicznych i zasadach, którymi kierowałem się podczas opracowywania rozwiązań. Dokument został utworzony za pomocą systemu LaTeX 2_ε, który pozwala na precyzyjną i estetyczną prezentację nie tylko tekstu, ale także formuł matematycznych, tabel i pseudokodów. Do złożenia tych ostatnich użyłem pakietu `clrscode` opracowanego przez Thomasa H. Cormena. Z pakietu tego korzystano również w oryginalnym tekście Podręcznika, a więc styl pseudokodów w rozwiązaniach jest identyczny jak w książce. Rozumowania w wielu miejscach ilustrują rysunki, przy tworzeniu których wykorzystywałem języki PGF/TikZ [10]. Dzięki zastosowaniu płaskiej numeracji mogę równocześnie i jednoznacznie odnosić się do rysunków i tabel zarówno z rozwiązań, jak i z Podręcznika, w którym to numeracja jest dwupoziomowa.

Niektóre konwencje notacji i nomenklatury pojęć matematycznych odbiegają nieco względem tych z Podręcznika. Głównym tego powodem były kwestie ujednolicenia zapisu, a także sprowadzenie ich do postaci częściej spotykanych w polskiej literaturze. Dla przykładu wszystkie ciągi, krotki i tablice obejmuję nawiasami trójkątnymi, a jako synonimu pojęcia „funkcja monotonicznie rosnąca" zdefiniowanego w Podręczniku używam pojęcia „funkcja niemalejąca". Notację $[a\,..\,b]$ stosuję do oznaczania zakresu liczb całkowitych od a do b (włącznie), natomiast znane z literatury matematycznej notacje (a, b), $(a, b]$, $[a, b)$, $[a, b]$ – dla zakresów (przedziałów) liczb rzeczywistych. Z kolei często wykorzystywanym przeze mnie skrótem myślowym jest zapis „indeksy $i < j$" w znaczeniu „indeksy i, j, gdzie $i < j$".

Rozwiązania często powołują się na fakty przedstawione w Podręczniku, wymagana jest zatem znajomość całego materiału przynajmniej z bieżącego rozdziału. W wielu tekstach rozwiązań

znajdziemy także odnośniki do innych zadań, szczególnie wówczas, gdy dane zadanie korzysta z wyniku innego we własnym rozwiązaniu. Na ogół wykorzystywane są rozwiązania zadań występujących w tekście wcześniej względem danego, choć nie jest to zasadą. W początkowych rozdziałach można zatem zaobserwować nieco większą koncentrację na szczegółach, a w późniejszych – więcej odsyłaczy do zadań, w których szczegóły te zostały już omówione.

O każdym znalezionym błędzie lub nieścisłości w treściach zadań zwracam uwagę w krótkich notkach przed rozwiązaniem danego zadania. Bazuję przede wszystkim na polskim tłumaczeniu, ale wskazuję, czy błąd występuje także w oryginale. Uwzględniam jednakże erratę do oryginału: `http://www.cs.dartmouth.edu/~thc/clrs-2e-bugs/bugs.php` – jeśli znajduje się w niej wpis o pewnej poprawce, to przyjmuję, że błąd został naprawiony i wspominam o jego istnieniu tylko wówczas, gdy występuje w tłumaczeniu.

Obecna wersja tekstu (oznaczona jako 0.5) w porównaniu z poprzednią (0.4), oprócz rozwiązań z części III, zawiera też pewną liczbę poprawek do zadań z części I, II i VIII. Poprawione zostały przede wszystkim znalezione błędy logiczne, a niektóre paragrafy zostały przeredagowane. Zmiany zaszły także w warstwie prezentacyjnej, między innymi dzięki wciągnięciu numerów zadań do pierwszych paragrafów rozwiązań, co zmniejszyło nieestetyczne puste odstępy między rozwiązaniami.

W ostatnim czasie zdecydowałem się na implementację algorytmów i struktur danych z Podręcznika i rozwiązań w rzeczywistym języku programowania i przeprowadzenia testów ich poprawności. W wyniku tego przedsięwzięcia powstał projekt w języku Java dostępny na platformie GitHub: `https://github.com/wojtask/CormenImpl`. Nie zalecam jednak wykorzystywania tej implementacji jako biblioteki algorytmów do prawdziwego systemu, ponieważ głównym zadaniem projektu było jak najwierniejsze odwzorowanie pseudokodów w języku programowania i przetestowanie powstałego kodu. W wyniku przeprowadzenia testów udało mi się znaleźć wiele błędów i innych niedoskonałości w pseudokodach z rozwiązań, które zostały oczywiście poprawione w obecnej wersji tekstu.

Opracowanie powstaje od 2009 roku w bardzo powolnym tempie, głównie z powodu mojego perfekcjonistycznego podejścia w ich opracowywaniu. Tymczasem w tym samym roku ukazało się kolejne, trzecie wydanie Podręcznika [5] zawierające nowy materiał, jak i wiele nowych zadań i problemów. Mam nadzieję na przyspieszenie prac nad wydaniem drugim i jak najszybszego przystąpienia do opracowywania rozwiązań dla wydania trzeciego. Innym planowanym projektem jest przetłumaczenie całego tekstu na język angielski, aby dotrzeć do szerszego grona odbiorców.

Dołożyłem wszelkich starań, aby każde rozwiązanie zostało dokładnie sprawdzone. Jeśli jednak znalazłeś błąd merytoryczny lub typograficzny, bądź twierdzisz, że znasz znacznie krótsze lub prostsze rozwiązanie jakiegoś zadania lub problemu, powiadom mnie o tym niezwłocznie, pisząc na adres `kwojtas@gmail.com`. Jeśli sugestia okaże się trafna, Twoje nazwisko pojawi się w podziękowaniach, a kolejne wersje tego opracowania będą dzięki Tobie bliższe ideału.

Kraków, czerwiec 2017 — Krzysztof Wojtas

Spis treści

Część I

Podstawy

Rozdział **1**

Rola algorytmów w obliczeniach

1.1. Algorytmy

1.1-1. Przykłady występowania poszczególnych problemów:

Sortowanie: Jest to problem bardzo powszechny – znajduje zastosowanie w wielu zagadnieniach obliczeniowych. Najczęstszym powodem sortowania danych jest przygotowanie ich do dalszego przetwarzania, które wówczas jest na ogół bardziej efektywne.

Najlepsza kolejność mnożenia macierzy: Problem występuje podczas wyznaczania transformacji graficznych (np. skalowań, obrotów); przekształcenia te opisane są za pomocą macierzy, a ich składanie oznacza obliczanie iloczynu tych macierzy.

Otoczka wypukła: Mając zbiór wbitych w ziemię słupków, chcemy otoczyć pewien obszar siatką ogrodzeniową opierając ją na niektórych słupkach tak, by obszar ten zmaksymalizować.

1.1-2. Miary efektywności algorytmu inne niż jego szybkość:

- zużycie pamięci (operacyjnej i masowej);
- stopień wykorzystania systemu operacyjnego;
- efektywność dostępu do bazy danych;
- stopień wykorzystania połączenia sieciowego;
- dostosowanie do konkretnej architektury sprzętowo-programowej;
- efektywność działania w architekturze równoległej lub rozproszonej.

1.1-3. Poniżej zestawiono zalety i wady listy dwukierunkowej w porównaniu ze zwykłą tablicą.

Zalety:

- przy definiowaniu listy nie trzeba z góry znać jej maksymalnej pojemności, jak to jest w przypadku definiowania tablicy;
- lista jest elastyczna – może się dowolnie powiększać i kurczyć w trakcie wykonywania na niej operacji wstawiania i usuwania;
- wstawianie i usuwanie elementów z dowolnej pozycji listy odbywa się w czasie stałym.

Wady:

- nie można uzyskać dostępu do dowolnego elementu listy w stałym czasie;
- lista potrzebuje nieco więcej pamięci niż tablica – oprócz danych pamiętane są wskaźniki na poprzedni i następny element listy;
- w przeciwieństwie do tablicy lista na ogół nie zajmuje spójnego obszaru pamięci, co może prowadzić do fragmentacji pamięci.

1.1-4. Obydwa problemy są problemami grafowymi mającymi na celu minimalizację pewnej ścieżki w grafie. W problemie najkrótszej ścieżki poszukuje się minimalnej ścieżki między dwoma wierzchołkami, a w problemie komiwojażera – minimalnego cyklu uwzględniającego wszystkie wierzchołki grafu (minimalny cykl Hamiltona); Problem najkrótszej ścieżki jest wielomianowy (istnieje dla niego szybki algorytm), podczas gdy problem komiwojażera jest NP-zupełny (prawdopodobnie nie istnieje efektywny algorytm rozwiązujący ten problem).

1.1-5. Rozwiązanie dokładne jest jedynym dopuszczalnym np. w problemie wyznaczenia trajektorii sztucznej satelity. Głównym powodem jest to, że ewentualne zniszczenie lub uszkodzenie satelity wiązałoby się z ogromnymi stratami finansowymi.

Natomiast przybliżone rozwiązanie jest wystarczające np. podczas modelowania pogody. W problemie tym występuje bardzo wiele parametrów i często są one dane tylko jako pewne przybliżenia rzeczywistych wielkości, przez co wyznaczenie dokładnego rozwiązania jest zazwyczaj niemożliwe. Poza tym pewna tolerancja rozwiązania jest całkowicie dopuszczalna.

1.2. Algorytmy jako technologia

1.2-1. Przykładem aplikacji, w której wykorzystywane są różnorodne algorytmy, jest współczesna gra komputerowa. Jej silnik grafiki trójwymiarowej jest zaawansowanym środowiskiem, w którym stosowane są algorytmy geometrii obliczeniowej i renderowania grafiki 3D, jak również wiele algorytmów numerycznych do wyznaczania interpolacji oraz algorytmy grafowe. Także w dziedzinie sztucznej inteligencji opracowano wiele zaawansowanych algorytmów. Ponadto powszechne problemy, takie jak wyszukiwanie elementu w tablicy czy sortowanie, są rozwiązywane w niemal każdej aplikacji.

1.2-2. Jeśli n jest liczbą naturalną, to nierówność $8n^2 < 64n \lg n$ jest spełniona dla $2 \leqslant n \leqslant 43$. Tablice o takich rozmiarach porządkowane są przez sortowanie przez wstawianie szybciej niż przez sortowanie przez scalanie.

1.2-3. Najmniejszą dodatnią liczbą całkowitą n spełniającą nierówność $100n^2 < 2^n$ jest $n = 15$.

Problemy

1-1. Porównanie czasów działania

W tabeli 1 zebrano wyznaczone wartości. Liczby przekraczające 10^7 zostały podane w przybliżeniu. Przyjęto, że na każdy miesiąc przypada 30 dni, a na każdy rok 365 dni.

$f(n)$	1 sekunda	1 minuta	1 godzina	1 dzień	1 miesiąc	1 rok	1 wiek
$\lg n$	2^{10^6}	$2^{6\cdot 10^7}$	$2^{3{,}6\cdot 10^9}$	$2^{8{,}64\cdot 10^{10}}$	$2^{2{,}59\cdot 10^{12}}$	$2^{3{,}15\cdot 10^{13}}$	$2^{3{,}15\cdot 10^{15}}$
$\sqrt{n}$	10^{12}	$3{,}6\cdot 10^{15}$	$1{,}3\cdot 10^{19}$	$7{,}46\cdot 10^{21}$	$6{,}72\cdot 10^{24}$	$9{,}95\cdot 10^{26}$	$9{,}95\cdot 10^{30}$
n	10^6	$6\cdot 10^7$	$3{,}6\cdot 10^9$	$8{,}64\cdot 10^{10}$	$2{,}59\cdot 10^{12}$	$3{,}15\cdot 10^{13}$	$3{,}15\cdot 10^{15}$
$n\lg n$	62746	$2{,}8\cdot 10^6$	$1{,}33\cdot 10^8$	$2{,}76\cdot 10^9$	$7{,}19\cdot 10^{10}$	$7{,}98\cdot 10^{11}$	$6{,}86\cdot 10^{13}$
n^2	1000	7745	60000	293938	$1{,}61\cdot 10^6$	$5{,}62\cdot 10^6$	$5{,}62\cdot 10^7$
n^3	100	391	1532	4420	13736	31593	146645
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

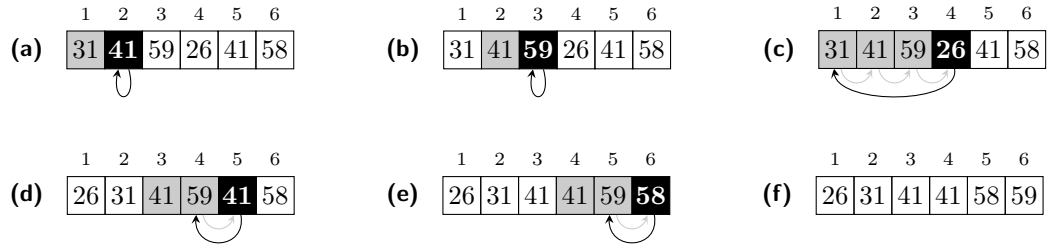
Tabela 1: Ograniczenia rozmiarów problemów.

Rozdział **2**

Zaczynamy

2.1. Sortowanie przez wstawianie

2.1-1. Rys. 1 przedstawia działanie algorytmu INSERTION-SORT dla tablicy A.



Rysunek 1: Działanie algorytmu INSERTION-SORT dla tablicy $A = \langle 31, 41, 59, 26, 41, 58 \rangle$. **(a)–(e)** Iteracje pętli **for** w wierszach 1–8. **(f)** Wynikowa posortowana tablica.

2.1-2. Aby sortować w porządku nierosnącym, wystarczy w warunku pętli **while** w linii 5 algorytmu INSERTION-SORT zmienić znak drugiej nierówności na przeciwny:

5 **while** $i > 0$ i $A[i] < key$

2.1-3. Przedstawiony opis prowadzi do następującego algorytmu wyszukiwania liniowego:

LINEAR-SEARCH(A, v)
1 $i \leftarrow 1$
2 **while** $i \leqslant length[A]$ i $A[i] \neq v$
3 **do** $i \leftarrow i + 1$
4 **if** $i \leqslant length[A]$
5 **then return** i
6 **return** NIL

Udowodnimy dla powyższej procedury niezmiennik pętli:

> Na początku każdej iteracji pętli **while** w wierszach 2–3 fragment tablicy $A[1 \,..\, i-1]$ nie zawiera elementu v.

Inicjowanie: Przed pierwszą iteracją $i = 1$, więc fragment $A[1 \,..\, i-1]$ jest pusty.

Utrzymanie: Załóżmy, że podtablica $A[1\,..\,i-1]$ nie zawiera elementu v. W warunku pętli **while** sprawdzamy, czy $A[i]$ jest różne od v. Jeśli tak, to i jest zwiększane o 1, więc niezmiennik jest zachowany. W przeciwnym przypadku (odnaleziono v) przerywamy pętlę.

Zakończenie: Pętla kończy swe działanie, kiedy zostanie odnaleziony indeks i taki, że $A[i] = v$ albo $i = \mathit{length}[A] + 1$. Pierwszy przypadek oznacza odnalezienie pierwszego wystąpienia v w tablicy A, a drugi – że przejrzeliśmy całą tablicę, nie znajdując v ($A[1\,..\,i-1]$ jest teraz całą tablicą A).

2.1-4. W tym problemie rozważać będziemy tylko liczby całkowite nieujemne, które są reprezentowane przez tablice bitów w kolejności od najmniej do najbardziej znaczącego.

Dane wejściowe: n-elementowe tablice A i B zawierające reprezentacje binarne n-bitowych liczb całkowitych nieujemnych a i b.
Wynik: $(n+1)$-elementowa tablica C zawierająca reprezentację binarną $(n+1)$-bitowej liczby całkowitej c takiej, że $c = a + b$.

```
BINARY-ADD(A, B)
1  n ← length[A]
2  for i ← 1 to n + 1
3        do C[i] ← 0
4  for i ← 1 to n
5        do sum ← A[i] + B[i] + C[i]
6           C[i] ← sum mod 2
7           C[i + 1] ← ⌊sum/2⌋
8  return C
```

Po utworzeniu tablicy C i wypełnieniu jej zerami, w pętli **for** w wierszach 4–7 procedura dodaje poszczególne bity liczb a i b w kolejności od najmniej do najbardziej znaczącego. W rzeczywistości przeprowadzane jest dodawanie modulo 2, a bit przeniesienia jest zapamiętywany na kolejnej pozycji w tablicy wynikowej i uczestniczy w kolejnej iteracji pętli (zakładając, że obecna iteracja nie jest ostatnią).

2.2. Analiza algorytmów

2.2-1.

$$n^3/1000 - 100n^2 - 100n + 3 = \Theta(n^3)$$

2.2-2. Poniższy algorytm implementuje sortowanie przez wybieranie:

```
SELECTION-SORT(A)
1  n ← length[A]
2  for j ← 1 to n − 1
3        do min ← j
4           for i ← j + 1 to n
5                 do if A[i] < A[min]
6                       then min ← i
7           zamień A[min] ↔ A[j]
```

Zewnętrzna pętla algorytmu zachowuje następujący niezmiennik:

> Na początku każdej iteracji pętli **for** w wierszach 2–7 podtablica $A[1\,..\,j-1]$ jest posortowana niemalejąco i zawiera $j-1$ najmniejszych elementów znajdujących się początkowo w tablicy A.

Nie trzeba wykonywać n iteracji pętli **for** z wierszy 2–7, gdyż po jej zakończeniu (po $n-1$ iteracjach) fragment $A[1\,..\,n-1]$ zawiera $n-1$ najmniejszych elementów tablicy A w porządku niemalejącym, zatem element $A[n]$ jest większy lub równy względem każdego elementu z podtablicy $A[1\,..\,n-1]$, a to oznacza, że cała tablica pozostaje posortowana niemalejąco.

Przeprowadzanych jest $n-1$ iteracji zewnętrznej pętli **for**, a wewnętrzna pętla **for** iteruje po wszystkich elementach aktualnie nieposortowanego fragmentu tablicy, szukając jego minimalnego elementu. Łącznie wykonywanych jest więc

$$\sum_{j=1}^{n-1}(n-j)=\sum_{i=1}^{n-1} j=\frac{n(n-1)}{2}$$

iteracji wewnętrznej pętli **for**, zatem zarówno pesymistyczny, jak i optymistyczny czas działania algorytmu wynosi $\Theta(n^2)$.

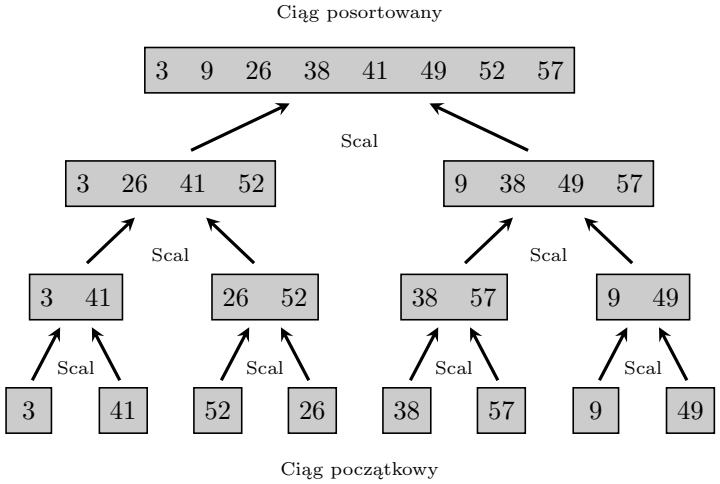
2.2-3. Z zad. C.3-2 mamy, że w średnim przypadku należy sprawdzić $(n+1)/2$ elementów tablicy, zatem średni czas działania algorytmu wyszukiwania liniowego wynosi $\Theta(n)$. W przypadku pesymistycznym procedura sprawdza wszystkie n elementów, nie znajdując szukanego, a więc otrzymujemy ten sam wynik.

2.2-4. Na początku działania algorytmu można wykrywać egzemplarze danych wejściowych, które stanowią dla niego przypadek optymistyczny i zwracać dla nich wyniki, które zostały przygotowane przed uruchomieniem algorytmu.

2.3. Projektowanie algorytmów

2.3-1. Na rys. 2 przedstawiono działanie algorytmu sortowania przez scalanie dla tablicy A.

2.3-2. Poniżej przedstawiono implementację procedury MERGE, w której nie wykorzystuje się wartowników.



Rysunek 2: Działanie algorytmu MERGE-SORT dla tablicy $A = \langle 3, 41, 52, 26, 38, 57, 9, 49\rangle$.

MERGE$'(A, p, q, r)$

```
 1  n_1 ← q − p + 1
 2  n_2 ← r − q
 3  utwórz tablice L[1 .. n_1] i R[1 .. n_2]
 4  for i ← 1 to n_1
 5        do L[i] ← A[p + i − 1]
 6  for j ← 1 to n_2
 7        do R[j] ← A[q + j]
 8  i ← 1
 9  j ← 1
10  for k ← p to r
11        do if j > n_2 lub L[i] ⩽ R[j]
12              then A[k] ← L[i]
13                   i ← i + 1
14              else A[k] ← R[j]
15                   j ← j + 1
```

Warunek w wierszu 11 w powyższej procedurze pozwala na poprawne rozmieszczenie elementów z tablic L i R w wynikowej tablicy A pomimo braku wartowników. Dzięki niemu do k-tej komórki tablicy A zostanie przekopiowana wartość z tablicy L w jednym z dwóch przypadków – gdy tablica R została już w całości przeglądnięta, bądź wtedy, gdy w tablicach L i R znajdują się jeszcze elementy do przekopiowania, ale mniejsza (lub równa) z liczb znajduje się na aktualnej pozycji w tablicy L. Gdy indeks j przekroczy n_2, to do końca działania procedury do tablicy A będą kopiowane elementy z tablicy L.

2.3-3. Przeprowadzimy dowód przez indukcję względem k. Dla $k = 1$ mamy $n = 2$ i $T(n) = 2 = 2\lg 2$, więc przypadek bazowy zachodzi. Załóżmy teraz, że $k > 1$, czyli $n > 2$ i że zachodzi $T(n/2) = (n/2)\lg(n/2)$. Mamy

$$T(n) = 2T(n/2) + n = 2(n/2)\lg(n/2) + n = n(\lg n - 1) + n = n\lg n,$$

co dowodzi rozwiązania rekurencji dla n będącego potęgą 2.

2.3-4. Niech $T(n)$ będzie czasem potrzebnym na posortowanie tablicy $A[1\,..\,n]$. Wstawienie elementu $A[n]$ w posortowaną podtablicę $A[1\,..\,n-1]$ odbywa się w najgorszym przypadku w czasie $\Theta(n)$, otrzymujemy więc następujące równanie rekurencyjne:

$$T(n) = \begin{cases} \Theta(1), & \text{jeśli } n = 1, \\ T(n-1) + \Theta(n), & \text{jeśli } n > 1. \end{cases}$$

2.3-5. Algorytm wyszukiwania binarnego przyjmuje na wejściu posortowaną niemalejąco tablicę A oraz szukaną wartość v. Podczas wyszukiwania v w podtablicy $A[low\,..\,high]$, v jest porównywane z $A[\lfloor (low + high)/2 \rfloor]$, czyli elementem środkowym tego fragmentu i na podstawie wyniku tego porównania eliminuje z dalszych rozważań odpowiednią połowę tej podtablicy.

Poniżej przedstawiono wersję rekurencyjną oraz iteracyjną algorytmu wyszukiwania binarnego. W przypadku odnalezienia wartości v w tablicy A zwracany jest taki indeks i, że $A[i] = v$. Jeśli elementu v nie ma w tablicy, to wynikiem procedury jest specjalna wartość NIL. Wersja rekurencyjna przyjmuje dodatkowo parametry *low* i *high* będące indeksami początku i końca przetwarzanego fragmentu tablicy A. Aby wyszukiwać w całej tablicy A, używając procedury rekurencyjnej, należy wywołać ją z parametrami $low = 1$ i $high = length[A]$.

```
RECURSIVE-BINARY-SEARCH(A, v, low, high)
1  if low > high
2     then return NIL
3  mid ← ⌊(low + high)/2⌋
4  if v = A[mid]
5     then return mid
6  if v < A[mid]
7     then return RECURSIVE-BINARY-SEARCH(A, v, low, mid − 1)
8     else  return RECURSIVE-BINARY-SEARCH(A, v, mid + 1, high)
```

```
ITERATIVE-BINARY-SEARCH(A, v)
 1  low ← 1
 2  high ← length[A]
 3  while low ⩽ high
 4        do mid ← ⌊(low + high)/2⌋
 5           if v = A[mid]
 6              then return mid
 7           if v < A[mid]
 8              then high ← mid − 1
 9              else  low ← mid + 1
10  return NIL
```

W obu wersjach algorytmu BINARY-SEARCH v jest przyrównywane do środkowego elementu fragmentu $A[low\,..\,high]$, odrzucana jest (w przybliżeniu) połowa tej podtablicy i v jest następnie poszukiwane w drugiej połowie. Procedury kończą swe działanie, gdy odnajdą v albo gdy zakres poszukiwań okaże się pusty (czyli $low > high$), co oznacza, że elementu v nie ma w A. Niech n będzie rozmiarem tablicy A. Rekurencja opisująca czas działania algorytmu w przypadku pesymistycznym ma postać

$$T(n) = \begin{cases} \Theta(1), & \text{jeśli } n \leqslant 1, \\ T(n/2) + \Theta(1), & \text{jeśli } n > 1. \end{cases}$$

Jej rozwiązaniem (z zad. 4.3-3) jest $T(n) = \Theta(\lg n)$.

2.3-6. Wyszukując binarnie, można odnaleźć pozycję tablicy, na którą należy umieścić kolejny element z nieposortowanego fragmentu, jednak wstawienie go wymaga przesunięcia pewnej części tablicy o jedną pozycję w prawo, co w najgorszym przypadku zajmuje czas $\Theta(n)$. Nie można zatem obniżyć czasu działania sortowania przez wstawianie poprzez zastosowanie wyszukiwania binarnego.

2.3-7. Będziemy traktować zbiór S jak tablicę $S[1\,..\,n]$. Dla każdego elementu $S[i]$ można wyszukiwać inny element w tablicy S, który po zsumowaniu z $S[i]$ daje x. Będziemy wyszukiwać binarnie po uprzednim posortowaniu S (procedura wyszukiwania binarnego została opisana w zad. 2.3-5). Dla $S[i]$ szukamy zatem elementu o wartości $x - S[i]$ w podtablicy $S[i+1\,..\,n]$. Podtablica ta jest pusta dla $i = n$, dlatego wyszukiwanie dla ostatniego elementu pomijamy. W zależności od wyniku wyszukiwania zwracana jest wartość logiczna TRUE lub FALSE. Algorytm zapisujemy w postaci pseudokodu:

```
SUM-SEARCH(S, x)
1  n ← length[S]
2  MERGE-SORT(S, 1, n)
3  for i ← 1 to n − 1
4        do if RECURSIVE-BINARY-SEARCH(S, x − S[i], i + 1, n) ≠ NIL
5              then return TRUE
6  return FALSE
```

Sortowanie S w wierszu 2 działa w czasie $\Theta(n \lg n)$, a procedura RECURSIVE-BINARY-SEARCH jest wywoływana w wierszu 4 dla tablic o rozmiarach, kolejno, 1, 2, ..., $n - 1$. Wywołania te łącznie zajmują czas

$$\sum_{i=1}^{n-1} \Theta(\lg i) = \Theta\left(\sum_{i=1}^{n} \lg i - \lg n\right) = \Theta\left(\lg\left(\prod_{i=1}^{n} i\right) - \lg n\right) = \Theta(\lg(n!) - \lg n) = \Theta(n \lg n),$$

ponieważ $\lg(n!) = \Theta(n \lg n)$ (ze wzoru (3.18)). Zatem pesymistyczny czas algorytmu SUM-SEARCH wynosi $\Theta(n \lg n)$.

Problemy

2-1. Sortowanie przez wstawianie dla małych tablic podczas sortowania przez scalanie

(a) Sortowanie przez wstawianie podlisty o długości k działa w czasie pesymistycznym $\Theta(k^2)$, a zastosowane osobno do n/k takich podlist zajmuje czas równy $(n/k) \cdot \Theta(k^2) = \Theta(nk)$.

(b) Uogólniając procedurę scalania dwóch podlist na jednoczesne scalanie n/k podlist, można osiągnąć czas $\Theta(n^2/k)$, ponieważ wszystkie podlisty należy przejrzeć łącznie n razy, szukając za każdym razem najmniejszego elementu do wstawienia na listę wynikową.

Lepszy czas można jednak uzyskać dzięki scalaniu podlist parami, następnie otrzymane większe podlisty również scalając parami itd., aż do uzyskania pojedynczej listy wynikowej. Na każdym poziomie scalanie wymaga czasu $\Theta(n)$, jest $\lceil \lg(n/k) \rceil$ poziomów, a zatem czas działania scalania n/k podlist przy użyciu tego pomysłu wynosi $\Theta(n \lg(n/k))$.

(c) Czas działania zmodyfikowanego algorytmu ma ten sam rząd złożoności co czas działania sortowania przez scalanie, o ile zachodzi $\Theta(nk + n\lg(n/k)) = \Theta(n\lg n)$. Zauważmy, że jeśli $k = \omega(\lg n)$, to zmodyfikowany algorytm działa w czasie $\omega(n\lg n)$. Zbadajmy więc, co się dzieje, gdy $k = \Theta(\lg n)$. Mamy

$$\Theta(nk + n\lg(n/k)) = \Theta(nk + n\lg n - n\lg k) = \Theta(2n\lg n - n\lg\lg n) = \Theta(n\lg n),$$

dzięki opuszczeniu składnika niższego rzędu i pominięciu stałego współczynnika. Maksymalnym rzędem k, dla którego czas zmodyfikowanego algorytmu jest równy czasowi zwykłego sortowania przez scalanie, jest zatem $\Theta(\lg n)$.

(d) W praktyce k powinno być największą długością listy, dla której sortowanie przez wstawianie działa szybciej od sortowania przez scalanie.

2-2. Poprawność sortowania bąbelkowego

(a) Należy jeszcze pokazać, że tablica A' stanowi permutację tablicy A.

(b) Niezmiennik wewnętrznej pętli **for**:

Przed każdą iteracją pętli **for** w wierszach 2–4 najmniejszym elementem podtablicy $A[j\,..\,\mathit{length}[A]]$ jest $A[j]$.

Inicjowanie: Przed pierwszą iteracją $j = \mathit{length}[A]$, więc $A[j\,..\,\mathit{length}[A]]$ zawiera tylko jeden element, który oczywiście jest najmniejszy w tej podtablicy i jest nim $A[j]$.

Utrzymanie: Załóżmy, że $A[j]$ jest najmniejszym elementem w $A[j\,..\,\mathit{length}[A]]$. Jeżeli $A[j-1]$ jest większe od $A[j]$, to $A[j]$ jest zamieniane z $A[j-1]$ w wierszu 4, więc w tym momencie podtablica $A[j-1\,..\,\mathit{length}[A]]$ posiada swój najmniejszy element w $A[j-1]$. Uaktualnienie j powoduje odtworzenie niezmiennika. W przeciwnym przypadku zamiana nie następuje, przez co $A[j-1]$ stanowi najmniejszy element $A[j-1\,..\,\mathit{length}[A]]$ i aktualizacja j także pozwala spełnić niezmiennik.

Zakończenie: Po zakończeniu wykonywania pętli zachodzi $j = i$, a więc $A[i]$ jest najmniejszym elementem podtablicy $A[i\,..\,\mathit{length}[A]]$.

(c) Niezmiennik zewnętrznej pętli **for**:

Przed każdą iteracją pętli **for** w wierszach 1–4 podtablica $A[1\,..\,i-1]$ jest posortowana niemalejąco.

Inicjowanie: Przed pierwszą iteracją $i = 1$, czyli podtablica $A[1\,..\,i-1]$ jest pusta, a więc jest trywialnie posortowana.

Utrzymanie: Z założenia, że podtablica $A[1\,..\,i-1]$ jest posortowana niemalejąco wynika, że $A[i-1]$ jest największym elementem tej podtablicy. Wewnętrzna pętla **for** wyszukuje w podtablicy $A[i\,..\,\mathit{length}[A]]$ najmniejszy element i umieszcza go na pozycji i (dowód w poprzednim punkcie). W podtablicy $A[i\,..\,\mathit{length}[A]]$ nie ma mniejszych elementów od $A[i-1]$, a zatem w szczególności zachodzi $A[i-1] \leqslant A[i]$. Stąd wnioskujemy, że podtablica $A[1\,..\,i]$ jest posortowana niemalejąco i po aktualizacji i niezmiennik zostaje odtworzony.

Zakończenie: Na końcu mamy $i = \mathit{length}[A] + 1$. Podtablica $A[1\,..\,i-1]$ jest całą tablicą A posortowaną niemalejąco, a zatem algorytm sortuje poprawnie.

(d) Niech $n = \mathit{length}[A]$. Dla wszystkich przypadków danych wejściowych pętla **for** z wierszy 2–4 wykonuje $n - i$ iteracji dla każdego $i = 1, 2, \ldots, n$. Pesymistyczny czas działania sortowania bąbelkowego wynosi zatem

$$T(n) = \sum_{i=1}^{n}(n-i) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2),$$

jest on więc równy pesymistycznemu czasowi sortowania przez wstawianie.

2-3. Poprawność schematu Hornera

(a) Pętla **while** w wierszach 3–5 wykonuje $n + 1$ iteracji, więc czas działania tego fragmentu kodu wynosi $\Theta(n)$.

(b) Następujący fragment kodu oblicza wartość $P(x)$ dla zadanych współczynników $a_0, a_1, \ldots, a_n$ wielomianu P i wartości x:

```
1  y ← 0
2  for i ← 0 to n
3        do s ← a_i
4           for j ← 1 to i
5                 do s ← s · x
6           y ← y + s
```

Pętla **for** w wierszach 4–5 wykonuje się i razy dla każdego $i = 0, 1, \ldots, n$, czyli łącznie $\sum_{i=0}^{n} i = n(n+1)/2$ razy. Czas działania powyższego kodu wynosi zatem $\Theta(n^2)$. Jest to więc mniej efektywny sposób obliczania wartości wielomianu od schematu Hornera.

(c) Dowodzimy w trzech krokach:

Inicjowanie: Przed pierwszą iteracją $i = n$, więc

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k = \sum_{k=0}^{-1} a_{k+n+1}x^k = 0,$$

co zgadza się z początkową wartością y.

Utrzymanie: Podczas kolejnych iteracji y przyjmuje wartość $a_i + xy$. Przy założeniu, że niezmiennik jest spełniony przed bieżącą iteracją, mamy

$$y = a_i + \sum_{k=0}^{n-(i+1)} a_{k+i+1}x^{k+1} = a_i x^0 + \sum_{k=1}^{n-i} a_{k+i}x^k = \sum_{k=0}^{n-i} a_{k+i}x^k$$

i po aktualizacji i niezmiennik zostaje odtworzony.

Zakończenie: Na końcu mamy $i = -1$, więc

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k = \sum_{k=0}^{n} a_k x^k = P(x),$$

zatem algorytm poprawnie oblicza wynik.

(d) Algorytm zwraca poprawny wynik, gdyż ustawia prawidłowe początkowe wartości, $y = 0$ oraz $i = n$, a poprawność pętli **while** została wykazana w poprzednim punkcie. Procedura posiada własność stopu, ponieważ zmienna i jest zmniejszana w kolejnych iteracjach pętli, zatem po skończonej liczbie iteracji i po skończonej liczbie kroków algorytmu będzie zachodzić $i = 0$, co jest warunkiem zakończenia pętli. Algorytm działa więc poprawnie.

2-4. Inwersje

(a) $\langle 1, 5\rangle$, $\langle 2, 5\rangle$, $\langle 3, 4\rangle$, $\langle 3, 5\rangle$, $\langle 4, 5\rangle$

(b) Największą możliwą liczbę inwersji ma tablica posortowana malejąco. Każdy element na pozycji i tworzy inwersję z każdym z $n - i$ elementów na prawo od niego w tej tablicy. Liczba inwersji wynosi zatem

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

(c) Załóżmy, że tablica A ma inwersję $\langle i, j\rangle$. To znaczy, że $i < j$ oraz $A[i] > A[j]$. Wtedy w procedurze INSERTION-SORT pewna iteracja pętli **while** w wierszach 5–7 przesunie $A[i]$ o jedną pozycję w prawo, podczas gdy element będący pierwotnie na pozycji j będzie znajdował się na lewo od niego, przez co wyeliminowana zostanie jedna z inwersji. Tak więc każda iteracja pętli **while** usuwa jedną inwersję tablicy A, skąd wnioskujemy, że ich liczba jest tego samego rzędu, co czas działania algorytmu sortowania przez wstawianie wykonanego na A.

(d) Niech a i b będą dwoma różnymi elementami tablicy A. Załóżmy, że podczas sortowania przez scalanie w procedurze MERGE w pewnym momencie $L[i] = a$ oraz $R[j] = b$. Jeśli warunek z wiersza 13 procedury MERGE zachodzi, to znaczy, że a i b nie tworzą inwersji. W przeciwnym przypadku $a > b$, a ponieważ scalane podtablice są posortowane, to b jest mniejsze od każdego dotychczas nieprzetworzonego elementu podtablicy L. Liczba elementów A należących do L wynosi n_1, zatem w momencie przetwarzania elementu b, jest w niej $n_1 - i + 1$ elementów nieprzetworzonych, a więc tyle inwersji tworzy z nimi b. Od tego momentu element ten będzie z nimi w jednej podtablicy, więc nie policzymy żadnej inwersji dwukrotnie.

W ten sposób, modyfikując algorytm sortowania przez scalanie, wyznaczamy liczbę inwersji n-elementowej tablicy A w czasie $\Theta(n \lg n)$, czego efektem ubocznym jest jej posortowanie. W procedurze MERGE-SORT wystarczy początkowo wyzerować licznik inwersji i następnie sumować częściowe wyniki zwracane z wywołań rekurencyjnych, a w MERGE – doliczać odpowiednią liczbę inwersji tworzonych przez element b. Poniższe pseudokody implementują to rozumowanie.

```
COUNT-INVERSIONS(A, p, r)
1  inversions ← 0
2  if p < r
3     then q ← ⌊(p + r)/2⌋
4          inversions ← inversions + COUNT-INVERSIONS(A, p, q)
5          inversions ← inversions + COUNT-INVERSIONS(A, q + 1, r)
6          inversions ← inversions + MERGE-INVERSIONS(A, p, q, r)
7  return inversions
```

```
MERGE-INVERSIONS(A, p, q, r)
 1  n_1 ← q − p + 1
 2  n_2 ← r − q
 3  utwórz tablice L[1 .. n_1 + 1] i R[1 .. n_2 + 1]
 4  for i ← 1 to n_1
 5        do L[i] ← A[p + i − 1]
 6  for j ← 1 to n_2
 7        do R[j] ← A[q + j]
 8  L[n_1 + 1] ← R[n_2 + 1] ← ∞
 9  i ← j ← 1
10  inversions ← 0
11  for k ← p to r
12        do if L[i] ⩽ R[j]
13              then A[k] ← L[i]
14                   i ← i + 1
15              else A[k] ← R[j]
16                   j ← j + 1
17                   inversions ← inversions + n_1 − i + 1
18  return inversions
```

Rozdział **3**

Rzędy wielkości funkcji

3.1. Notacja asymptotyczna

3.1-1. Załóżmy, że dziedziną funkcji $f(n)$ i $g(n)$ jest $\mathbb{N}$. Niech $A = \{\, n \in \mathbb{N} : f(n) \geqslant g(n) \,\}$. Dla odpowiednio dużych $n \in A$, dla których obie funkcje przyjmują wartości nieujemne, mamy

$$\max(f(n), g(n)) = f(n) \leqslant f(n) + g(n) = O(f(n) + g(n)).$$

Z kolei

$$\max(f(n), g(n)) = f(n) \geqslant \frac{f(n) + g(n)}{2} = \Omega(f(n) + g(n))$$

i na mocy tw. 3.1 otrzymujemy, że $\max(f(n), g(n)) = \Theta(f(n)+g(n))$. Identyczny rezultat można uzyskać dla $n \in \mathbb{N} \setminus A$, tzn. gdy $f(n) < g(n)$.

3.1-2. Aby pokazać, że $(n + a)^b = \Theta(n^b)$, należy znaleźć stałe c_1, c_2, $n_0 > 0$ takie, że

$$0 \leqslant c_1 n^b \leqslant (n + a)^b \leqslant c_2 n^b$$

dla wszystkich $n \geqslant n_0$. Zauważmy, że $n+a \leqslant n+|a| \leqslant 2n$, gdy $|a| \leqslant n$ oraz $n+a \geqslant n-|a| \geqslant n/2$, o ile $|a| \leqslant n/2$. Stąd, jeśli $n \geqslant 2|a|$, to zachodzi

$$0 \leqslant n/2 \leqslant n + a \leqslant 2n.$$

Ponieważ $b > 0$, to powyższe nierówności możemy podnieść do potęgi b:

$$0 \leqslant (n/2)^b \leqslant (n + a)^b \leqslant (2n)^b,$$
$$0 \leqslant 2^{-b} n^b \leqslant (n + a)^b \leqslant 2^b n^b.$$

Widać zatem, że szukanym stałym można nadać wartości $c_1 = 2^{-b}$, $c_2 = 2^b$ oraz $n_0 = 2|a|$. Prawdą jest zatem, że $(n + a)^b = \Theta(n^b)$.

3.1-3. Niech $T(n)$ będzie czasem działania algorytmu A. Stwierdzenie „$T(n)$ wynosi co najmniej $O(n^2)$” oznacza, że począwszy od pewnego n, $T(n) \geqslant f(n)$ dla pewnej funkcji $f(n)$ z klasy $O(n^2)$. Zdanie to pozostaje prawdziwe dla dowolnego T, wystarczy bowiem wybrać funkcję $f(n)$ tożsamościowo równą 0, która oczywiście jest w $O(n^2)$. Widać więc, że takie określenie nie przekazuje żadnej użytecznej informacji o czasie działania algorytmu.

3.1-4. Znajdziemy stałe c, $n_0 > 0$ takie, że $0 \leqslant 2^{n+1} \leqslant c2^n$ dla każdego $n \geqslant n_0$. Ponieważ $2^{n+1} = 2 \cdot 2^n$ dla każdego $n \geqslant 1$, to można przyjąć $c = 2$ oraz $n_0 = 1$. A zatem $2^{n+1} = O(2^n)$.

Spróbujmy teraz wyznaczyć te same stałe, ale spełniające zależność $0 \leqslant 2^{2n} \leqslant c2^n$ dla wszystkich $n \geqslant n_0$. Mamy $2^{2n} = 2^n \cdot 2^n \leqslant c2^n$, z czego wynika, że $c \geqslant 2^n$, co jednak uzależnia c od funkcji zmiennej n przyjmującej dowolnie duże wartości, a więc c nie może być stałą. Stąd otrzymujemy, że $2^{2n} \neq O(2^n)$.

3.1-5. Z definicji notacji Θ mamy, że $f(n) = \Theta(g(n))$ wtedy i tylko wtedy, gdy istnieją takie stałe c_1, c_2, $n_0 > 0$, że nierówności

$$0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$$

zachodzą dla wszystkich $n \geqslant n_0$. Możemy je zapisać w formie układu

$$\begin{cases} 0 \leqslant c_1 g(n) \leqslant f(n) \\ 0 \leqslant f(n) \leqslant c_2 g(n) \end{cases}.$$

Z pierwszej nierówności układu dostajemy, że $f(n) = \Omega(g(n))$, a z drugiej, że $f(n) = O(g(n))$.

3.1-6. Niech c_1, c_2, n_1, n_2 będą pewnymi stałymi dodatnimi. Pesymistyczny czas działania algorytmu wynosi $O(g(n))$ wtedy i tylko wtedy, gdy dla dowolnych danych wejściowych rozmiaru $n \geqslant n_1$ czas jego działania $f(n)$ nie przekracza $c_1 g(n)$. Z kolei to, że optymistyczny czas wynosi $\Omega(g(n))$, oznacza, że dla dowolnych danych wejściowych rozmiaru $n \geqslant n_2$ czas działania algorytmu $f(n)$ jest nie mniejszy niż $c_2 g(n)$. Widać zatem, że dla dowolnych danych rozmiaru $n \geqslant \max(n_1, n_2)$ mamy $0 \leqslant c_2 g(n) \leqslant f(n) \leqslant c_1 g(n)$, a to jest równoważne z tym, że $f(n) = \Theta(g(n))$.

3.1-7. Załóżmy niepustość tego zbioru i rozważmy pewne $f(n) \in o(g(n)) \cap \omega(g(n))$. Zachodzi zatem zarówno $f(n) = o(g(n))$, jak i $f(n) = \omega(g(n))$, co oznacza, że dla każdych dodatnich stałych c_1 i c_2 istnieje pewne dodatnie n_0, że

$$c_1 g(n) < f(n) < c_2 g(n)$$

dla wszystkich $n \geqslant n_0$. Dochodzimy do sprzeczności, bowiem nieprawdą jest, że każde liczby c_1 i c_2 spełniają $c_1 < c_2$. Stąd $o(g(n)) \cap \omega(g(n)) = \emptyset$.

Udowodniona własność pokazuje, że nie ma potrzeby definiowania notacji θ odpowiadającej Θ i analogicznej do o i ω.

3.1-8. *Notacja O dla funkcji dwóch zmiennych jest błędnie zdefiniowana – warunek powinien zachodzić dla wszystkich $n \geqslant n_0$* ***lub*** *$m \geqslant m_0$.*

Definicje notacji Ω i Θ dla funkcji dwóch zmiennych:

$$\Omega(g(n,m)) = \big\{ f(n,m) : \text{istnieją dodatnie stałe } c,\ n_0,\ m_0 \text{ takie, że} \\ 0 \leqslant cg(n,m) \leqslant f(n,m) \text{ dla wszystkich } n \geqslant n_0 \text{ lub } m \geqslant m_0 \big\},$$

$$\Theta(g(n,m)) = \big\{ f(n,m) : \text{istnieją dodatnie stałe } c_1,\ c_2,\ n_0,\ m_0 \text{ takie, że} \\ 0 \leqslant c_1 g(n,m) \leqslant f(n,m) \leqslant c_2 g(n,m) \text{ dla wszystkich } n \geqslant n_0 \text{ lub } m \geqslant m_0 \big\}.$$

3.2. Standardowe notacje i typowe funkcje

3.2-1. Z założenia, jeśli $n_1 \leqslant n_2$, to zachodzi $f(n_1) \leqslant f(n_2)$ oraz $g(n_1) \leqslant g(n_2)$, więc po dodaniu tych nierówności stronami otrzymujemy $f(n_1)+g(n_1) \leqslant f(n_2)+g(n_2)$, czyli że funkcja $f(n)+g(n)$ jest monotonicznie rosnąca. Traktując wartości funkcji $g(n)$ jako argumenty funkcji $f(n)$, otrzymamy $f(g(n_1)) \leqslant f(g(n_2))$, zatem $f(g(n))$ także jest funkcją monotonicznie rosnącą. Jeśli ponadto założymy, że funkcje $f(n)$ i $g(n)$ są nieujemne, to początkowe nierówności można pomnożyć stronami, co daje $f(n_1) \cdot g(n_1) \leqslant f(n_2) \cdot g(n_2)$, a to oznacza, że również funkcja $f(n) \cdot g(n)$ jest monotonicznie rosnąca.

3.2-2. Wykorzystując podstawowe własności logarytmów, otrzymujemy

$$\log_b a^{\log_b c} = \log_b c \cdot \log_b a = \log_b c^{\log_b a},$$

skąd na podstawie różnowartościowości funkcji logarytmicznej wynika tożsamość

$$a^{\log_b c} = c^{\log_b a}.$$

3.2-3.

Dowód wzoru (3.18). Górne oszacowanie na $\lg(n!)$ dostajemy dzięki wykorzystaniu własności logarytmów:

$$\lg(n!) = \lg\left(\prod_{i=1}^{n} i\right) = \sum_{i=1}^{n} \lg i \leqslant \sum_{i=1}^{n} \lg n = n \lg n = O(n \lg n).$$

Wykorzystując wzór Stirlinga i wybierając pewną stałą $c > 0$, ograniczamy $\lg(n!)$ od dołu:

$$\begin{aligned}
\lg(n!) &\geqslant \lg\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\frac{c}{n}\right)\right) \\
&= \lg\sqrt{2\pi n} + \lg\left(\frac{n}{e}\right)^n + \lg\left(1+\frac{c}{n}\right) \\
&> n\lg n - n\lg e \\
&\geqslant n\lg n - \frac{n\lg n}{2} \\
&= \frac{n\lg n}{2}.
\end{aligned}$$

Przedostatnia nierówność zachodzi, o ile $n \geqslant e^2$. Otrzymany wynik dowodzi, że $\lg(n!) = \Omega(n \lg n)$ i po skorzystaniu z twierdzenia 3.1 dostajemy $\lg(n!) = \Theta(n \lg n)$. □

Dowód tożsamości $n! = \omega(2^n)$. Równoważnie należy pokazać, że zachodzi

$$\lim_{n\to\infty} \frac{n!}{2^n} = \infty.$$

Zauważmy, że

$$\frac{n!}{2^n} = \left(\frac{1}{2}\right)\left(\frac{2}{2}\right)\cdots\left(\frac{n}{2}\right).$$

Wszystkie czynniki powyższego iloczynu są dodatnie, a przy coraz większym n ostatnie czynniki rosną nieograniczenie, zatem cały iloczyn dąży do ∞. □

Dowód tożsamości $n! = o(n^n)$. Na podstawie wzoru (3.1) dowód sprowadza się do pokazania, że

$$\lim_{n\to\infty} \frac{n!}{n^n} = 0.$$

Mamy

$$\frac{n!}{n^n} = \left(\frac{1}{n}\right)\left(\frac{2}{n}\right)\cdots\left(\frac{n}{n}\right).$$

Każdy czynnik po prawej stronie znaku równości jest dodatni i nie przekracza 1. Ponadto dla n dążącego do ∞ początkowe czynniki zmierzają do 0, a zatem granicą tego iloczynu jest 0. □

3.2-4. Funkcja $f(n)$ jest ograniczona wielomianowo, jeżeli istnieją stałe c, k, $n_0 > 0$ takie, że dla każdego $n \geqslant n_0$ zachodzi $f(n) \leqslant cn^k$. Stąd $\lg f(n) \leqslant k \lg n + \lg c \leqslant (k+1)\lg n$, o ile $n \geqslant c$, a więc $\lg f(n) = O(\lg n)$. Stwierdzenie, że funkcja $f(n)$ jest ograniczona wielomianowo, jest więc równoważne stwierdzeniu, że $\lg f(n) = O(\lg n)$.

Zanim przejdziemy do głównego dowodu, zauważmy, że $\lceil \lg n \rceil = \Theta(\lg n)$. Zachodzi bowiem $\lceil \lg n \rceil \geqslant \lg n$ oraz $\lceil \lg n \rceil < \lg n + 1 \leqslant 2 \lg n$ dla każdego $n \geqslant 2$.

Logarytmując pierwszą badaną funkcję przy wykorzystaniu wzoru (3.18), dostajemy

$$\lg(\lceil \lg n \rceil !) = \Theta(\lceil \lg n \rceil \lg \lceil \lg n \rceil) = \Theta(\lg n \lg \lg n) = \omega(\lg n),$$

a zatem $\lg(\lceil \lg n \rceil !) \neq O(\lg n)$ i funkcja $\lceil \lg n \rceil !$ nie jest ograniczona wielomianowo.

Dla drugiej funkcji mamy

$$\lg(\lceil \lg \lg n \rceil !) = \Theta(\lceil \lg \lg n \rceil \lg \lceil \lg \lg n \rceil) = \Theta(\lg \lg n \lg \lg \lg n) = o((\lg \lg n)^2) = o(\lg n).$$

Ostatni krok wynika z tożsamości $\lg^b n = o(n^a)$ prawdziwej dla stałych a, $b > 0$, w której podstawiono $\lg n$ w miejsce n oraz przyjęto $a = 1$ i $b = 2$. Otrzymany rezultat potwierdza, że $\lg(\lceil \lg \lg n \rceil !) = O(\lg n)$, a zatem funkcja $\lceil \lg \lg n \rceil !$ jest ograniczona wielomianowo.

3.2-5. Zdefiniujmy n jako

$$2^{2^{\cdot^{\cdot^{\cdot^{2^\epsilon}}}}}\Big\}k,$$

przy czym $0 < \epsilon \leqslant 1$, a $k \geqslant 1$ oznacza liczbę dwójek w powyższym zapisie. Zachodzi

$$\lg^* n = k \quad \text{oraz} \quad \lg n = 2^{2^{\cdot^{\cdot^{\cdot^{2^\epsilon}}}}}\Big\}k-1,$$

a zatem

$$\lg \lg^* n = \lg k \quad \text{oraz} \quad \lg^* \lg n = k - 1.$$

Oczywiście $k - 1 = \omega(\lg k)$, więc otrzymujemy, że $\lg^* \lg n = \omega(\lg \lg^* n)$.

3.2-6. Łatwo sprawdzić, że dla $i = 0$ oraz $i = 1$ wzór jest prawdziwy. Załóżmy teraz, że zachodzi

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} \quad \text{oraz} \quad F_{i+1} = \frac{\phi^{i+1} - \widehat{\phi}^{i+1}}{\sqrt{5}}$$

dla pewnego $i \geqslant 0$. Po wykorzystaniu zależności $\phi + 1 = \phi^2$ i $\widehat{\phi} + 1 = \widehat{\phi}^2$ otrzymujemy

$$F_{i+2} = F_{i+1} + F_i = \frac{\phi^{i+1} - \widehat{\phi}^{i+1}}{\sqrt{5}} + \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} = \frac{\phi^i(\phi+1) - \widehat{\phi}^i\big(\widehat{\phi}+1\big)}{\sqrt{5}} = \frac{\phi^{i+2} - \widehat{\phi}^{i+2}}{\sqrt{5}},$$

a zatem wzór jest spełniony dla każdego $i \geqslant 0$.

3.2-7. Korzystając z wyniku z poprzedniego zadania, mamy

$$\begin{aligned} F_{i+2} - \phi^i &= \frac{\phi^{i+2} - \widehat{\phi}^{i+2}}{\sqrt{5}} - \phi^i \\ &= \frac{\phi^i(\phi^2 - \sqrt{5}) - \widehat{\phi}^{i+2}}{\sqrt{5}} \\ &= \frac{\phi^i \cdot \frac{3-\sqrt{5}}{2} - \widehat{\phi}^i \cdot \frac{3-\sqrt{5}}{2}}{\sqrt{5}} \\ &= \frac{3-\sqrt{5}}{2\sqrt{5}} \left(\phi^i - \widehat{\phi}^i\right). \end{aligned}$$

Ponieważ $\phi > |\widehat{\phi}|$, to otrzymane wyrażenie jest nieujemne dla każdego $i \geqslant 0$ (równość zachodzi tylko wówczas, gdy $i = 0$), a zatem $F_{i+2} \geqslant \phi^i$ dla dowolnego $i \geqslant 0$.

Problemy

3-1. Asymptotyczne zachowanie wielomianów

Udowodnimy najpierw fakt, że $p(n) = \Theta(n^d)$. Należy znaleźć stałe c_1, c_2, $n_0 > 0$ takie, że dla $n \geqslant n_0$ prawdziwe są nierówności:

$$0 \leqslant c_1 n^d \leqslant a_d n^d + a_{d-1} n^{d-1} + \cdots + a_0 \leqslant c_2 n^d.$$

Po podzieleniu ich przez n^d dostajemy

$$0 \leqslant c_1 \leqslant a_d + \underbrace{\frac{a_{d-1}}{n} + \cdots + \frac{a_0}{n^d}}_{\delta} \leqslant c_2,$$

a ponieważ składnik δ można dowolnie zbliżyć do zera, zwiększając parametr n_0, to stąd obie wartości c_1 i c_2 mogą być dowolnie bliskie a_d.

(a) Zachodzi $p(n) = \Theta(n^d)$, zatem w szczególności $p(n) = O(n^d)$, co oznacza, że istnieją stałe c, $n_0 > 0$, że dla wszystkich $n \geqslant n_0$ prawdą jest $0 \leqslant p(n) \leqslant cn^d$. Z kolei $k \geqslant d$, więc $n^k \geqslant n^d$ dla $n \geqslant 1$, a zatem $0 \leqslant p(n) \leqslant cn^d \leqslant cn^k$, skąd dostajemy, że $p(n) = O(n^k)$.

(b) Nierówność $k \leqslant d$ implikuje $n^k \leqslant n^d$ dla $n \geqslant 1$. Korzystając z tego, że $p(n) = \Omega(n^d)$, mamy $0 \leqslant cn^k \leqslant cn^d \leqslant p(n)$, skąd wynika $p(n) = \Omega(n^k)$.

(c) Dla $k = d$ tożsamość $p(n) = \Theta(n^d) = \Theta(n^k)$ zachodzi w oczywisty sposób.

(d) Zachodzi $p(n) = O(n^d)$, tzn. istnieją stałe c, $n_0 > 0$ takie, że dla każdego $n \geqslant n_0$ spełniona jest nierówność $0 \leqslant p(n) \leqslant cn^d$. Niech $b > 0$ będzie dowolną stałą. Zbadajmy, dla jakich n zachodzi $cn^d < bn^k$. Ponieważ $k > d$, to nierówność sprowadzamy do postaci $c/b < n^{k-d}$, skąd $n > n_1 = (c/b)^{1/(k-d)}$. A zatem dla każdego $b > 0$ istnieje $n_2 = \max(n_0, n_1+1)$, że dla wszystkich $n \geqslant n_2$ zachodzi $0 \leqslant p(n) \leqslant cn^d < bn^k$, co oznacza, że $p(n) = o(n^k)$.

(e) Rozumowanie jest analogiczne do tego z poprzedniego punktu. Wystarczy wykorzystać fakt, że $p(n) = \Omega(n^d)$ i pokazać, że dla każdej stałej $b > 0$ odpowiednio duże n spełniają nierówność $0 \leqslant bn^k < cn^d \leqslant p(n)$, gdzie $c > 0$ jest stałą ukrytą w notacji Ω.

3-2. Względny rząd asymptotyczny

(a) Ponieważ każdy wielomian rośnie szybciej niż dowolna funkcja polilogarytmiczna, czyli $\lg^k n = o(n^\epsilon)$, to stąd wynika, że również $\lg^k n = O(n^\epsilon)$.

(b) Podobnie, ze wzoru (3.9), mamy, że $n^k = o(c^n)$, co implikuje również $n^k = O(c^n)$.

(c) Wyrażenie $\sqrt{n}$ nie jest w żadnej rozważanej relacji z wyrażeniem $n^{\sin n}$, gdyż wartość wykładnika tego ostatniego przyjmuje wszystkie wartości między -1 a 1, podczas gdy $\sqrt{n} \equiv n^{1/2}$.

(d) Zachodzi $2^n = \omega(2^{n/2})$, bo

$$\lim_{n\to\infty} \frac{2^n}{2^{n/2}} = \lim_{n\to\infty} 2^{n/2} = \infty,$$

a stąd wynika też $2^n = \Omega(2^{n/2})$.

(e) Funkcje $n^{\lg c}$ i $c^{\lg n}$ dla $n > 0$ są równoważne na podstawie tożsamości (3.15).

(f) Ze wzoru (3.18) mamy $\lg(n!) = \Theta(n \lg n)$, z kolei $\lg n^n = n \lg n = \Theta(n \lg n)$, a zatem obie funkcje są asymptotycznie równoważne.

Na podstawie powyższych uzasadnień dostajemy tabelę 2.

A	B	O	o	Ω	ω	Θ
$\lg^k n$	n^ϵ	tak	tak	nie	nie	nie
n^k	c^n	tak	tak	nie	nie	nie
$\sqrt{n}$	$n^{\sin n}$	nie	nie	nie	nie	nie
2^n	$2^{n/2}$	nie	nie	tak	tak	nie
$n^{\lg c}$	$c^{\lg n}$	tak	nie	tak	nie	tak
$\lg(n!)$	$\lg n^n$	tak	nie	tak	nie	tak

Tabela 2: Porównanie funkcji na podstawie rzędu asymptotycznego.

3-3. Porządkowanie ze względu na rząd wielkości funkcji

(a) Poniższe uzasadnienia stanowią dowody, że $g_i(n) = \Omega(g_{i+1}(n))$ dla $i = 1, 2, \ldots, 29$, gdzie $g_i(n)$ to rozważane kolejno funkcje. W niektórych dowodach korzystamy z obserwacji, że jeśli $f(n) = g(n)h(n)$ i $h(n) = \omega(1)$, to $f(n) = \omega(g(n))$. Ponadto wykorzystujemy fakt, że $\lg f(n) = \omega(\lg g(n))$ implikuje $f(n) = \omega(g(n))$, co można łatwo wykazać.

- $2^{2^{n+1}} = \Omega\big(2^{2^n}\big)$

  $$2^{2^{n+1}} = 2^{2^n} \cdot 2^{2^n} = \omega\big(2^{2^n}\big), \quad \text{bo } 2^{2^n} = \omega(1).$$

- $2^{2^n} = \Omega((n+1)!)$
  Logarytmując obie funkcje i wykorzystując wzór (3.18), otrzymujemy

  $$\lg 2^{2^n} = 2^n \quad \text{oraz} \quad \lg((n+1)!) = \Theta((n+1)\lg(n+1)) = \Theta(n \lg n).$$

  Ponieważ $2^n = \omega(n^2)$ i $n^2 = \omega(n \lg n)$, to mamy, że $2^n = \omega(n \lg n)$. Powracając do początkowych funkcji, dostajemy $2^{2^n} = \omega((n+1)!)$, skąd wynika prawdziwość dowodzonej zależności.

- $(n+1)! = \Omega(n!)$

  $$(n+1)! = (n+1) \cdot n! = \omega(n!), \quad \text{bo } n+1 = \omega(1).$$

- $n! = \Omega(e^n)$
  Zbadajmy logarytmy obu funkcji. Ze wzoru (3.18) otrzymujemy $\lg(n!) = \Theta(n \lg n) = \omega(n)$, a $\lg e^n = \Theta(n)$. Prawdą jest zatem, że $\lg(n!) = \omega(\lg e^n)$ i stąd wynika zależność $n! = \omega(e^n)$.

- $e^n = \Omega(n \cdot 2^n)$

  $$e^n = (e/2)^n 2^n = \omega(n \cdot 2^n), \quad \text{bo } (e/2)^n = \omega(n),$$

  ponieważ funkcje wykładnicze rosną szybciej niż wielomiany.

- $n \cdot 2^n = \Omega(2^n)$
  Tożsamość zachodzi, bo $n = \omega(1)$.

- $2^n = \Omega((3/2)^n)$

  $$2^n = (4/3)^n (3/2)^n = \omega((3/2)^n), \quad \text{bo } (4/3)^n = \omega(1).$$

- $(3/2)^n = \Omega\big(n^{\lg \lg n}\big)$
  Logarytmując obie funkcje, dostajemy

  $$\lg(3/2)^n = \Theta(n) \quad \text{oraz} \quad \lg n^{\lg \lg n} = \lg n \lg \lg n = o(\lg^2 n).$$

  Wystarczy pokazać, że $n = \omega(\lg^2 n)$. Po podstawieniu $n = 2^h$ wzór przyjmuje postać $2^h = \omega(h^2)$, co jest prawdą, ponieważ funkcja wykładnicza rośnie szybciej niż wielomian.

- $n^{\lg \lg n} = \Omega\big((\lg n)^{\lg n}\big)$
  Na mocy tożsamości (3.15) funkcje są równoważne.

- $(\lg n)^{\lg n} = \Omega((\lg n)!)$
  Jeśli podstawimy $n = 2^h$, to otrzymamy wzór $h^h = \Omega(h!)$, który jest prawdziwy na podstawie zależności $n! = o(n^n)$, zaprezentowanej w Podręczniku.

- $(\lg n)! = \Omega(n^3)$
  Korzystając z logarytmu pierwszej funkcji oszacowanego w poprzednim uzasadnieniu oraz z tego, że $\lg n^3 = \Theta(\lg n)$, dostajemy żądany wynik, ponieważ $\lg \lg n = \omega(1)$.

- $n^3 = \Omega(n^2)$
  Tożsamość zachodzi wprost z punktu (b) problemu 3-1.

- $n^2 = \Omega\big(4^{\lg n}\big)$
  Funkcje są tożsame – na podstawie wzoru (3.15) mamy $4^{\lg n} = n^{\lg 4} = n^2$.

- $4^{\lg n} = \Omega(n \lg n)$
  Wzór zachodzi, bo $4^{\lg n} = n^2$, a $n = \omega(\lg n)$.

- $n \lg n = \Omega(\lg(n!))$
  Obie funkcje są asymptotycznie równoważne na podstawie wzoru (3.18).

- $\lg(n!) = \Omega(n)$
  Ze wzoru (3.18) mamy, że $\lg(n!) = \Theta(n \lg n)$, więc tożsamość jest prawdziwa, bo $\lg n = \omega(1)$.

- $n = \Omega\big(2^{\lg n}\big)$
  Na mocy wzoru (3.15) zachodzi $2^{\lg n} = n^{\lg 2} = n$, a więc funkcje są tożsame.

- $2^{\lg n} = \Omega\big(\big(\sqrt{2}\big)^{\lg n}\big)$
  Z poprzedniego uzasadnienia mamy, że $2^{\lg n} = n$, a $\big(\sqrt{2}\big)^{\lg n} = n^{\lg \sqrt{2}} = \sqrt{n}$ (ze wzoru (3.15)), więc tożsamość zachodzi, ponieważ $\sqrt{n} = \omega(1)$.

- $\big(\sqrt{2}\big)^{\lg n} = \Omega\big(2^{\sqrt{2 \lg n}}\big)$
  Rozważmy tożsamość $2^{\lg n} = n$ i podnieśmy ją do potęgi $\sqrt{2/\lg n}$. Otrzymujemy $2^{\sqrt{2 \lg n}} = n^{\sqrt{2/\lg n}}$, a zatem $2^{\sqrt{2 \lg n}} = \Theta\big(n^{\sqrt{2/\lg n}}\big)$. Ponieważ $\big(\sqrt{2}\big)^{\lg n} = n^{1/2}$, to wystarczy pokazać, że $1/2 = \Omega\big(\sqrt{2/\lg n}\big)$. Wzór oczywiście zachodzi, gdyż funkcja z prawej strony jest malejąca i dąży do 0 wraz ze wzrostem n.

- $2^{\sqrt{2 \lg n}} = \Omega(\lg^2 n)$
  Biorąc logarytmy obu funkcji, dostajemy

$$\lg 2^{\sqrt{2 \lg n}} = \sqrt{2 \lg n} = \Theta\big(\lg^{1/2} n\big) \quad \text{oraz} \quad \lg \lg^2 n = \Theta(\lg \lg n).$$

  Pozostaje zatem zbadać prawdziwość wzoru $\lg^{1/2} n = \Omega(\lg \lg n)$. Przyjmując $n = 2^{4^h}$, sprowadzamy go do postaci $2^h = \Omega(h)$, co oczywiście zachodzi.

- $\lg^2 n = \Omega(\ln n)$
  Zależność jest prawdziwa, ponieważ $\ln n = \Theta(\lg n)$ oraz $\lg n = \omega(1)$.

- $\ln n = \Omega\big(\sqrt{\lg n}\big)$
  Wystarczy przyjąć $n = e^h$, aby otrzymać tożsamość $h = \Omega\big(\sqrt{h}\big)$, która zachodzi na mocy tego, że $\sqrt{h} = \omega(1)$.

- $\sqrt{\lg n} = \Omega(\ln \ln n)$
  Prawa strona jest identyczna z $\Omega(\lg \lg n)$, zatem przyjmując $n = 2^{4^h}$, dostajemy tożsamość $2^h = \Omega(h)$.

- $\ln \ln n = \Omega\big(2^{\lg^* n}\big)$
  Logarytmując funkcje, dostajemy

$$\lg \ln \ln n = \Theta\big(\lg^{(3)} n\big) \quad \text{oraz} \quad \lg\big(2^{\lg^* n}\big) = \lg^* n.$$

  By wykazać prawdziwość tożsamości, dokonajmy podstawienia

$$n = 2^{2^{\cdot^{\cdot^{\cdot^{2^\epsilon}}}}\Big\}k},$$

gdzie $0 < \epsilon \leqslant 1$, a $k \geqslant 3$ jest liczbą dwójek. Wyliczając wartości obu funkcji dla takiego argumentu, otrzymujemy

$$\lg^{(3)} n = 2^{2^{\cdot^{\cdot^{\cdot^{2^{\epsilon}}}}}\Big\}k-3} \quad \text{oraz} \quad \lg^* n = k.$$

Oczywistym jest, że funkcja po lewej stronie jest asymptotycznie większa od funkcji po prawej stronie.

- $2^{\lg^* n} = \Omega(\lg^* n)$
  Po zlogarytmowaniu obu funkcji i wykorzystaniu wzoru (3.15), otrzymujemy

  $$\lg 2^{\lg^* n} = \lg^* n \quad \text{oraz} \quad \lg \lg^* n.$$

  Biorąc $h = \lg^* n$, sprowadzamy tożsamość do udowodnionej wcześniej $h = \Omega(\lg h)$, a zatem dowodzona zależność jest spełniona.

- $\lg^* n = \Omega(\lg^* \lg n)$
  Funkcje są asymptotycznie równoważne, ponieważ $\lg^* \lg n = \lg^* n - 1$ dla $n \geqslant 2$.

- $\lg^* \lg n = \Omega(\lg \lg^* n)$
  Tożsamość zachodzi na podstawie rozwiązania zad. 3.2-5.

- $\lg \lg^* n = \Omega\big(n^{1/\lg n}\big)$
  Z własności logarytmów mamy, że $1/\lg n = \log_n 2$, a więc wykorzystując wzór (3.15), dostajemy $n^{1/\lg n} = n^{\log_n 2} = 2^{\log_n n} = 2 = \Theta(1)$, skąd wynika prawdziwość zależności.

- $n^{1/\lg n} = \Omega(1)$
  Tożsamość zachodzi, gdyż z poprzedniego uzasadnienia $n^{1/\lg n} = \Theta(1)$.

Tabela 3 przedstawia badane funkcje uporządkowane względem notacji Ω na podstawie powyższych dowodów.

$g_1(n) =$	$2^{2^{n+1}}$	$g_{11}(n) =$	$(\lg n)!$	$g_{21}(n) =$	$\lg^2 n$
$g_2(n) =$	2^{2^n}	$g_{12}(n) =$	n^3	$g_{22}(n) =$	$\ln n$
$g_3(n) =$	$(n+1)!$	$g_{13}(n) =$	n^2	$g_{23}(n) =$	$\sqrt{\lg n}$
$g_4(n) =$	$n!$	$g_{14}(n) =$	$4^{\lg n}$	$g_{24}(n) =$	$\ln \ln n$
$g_5(n) =$	e^n	$g_{15}(n) =$	$n \lg n$	$g_{25}(n) =$	$2^{\lg^* n}$
$g_6(n) =$	$n \cdot 2^n$	$g_{16}(n) =$	$\lg(n!)$	$g_{26}(n) =$	$\lg^* n$
$g_7(n) =$	2^n	$g_{17}(n) =$	n	$g_{27}(n) =$	$\lg^* \lg n$
$g_8(n) =$	$(3/2)^n$	$g_{18}(n) =$	$2^{\lg n}$	$g_{28}(n) =$	$\lg \lg^* n$
$g_9(n) =$	$n^{\lg \lg n}$	$g_{19}(n) =$	$\big(\sqrt{2}\big)^{\lg n}$	$g_{29}(n) =$	$n^{1/\lg n}$
$g_{10}(n) =$	$(\lg n)^{\lg n}$	$g_{20}(n) =$	$2^{\sqrt{2 \lg n}}$	$g_{30}(n) =$	1

Tabela 3: Uporządkowanie funkcji względem asymptotycznego tempa wzrostu. Funkcje znajdujące się w tej samej komórce są asymptotycznie równoważne.

(b) Oto przykład funkcji, która nie jest ani $O(g_i(n))$, ani $\Omega(g_i(n))$, gdzie $i = 1, 2, \ldots, 30$:

$$f(n) = \begin{cases} 2^{2^{n+2}}, & \text{jeśli } n \text{ jest parzyste,} \\ 0, & \text{jeśli } n \text{ jest nieparzyste.} \end{cases}$$

Gdyby rozważać funkcję $f(n)$ w dziedzinie liczb parzystych, to byłaby ona $\Omega(g_1(n))$, z kolei po obcięciu dziedziny do liczb nieparzystych, $f(n)$ byłoby na końcu listy funkcji w uporządkowaniu z poprzedniego punktu. Dlatego wraz ze wzrostem n, w zależności od jego parzystości, $f(n)$ jest asymptotycznie większe od wszystkich funkcji $g_i(n)$ albo od nich asymptotycznie mniejsze.

3-4. Własności notacji asymptotycznej

(a) Fałsz. Niech np. $f(n) = n$ i $g(n) = n^2$. Wtedy $f(n) = O(g(n))$, ale $g(n) \neq O(f(n))$.

(b) Fałsz. Jako kontrprzykład rozważmy $f(n) = n$ i $g(n) = n^2$. Dla $n \geqslant 1$ zachodzi wtedy

$$\min(f(n), g(n)) = f(n) \quad \text{oraz} \quad f(n) + g(n) = \Theta(g(n)) \neq \Theta(f(n)).$$

(c) Prawda. Z faktu, że $f(n) = O(g(n))$, wynika $f(n) \leqslant cg(n)$ dla $n \geqslant n_0$, gdzie c, $n_0 > 0$ są pewnymi stałymi. Z założenia, że $\lg g(n) \geqslant 1$ i $f(n) \geqslant 1$, otrzymujemy

$$0 \leqslant \lg f(n) \leqslant \lg c + \lg g(n) \leqslant \lg c \lg g(n) + \lg g(n) = (\lg c + 1) \lg g(n) = O(\lg g(n)).$$

(d) Fałsz. Niech $f(n) = 2n$ oraz $g(n) = n$. Zachodzi $f(n) = O(g(n))$, jednak $2^{f(n)} \neq O\big(2^{g(n)}\big)$ (z zad. 3.1-4).

(e) Fałsz. Dla $f(n) = 1/n$ mamy $f^2(n) = 1/n^2$. Ponieważ nie istnieje żadna dodatnia stała c spełniająca nierówność $1/n \leqslant c/n^2$ dla dowolnie dużych n, to stąd $f(n) \neq O\big(f^2(n)\big)$.

(f) Prawda. Z definicji notacji O, jeśli $f(n) = O(g(n))$, to istnieją stałe c, $n_0 > 0$, że dla każdego $n \geqslant n_0$ zachodzi $0 \leqslant f(n) \leqslant cg(n)$. Dzieląc nierówność przez c, otrzymujemy $0 \leqslant f(n)/c \leqslant g(n)$, przy czym $1/c > 0$, a więc $g(n) = \Omega(f(n))$.

(g) Fałsz. Niech np. $f(n) = 2^n$. Wtedy $f(n/2) = 2^{n/2} = \sqrt{2^n}$. Gdyby zachodziło $2^n = O\big(\sqrt{2^n}\big)$, to dla pewnej stałej $c > 0$ i odpowiednio dużych n mielibyśmy $2^n \leqslant c\sqrt{2^n}$. Ale wówczas $c \geqslant \sqrt{2^n}$ nie mogłoby być stałą.

(h) Prawda. Niech $h(n) = o(f(n))$. Wtedy, na podstawie definicji notacji o mamy, że dla każdej stałej $c > 0$ istnieje stała $n_0 > 0$ taka, że nierówności $0 \leqslant h(n) < cf(n)$ zachodzą dla wszystkich $n \geqslant n_0$. To znaczy, że

$$f(n) \leqslant f(n) + o(f(n)) = f(n) + h(n) < (c+1)f(n).$$

Ponieważ $c + 1 > 1$, to można wyrażenie $f(n) + o(f(n))$ ograniczyć od góry przez $c_2 f(n)$, wybierając np. $c_2 = 2$. Jego dolnym ograniczeniem jest $f(n)$, więc ustalamy $c_1 = 1$. Stałe c_1, c_2, n_0 spełniają założenia definicji notacji Θ, więc wnioskujemy, że $f(n) + o(f(n)) = \Theta(f(n))$.

3-5. Wariacje na temat notacji O i Ω

(a) Niech $c > 0$ będzie pewną stałą. Załóżmy, że $f(n) \geqslant cg(n)$ dla n z pewnego skończonego zbioru, w przeciwnym razie zachodzi bowiem $f(n) = \overset{\infty}{\Omega}(g(n))$. Ponieważ zbiór ten jest skończony,

to wybierzmy największe n, dla którego nierówność ta jest spełniona i oznaczmy je przez n_0. Mamy zatem $0 \leqslant f(n) < cg(n)$ dla $n \geqslant n_0 + 1$, czyli $f(n) = O(g(n))$.

Natomiast nie jest prawdą podobne twierdzenie, gdyby zastosować notację Ω zamiast $\overset{\infty}{\Omega}$ – jeśli np. $f(n) = n$ oraz $g(n) = n^{1+\sin n}$, to $f(n) = \overset{\infty}{\Omega}(g(n))$, ale $f(n) \neq \Omega(g(n))$ i $f(n) \neq O(g(n))$.

(b) Zaletą nowej notacji jest fakt, że jeśli o pewnej funkcji $f(n)$ wiemy, że nie jest klasy $O(g(n))$, to jest klasy $\overset{\infty}{\Omega}(g(n))$ (z poprzedniego punktu) – nie istnieją zatem funkcje, których nie da się porównać z innymi za pomocą pary notacji O i $\overset{\infty}{\Omega}$.

Niestety, jeśli $f(n) = \overset{\infty}{\Omega}(g(n))$, to funkcja $f(n)$ niekoniecznie dominuje nad funkcją $g(n)$, gdyż w nieskończenie wielu punktach może przyjmować wartości mniejsze od wartości $g(n)$. Nieskończenie wiele punktów, o których mowa w definicji $\overset{\infty}{\Omega}$, może też występować daleko ponad maksymalnym rozmiarem danych wejściowych algorytmu, którego czas działania opisujemy przy użyciu nowej notacji. Wady te sprawiają, że nowa notacja ma niewielkie zastosowanie praktyczne i jest użyteczna głównie w rozważaniach teoretycznych.

(c) W przypadku implikacji w lewą stronę z warunku $f(n) = \Theta(g(n))$ wynika, że $f(n) \geqslant 0$ od pewnego dodatniego n_0, a więc $f(n)$ jest funkcją asymptotycznie nieujemną i twierdzenie stosuje się bez zmian, czyli implikacja zachodzi.

Zbadajmy teraz implikację w prawo. Wprost z definicji, jeśli $f(n) = O'(g(n))$, to istnieją takie stałe c, $n_0 > 0$, że dla wszystkich $n \geqslant n_0$ zachodzi

$$\begin{cases} 0 \leqslant f(n) \leqslant cg(n), & \text{jeśli } f(n) \geqslant 0, \\ 0 \leqslant -f(n) \leqslant cg(n), & \text{jeśli } f(n) < 0. \end{cases}$$

Załóżmy, że istnieje nieskończenie wiele takich $n \geqslant n_0$, że $f(n) < 0$, ponieważ w przeciwnym przypadku mielibyśmy do czynienia z funkcją asymptotycznie nieujemną, dla której jest $f(n) = O(g(n))$. Teraz jednak nierówności $0 \leqslant c_1 g(n) \leqslant f(n)$ wynikające z założenia, że $f(n) = \Omega(g(n))$ nie są spełnione dla nieskończenie wielu $n \geqslant n_0$ i dowolnej stałej $c_1 > 0$. Wynika stąd wniosek, że jeśli spełnione są założenia, to funkcja $f(n)$ jest asymptotycznie nieujemna i implikacja stosuje się bez zmian.

Widać, że zastosowanie notacji O' w miejsce O nie pozbawia prawdziwości twierdzenia 3.1.

(d) Oto definicje notacji $\widetilde{\Omega}$ i $\widetilde{\Theta}$:

$$\begin{aligned} \widetilde{\Omega}(g(n)) = \big\{ & f(n) : \text{istnieją dodatnie stałe } c,\ k,\ n_0 \text{ takie, że} \\ & 0 \leqslant cg(n) \lg^k n \leqslant f(n) \text{ dla wszystkich } n \geqslant n_0 \big\}, \\ \widetilde{\Theta}(g(n)) = \big\{ & f(n) : \text{istnieją dodatnie stałe } c_1,\ c_2,\ k_1,\ k_2,\ n_0 \text{ takie, że} \\ & 0 \leqslant c_1 g(n) \lg^{k_1} n \leqslant f(n) \leqslant c_2 g(n) \lg^{k_2} n \text{ dla wszystkich } n \geqslant n_0 \big\}. \end{aligned}$$

Dowód twierdzenia 3.1 dla notacji $\widetilde{O}$, $\widetilde{\Omega}$ i $\widetilde{\Theta}$ przebiega analogicznie do dowodu jego oryginalnego odpowiednika przeprowadzonego w zad. 3.1-5. Z definicji notacji $\widetilde{\Theta}$ mamy, że $f(n) = \widetilde{\Theta}(g(n))$ wtedy i tylko wtedy, gdy istnieją takie stałe c_1, c_2, k_1, k_2, $n_0 > 0$, że nierówności

$$0 \leqslant c_1 g(n) \lg^{k_1} n \leqslant f(n) \leqslant c_2 g(n) \lg^{k_2} n$$

zachodzą dla wszystkich $n \geqslant n_0$. Zapiszmy je w postaci układu

$$\begin{cases} 0 \leqslant c_1 g(n) \lg^{k_1} n \leqslant f(n) \\ 0 \leqslant f(n) \leqslant c_2 g(n) \lg^{k_2} n \end{cases}.$$

Z pierwszej nierówności mamy, że $f(n) = \widetilde{\Omega}(g(n))$, a z drugiej, że $f(n) = \widetilde{O}(g(n))$.

3-6. Funkcje iterowane

W każdym punkcie tego problemu za dziedzinę funkcji $f_c^*(n)$ przyjęto dziedzinę $f(n)$. W celu wyznaczenia $f_c^*(n)$ szukamy najmniejszego $i \geqslant 0$, dla którego zachodzi $f^{(i)}(n) \leqslant c$.

(a) *Rozwiązanie dotyczy przykładu z tekstu oryginalnego, w którym* $f(n) = n - 1$ *oraz* $c = 0$.

Ponieważ $(n-1)^{(i)} \equiv n - i$, to otrzymujemy, że $f_0^*(n) = \max(0, \lceil n \rceil)$ i oszacowaniem dokładnym jest $f_0^*(n) = \Theta(n)$.

(b) *Rozwiązanie dotyczy przykładu z tekstu oryginalnego, w którym* $f(n) = \lg n$ *oraz* $c = 1$.

Z definicji logarytmu iterowanego mamy $f_1^*(n) = \lg^* n = \Theta(\lg^* n)$.

(c) Oczywiście $(n/2)^{(i)} \equiv n/2^i$, więc:

$$f_1^*(n) = \begin{cases} 0, & \text{jeśli } n \leqslant 1, \\ \lceil \lg n \rceil, & \text{jeśli } n > 1. \end{cases}$$

Oszacowanie dokładne: $f_1^*(n) = \Theta(\lg n)$.

(d) Korzystając z poprzedniego punktu, ale dla $c = 2$, mamy:

$$f_2^*(n) = \begin{cases} 0, & \text{jeśli } n \leqslant 2, \\ \lceil \lg n \rceil - 1, & \text{jeśli } n > 2. \end{cases}$$

Oszacowanie dokładne: $f_2^*(n) = \Theta(\lg n)$.

(e) Ponieważ $\sqrt{n} \equiv n^{1/2}$, to mamy $\left(\sqrt{n}\right)^{(i)} \equiv n^{1/2^i}$. Jeśli $n > 2$, to rozwiązaniem nierówności $n^{1/2^i} \leqslant 2$ ze względu na i jest $i \geqslant \lg \lg n$, skąd dostajemy następujący wynik:

$$f_2^*(n) = \begin{cases} 0, & \text{jeśli } 0 \leqslant n \leqslant 2, \\ \lceil \lg \lg n \rceil, & \text{jeśli } n > 2. \end{cases}$$

Oszacowanie dokładne: $f_2^*(n) = \Theta(\lg \lg n)$.

(f) Bieżący punkt różni się od poprzedniego jedynie stałą c, jednak ta zmiana mocno wpływa na postać funkcji $f_c^*(n)$. Jeśli $0 \leqslant n \leqslant 1$, to $f_1^*(n) = 0$, a jeśli $n > 1$, to wartości tej funkcji są nieokreślone, ponieważ dla każdego całkowitego $i \geqslant 0$, $\left(\sqrt{n}\right)^{(i)} > 1$.

(g) Postępowanie jest analogiczne do tego z punktu (e). Mamy $\left(n^{1/3}\right)^{(i)} \equiv n^{1/3^i}$, a stąd:

$$f_2^*(n) = \begin{cases} 0, & \text{jeśli } n \leqslant 2, \\ \lceil \log_3 \lg n \rceil, & \text{jeśli } n > 2. \end{cases}$$

Oszacowanie dokładne: $f_2^*(n) = \Theta(\lg \lg n)$.

(h) Kolejne iteracje funkcji $f(n) = n/\lg n$ mają zbyt skomplikowaną postać, aby można było je badać podobnie jak w poprzednich punktach. Dlatego naszą analizę przeprowadzimy dla oszacowań górnego i dolnego funkcji $f(n)$. Niech $g(n)$ i $h(n)$ będą funkcjami monotonicznie rosnącymi. Dla $n \geqslant c$, jeśli $f(n) \leqslant g(n)$ oraz $f_c^*(n)$ i $g_c^*(n)$ są dobrze określone, to $f_c^*(n) \leqslant g_c^*(n)$, gdyż argument w kolejnych iteracjach funkcji $f(n)$ maleje szybciej niż w iteracjach funkcji $g(n)$. Analogiczny wniosek można uzyskać w przypadku, gdy $f(n) \geqslant h(n)$.

Jeśli $n \geqslant 4$, to zachodzi $n/\lg n \leqslant n/2$. Niech $g(n) = n/2$. Wówczas z punktu (d) mamy, że $g_4^*(n) = g_2^*(n) - 1 = \Theta(\lg n)$, a zatem $f_2^*(n) = f_4^*(n) + 1 \leqslant g_4^*(n) + 1 = O(\lg n)$.

Jeśli z kolei $n \geqslant 16$, to prawdziwa jest nierówność $n/\lg n \geqslant \sqrt{n}$. Biorąc $h(n) = \sqrt{n}$ i wykorzystując wynik z punktu (e), otrzymujemy $h_{16}^*(n) = h_2^*(n) - 2 = \Theta(\lg \lg n)$, a zatem $f_2^*(n) = f_{16}^*(n) + 2 \geqslant h_{16}^*(n) + 2 = \Omega(\lg \lg n)$.

Tempo wzrostu funkcji $f_2^*(n)$ znajduje się zatem pomiędzy $\Omega(\lg \lg n)$ a $O(\lg n)$. Niestety na podstawie przeprowadzonej analizy nie można podać dokładniejszego oszacowania.

Rozdział **4**

Rekurencje

4.1. Metoda podstawiania

4.1-1. Niech $c > 0$ będzie stałą. Przyjmujemy założenie, że

$$T(\lceil n/2 \rceil) \leqslant c \lg \lceil n/2 \rceil$$

i że $n \geqslant 4$. Na podstawie założeń i wzoru (3.3) otrzymujemy:

$$\begin{aligned} T(n) &\leqslant c \lg \lceil n/2 \rceil + 1 \\ &< c \lg(n/2 + 1) + 1 \\ &\leqslant c \lg(3n/4) + 1 \\ &= c \lg n + c \lg(3/4) + 1 \\ &\leqslant c \lg n, \end{aligned}$$

przy czym ostatnia nierówność jest spełniona, gdy $c \geqslant \log_{4/3} 2$.

Niech $T(1) = 1$. Wówczas $T(2) = 2$, $T(3) = 3$. Nierówności $T(2) \leqslant c \lg 2$ oraz $T(3) \leqslant c \lg 3$ zachodzą dla $c \geqslant 2$, a więc w szczególności dla $c \geqslant \log_{4/3} 2$. Można zatem przyjąć $n = 2$ i $n = 3$ za podstawę indukcji, gdyż dla $n \geqslant 4$ rekurencja nie zależy bezpośrednio od $T(1)$. Na mocy dowodu indukcyjnego wynika zatem, że $T(n) = O(\lg n)$.

4.1-2. Przyjmijmy założenie, że

$$T(\lfloor n/2 \rfloor) \geqslant c \lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor$$

dla pewnej dodatniej stałej c. Korzystając ze wzoru (3.3), mamy:

$$\begin{aligned} T(n) &\geqslant 2c \lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor + n \\ &> 2c(n/2 - 1) \lg(n/4) + n \\ &= 2c((n/2) \lg n - n - \lg n + 2) + n \\ &= cn \lg n - 2cn - 2c \lg n + 4c + n \\ &> cn \lg n - 2c(n + \lg n) + n \\ &\geqslant cn \lg n. \end{aligned}$$

Ostatni krok uzasadniamy, rozwiązując nierówność $-2c(n + \lg n) + n \geqslant 0$ ze względu na c:

$$c \leqslant \frac{n}{2(n + \lg n)} \leqslant \frac{n}{2n} = \frac{1}{2}.$$

Wybierając dowolne $0 < c \leqslant 1/2$, spełniamy ostatnią nierówność wyprowadzenia. Można przyjąć $T(1) = 1$ i $n = 1$ za podstawę indukcji, bo wówczas $T(1) \geqslant c \cdot 1 \cdot \lg 1 = 0$. Wykazaliśmy, że $T(n) = \Omega(n \lg n)$, więc na mocy tw. 3.1 mamy $T(n) = \Theta(n \lg n)$.

4.1-3. Udowodnimy, że $T(n) = O(n^2)$. Przyjmijmy w tym celu założenie dla $n \geqslant 2$, że

$$T(\lfloor n/2 \rfloor) \leqslant c \lfloor n/2 \rfloor^2,$$

gdzie $c > 0$ jest stałą. Mamy teraz

$$T(n) \leqslant 2c \lfloor n/2 \rfloor^2 + n \leqslant 2cn^2/4 + n = cn^2/2 + n \leqslant cn^2,$$

co jest prawdą, o ile $c \geqslant 1$. Definiując $T(1) = 1$, dzięki mocniejszemu założeniu indukcyjnemu można przyjąć $n = 1$ w podstawie indukcji, bo $T(1) \leqslant c \cdot 1^2 = c$.

4.1-4. Rekurencję (4.2) można przedstawić w alternatywny sposób:

$$T(n) = \begin{cases} d_1, & \text{jeśli } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + d_2 n, & \text{jeśli } n > 1, \end{cases}$$

gdzie $d_1, d_2 > 0$ są stałymi. Udowodnimy oszacowania górne i dolne dla $T(n)$ przy użyciu indukcji, przy czym założymy, że $n \geqslant 4$ i skorzystamy z obserwacji, że $T(n)$ jest funkcją niemalejącą. Dla stałych c_1, $c_2 > 0$ przyjmijmy założenia

$$T(\lfloor n/2 \rfloor) \geqslant c_1 \lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor \quad \text{oraz} \quad T(\lceil n/2 \rceil) \leqslant c_2 \lceil n/2 \rceil \lg \lceil n/2 \rceil.$$

W przypadku dolnego oszacowania mamy:

$$\begin{aligned} T(n) &\geqslant 2T(\lfloor n/2 \rfloor) + d_2 n \\ &\geqslant 2c_1 \lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor + d_2 n \\ &> 2c_1 (n/2 - 1) \lg(n/4) + d_2 n \\ &= c_1 n \lg n - 2c_1 n - 2c_1 \lg(n/4) + d_2 n \\ &\geqslant c_1 n \lg n, \end{aligned}$$

ponieważ zachodzi $\lfloor n/2 \rfloor > n/2 - 1$ i $\lfloor n/2 \rfloor \geqslant n/4$ oraz można tak dobrać stałą c_1, aby prawdziwa była ostatnia nierówność powyższego wyprowadzenia. Po podstawieniu $c_1 = d_2/4$ sprowadza się ona do postaci $n \geqslant \lg(n/4)$, co jest prawdą dla wszystkich n dodatnich. Podstawę indukcji stanowi $n = 1$, gdyż $T(1) \geqslant c_1 \cdot 1 \cdot \lg 1 = 0$, a zatem $T(n) = \Omega(n \lg n)$.

Wykorzystując nierówność $\lceil n/2 \rceil < n/2 + 1$, dowodzimy górnego oszacowania:

$$\begin{aligned} T(n) &\leqslant 2T(\lceil n/2 \rceil) + d_2 n \\ &\leqslant 2c_2 \lceil n/2 \rceil \lg \lceil n/2 \rceil + d_2 n \\ &< 2c_2 (n/2 + 1) \lg \frac{n \lceil n/2 \rceil}{n} + d_2 n \\ &\leqslant c_2 n \lg n + c_2 (n + 2) \lg \frac{\lceil n/2 \rceil}{n} + 2c_2 \lg n + d_2 n \\ &\leqslant c_2 n \lg n. \end{aligned}$$

W ostatniej nierówności skorzystano z faktu, że dla $n \geqslant 2$ wyrażenie $\lg \frac{\lceil n/2 \rceil}{n}$ jest ujemne, a zatem, dobierając odpowiednio duże c_2, można uzasadnić nierówność, gdyż funkcja liniowa rośnie

szybciej od logarytmicznej. Dokładniej, okazuje się, że przyjęcie $c_2 \geqslant 10d_2$ wystarcza, aby spełnić nierówność dla $n \geqslant 4$.

Ponieważ dla $n \geqslant 4$ rekurencja nie zależy bezpośrednio od $T(1)$, to za podstawę indukcji można przyjąć $n = 2$ i $n = 3$. Mamy $T(2) = 2d_1 + 2d_2$ oraz $T(3) = 3d_1 + 5d_2$. Łatwo sprawdzić, że otrzymane oszacowanie zachodzi dla tych dwóch wartości, o ile d_1 jest dostatecznie małe. W przeciwnym przypadku górne oszacowanie może nie być wystarczające, jednak zwiększenie c_2 pozwala na dowolne ograniczenie rekurencji od góry w zależności od wartości stałych d_1 i d_2. Niezależnie od ich doboru oszacowaniem górnym rekurencji $T(n)$ jest $O(n \lg n)$, co na mocy wcześniejszego wyniku dolnego oszacowania implikuje $T(n) = \Theta(n \lg n)$.

4.1-5. Wykorzystując założenie

$$T(\lfloor n/2 \rfloor + 17) \leqslant c(\lfloor n/2 \rfloor + 17) \lg(\lfloor n/2 \rfloor + 17)$$

dla pewnej stałej $c > 0$, otrzymujemy:

$$\begin{aligned} T(n) &\leqslant 2c(\lfloor n/2 \rfloor + 17) \lg(\lfloor n/2 \rfloor + 17) + n \\ &\leqslant 2c(n/2 + 17) \lg(n/2 + 17) + n \\ &\leqslant c(n + 34) \lg(11n/20) + n \\ &< cn \lg n + cn \lg(11/20) + 34c \lg n + n \\ &\leqslant cn \lg n. \end{aligned}$$

Nierówności zachodzą, o ile $n/2 + 17 \leqslant 11n/20$, czyli $n \geqslant 340$, oraz

$$cn \lg(11/20) + 34c \lg n + n \leqslant 0.$$

Badając ostatnią nierówność, można dojść do rezultatu, że przyjęcie $c \geqslant 47$ wystarcza, aby spełnić ją dla wszystkich $n \geqslant n_0$, gdzie $n_0 = 340$.

Zauważmy, że stosowanie wzoru $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ dla $n \leqslant 34$ nie ma sensu, bo wtedy $T(n)$ nie zależy od niższych wyrazów. Przyjmijmy zatem, że $T(n) = 1$ dla wszystkich $n \leqslant 34$ i niech stanowi to przypadek brzegowy rekurencji. Za podstawę indukcji musimy wtedy przyjąć wszystkie $n = 187, 188, \ldots, 339$. Można pokazać, że dla $c \geqslant 47$ wszystkie te wartości spełniają oszacowanie, a zatem $T(n) = O(n \lg n)$.

Analiza tej rekurencji z każdą inną stałą w miejscu 17 przebiega analogicznie, zmianie ulegają natomiast wartości n_0 i c oraz wartości brzegowe rekursji i podstawa indukcji, jednak w każdym takim przypadku rekurencja jest klasy $O(n \lg n)$.

4.1-6. Przyjmijmy, że $n = 2^m$, skąd $m = \lg n$. Rekurencja przyjmuje teraz postać

$$T(2^m) = 2T(2^{m/2}) + 1.$$

Z kolei podstawiając $S(m)$ za $T(2^m)$, dostajemy nową rekurencję

$$S(m) = 2S(m/2) + 1,$$

dla której udowodnimy rozwiązanie $\Theta(\lg m)$.

Ponieważ całkowitość argumentów nie jest dla nas istotna, to możemy rekurencję potraktować jako $S(m) = S(\lfloor m/2 \rfloor) + S(\lceil m/2 \rceil) + 1$, co, według Podręcznika, jest $O(m)$.

Aby uzyskać oszacowanie dokładne, pozostaje udowodnić, że $S(m) = \Omega(m)$. Przyjmujemy zatem założenie, że $S(m/2) \geqslant cm/2$ dla $c > 0$ i na jego podstawie otrzymujemy:

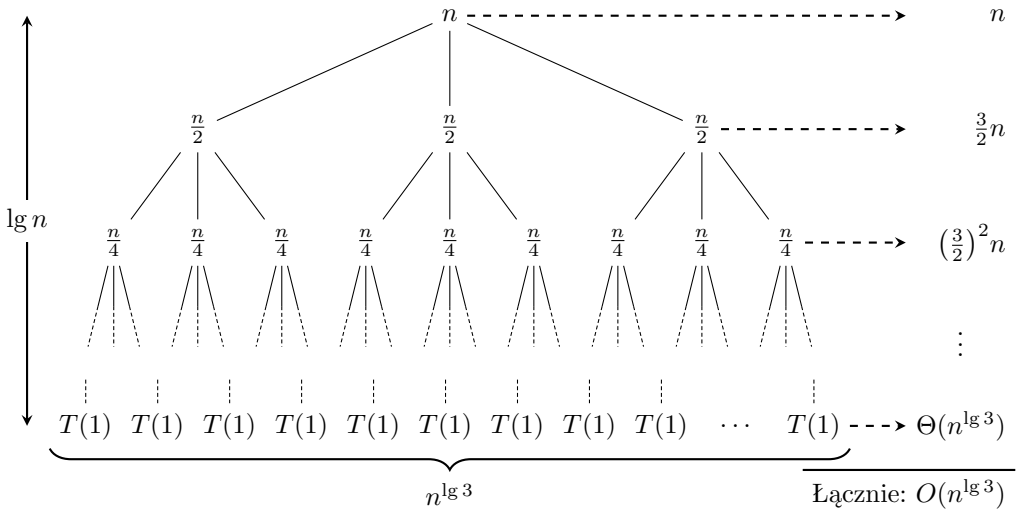
$$S(m) \geqslant 2cm/2 + 1 = cm + 1 > cm,$$

co zachodzi dla dowolnej wartości c. Niech $S(1) = 1$. Przyjęcie $m = 1$ na podstawę indukcji wystarcza, bo warunek $S(1) \geqslant c$ spełnia każde $c \leqslant 1$, a zatem $S(m) = \Omega(m)$.

Stosując tw. 3.1, mamy $S(m) = \Theta(m)$. Wracając teraz do oryginalnej rekurencji i starej zmiennej, dostajemy $T(n) = T(2^m) = S(m) = \Theta(m) = \Theta(\lg n)$.

4.2. Metoda drzewa rekursji

4.2-1. Dokonamy pewnego uproszczenia, pomijając podłogę w argumencie rekurencji i rozważając zależność $T(n) = 3T(n/2) + n$. Drzewo tej rekursji przedstawione na rys. 3 ma wysokość



Rysunek 3: Drzewo rekursji dla równania rekurencyjnego $T(n) = 3T(n/2) + n$.

równą $\lg n$, czyli jest w nim $\lg n + 1$ poziomów. Na i-tym poziomie znajduje się 3^i węzłów, zatem jest $n^{\lg 3}$ liści. Koszt węzła na poziomie i wynosi $n/2^i$, skąd wynika, że łączny koszt wszystkich węzłów na i-tej głębokości jest równy $(3/2)^i n$. Przyjmujemy, że $T(1) = 1$, a więc na ostatnim poziomie jest $\Theta(n^{\lg 3})$. Na podstawie tych wartości rozwiązujemy rekurencję:

$$\begin{aligned} T(n) &= n + \frac{3}{2}n + \left(\frac{3}{2}\right)^2 n + \cdots + \left(\frac{3}{2}\right)^{\lg n - 1} n + \Theta(n^{\lg 3}) \\ &= \sum_{i=0}^{\lg n - 1} \left(\frac{3}{2}\right)^i n + \Theta(n^{\lg 3}) \\ &= \frac{(3/2)^{\lg n} - 1}{(3/2) - 1}\, n + \Theta(n^{\lg 3}) \\ &= 2n(n^{\lg 3 - 1} - 1) + \Theta(n^{\lg 3}) \\ &= O(n^{\lg 3}). \end{aligned}$$

Pokażemy teraz, że otrzymany wynik stanowi górne oszacowanie dla oryginalnej rekurencji. Przyjmujemy założenie, że

$$T(\lfloor n/2 \rfloor) \leqslant c \lfloor n/2 \rfloor^{\lg 3}$$

dla pewnej stałej $c > 0$. Dowodzimy oszacowania dla oryginalnej rekurencji metodą podstawiania:

$$T(n) \leqslant 3c\lfloor n/2 \rfloor^{\lg 3} + n \leqslant 3c(n/2)^{\lg 3} + n = cn^{\lg 3} + n.$$

Nie możemy jednak na podstawie tego wyniku wywnioskować szukanego oszacowania. Wzmocnijmy zatem nasze założenie, niech

$$T(\lfloor n/2 \rfloor) \leqslant c\lfloor n/2 \rfloor^{\lg 3} - b\lfloor n/2 \rfloor,$$

gdzie $b \geqslant 0$ jest nową stałą. Korzystając ze wzoru (3.3), mamy teraz:

$$\begin{aligned} T(n) &\leqslant 3c\lfloor n/2 \rfloor^{\lg 3} - 3b\lfloor n/2 \rfloor + n \\ &< 3c(n/2)^{\lg 3} - 3b(n/2 - 1) + n \\ &= cn^{\lg 3} - 3bn/2 + 3b + n \\ &\leqslant cn^{\lg 3} - bn, \end{aligned}$$

co zachodzi dla $n \geqslant 7$, o ile $b \geqslant 14$. Przyjmujemy wszystkie $n = 3, 4, 5, 6$ na podstawę indukcji, ponieważ dla dowolnie ustalonego b można dobrać dla stałej c odpowiednią wartość tak, aby pokazane oszacowanie zachodziło dla takich n. To kończy dowód faktu, że górnym oszacowaniem rekurencji $T(n)$ jest $O(n^{\lg 3})$.

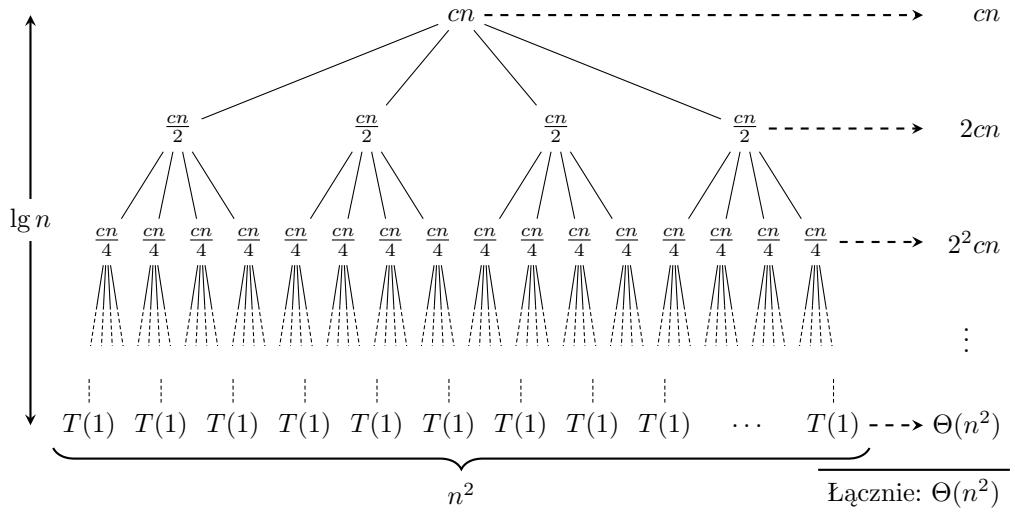
4.2-2. Drzewo rekursji $T(n)$ nie jest pełnym drzewem binarnym. Najkrótszą ścieżką od korzenia do liścia jest $cn \to cn/3 \to cn/9 \to \cdots \to cn/3^i \to \cdots \to T(1)$. Liść tej gałęzi znajduje się na poziomie $h = \log_3 n$. Gdybyśmy pominęli wszystkie poziomy tego drzewa poniżej h-tego, to uzyskalibyśmy pełne drzewo binarne o wysokości h i łącznym koszcie $\Theta(n \lg n)$. Ale tak zmodyfikowane drzewo jest mniejsze niż drzewo rekursji $T(n)$, przez co jego koszt stanowi oszacowanie dolne rekurencji $T(n)$, czyli $T(n) = \Omega(n \lg n)$.

4.2-3. W metodzie drzewa rekursji dla uproszczenia pominiemy branie części całkowitych. W drzewie rekursji z rys. 4 na i-tym poziomie jest 4^i węzłów, z których każdy wnosi koszt równy $cn/2^i$. Stąd kosztem całego poziomu jest $2^i cn$. Współczynnik przy n w koszcie węzła maleje dwukrotnie wraz z głębokością drzewa, więc jego wysokością jest $\lg n$. Wnioskujemy zatem, że liczbą liści w tym drzewie jest $4^{\lg n} = n^2$ i że koszt ostatniego poziomu wynosi $\Theta(n^2)$. Sumując koszty z każdego poziomu, otrzymujemy:

$$\begin{aligned} T(n) &= cn + 2cn + 2^2 cn + \cdots + 2^{\lg n - 1} cn + \Theta(n^2) \\ &= cn \sum_{i=0}^{\lg n - 1} 2^i + \Theta(n^2) \\ &= cn(2^{\lg n} - 1) + \Theta(n^2) \\ &= cn(n - 1) + \Theta(n^2) \\ &= \Theta(n^2). \end{aligned}$$

Sprawdzimy teraz otrzymany wynik, używając w tym celu metody podstawiania dla oryginalnej rekurencji. Przyjmiemy ponadto, że $T(1) = 1$ stanowi jej przypadek brzegowy. Badamy najpierw oszacowanie dolne $T(n)$, wykorzystując założenie

$$T(\lfloor n/2 \rfloor) \geqslant c_1 \lfloor n/2 \rfloor^2$$



Rysunek 4: Drzewo rekursji dla równania rekurencyjnego $T(n) = 4T(n/2) + cn$.

dla pewnej stałej $c_1 > 0$. Stąd

$$T(n) \geqslant 4c_1\lfloor n/2\rfloor^2 + cn > 4c_1(n/2-1)^2 + cn = c_1n^2 - 4c_1n + 4c_1 + cn \geqslant c_1n^2,$$

co jest prawdą dla $n \geqslant 2$, jeśli przyjmiemy, że $c_1 \leqslant c/4$. Podstawą indukcji jest $n = 1$, bo $T(1)$ spełnia oszacowanie dla $c_1 \leqslant 1$. A więc istotnie $T(n) = \Omega(n^2)$.

Pokażemy teraz, że $T(n) = O(n^2)$. W tym celu można przyjąć analogiczne założenie indukcyjne jak przy dowodzie dolnego oszacowania. Niestety założenie to okazuje się zbyt słabe i nie prowadzi do żądanego wyniku. Przyjmijmy zatem, że dla stałych $c_2 > 0$ i $c_3 \geqslant 0$ zachodzi mocniejszy warunek

$$T(\lfloor n/2\rfloor) \leqslant c_2\lfloor n/2\rfloor^2 - c_3\lfloor n/2\rfloor.$$

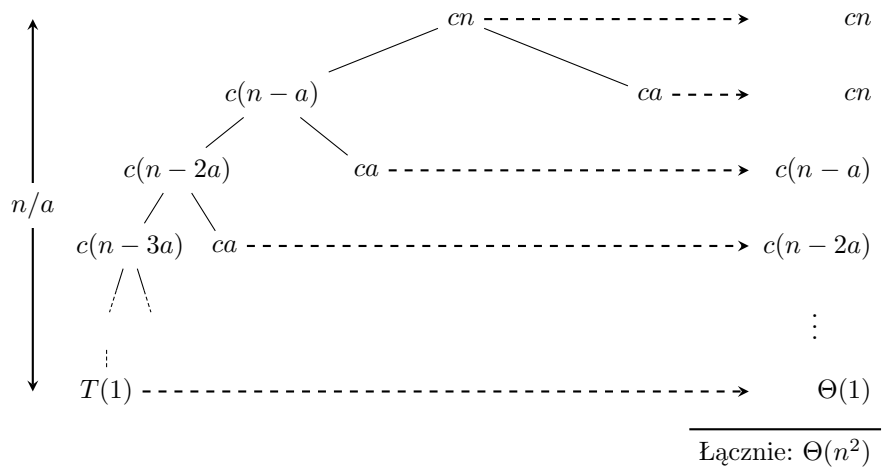
Wówczas:

$$\begin{aligned} T(n) &\leqslant 4(c_2\lfloor n/2\rfloor^2 - c_3\lfloor n/2\rfloor) + cn \\ &< 4(c_2(n/2)^2 - c_3(n/2-1)) + cn \\ &= c_2n^2 - 2c_3n + 4c_3 + cn \\ &\leqslant c_2n^2 - c_3n. \end{aligned}$$

Ostatnia nierówność jest prawdziwa dla $n \geqslant 5$, o ile $c_3 \geqslant 5c$.

Zajmiemy się teraz ustaleniem stałych c_2 i c_3, aby spełnić podstawę tej indukcji. Początkowe wyrazy rekurencji wynoszą $T(1) = 1$, $T(2) = 4 + 2c$, $T(3) = 4 + 3c$, $T(4) = 16 + 12c$. Jeśli przyjmiemy $c_3 = 5c$, to dla dowolnego $c_2 \geqslant 5c + 1$ wyrażenie $c_2n^2 - c_3n$ będzie stanowić oszacowanie górne dla tych wartości. Zachodzi zatem $T(n) = O(n^2)$ i dowód dokładnego oszacowania na $T(n)$ jest zakończony.

4.2-4. Dla uproszczenia przyjmijmy, że $T(n) = ca$ w przypadku, gdy $n \leqslant a$, czyli że dla dostatecznie małego n rekurencja przyjmuje wartość stałą równą ca. Drzewo rekursji $T(n)$ zostało zilustrowane na rys. 5. Wysokością tego drzewa jest n/a. Koszt wnoszony przez i-ty poziom



Rysunek 5: Drzewo rekursji dla równania rekurencyjnego $T(n) = T(n-a) + T(a) + cn$.

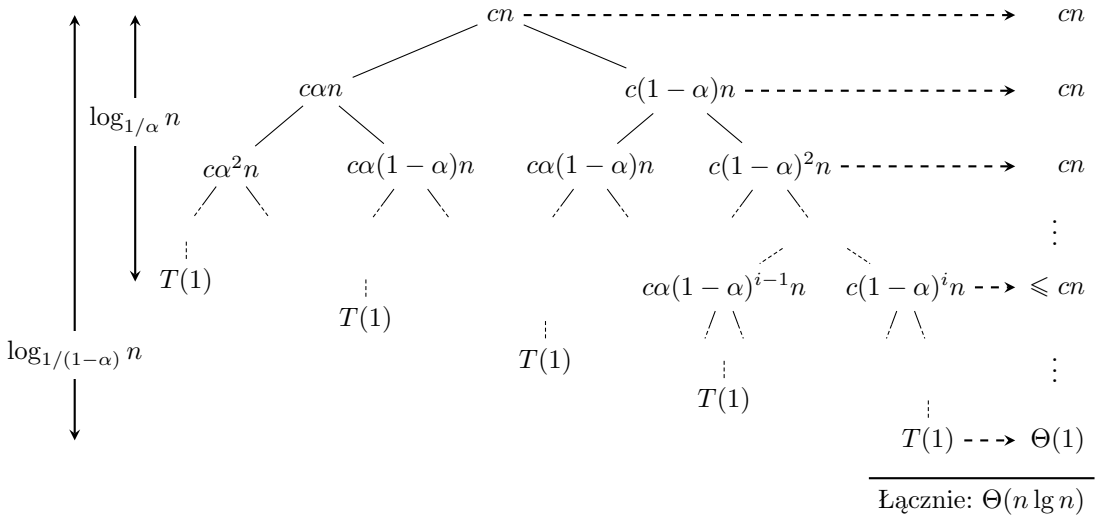
(z wyjątkiem zerowego i ostatniego) wynosi $c(n-a(i-1))$, przy czym na ostatnim poziomie jest tylko jeden liść, który kosztuje $\Theta(1)$. Mamy zatem:

$$\begin{aligned} T(n) &= cn + \sum_{i=1}^{n/a-1} c(n-a(i-1)) + \Theta(1) \\ &= cn + c\sum_{i=0}^{n/a-2} (n-ai) + \Theta(1) \\ &= cn + cn\sum_{i=0}^{n/a-2} 1 - ca\sum_{i=0}^{n/a-2} i + \Theta(1) \\ &= cn + cn(n/a-1) - \frac{ca(n/a-2)(n/a-1)}{2} + \Theta(1) \\ &= \Theta(n^2). \end{aligned}$$

4.2-5. Zauważmy, że drzewo rekurencji $T(n)$ z rys. 6 dla parametru α jest symetryczne do tego samego drzewa z parametrem $1-\alpha$ – przyjmijmy więc, że $0 < \alpha \leqslant 1/2$.

Wyznaczmy najpierw oszacowanie dolne rekurencji. Na i-tym poziomie drzewa najmniejszy koszt wnoszą elementy o wartościach $c\alpha^i n$, a więc węzły ze skrajnie lewej gałęzi. Najgłębszym poziomem o komplecie węzłów w tym drzewie jest ten, na którym element ze skrajnie lewej gałęzi drzewa osiąga wartość stałą $d > 0$. Oznaczmy przez h głębokość tego poziomu. Wtedy $c\alpha^h n = d$, skąd otrzymujemy $h = \log_{1/\alpha}(cn/d)$. Ponieważ α jest stałe, to $h = \Theta(\lg n)$. Sumując koszty węzłów drzewa $T(n)$ znajdujące się na poziomach wyższych niż h-ty, uzyskujemy oszacowanie dolne rekurencji, które wynosi $T(n) \geqslant cnh = \Omega(n \lg n)$.

Badając teraz skrajnie prawą gałąź, której elementy wnoszą największy koszt na każdym poziomie, możemy dojść do oszacowania górnego dla $T(n)$. Na głębokości H równej wysokości drzewa mamy $c(1-\alpha)^H n = d$, skąd $H = \log_{1/(1-\alpha)}(cn/d) = \Theta(\lg n)$, a więc $T(n) \leqslant cn(H+1) = O(n \lg n)$. Asymptotycznie dokładnym oszacowaniem rekurencji jest zatem $\Theta(n \lg n)$.



Rysunek 6: Drzewo rekursji dla równania rekurencyjnego $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$, gdzie $0 < \alpha \leqslant 1/2$.

4.3. Metoda rekurencji uniwersalnej

4.3-1.

(a) We wzorze (4.5) przyjmujemy $a = 4$, $b = 2$ oraz $f(n) = n$. Ponieważ $f(n) = O(n^{2-\epsilon})$ dla $\epsilon \leqslant 1$, to z tw. o rekurencji uniwersalnej mamy $T(n) = \Theta(n^2)$.

(b) Tutaj mamy te same wartości a i b jak w poprzednim punkcie, ale $f(n) = n^2$. Z tego, że $f(n) = \Theta(n^2)$, dostajemy $T(n) = \Theta(n^2 \lg n)$.

(c) Dla tych samych a i b oraz $f(n) = n^3$ mamy $f(n) = \Omega(n^{2+\epsilon})$, gdzie $\epsilon \leqslant 1$ oraz

$$4f(n/2) = 4n^3/8 = n^3/2 = f(n)/2 \leqslant cf(n),$$

o ile $c \geqslant 1/2$. A zatem warunek regularności jest spełniony i $T(n) = \Theta(n^3)$.

4.3-2. Rozwiążmy $T(n)$, korzystając z metody rekurencji uniwersalnej. Stosujemy przypadek 1 tw. 4.1 i stwierdzamy, że $n^2 = O(n^{\lg 7 - \epsilon})$ dla $\epsilon \leqslant \lg 7 - 2$, a więc zachodzi $T(n) = \Theta(n^{\lg 7})$.

Pozostaje teraz zbadanie nierówności $T'(n) < T(n)$ w zależności od parametru a, bo w rekurencji $T'(n)$ jest $n^{\log_b a} = n^{\log_4 a}$ oraz $f(n) = n^2$. Załóżmy, że dla pewnej stałej $\epsilon > 0$, $f(n) = O(n^{\log_4 a - \epsilon})$, co jest prawdą dla $a > 16$ i wtedy $T'(n) = \Theta(n^{\log_4 a})$. Algorytm A' jest efektywniejszy od algorytmu A, gdy $\log_4 a < \lg 7$, skąd $16 < a < 49$. Pozostałe przypadki tw. 4.1 można stosować, o ile $a \leqslant 16$, ale pominiemy ich sprawdzanie, bo dążymy do maksymalizacji a.

Największym całkowitym a, dla którego algorytm A' jest bardziej efektywny od algorytmu A, jest $a = 48$.

4.3-3. Ponieważ $a = 1$ oraz $b = 2$, to $n^{\log_b a} = n^0 = 1$ jest funkcją stałą. W rekurencji tej $f(n)$ także jest stałe, a więc $f(n) = \Theta(n^{\log_b a})$ i $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(\lg n)$.

4.3-4. Dla rekurencji $T(n)$ mamy $a = 4$, $b = 2$ oraz $f(n) = n^2 \lg n$, a więc $n^{\log_b a} = n^2$, ale nie istnieje takie $\epsilon > 0$, że $f(n) = \Omega(n^{2+\epsilon})$. Nie można zatem zastosować w rozwiązaniu twierdzenia o rekurencji uniwersalnej. Asymptotyczne górne oszacowanie na $T(n)$ znajdziemy natomiast, zgadując rozwiązanie, a następnie dowodząc jego poprawności metodą podstawiania.

W pierwszym wywołaniu rekurencja wnosi koszt równy $n^2 \lg n$. Następnie mamy 4 wywołania $T(n/2)$, co daje koszt

$$4T(n/2) = 4(n/2)^2 \lg(n/2) = n^2 \lg n - n^2.$$

Kolejne poziomy wywołań kosztują:

$$\begin{aligned} 16T(n/4) &= 16(n/4)^2 \lg(n/4) = n^2 \lg n - 2n^2, \\ 64T(n/8) &= 64(n/8)^2 \lg(n/8) = n^2 \lg n - 3n^2 \qquad \text{itd.} \end{aligned}$$

Z otrzymanych wyników wnioskujemy, że i-ty poziom wprowadza koszt równy $n^2 \lg n - in^2$. Liście o koszcie stałym znajdują się na poziomie $\lg n$ i jest ich $4^{\lg n} = n^2$, więc dostajemy:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\lg n - 1} (n^2 \lg n - in^2) + \Theta(n^2) \\ &= n^2 \lg^2 n - n^2 \sum_{i=0}^{\lg n - 1} i + \Theta(n^2) \\ &= n^2 \lg^2 n - \frac{n^2 \lg n(\lg n - 1)}{2} + \Theta(n^2) \\ &= O(n^2 \lg^2 n). \end{aligned}$$

Udowodnimy teraz metodą podstawiania, że otrzymane przypuszczenie jest istotnie oszacowaniem górnym dla $T(n)$. Przyjmijmy założenie

$$T(n/2) \leqslant c(n/2)^2 \lg^2(n/2)$$

dla pewnej stałej $c > 0$. Mamy teraz:

$$\begin{aligned} T(n) &\leqslant 4c(n/2)^2 \lg^2(n/2) + n^2 \lg n \\ &= cn^2(\lg n - 1)^2 + n^2 \lg n \\ &= cn^2 \lg^2 n - 2cn^2 \lg n + cn^2 + n^2 \lg n \\ &\leqslant cn^2 \lg^2 n. \end{aligned}$$

Ostatnią nierówność dla wszystkich $n \geqslant 4$ spełniamy, dobierając $c \geqslant 2/3$. Przyjmujemy $T(1) = 1$ jako warunek brzegowy rekurencji, zaś $n = 2$ oraz $n = 3$ jako podstawę indukcji, ponieważ dla $n \geqslant 4$ rekurencja nie zależy już bezpośrednio od $T(1)$, a otrzymane oszacowanie dla $T(2) = 8$ i $T(3) = 4+9 \lg 3$ jest spełnione, o ile $c \geqslant 2$. Rozwiązaniem rekurencji $T(n)$ jest zatem $O(n^2 \lg^2 n)$.

4.3-5. Przyjmując $a = 1$, $b = 2$ oraz $f(n) = n(2 - \cos n)$, dostajemy, że $f(n) = \Omega(n^\epsilon)$ dla pewnego $0 < \epsilon \leqslant 1$. Jednak dla $n = 2(2k + 1)\pi$, gdzie $k = 0, 1, \ldots$, mamy

$$af(n/b) = 3(2k + 1)\pi \quad \text{oraz} \quad f(n) = 2(2k + 1)\pi$$

i nierówność $af(n/b) \leqslant cf(n)$ zachodzi dla $c \geqslant 3/2$, dlatego warunek regularności nie jest spełniony.

4.4. Dowód twierdzenia o rekurencji uniwersalnej

4.4-1. Wykażemy metodą indukcji, że $n_j = \lceil n/b^j \rceil$. Z definicji (4.12) bezpośrednio wynika prawdziwość tego wzoru dla $j = 0$. Przyjmijmy zatem, że dla $j > 0$ zachodzi $n_{j-1} = \lceil n/b^{j-1} \rceil$. Wykorzystując tożsamość (3.4), otrzymujemy

$$n_j = \lceil n_{j-1}/b \rceil = \lceil \lceil n/b^{j-1} \rceil / b \rceil = \lceil n/b^j \rceil,$$

co należało pokazać.

4.4-2. Zastąpmy przypadek 2 tw. 4.1 ogólniejszym warunkiem, tzn. jeśli $f(n) = \Theta(n^{\log_b a} \lg^k n)$ dla $k \geqslant 0$, to zachodzi $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. Dla tak zmodyfikowanego twierdzenia przeprowadzimy dowody analogicznie zmodyfikowanych lematów 4.3 i 4.4 (oznaczonych poniżej przez 4.3′ i 4.4′).

Dowód lematu 4.3′. Przy założeniu, że $f(n) = \Theta(n^{\log_b a} \lg^k n)$, otrzymujemy

$$f(n/b^j) = \Theta\big((n/b^j)^{\log_b a} \lg^k (n/b^j)\big)$$

i podstawiamy do wzoru (4.7):

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j} \right)^{\log_b a} \lg^k \frac{n}{b^j} \right).$$

Mamy dalej:

$$\begin{aligned} \sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j} \right)^{\log_b a} \lg^k \frac{n}{b^j} &= n^{\log_b a} \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^{\log_b a}} \right)^j \lg^k \frac{n}{b^j} \\ &= n^{\log_b a} \sum_{j=0}^{\log_b n - 1} (\lg n - j \lg b)^k \\ &= n^{\log_b a} \cdot \Theta(\lg^{k+1} n) \\ &= \Theta(n^{\log_b a} \lg^{k+1} n). \end{aligned}$$

Skorzystano ze wzoru (3.2), podstawiając $\lg n$ w miejsce n, a następnie z tego, że

$$\sum_{j=0}^{\log_b n - 1} \Theta(\lg^k n) = \log_b n \cdot \Theta(\lg^k n) = \Theta(\lg^{k+1} n).$$

Pokazano, że $g(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$, a więc lemat jest prawdziwy. □

Dowód lematu 4.4′. Wystarczy wykazać jedynie drugi przypadek, bo $f(n) = \Theta(n^{\log_b a} \lg^k n)$. Z lematów 4.2 i 4.3′:

$$T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg^{k+1} n) = \Theta(n^{\log_b a} \lg^{k+1} n),$$

co należało udowodnić. □

Twierdzenie wynika bezpośrednio z lematu 4.4′.

4.4-3. *Aby udowodnić to wynikanie, musimy założyć dodatkowo, że $c > 0$ i że funkcja $f(n)$ jest asymptotycznie dodatnia. W tw. 4.1 oba te warunki wynikają z faktu, że $f(n) = \Omega(n^{\log_b a+\epsilon})$.*

Załóżmy, że dla danej funkcji asymptotycznie dodatniej $f(n)$ oraz stałych $a \geqslant 1$, $b > 1$, zachodzi warunek regularności, tzn. istnieje stała $0 < c < 1$ i stała $n_0 > 0$, że dla każdego $n \geqslant n_0$ spełnione jest $af(n/b) \leqslant cf(n)$. Przyjmiemy, że n_0 zostało wybrane w taki sposób, że dla wszystkich $n \geqslant n_0/b$ funkcja $f(n)$ osiąga wartości dodatnie. Dla $n \geqslant n_0$ mamy

$$f(n) \geqslant (a/c)f(n/b),$$

a zatem, iterując tę nierówność, dostajemy

$$f(n) \geqslant (a/c)f(n/b) \geqslant (a/c)^2 f(n/b^2) \geqslant \cdots \geqslant (a/c)^i f(n/b^i).$$

Zakładamy, że nie można wykonać większej ilości iteracji, tzn. $n/b^{i-1} \geqslant n_0$ oraz $n/b^i < n_0$. Stąd $\log_b(n/n_0) < i \leqslant \log_b(n/n_0) + 1$, czyli, ze wzoru (3.3), $i = \lfloor \log_b(n/n_0) + 1 \rfloor$. Zachodzi więc

$$f(n) \geqslant (a/c)^{\lfloor \log_b(n/n_0)+1 \rfloor} f\big(n/b^{\lfloor \log_b(n/n_0)+1 \rfloor}\big).$$

Zauważmy, że przy zadanych ograniczeniach na a i c, $a/c > 1$, więc $f(n)$ musi być funkcją rosnącą dla $n \geqslant n_0/b$, skąd

$$f\big(n/b^{\lfloor \log_b(n/n_0)+1 \rfloor}\big) > f(n/b^{\log_b(n/n_0)+1}) = f(n_0/b).$$

Mamy następnie

$$\begin{aligned} f(n) &> (a/c)^{\log_b(n/n_0)} f(n/b^{\log_b(n/n_0)+1}) \\ &= \frac{(n/n_0)^{\log_b a}}{(n/n_0)^{\log_b c}} f(n_0/b) \\ &= n^{\log_b a - \log_b c} \cdot n_0^{\log_b c - \log_b a} \cdot f(n_0/b). \end{aligned}$$

Dwa ostatnie czynniki powyższego wyrażenia są dodatnie i niezależne od n, możemy więc potraktować ich iloczyn jako stałą $d > 0$. Ponieważ $0 < c < 1$ oraz $b > 1$, to $\log_b c < 0$, przyjmijmy więc $\epsilon = -\log_b c$. Otrzymujemy zatem, że dla dowolnego $n \geqslant n_0$

$$f(n) > n^{\log_b a+\epsilon} \cdot d = \Omega(n^{\log_b a+\epsilon}),$$

co należało wykazać.

Problemy

4-1. Przykłady rekurencji

W punktach (a)–(f) skorzystano z twierdzenia o rekurencji uniwersalnej.

(a) Mamy $n^{\log_b a} = n^{\log_2 2} = n$ oraz $f(n) = n^3 = \Omega(n^{1+\epsilon})$ dla $\epsilon \leqslant 2$. Ponieważ warunek regularności jest spełniony:

$$\begin{aligned} 2f(n/2) &\leqslant cf(n), \\ 2n^3/8 &\leqslant cn^3, \\ c &\geqslant 1/4, \end{aligned}$$

to stąd wnioskujemy, że $T(n) = \Theta(n^3)$.

(b) Mamy $n^{\log_b a} = n^{\log_{10/9} 1} = n^0 = 1$ oraz $f(n) = n = \Omega(n^\epsilon)$ dla $\epsilon \leqslant 1$. Badamy warunek regularności:

$$\begin{aligned} f(9n/10) &\leqslant cf(n), \\ 9n/10 &\leqslant cn, \\ c &\geqslant 9/10 \end{aligned}$$

i stwierdzamy, że $T(n) = \Theta(n)$.

(c) Mamy $n^{\log_b a} = n^{\log_4 16} = n^2$ oraz $f(n) = n^2 = \Theta(n^2)$, a stąd $T(n) = \Theta(n^2 \lg n)$.

(d) Mamy $n^{\log_b a} = n^{\log_3 7}$ oraz $f(n) = n^2 = \Omega(n^{\log_3 7 + \epsilon})$ dla $\epsilon \leqslant 2 - \log_3 7$. Warunek regularności zachodzi:

$$\begin{aligned} 7f(n/3) &\leqslant cf(n), \\ 7n^2/9 &\leqslant cn^2, \\ c &\geqslant 7/9, \end{aligned}$$

a zatem $T(n) = \Theta(n^2)$.

(e) Mamy $n^{\log_b a} = n^{\lg 7}$ oraz $f(n) = n^2 = O(n^{\lg 7 - \epsilon})$ dla $\epsilon \leqslant \lg 7 - 2$, a stąd $T(n) = \Theta(n^{\lg 7})$.

(f) Mamy $n^{\log_b a} = n^{\log_4 2} = n^{1/2}$ oraz $f(n) = \sqrt{n} = \Theta(n^{1/2})$, a stąd $T(n) = \Theta\bigl(\sqrt{n} \lg n\bigr)$.

(g) Zauważmy, że rekurencja rozwija się następująco:

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &\;\;\vdots \\ &= c + 3 + 4 + \cdots + n \\ &= c + \sum_{i=3}^{n} i \\ &= c + \frac{n(n+1)}{2} - 3, \end{aligned}$$

gdzie $c = T(2)$ jest pewną stałą. Otrzymujemy zatem, że $T(n) = \Theta(n^2)$.

(h) Niech $n = 2^m$, skąd $m = \lg n$. Rekurencja przyjmuje postać

$$T(2^m) = T(2^{m/2}) + 1.$$

Podstawiając $S(m)$ za $T(2^m)$, otrzymujemy

$$S(m) = S(m/2) + 1.$$

Ponieważ rozwiązaniem ostatniej rekurencji jest $\Theta(\lg m)$ (co pokazano w zad. 4.3-3), to stąd mamy, że $T(n) = T(2^m) = S(m) = \Theta(\lg m) = \Theta(\lg \lg n)$.

4-2. Szukanie brakującej liczby całkowitej

Poprzez badanie najmniej znaczącego bitu liczby całkowitej możemy sprawdzić parzystość tej liczby. Pobierzemy więc najmniej znaczące bity wszystkich liczb z tablicy A i policzymy, ile z nich jest zerami. W zakresie $0\,..\,n$ jest $\lfloor n/2 \rfloor + 1$ liczb parzystych. Jeśli więc wśród pobranych bitów znajdziemy dokładnie tyle samo zer, to wiemy, że szukana liczba jest nieparzysta, a jeśli zer będzie $\lfloor n/2 \rfloor$, to brakuje liczby parzystej. W zależności od przypadku z dalszej analizy odrzucimy liczby o parzystości innej niż parzystość brakującej liczby, a pozostałe będziemy przeszukiwać w kolejnym wywołaniu rekurencyjnym, badając kolejny najmniej znaczący bit tych liczb. W momencie osiągnięcia pustego zbioru będziemy znać wszystkie bity brakującej liczby.

Spośród n liczb, do wywołania rekurencyjnego zostaje ich przekazanych $n - (\lfloor n/2 \rfloor + 1) = \lfloor (n-1)/2 \rfloor$ albo $\lfloor n/2 \rfloor$. Pesymistyczny czas działania opisanego tutaj algorytmu można więc zapisać w postaci rekurencji $T(n) = T(\lfloor n/2 \rfloor) + O(n)$, której rozwiązaniem jest $\Theta(n)$ na podstawie tw. 4.1.

4-3. Koszty przekazywania parametrów

(a) W pierwszej strategii rekurencja przyjmuje postać z zad. 2.3-5, której rozwiązaniem jest $T(n) = \Theta(\lg n)$. Po przyjęciu $n = N$ dostajemy $T(N) = \Theta(\lg N)$.

W przypadku drugiej strategii na każdym poziomie rekursji należy dodać składnik $\Theta(N)$ odpowiedzialny za przekazywanie tablicy do wywołań rekurencyjnych. Otrzymujemy zatem

$$T(n) = \begin{cases} \Theta(1), & \text{jeśli } n \leqslant 1, \\ T(\lfloor n/2 \rfloor) + \Theta(N), & \text{jeśli } n > 1. \end{cases}$$

Ponieważ $\Theta(N)$ nie zależy od rozmiaru podproblemu n, to stąd rozwiązaniem powyższej rekurencji jest $T(n) = \Theta(N \lg n)$, a więc $T(N) = \Theta(N \lg N)$.

W ostatnim przypadku przekazujemy podtablicę o rozmiarze równym rozmiarowi podproblemu, co prowadzi do rekurencji

$$T(n) = \begin{cases} \Theta(1), & \text{jeśli } n \leqslant 1, \\ T(\lfloor n/2 \rfloor) + \Theta(\lfloor n/2 \rfloor), & \text{jeśli } n > 1, \end{cases}$$

którą można rozwiązać przy użyciu twierdzenia o rekurencji uniwersalnej. Otrzymujemy wynik $T(n) = \Theta(n)$, skąd $T(N) = \Theta(N)$.

(b) W przypadku zwykłego przekazywania wskaźnika mamy oryginalną postać rekurencji, której rozwiązaniem dla $n = N$ jest $T(N) = \Theta(N \lg N)$.

Ponieważ w drugiej strategii należy przekazać całą tablicę do obu wywołań rekurencyjnych, to czas działania wygląda następująco:

$$T(n) = \begin{cases} \Theta(1), & \text{jeśli } n = 1, \\ 2T(\lfloor n/2 \rfloor) + \Theta(n) + 2\Theta(N), & \text{jeśli } n > 1. \end{cases}$$

Na każdym poziomie dodajemy składnik rzędu $\Theta(N)$, zatem rozwiązaniem tej rekurencji jest iloczyn tegoż składnika i rozwiązania rekurencji z pierwszego przypadku, czyli $T(n) = \Theta(Nn \lg n)$, a stąd mamy $T(N) = \Theta(N^2 \lg N)$.

W ostatniej strategii do każdego wywołania rekurencyjnego przekazujemy podtablicę o rozmiarze podproblemu, jednak czas na to poświęcany jest identyczny ze składnikiem liniowym odpowiadającym za czas przeznaczany na procedurę Merge. Rekurencja ma zatem identyczną postać jak w oryginalnej analizie tego algorytmu i jej rozwiązaniem jest $T(N) = \Theta(N \lg N)$.

4-4. Więcej przykładów rekurencji

(a) Wykorzystując twierdzenie o rekurencji uniwersalnej, mamy $n^{\log_b a} = n^{\lg 3}$ oraz $f(n) = n \lg n = O(n^{\lg 3 - \epsilon})$, gdzie $\epsilon < \lg 3 - 1$, a stąd $T(n) = \Theta(n^{\lg 3})$.

(b) Z twierdzenia o rekurencji uniwersalnej obliczamy $n^{\log_b a} = n^{\log_5 5} = n$, jednak dla żadnego $\epsilon > 0$ nie jest prawdą, że

$$f(n) = \frac{n}{\lg n} = O(n^{1-\epsilon}),$$

ponieważ dla pewnej stałej $c > 0$ i dostatecznie dużych n musiałoby zachodzić

$$\frac{n^\epsilon}{\lg n} \leqslant c,$$

a ponieważ $n^\epsilon = \omega(\lg n)$, to niezależnie od wyboru ϵ dla dużych wartości n licznik tego ułamka jest dowolnie większy od mianownika i ułamka nie da się z tego powodu ograniczyć stałą.

Skorzystajmy zatem z innego sposobu na obliczenie $T(n)$, rozważając rodzinę rekurencji postaci

$$T_a(n) = \begin{cases} \Theta(1), & \text{jeśli } 1 \leqslant n < a, \\ aT_a(n/a) + n/\lg n, & \text{jeśli } n \geqslant a, \end{cases}$$

gdzie $a \geqslant 2$ jest liczbą całkowitą. Wykorzystamy technikę zamiany zmiennych, podstawiając $m = \log_a n$, skąd $n = a^m$, dzięki czemu otrzymujemy

$$T_a(a^m) = aT_a(a^{m-1}) + \frac{a^m}{m \lg a}.$$

Możemy teraz podstawić $S_a(m) = T_a(a^m)$, otrzymując nową rekurencję

$$S_a(m) = aS_a(m-1) + \frac{a^m}{m \lg a},$$

którą rozwiązujemy następująco (po przyjęciu $S_a(0) = T_a(1) = d$ dla pewnej stałej $d > 0$):

$$\begin{aligned} S_a(m) &= \frac{1}{\lg a}\left(\frac{a^m}{m} + a \cdot \frac{a^{m-1}}{m-1} + a^2 \cdot \frac{a^{m-2}}{m-2} + \cdots + a^{m-1} \cdot \frac{a^1}{1} + a^m d\right) \\ &= \frac{1}{\lg a}\left(\sum_{k=1}^{m} \frac{a^m}{k} + a^m d\right) \\ &= \frac{a^m (H_m + d)}{\lg a} \\ &= \Theta(a^m \lg m). \end{aligned}$$

Zamieniając z powrotem $S_a(m)$ na $T_a(n)$, otrzymujemy rozwiązanie

$$T_a(n) = T_a(a^m) = S_a(m) = \Theta(a^m \lg m) = \Theta(n \lg \log_a n) = \Theta(n \lg \lg n).$$

Stosując znalezione oszacowanie do rekurencji $T(n) \equiv T_5(n)$ z treści zadania, dostajemy oczywiście $T(n) = \Theta(n \lg \lg n)$.

(c) Z twierdzenia o rekurencji uniwersalnej mamy $n^{\log_b a} = n^{\lg 4} = n^2$ oraz $f(n) = n^{5/2} = \Omega(n^{2+\epsilon})$, gdzie $\epsilon \leqslant 1/2$. Badamy warunek regularności:

$$\begin{aligned} 4f(n/2) &\leqslant cf(n), \\ \frac{4n^{5/2}}{2^{5/2}} &\leqslant cn^{5/2}, \\ c &\geqslant \frac{1}{\sqrt{2}} \end{aligned}$$

i stwierdzamy, że $T(n) = \Theta(n^{5/2})$.

(d) Wykażemy, że stała 5 w argumencie rekurencji $T(n)$ nie wpływa na postać rozwiązania. Rozważając rekurencję $T'(n) = 3T'(n/3) + n/2$ i rozwiązując ją za pomocą twierdzenia o rekurencji uniwersalnej, dostajemy $T'(n) = \Theta(n \lg n)$. Udowodnimy teraz metodą podstawiania, że identyczny wynik jest rozwiązaniem $T(n)$.

Wykorzystując założenie

$$T(n/3 + 5) \leqslant c_1(n/3 + 5) \lg(n/3 + 5)$$

dla pewnej stałej $c_1 > 0$, otrzymujemy:

$$\begin{aligned} T(n) &\leqslant 3c_1(n/3 + 5) \lg(n/3 + 5) + n/2 \\ &\leqslant c_1(n + 15) \lg(2n/5) + n/2 \\ &< c_1 n \lg n + c_1 n \lg(2/5) + 15c_1 \lg n + n/2 \\ &\leqslant c_1 n \lg n. \end{aligned}$$

Powyższe wnioskowanie jest prawdziwe, o ile $n/3 + 5 \leqslant 2n/5$, skąd $n \geqslant 75$, oraz

$$c_1 n \lg(2/5) + 15c_1 \lg n + n/2 \leqslant 0.$$

Przeprowadzając podobną analizę jak w zad. 4.1-5, dostajemy, że dowolne $c_1 \geqslant 7$ spełnia ostatnią nierówność dla $n \geqslant 75$.

Analogicznie dla dolnego oszacowania przyjmujemy, że

$$T(n/3 + 5) \geqslant c_2(n/3 + 5) \lg(n/3 + 5),$$

gdzie $c_2 > 0$ jest pewną stałą. Wówczas:

$$\begin{aligned} T(n) &\geqslant 3c_2(n/3 + 5) \lg(n/3 + 5) + n/2 \\ &> 3c_2(n/3) \lg(n/3) + n/2 \\ &= c_2 n \lg n - c_2 n \lg 3 + n/2 \\ &\geqslant c_2 n \lg n, \end{aligned}$$

przy czym ostatnia nierówność zachodzi, o ile

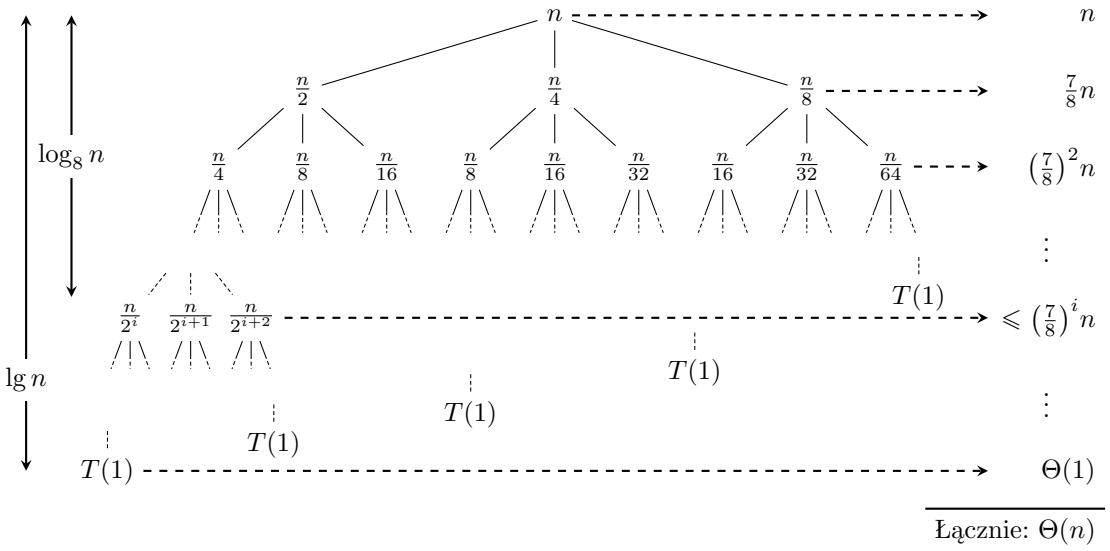
$$-c_2 n \lg 3 + n/2 \geqslant 0.$$

Warunek ten spełniamy poprzez przyjęcie $c_2 \leqslant 0{,}3$.

Dowód asymptotycznie dokładnego oszacowania dla $T(n)$ kończymy, wybierając odpowiednie wartości dla podstaw obu indukcji. Zauważmy, że obliczanie wartości rekurencji ze wzoru $T(n) = 3T(n/3 + 5) + n/2$ dla $n \leqslant 7$ nie ma sensu, bo wtedy $T(n)$ nie zależy od niższych wyrazów. Przyjmijmy zatem, że $T(n) = 1$ dla wszystkich $n \leqslant 7$ będzie przypadkiem brzegowym rekurencji. W dowodzie górnego oszacowania założyliśmy, że $n \geqslant 75$, więc podstawą obu indukcji możemy uczynić wszystkie $n = 30, 31, \ldots, 74$. Można sprawdzić, że dla takich wartości oba oszacowania są spełnione, co uzasadnia poprawny dobór stałych c_1 i c_2. A zatem $T(n) = \Theta(n \lg n)$.

(e) Mamy do czynienia z rekurencją $T_a(n)$ dla $a = 2$, którą rozważaliśmy w punkcie (b). Zgodnie z przedstawionym tam rozumowaniem wnioskujemy, że $T(n) = \Theta(n \lg \lg n)$.

(f) W celu rozwiązania rekurencji wykorzystamy metodę drzewa rekursji do odgadnięcia rozwiązania, które następnie udowodnimy. Drzewo zostało przedstawione na rys. 7.



Rysunek 7: Drzewo rekursji dla równania rekurencyjnego $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

Zauważmy, że najszybciej maleją argumenty na skrajnie prawej gałęzi. Niech $T(1) = 1$. Ponieważ koszt węzła z tej gałęzi znajdującego się na i-tym poziomie wynosi $n/8^i$, to liść zajmuje poziom $h = \log_8 n$. Łącznym kosztem na i-tym poziomie jest $(7/8)^i n$, więc sumując te wartości od korzenia aż do poziomu h-tego, otrzymujemy oszacowanie rekurencji od dołu:

$$T(n) \geqslant \sum_{i=0}^{\log_8 n} \left(\frac{7}{8}\right)^i n = \frac{1 - (7/8)^{\log_8 n + 1}}{1 - 7/8}\, n = 8n(1 - (7/8)n^{\log_8(7/8)}) \geqslant 8n(1 - 7/8) = \Omega(n).$$

Zbadajmy teraz górne oszacowanie rekurencji $T(n)$ poprzez dokonanie obserwacji, że najwolniej malejącymi węzłami drzewa są te ze skrajnie lewej gałęzi. Węzły te wnoszą koszt równy $n/2^i$ na i-tym poziomie, więc liść znajduje się na poziomie $H = \lg n$. Mamy

$$T(n) \leqslant \sum_{i=0}^{\lg n} \left(\frac{7}{8}\right)^i n < \sum_{i=0}^{\infty} \left(\frac{7}{8}\right)^i n = \frac{n}{1 - 7/8} = O(n),$$

co pozwala przypuszczać, że oszacowaniem dokładnym na $T(n)$ jest $\Theta(n)$.

Przeprowadźmy teraz dowód tego wyniku metodą podstawiania. Załóżmy, że:

$$\begin{aligned} c_1(n/2) &\leqslant T(n/2) \leqslant c_2(n/2), \\ c_1(n/4) &\leqslant T(n/4) \leqslant c_2(n/4), \\ c_1(n/8) &\leqslant T(n/8) \leqslant c_2(n/8), \end{aligned}$$

gdzie c_1, $c_2 > 0$ to pewne stałe. Mamy zatem

$$T(n) \geqslant c_1 n/2 + c_1 n/4 + c_1 n/8 + n = 7c_1 n/8 + n \geqslant c_1 n,$$

co jest spełnione dla dowolnego n, o ile $c_1 \leqslant 8$. Dowód górnego oszacowania przebiega analogicznie, przy czym zakładamy, że $c_2 \geqslant 8$.

Jako warunek brzegowy rekurencji przyjmijmy $T(1) = T(2) = \cdots = T(7) = 1$. Aby spełnić przypadek bazowy indukcji dla $n = 1, 2, \ldots, 7$, musimy przyjąć dodatkowo, że $c_1 \leqslant 1/7$. Udowodniliśmy zatem, że dokładnym rozwiązaniem rekurencji jest $T(n) = \Theta(n)$.

(g) Załóżmy dla uproszczenia rachunków, że $T(1) = 1$. Rozwijając rekurencję, otrzymujemy

$$T(n) = \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} + \frac{1}{1} = H_n,$$

a zatem, korzystając ze wzoru (A.7), mamy $T(n) = \Theta(\lg n)$.

(h) Dla ułatwienia przyjmijmy, że $T(1) = 0$. Wówczas

$$T(n) = \lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1 = \lg\left(\prod_{i=1}^{n} i\right) = \lg(n!)$$

i ze wzoru (3.18) dostajemy $T(n) = \Theta(n \lg n)$.

(i) Przyjmijmy, że $T(2) = 2$ i rozważmy przypadek, gdy n jest parzyste. Rozwijamy rekurencję, otrzymując

$$T(n) = 2\lg n + 2\lg(n-2) + \cdots + 2\lg 4 + 2\lg 2 = 2\lg\left(\prod_{i=1}^{n/2} 2i\right) = 2\big(\lg((n/2)!) + n/2\big).$$

Wykorzystując wzór (3.18), dostajemy

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

Dla n nieparzystego, po przyjęciu $T(1) = 0$ sprowadzamy rekurencję do sumy

$$T(n) = 2\lg n + 2\lg(n-2) + \cdots + 2\lg 3 + 2\lg 1 = 2\lg\left(\prod_{i=1}^{(n+1)/2} (2i-1)\right),$$

którą można ograniczyć z góry przez $2\lg\left(\prod_{i=1}^{(n+1)/2} 2i\right)$, a z dołu przez $2\lg\left(\prod_{i=1}^{(n-1)/2} 2i\right)$. Po pewnych przekształceniach i po skorzystaniu ze wzoru (3.18) oba te wyrażenia sprowadzamy do postaci $\Theta(n \lg n)$. Na podstawie tego faktu i uzasadnienia z poprzedniego paragrafu wnioskujemy, że $T(n)$ jest klasy $\Theta(n \lg n)$.

(j) Po podzieleniu rekurencji $T(n)$ przez n dostajemy

$$\frac{T(n)}{n} = \frac{T(\sqrt{n})}{\sqrt{n}} + 1.$$

Podstawmy teraz $S(n) = T(n)/n$, otrzymując nową rekurencję

$$S(n) = S(\sqrt{n}) + 1.$$

Potraktujmy n jako 2^m, skąd $m = \lg n$ i podstawmy $R(m) = S(2^m)$:

$$R(m) = R(m/2) + 1.$$

Rozwiązanie ostatniej rekurencji zostało wyznaczone w zad. 4.3-3 i wynosi $R(m) = \Theta(\lg m)$. Powracamy do oryginalnej rekurencji $T(n)$, dostając ostatecznie

$$T(n) = nS(n) = nR(\lg n) = n \cdot \Theta(\lg \lg n) = \Theta(n \lg \lg n).$$

4-5. Liczby Fibonacciego

(a) Wprost z definicji $\mathcal{F}(z)$ mamy:

$$\begin{aligned}
\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\
&= F_0 + zF_1 + \sum_{i=2}^{\infty} (F_{i-1} + F_{i-2}) z^i \\
&= z + \sum_{i=2}^{\infty} F_{i-1} z^i + \sum_{i=2}^{\infty} F_{i-2} z^i \\
&= z + \sum_{i=1}^{\infty} F_i z^{i+1} + \sum_{i=0}^{\infty} F_i z^{i+2} \\
&= z + z\mathcal{F}(z) + z^2 \mathcal{F}(z),
\end{aligned}$$

a zatem tożsamość zachodzi.

(b) Ze wzoru z poprzedniego punktu wynika pierwsza równość:

$$\begin{aligned}
\mathcal{F}(z) &= z + z\mathcal{F}(z) + z^2\mathcal{F}(z), \\
(1 - z - z^2)\mathcal{F}(z) &= z, \\
\mathcal{F}(z) &= \frac{z}{1 - z - z^2}.
\end{aligned}$$

Mianownik prawej strony ostatniego wyrażenia jest trójmianem kwadratowym o miejscach zerowych ϕ i $\widehat{\phi}$, który zapisujemy w równoważnej postaci

$$1 - z - z^2 = -(z + \phi)\big(z + \widehat{\phi}\big) = (1 - \phi z)\big(1 - \widehat{\phi} z\big),$$

co można uzasadnić wzorem $\phi \cdot \widehat{\phi} = -1$ i tym samym dowieść drugiej równości. Ostatnią z nich otrzymujemy, zauważając, że

$$\frac{1}{1 - \phi z} - \frac{1}{1 - \widehat{\phi} z} = \frac{1 - \widehat{\phi} z - 1 + \phi z}{(1 - \phi z)\big(1 - \widehat{\phi} z\big)} = \frac{z\big(\phi - \widehat{\phi}\big)}{(1 - \phi z)\big(1 - \widehat{\phi} z\big)} = \frac{z\sqrt{5}}{(1 - \phi z)\big(1 - \widehat{\phi} z\big)},$$

a stąd mamy

$$\frac{z}{(1 - \phi z)\big(1 - \widehat{\phi} z\big)} = \frac{1}{\sqrt{5}} \cdot \frac{z\sqrt{5}}{(1 - \phi z)\big(1 - \widehat{\phi} z\big)} = \frac{1}{\sqrt{5}} \left(\frac{1}{1 - \phi z} - \frac{1}{1 - \widehat{\phi} z} \right).$$

(c) Tezę otrzymujemy natychmiast, jeśli w definicji $\mathcal{F}(z)$ podstawimy $F_i = \left(\phi^i - \widehat{\phi}^i\right)/\sqrt{5}$, co wykazano w zad. 3.2-6.

(d) Ponieważ $\left|\widehat{\phi}\right| < 1$, to prawdą jest, że $\left|\widehat{\phi}^i\right| < 1$ dla $i > 0$ oraz $\left|\widehat{\phi}^i\right|/\sqrt{5} < 1/\sqrt{5} < 1/2$. Mamy

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} = \frac{\phi^i}{\sqrt{5}} - \frac{\widehat{\phi}^i}{\sqrt{5}},$$

skąd

$$\frac{\phi^i}{\sqrt{5}} - \frac{1}{2} < F_i < \frac{\phi^i}{\sqrt{5}} + \frac{1}{2},$$

a zatem F_i jest liczbą całkowitą najbliższą wartości $\phi^i/\sqrt{5}$.

(e) Nierówność została udowodniona w zad. 3.2-7.

4-6. Testowanie układów VLSI

(a) Niech D i Z będą zbiorami, odpowiednio, układów dobrych i układów złych. Podczas testowania każdy zły układ może twierdzić, że każdy inny układ ze zbioru Z jest dobry, a każdy układ ze zbioru D jest zły. Z kolei dobry układ może orzekać o każdym układzie z Z, że jest zły, natomiast o każdym innym z D, że jest dobry. Inaczej ujmując, zbiór Z będzie wskazywał, że sam jest zbiorem układów dobrych, a zbiór D złych i symetrycznie dla zbioru D. W ogólności Z może składać się z podzbiorów układów, które będą twierdzić, że wyłącznie one są dobre. Nie da się natomiast rozdzielić w taki sposób zbioru D, ponieważ jego układy zawsze orzekają prawdziwie. Aby zatem jednoznacznie wskazać zbiór dobrych układów, wymagane jest założenie, że $|D| > |Z|$.

(b) Potraktujmy wynik każdego testu dwóch układów jako parę $\langle a, b\rangle \in \{\text{D}, \text{Z}\}^2$. Pierwszy element pary jest stwierdzeniem pierwszego układu o drugim, a drugi element – drugiego o pierwszym, przy czym D oznacza pozytywny wynik testu, a Z – negatywny. Możliwe jest uzyskanie jednego z czterech wyników:

- $\langle \text{D}, \text{D}\rangle$ – oba układy są dobre albo oba są złe;
- $\langle \text{D}, \text{Z}\rangle$ – tylko pierwszy z układów jest zły;
- $\langle \text{Z}, \text{D}\rangle$ – tylko drugi z układów jest zły;
- $\langle \text{Z}, \text{Z}\rangle$ – co najmniej jeden z układów jest zły.

Podzielmy zbiór układów na pary i przetestujmy wzajemnie układy w każdej takiej parze, wykonując przy tym $\lfloor n/2 \rfloor$ testów. Zauważmy, że otrzymawszy dla pewnej pary wynik inny niż $\langle \text{D}, \text{D}\rangle$, możemy ją odrzucić, gdyż co najmniej jeden układ z tej pary jest zły i wśród nieodrzuconych układów będzie nadal więcej dobrych niż złych. Dostaniemy w rezultacie od jednej do $\lfloor n/2 \rfloor$ par o tej własności, że w każdej z nich oba układy są dobre albo oba są złe. Odrzucając zatem po jednym układzie z każdej pozostawionej pary, nadal zachowujemy własność o większej liczbie dobrych układów w pozostawionym zbiorze układów, którego rozmiar wynosi co najwyżej $\lceil n/2 \rceil$.

Powyżej opisany proces przeprowadzamy rekurencyjnie, dostając w końcu zbiór jednoelementowy, który na mocy założenia zawiera dobry układ.

(c) Wykorzystując wynik poprzedniego punktu, dostajemy następującą rekurencję opisującą liczbę testów koniecznych do znalezienia jednego dobrego układu w pesymistycznym przypadku (tzn. kiedy każda para układów zwraca wynik $\langle \text{D}, \text{D} \rangle$):

$$T(n) = \begin{cases} \Theta(1), & \text{jeśli } n = 1, \\ T(\lceil n/2 \rceil) + \lfloor n/2 \rfloor, & \text{jeśli } n > 1. \end{cases}$$

Stosując twierdzenie o rekurencji uniwersalnej, dostajemy rozwiązanie: $T(n) = \Theta(n)$. Wynik ten jest prawdziwy także w przypadku optymistycznym – wówczas dobry układ znajdujemy w jednym zejściu rekurencyjnym po uprzednim wykonaniu $\Theta(n)$ testów.

Potrafimy wyznaczyć dobry układ u – wykorzystajmy go więc do znalezienia kolejnych. Testujemy tenże układ z pozostałymi $n-1$. Wynikiem testu u z pewnym innym układem v nie może być $\langle \text{D}, \text{Z} \rangle$, a więc możliwe są trzy sytuacje. Jeśli otrzymamy $\langle \text{D}, \text{D} \rangle$, to oznacza to, że oba układy są tak samo dobre, a więc v również jest dobry. Uzyskując wynik $\langle \text{Z}, \text{D} \rangle$, mamy natychmiast, że v jest zły, podobnie w wypadku, gdy wynikiem testu będzie $\langle \text{Z}, \text{Z} \rangle$ – wtedy co najmniej jeden z testowanych układów jest zły, ale nie może nim być u. Wynika stąd, że po wykonaniu $n-1$ takich testów wyznaczymy pozostałe dobre układy. A zatem łączna liczba testów wymaganych do zidentyfikowania wszystkich dobrych układów wynosi $\Theta(n)$.

4-7. Tablice Monge'a

(a) Elementy tablicy Monge'a A spełniają nierówność

$$A[i,j] + A[k,l] \leqslant A[i,l] + A[k,j],$$

gdzie $1 \leqslant i < k \leqslant m$ oraz $1 \leqslant j < l \leqslant n$. W szczególności więc może być $k = i+1$ oraz $l = j+1$, zatem implikacja w prawo zachodzi.

Implikację w przeciwną stronę dowodzimy przez indukcję względem liczby wierszy m. Zauważmy, że warunek tablicy Monge'a nie ma większego sensu dla tablic o jednej kolumnie lub jednym wierszu, zatem przyjmijmy $m = 2$ za podstawę indukcji. Wówczas może być tylko $i = 1$ oraz $k = 2$. Na podstawie założenia spełnione są zatem wszystkie nierówności z następującego układu:

$$\begin{cases} A[1,1] + A[2,2] & \leqslant & A[1,2] + A[2,1] \\ A[1,2] + A[2,3] & \leqslant & A[1,3] + A[2,2] \\ & \vdots & \\ A[1,n-1] + A[2,n] & \leqslant & A[1,n] + A[2,n-1] \end{cases}.$$

Dodając stronami nierówności z tego układu od j-tej do l-tej włącznie, a następnie redukując powtarzające się składniki, otrzymujemy nierówność $A[1,j] + A[2,l] \leqslant A[1,l] + A[2,j]$ z tezy twierdzenia.

Niech teraz $m > 2$. Przyjmujemy założenie indukcyjne, że tablica A pozbawiona ostatniego wiersza stanowi tablicę Monge'a. Pozostaje zatem wykazać, że zachodzą nierówności z definicji tablicy Monge'a, przy czym $k = m$. Wykorzystując pomysł z pierwszego kroku indukcji, z układu

$$\begin{cases} A[m-1,1] + A[m,2] & \leqslant & A[m-1,2] + A[m,1] \\ A[m-1,2] + A[m,3] & \leqslant & A[m-1,3] + A[m,2] \\ & \vdots & \\ A[m-1,n-1] + A[m,n] & \leqslant & A[m-1,n] + A[m,n-1] \end{cases}$$

możemy uzyskać wszystkie nierówności postaci $A[m-1,j]+A[m,l] \leqslant A[m-1,l]+A[m,j]$, gdzie $1 \leqslant j < l \leqslant n$. Jeśli teraz dodamy stronami każdą z nich do każdej nierówności postaci $A[i,j]+A[m-1,l] \leqslant A[i,l]+A[m-1,j]$, gdzie $1 \leqslant j < l \leqslant n$ oraz $1 \leqslant i < m-1$, zachodzących na mocy założenia indukcyjnego, to po zredukowaniu zbędnych składników dostaniemy wszystkie nierówności z tezy, w których $k = m$.

Widać zatem, że implikacja jest prawdziwa dla tablic o ustalonej liczbie kolumn. Dowód dla zmiennej liczby kolumn przeprowadza się analogicznie przez indukcję po $n \geqslant 2$, pokazując tym samym, że twierdzenie zachodzi dla tablic o dowolnych wymiarach.

(b) Na podstawie poprzedniego punktu można sprawdzić, że nierówność

$$A[1,2]+A[2,3] \leqslant A[1,3]+A[2,2]$$

jest fałszywa, przez co własność tablicy Monge'a jest zaburzona. By przywrócić tę własność, można zamienić $A[1,3]$ np. na 24.

(c) Korzystając z poniższej nierówności prawdziwej dla tablicy Monge'a A:

$$A[i,j]+A[i+1,j+r] \leqslant A[i,j+r]+A[i+1,j],$$

gdzie $0 < r \leqslant n-j$, wnioskujemy w następujący sposób. Znajdujemy w pierwszym wierszu tablicy A pierwsze minimum z lewej strony, które oznaczymy przez μ. Indeksem μ jest oczywiście $f(1)$. Wstawiając teraz $i = 1$ oraz $r = f(1) - j$ do powyższej nierówności, otrzymujemy

$$A[1,j]+A[2,f(1)] \leqslant \mu + A[2,j].$$

Z drugiej strony $\mu < A[1,j]$ dla każdego $1 \leqslant j < f(1)$. Łącząc oba fakty, otrzymujemy, że $A[2,f(1)] < A[2,j]$, a to oznacza, że pierwsze z lewej strony minimum wiersza 2 występuje w nim na pozycji nie mniejszej niż $f(1)$, skąd $f(1) \leqslant f(2)$.

Dowód kolejnych nierówności przebiega analogicznie.

(d) Aby odnaleźć minimum wiersza i-tego (nieparzystego), sprawdzamy indeksy minimów wierszy $(i-1)$-szego oraz $(i+1)$-szego (parzystych). Wartości te zostały wyznaczone w pierwszej części algorytmu. Na podstawie poprzedniego punktu mamy, że $f(i-1) \leqslant f(i) \leqslant f(i+1)$, wystarczy więc sprawdzić komórki $A[i,f(i-1)]$, $A[i,f(i-1)+1]$, $\ldots$, $A[i,f(i+1)]$ i wybrać spośród nich element minimalny. Oczywiście nie istnieje wiersz zerowy, dlatego przetwarzając pierwszy wiersz, nie szukamy minimum poprzedniego, ale przyjmujemy dla uproszczenia procedury, że $f(0) = 1$. Podobny przypadek może się zdarzyć dla ostatniego nieparzystego wiersza, jeśli jest on ostatnim wierszem tablicy – wtedy wystarczy przyjąć $f(m+1) = n$.

Poszukując minimum wiersza i-tego, sprawdzamy $f(i+1)-f(i-1)+1$ komórek w tym wierszu. Podczas działania procedury w pesymistycznym przypadku zostanie sprawdzonych

$$\begin{aligned}
\sum_{\substack{1 \leqslant i \leqslant m \\ 2 \nmid i}} (f(i+1)-f(i-1)+1) &= \sum_{k=0}^{\lceil m/2 \rceil - 1} (f(2k+2)-f(2k)+1) \\
&= \lceil m/2 \rceil + \sum_{k=0}^{\lceil m/2 \rceil - 1} (f(2k+2)-f(2k)) \\
&= \lceil m/2 \rceil + f(2\lceil m/2 \rceil) - f(0) \\
&\leqslant \lceil m/2 \rceil + n - 1
\end{aligned}$$

komórek, co jest rzędu $O(m+n)$.

(e) Na ostatnim poziomie rekursji wyznaczenie minimum jednego wiersza tablicy wymaga sprawdzenia co najwyżej $O(n)$ komórek. Na podstawie tego faktu i oszacowania pokazanego w poprzednim punkcie formułujemy następującą rekurencję opisującą pesymistyczny czas działania algorytmu:

$$T(m,n) = \begin{cases} O(n), & \text{jeśli } m = 1, \\ T(\lceil m/2 \rceil, n) + O(m+n), & \text{jeśli } m > 1. \end{cases}$$

Dla uproszczenia pominiemy sufit w argumencie rekurencji. Na i-tym poziomie rekurencja ta wprowadza koszt równy $O(m/2^i + n)$. Łatwo zauważyć, że jest $\lg m + 1$ poziomów, a zatem całkowity koszt wynosi:

$$\begin{aligned} T(m,n) &= \sum_{i=0}^{\lg m - 1} O\left(\frac{m}{2^i} + n\right) + O(n) \\ &= O\left(m \sum_{i=0}^{\lg m - 1} \frac{1}{2^i}\right) + O(n \lg m) \\ &\leqslant O\left(m \sum_{i=0}^{\infty} \frac{1}{2^i}\right) + O(n \lg m) \\ &= O(m + n \lg m). \end{aligned}$$

Rozdział **5**

Analiza probabilistyczna i algorytmy randomizowane

5.1. Problem zatrudnienia sekretarki

5.1-1. Niech $\preceq$ będzie relacją określoną na zbiorze rang kandydatek, za pomocą której rozstrzygamy, która kandydatka z dwóch testowanych jest lepsza. Na wejściu procedury HIRE-ASSISTANT może pojawić się każda permutacja kandydatek, więc jesteśmy w stanie rozstrzygać o każdej parze kandydatek. Pozostaje zatem udowodnić, że $\preceq$ jest porządkiem częściowym.

Możemy bezpiecznie założyć, że relacja $\preceq$ jest zwrotna, jako że nie testujemy żadnej kandydatki z nią samą. Jeśli $\preceq$ nie byłoby antysymetryczne, to w zależności od kolejności pojawienia się na wejściu procedury pewnych dwóch kandydatek, za lepszą mogłaby zostać uznana którakolwiek z tej pary, co przeczyłoby założeniu. Podobnie można wykazać, że $\preceq$ jest przechodnie, bowiem w przeciwnym przypadku dla pewnych trzech kandydatek, o tym, która z nich jest najlepsza, decydowałaby ich permutacja wejściowa.

5.1-2. Poniższy algorytm implementuje generator liczb losowych z zakresu $a\,..\,b$, korzystając jedynie z pomocniczych wywołań RANDOM$(0, 1)$.

```
RANDOM(a, b)
1  while a < b
2        do mid ← ⌊(a + b)/2⌋
3           if RANDOM(0, 1) = 0
4              then a ← mid + 1
5              else  b ← mid
6  return a
```

Niech $n = b - a + 1$ będzie długością zakresu generowania. W każdym wywołaniu rekurencyjnym odrzucana jest połowa zakresu z dokładnością do jednego elementu. Działanie procedury jest więc analogiczne do pesymistycznego przypadku wyszukiwania binarnego w n-elementowej tablicy, przez co średnio działa ona w czasie opisanym przez rekurencję z zad. 4.3-3. Rozwiązaniem tej rekursji jest $T(n) = \Theta(\lg n)$, a zatem oczekiwany czas działania procedury RANDOM(a, b) w zależności od a i b wynosi $\Theta(\lg(b - a))$.

5.1-3. Zauważmy, że prawdopodobieństwo uzyskania najpierw orła, a potem reszki w dwóch rzutach monetą jest takie samo, jak uzyskanie najpierw reszki, a potem orła i wynosi $p(1 - p)$.

Będziemy zatem rzucać monetą po dwa razy, aż do uzyskania różnych wyników. Jako wynik procedury przyjmiemy wynik pierwszego rzutu w ostatniej parze rzutów.

Następujący algorytm implementuje powyższy opis:

```
UNBIASED-RANDOM
1  repeat  x ← BIASED-RANDOM
2          y ← BIASED-RANDOM
3     until x ≠ y
4  return x
```

Załóżmy, że każda iteracja pętli **repeat** odbywa się w czasie stałym. Kolejne iteracje tworzą ciąg prób Bernoulliego, w których sukcesem jest warunek z wiersza 3, zachodzący z prawdopodobieństwem $2p(1-p)$. Oczekiwana liczba prób aż do osiągnięcia sukcesu jest zadana wzorem (C.31) i wynosi $1/(2p(1-p))$. Stąd wnioskujemy, że oczekiwanym czasem działania algorytmu jest $\Theta(1/(p(1-p)))$.

5.2. Zmienne losowe wskaźnikowe

5.2-1. Zatrudnienie tylko jednej kandydatki jest równoważne przyjęciu pierwszej z nich i tylko jej. Zauważmy, że pierwszą kandydatkę przyjmujemy w procedurze HIRE-ASSISTANT w każdym przypadku. Jeśli ma ona być jedyną zatrudnioną osobą, to powinna być najbardziej wykwalifikowaną w zbiorze wszystkich kandydatek (czyli mieć największą wartość *rank*). Najlepsza kandydatka może znajdować się na każdym z n miejsc w ciągu wejściowym, zatem prawdopodobieństwo tego, że będzie zajmować pierwszą pozycję, jest równe $1/n$.

By dokonać zatrudnienia wszystkich n kandydatek, musimy przesłuchiwać je w kolejności rosnących rang. Jest tylko jedna taka permutacja wejściowa, zatem prawdopodobieństwo tego zdarzenia wynosi $1/n!$.

5.2-2. Zauważmy, że zarówno kandydatka z pierwszej pozycji w ciągu wejściowym, jak również ta o najwyższej randze, są zatrudniane w każdym przypadku. Jeśli procedura HIRE-ASSISTANT ma dokonać dokładnie dwóch zatrudnień, to kandydatka z numerem 1 powinna mieć rangę $i \leqslant n-1$, a wszystkie kandydatki o rangach $i+1, i+2, \ldots, n-1$ powinny występować w ciągu po kandydatce z rangą równą n.

Oznaczmy przez E_i zdarzenie, że pierwsza kandydatka ma rangę równą i. Zachodzi oczywiście $\Pr(E_i) = 1/n$ dla każdego $i = 1, 2, \ldots, n$. Przyjmijmy, że j jest pozycją najlepszej kandydatki w ciągu i niech F będzie zdarzeniem polegającym na tym, że kandydatki o numerach $2, 3, \ldots, j-1$ mają rangi mniejsze od rangi kandydatki numer 1. Jeśli zachodzi E_i, to F zachodzi tylko wtedy, gdy $i \neq n$, a spośród $n-i$ kandydatek, których rangi są większe niż i, ta z rangą równą n przesłuchiwana jest najwcześniej. Stąd mamy $\Pr(F \mid E_i) = 1/(n-i)$, o ile $i \neq n$. Niech w końcu A oznacza zdarzenie, że w procedurze HIRE-ASSISTANT zatrudniane są dokładnie dwie osoby. Ponieważ zdarzenia $E_1, E_2, \ldots, E_n$ są rozłączne, to zachodzi

$$A = F \cap (E_1 \cup E_2 \cup \cdots \cup E_{n-1}) = (F \cap E_1) \cup (F \cap E_2) \cup \cdots \cup (F \cap E_{n-1})$$

oraz

$$\Pr(A) = \sum_{i=1}^{n-1} \Pr(F \cap E_i).$$

Z tożsamości (C.14),

$$\Pr(F \cap E_i) = \Pr(F \mid E_i)\Pr(E_i) = \frac{1}{n-i} \cdot \frac{1}{n},$$

a zatem

$$\Pr(A) = \sum_{i=1}^{n-1} \frac{1}{n-i} \cdot \frac{1}{n} = \frac{1}{n} \sum_{i=1}^{n-1} \frac{1}{n-i} = \frac{1}{n} \sum_{i=1}^{n-1} \frac{1}{i} = \frac{H_{n-1}}{n}.$$

5.2-3. Obliczmy wartość oczekiwaną liczby oczek w jednym rzucie kostką. Definiując zmienną losową X_i jako liczbę oczek na i-tej kostce ($i = 1, 2, \ldots, n$), obliczamy $\mathrm{E}(X_i)$, przyjmując, że zmienne X_i mają rozkład jednostajny (prawdopodobieństwo każdego wyniku jest równe $1/6$):

$$\mathrm{E}(X_i) = \sum_x x \Pr(X_i = x) = \frac{1+2+3+4+5+6}{6} = 3{,}5.$$

Niech zmienna losowa X oznacza sumę oczek na n kostkach. Mamy $X = X_1 + X_2 + \cdots + X_n$, więc z liniowości wartości oczekiwanej

$$\mathrm{E}(X) = \mathrm{E}\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} \mathrm{E}(X_i) = 3{,}5n.$$

5.2-4. Niech S_i, dla $i = 1, 2, \ldots, n$, będzie zdarzeniem oznaczającym, że i-ta osoba otrzymała swój kapelusz. Definiujemy teraz zmienne losowe $X_i = \mathrm{I}(S_i)$ oraz $X = X_1 + X_2 + \cdots + X_n$, przy czym X oznacza liczbę osób, którym zwrócono właściwe kapelusze. Mamy

$$\mathrm{E}(X) = \mathrm{E}\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} \mathrm{E}(X_i) = \sum_{i=1}^{n} \Pr(X_i = 1) = \sum_{i=1}^{n} \frac{1}{n} = 1,$$

a zatem swój kapelusz otrzyma średnio tylko jedna osoba.

5.2-5. Dla wszystkich całkowitych i, j takich, że $1 \leqslant i < j \leqslant n$, zdefiniujmy zdarzenia S_{ij} – w tablicy A występuje inwersja $\langle i, j \rangle$. Szanse na to, aby elementy na pozycjach i oraz j tworzyły inwersję, są równe $1/2$. Definiujemy zmienne losowe $X_{ij} = \mathrm{I}(S_{ij})$ oraz $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$, przy czym zmienna X oznacza łączną liczbę inwersji tablicy A. Jej wartością oczekiwaną jest

$$\mathrm{E}(X) = \mathrm{E}\left(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}(X_{ij}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(X_{ij} = 1)$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} = \frac{1}{2} \sum_{i=1}^{n-1} (n-i) = \frac{1}{2} \sum_{i=1}^{n-1} i = \frac{n(n-1)}{4}.$$

5.3. Algorytmy randomizowane

5.3-1. Oto zmodyfikowana procedura RANDOMIZE-IN-PLACE:

RANDOMIZE-IN-PLACE′(A)

```
1  n ← length[A]
2  zamień A[1] ↔ A[RANDOM(1, n)]
3  for i ← 2 to n
4        do zamień A[i] ↔ A[RANDOM(i, n)]
```

Treść niezmiennika pozostaje taka sama (z wyjątkiem fragmentu, który podaje linie kodu zawierające ciało pętli). Modyfikacji wymaga jedynie dowód jego pierwszej własności.

Inicjowanie: Gdy $i = 2$, niezmiennik pętli mówi, że dla każdej 1-permutacji fragment tablicy $A[1\,..\,1]$ zawiera tę permutację z prawdopodobieństwem $(n-1)!/n! = 1/n$. Podtablica $A[1\,..\,1]$ stanowi tylko jeden element $A[1]$, który z prawdopodobieństwem $1/n$ jest pewnym ustalonym elementem spośród n elementów tablicy. A więc niezmiennik jest spełniony przed pierwszą iteracją.

5.3-2. *Przedstawiony w treści zadania algorytm jest podany niepoprawnie, ponieważ wynik wywołania* RANDOM$(i+1, n)$ *jest niezdefiniowany, gdy i przyjmuje wartość n. Pętla* ***for*** *w tej procedurze powinna iterować po wszystkich i od* 1 *do* $n-1$.

Algorytm ten nie działa zgodnie z zamierzeniem. Jako przykład weźmy dowolną tablicę o $n = 3$ elementach. Istnieje $n! - 1 = 5$ permutacji tej tablicy różnych niż identycznościowa. Pętla **for** w pierwszej iteracji zamienia pierwszy element tablicy z losowo wybranym z pozostałych dwóch. W drugiej iteracji może zostać wybrana tylko jedna wartość na drugi element. Za pomocą tej procedury jesteśmy więc w stanie utworzyć tylko dwie permutacje wejściowej tablicy.

5.3-3. Zauważmy, że kolejne wywołania generatora liczb losowych w procedurze PERMUTE-WITH-ALL generują jeden z n^n możliwych ciągów pozycji tablicy, podczas gdy istnieje $n!$ możliwych wyników procedury. Załóżmy, że $n > 2$ i że procedura generuje każdą permutację z jednakowym prawdopodobieństwem. A zatem każdej permutacji na wyjściu odpowiada stała liczba c ciągów indeksów, czyli $n^n = cn!$. W tym wzorze $n-1$ dzieli prawą stronę, a więc powinno dzielić także lewą. Ale to nie jest prawdą, gdyż ze wzoru (A.5) dla $x = n$ mamy

$$\sum_{k=0}^{n-1} n^k = \frac{n^n - 1}{n-1},$$

skąd dostajemy

$$n^n = (n-1)\sum_{k=0}^{n-1} n^k + 1,$$

czyli n^n daje resztę 1 przy dzieleniu przez $n-1$. Na podstawie otrzymanej sprzeczności wnioskujemy, że procedura PERMUTE-WITH-ALL nie generuje permutacji losowych zgodnie z rozkładem jednostajnym.

5.3-4. Na początku działania procedury losowana jest liczba *offset*, o jaką zostaną przesunięte elementy tablicy A cyklicznie w prawo. Element z pozycji i znajdzie się w wyniku tego przesunięcia na pozycji $dest = (i + offset) \bmod n$ w tablicy B. Ponieważ istnieje n możliwych wartości zmiennej *offset*, to szanse, że element $A[i]$ znajdzie się na pewnej ustalonej pozycji w B, są równe $1/n$.

Ponieważ nie jest zmieniana wzajemna kolejność elementów, to nie każdą permutację można otrzymać w wyniku działania tej procedury – na przykład nie dostaniemy nigdy permutacji będącej odwróceniem tablicy wejściowej, o ile jej rozmiar jest większy niż 2.

5.3-5. Spróbujmy skonstruować tablicę P, w której wszystkie elementy są różne. Na pierwszy element tej tablicy możemy wybrać jedną z n^3 liczb, drugi element może przyjąć jedną z $n^3 - 1$ pozostałych wartości, trzeci – jedną z $n^3 - 2$ pozostałych itd. Ogólnie, i-ty z kolei element

tablicy P może być jedną z $n^3 - i + 1$ liczb pozostałych po poprzednich wyborach. A zatem prawdopodobieństwo tego, że wszystkie elementy tablicy P są różne, wynosi

$$\prod_{i=1}^{n} \frac{n^3 - i + 1}{n^3} = \prod_{i=0}^{n-1} \frac{n^3 - i}{n^3} = \prod_{i=0}^{n-1} \left(1 - \frac{i}{n^3}\right) > \prod_{i=0}^{n-1} \left(1 - \frac{n}{n^3}\right) = \left(1 - \frac{1}{n^2}\right)^n.$$

Wykorzystując teraz fakt, że ciąg $e_n = (1 - 1/n)^n$ jest rosnący, otrzymujemy, że

$$\left(1 - \frac{1}{n^2}\right)^{n^2} \geqslant \left(1 - \frac{1}{n}\right)^n$$

i po zastosowaniu pierwiastka n-tego stopnia do obu stron nierówności otrzymujemy żądany wynik.

5.3-6. Gdy dwa priorytety powtarzają się, czyli $P[i] = P[j]$ dla pewnych $i \neq j$, to deterministyczny algorytm sortujący szereguje odpowiadające im elementy $A[i]$ oraz $A[j]$ zawsze w tej samej kolejności. W skrajnym przypadku, jeśli wszystkie priorytety w P są identyczne, to generowana będzie tylko jedna permutacja tablicy A.

Rozwiązaniem problemu powtarzających się priorytetów jest użycie randomizowanego algorytmu sortującego wykorzystującego porównania. Za każdym razem, gdy porównywane elementy x i y okażą się równe, algorytm ten będzie losowo – z równym prawdopodobieństwem – wybierał, czy potraktować relację między nimi jako $x < y$, czy jako $x > y$. Dzięki temu każda wejściowa tablica priorytetów z punktu widzenia algorytmu sortowania będzie zawierała liczby parami różne, których każda permutacja może pojawić się na wejściu z jednakowym prawdopodobieństwem.

5.4. Analiza probabilistyczna i dalsze zastosowania zmiennych losowych wskaźnikowych

5.4-1. Podobnie jak w analizie paradoksu dnia urodzin przyjmiemy, że n jest liczbą dni w roku i ponumerujemy osoby znajdujące się w pokoju liczbami całkowitymi 1, 2, ..., k. Dla $i = 1, 2, \ldots, k$ niech A_i będzie zdarzeniem polegającym na tym, że osoba i ma urodziny kiedy indziej niż ja. Wówczas

$$B_k = \bigcap_{i=1}^{k} A_i$$

jest zdarzeniem, że żadna z k osób nie ma urodzin wtedy co ja. Dla każdego $i = 1, 2, \ldots, k$ zachodzi $\Pr(A_i) = 1 - 1/n$. Przy założeniu, że zdarzenia $A_1, A_2, \ldots, A_k$ są wzajemnie niezależne, mamy

$$\Pr(B_k) = \Pr\left(\bigcap_{i=1}^{k} A_i\right) = \prod_{i=1}^{k} \Pr(A_i) = \prod_{i=1}^{k} \left(1 - \frac{1}{n}\right) = \left(1 - \frac{1}{n}\right)^k.$$

Prawdopodobieństwo zdarzenia, że wśród k osób jest przynajmniej jedna, która ma urodziny tego samego dnia co ja, ma być mniejsze niż $1/2$, czyli

$$\left(1 - \frac{1}{n}\right)^k \leqslant \frac{1}{2}.$$

Rozwiązując tę nierówność ze względu na k, otrzymujemy $k \geqslant \log_{1-1/n}(1/2)$ i po przyjęciu $n = 365$ mamy, że najmniejszym całkowitym k spełniającym tę nierówność jest $k = 253$.

W rozwiązaniu drugiej części zadania pozostaniemy przy poprzednim znaczeniu symbolu n i numeracji osób kolejnymi liczbami całkowitymi. Ponadto przez r oznaczymy dzień 3 maja. Niech teraz B_k będzie zdarzeniem polegającym na tym, że wśród k osób co najwyżej jedna ma urodziny w dniu r oraz A_i, dla $i = 1, 2, \ldots, k$, niech będzie zdarzeniem, że osoba i ma urodziny w inny dzień niż r. Podobnie jak w pierwszej części zadania zachodzi $\Pr(A_1) = \Pr(A_2) = \cdots = \Pr(A_k) = 1 - 1/n$. Dla $i = 1, 2, \ldots, k$ zdefiniujmy jeszcze

$$C_i = \overline{A_i} \cap \bigcap_{\substack{j=1 \\ j \neq i}}^{k} A_j$$

jako zdarzenie polegające na tym, że wśród k osób tylko i-ta obchodzi urodziny dnia r. Zachodzi

$$B_k = \bigcap_{i=1}^{k} A_i \cup \bigcup_{i=1}^{k} C_i.$$

Obliczmy $\Pr(C_i)$, zakładając, że zdarzenia $A_1, A_2, \ldots, A_k$ są wzajemnie niezależne:

$$\Pr(C_i) = (1 - \Pr(A_i)) \prod_{\substack{j=1 \\ j \neq i}}^{k} \Pr(A_j) = \frac{1}{n} \prod_{\substack{j=1 \\ j \neq i}}^{k} \left(1 - \frac{1}{n}\right) = \frac{1}{n}\left(1 - \frac{1}{n}\right)^{k-1}.$$

Zdarzenia $\bigcap_{i=1}^{k} A_i, C_1, C_2, \ldots, C_k$ wzajemnie się wykluczają, a więc dostajemy

$$\begin{aligned}
\Pr(B_k) &= \Pr\left(\bigcap_{i=1}^{k} A_i\right) + \Pr\left(\bigcup_{i=1}^{k} C_i\right) \\
&= \prod_{i=1}^{k} \Pr(A_i) + \sum_{i=1}^{k} \Pr(C_i) \\
&= \left(1 - \frac{1}{n}\right)^k + \frac{k}{n}\left(1 - \frac{1}{n}\right)^{k-1} \\
&= \left(1 + \frac{k-1}{n}\right)\left(1 - \frac{1}{n}\right)^{k-1}.
\end{aligned}$$

To czego poszukujemy, to najmniejsze k takie, że $\Pr(B_k) < 1/2$. Można w tym momencie przyjąć $n = 365$ i obliczać prawdopodobieństwa B_k dla kolejnych naturalnych wartości k. W wyniku takich obliczeń można ustalić, że szukaną wartością jest $k = 613$.

5.4-2. Niech X oznacza liczbę potrzebnych rzutów, zanim w pewnej urnie znajdą się dwie kule. Załóżmy, że po k rzutach ($k = 1, 2, \ldots, b$) nie było urny z więcej niż jedną kulą i obliczmy szanse, że również po $(k+1)$-szym rzucie nie będzie kolizji. Ponieważ jest zajętych k urn, to $(k+1)$-sza kula wpada do pustej urny z prawdopodobieństwem równym

$$\Pr(X > k+1 \mid X > k) = \frac{b-k}{b}.$$

Zachodzi

$$\Pr(X > k+1) = \prod_{i=1}^{k} \Pr(X > i+1 \mid X > i) = \frac{(b-1)(b-2)\ldots(b-k)}{b^k}.$$

Oczywiście $\Pr(X = 1) = 0$ i $\Pr(X > 1) = 1$. Dla $k = 1, 2, \ldots, b$ mamy

$$\Pr(X = k+1) = \Pr(X > k) - \Pr(X > k-1) = \frac{(b-1)(b-2)\ldots(b-k+1)k}{b^k} = \frac{b!\,k}{(b-k)!\,b^{k+1}}.$$

Zajmijmy się teraz wartością oczekiwaną zmiennej losowej X:

$$\begin{aligned}
\mathrm{E}(X) &= \sum_{k=0}^{b}(k+1)\Pr(X = k+1) \\
&= \frac{b!}{b^b}\sum_{k=0}^{b}\frac{b^{b-k}k}{(b-k)!\,b}(k+1) \\
&= \frac{b!}{b^b}\sum_{k=0}^{b}\frac{b^k(b-k)}{k!\,b}(b-k+1) \\
&= \frac{b!}{b^b}\left(\sum_{k=0}^{b}\frac{b^k}{k!}(b-k+1) - \sum_{k=0}^{b}\frac{b^k k}{k!\,b}(b-k+1)\right) \\
&= \frac{b!}{b^b}\left(\sum_{k=0}^{b}\frac{b^k}{k!}(b-k+1) - \sum_{k=1}^{b}\frac{b^{k-1}}{(k-1)!}(b-k+1)\right) \\
&= \frac{b!}{b^b}\left(\sum_{k=0}^{b}\frac{b^k}{k!}(b-k+1) - \sum_{k=0}^{b-1}\frac{b^k}{k!}(b-k)\right) \\
&= \frac{b!}{b^b}\left(\sum_{k=0}^{b}\frac{b^k}{k!}(b-k+1) - \sum_{k=0}^{b}\frac{b^k}{k!}(b-k)\right) \\
&= \frac{b!}{b^b}\sum_{k=0}^{b}\frac{b^k}{k!}.
\end{aligned}$$

Ostatnie wyrażenie jest badane w [8], gdzie wyprowadzono jego oszacowanie $\sqrt{b\pi/2} + O(1)$, a zatem $E(X) = \Theta(\sqrt{b})$.

5.4-3. W analizie paradoksu dnia urodzin niezależność urodzin wykorzystuje się jedynie we wzorze

$$\Pr(b_i = r \text{ i } b_j = r) = \Pr(b_i = r)\Pr(b_j = r) = 1/n^2.$$

Wystarczy zatem założenie, że zdarzenia te są parami niezależne.

5.4-4. Niech n będzie liczbą dni w roku. Oznaczmy przez $P_1(k, n)$ prawdopodobieństwo tego, że wszystkie osoby z k-osobowej grupy mają urodziny w różne dni, a $P_2(k, n)$ niech będzie prawdopodobieństwem tego, że pewnego dnia w roku urodziły się dokładnie dwie osoby z tej grupy. Szanse na to, aby wśród tych k osób co najmniej troje miało urodziny tego samego dnia, są równe

$$P(k, n) = 1 - (P_1(k, n) + P_2(k, n)).$$

W klasycznym problemie dnia urodzin zostało wyznaczone

$$P_1(k, n) = \frac{n!}{(n-k)!\,n^k} = \frac{k!}{n^k}\binom{n}{k},$$

pozostaje zatem obliczyć $P_2(k,n)$.

Spośród n dni w roku wybierzmy jeden, który będzie urodzinami pewnych dwóch osób z naszej grupy. Pozostałe $k-2$ osób możemy rozdzielić między $n-1$ dni oznaczających ich urodziny. Ponieważ rozróżniamy osoby, to liczbę sposobów takiego wyboru należy pomnożyć przez liczbę ich permutacji $k!$ i podzielić przez 2 z racji tego, że zmiana kolejności dwóch osób wybranych do tego samego dnia w roku nie jest odrębnym przypadkiem. Ale takich par może być więcej. Można tak naprawdę wybrać $i \leqslant \lfloor k/2 \rfloor$ różnych dni, którym przypiszemy pary osób wtedy urodzonych – liczba możliwości wynosi $\binom{n}{i}$. Pozostałe osoby rozdzielamy między pozostałe dni w roku tak, aby każdej przypadł inny dzień, co da się wykonać na $\binom{n-i}{k-2i}$ sposobów. Podobnie jak wcześniej rozróżnianie osób wprowadza czynnik $k!/2^i$ – kolejność w parze w ramach tego samego dnia jest bowiem nieistotna. Ponieważ liczba możliwości przypisania k osób do n różnych dni wynosi n^k, to ostatecznie otrzymujemy wzór

$$P_2(k,n) = \frac{k!}{n^k} \sum_{i=1}^{\lfloor k/2 \rfloor} \frac{1}{2^i} \binom{n}{i} \binom{n-i}{k-2i}.$$

Ustalając wartość n na 365, można teraz obliczyć prawdopodobieństwa $P(k,n)$ dla wszystkich $k = 1, 2, \ldots, n$, po czym wyznaczyć najmniejszą wartość k, dla której $P(k,n) \geqslant 1/2$. Okazuje się, że rozwiązaniem jest $k = 88$.

5.4-5. Wszystkich możliwych k-słów nad zbiorem n-elementowym jest n^k. Aby k-słowo było w istocie k-permutacją, na pierwszy z jego elementów należy wybrać jeden z n elementów zbioru, na drugi element jeden z $n-1$ dotychczas niewybranych itd. Istnieje zatem $n(n-1)\ldots(n-k+1)$ możliwych k-permutacji, więc prawdopodobieństwo, że dane k-słowo będzie jedną z nich wynosi

$$\frac{n(n-1)\ldots(n-k+1)}{n^k} = \left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\ldots\left(1-\frac{k-1}{n}\right).$$

Problem jest analogiczny do pytania o prawdopodobieństwo zdarzenia, że wśród k osób nie ma dwóch takich, które urodziły się tego samego dnia roku, gdzie n jest liczbą dni w roku.

5.4-6. Obliczmy najpierw oczekiwaną liczbę pustych urn. Niech S_i, dla $i = 1, 2, \ldots, n$, będzie zdarzeniem, że i-ta urna jest pusta po wykonaniu n rzutów. Definiujemy zmienną losową $X_i = \mathrm{I}(S_i)$ oraz $X = \sum_{i=1}^{n} X_i$, która oznacza liczbę pustych urn. Wtedy

$$\mathrm{E}(X) = \mathrm{E}\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} \mathrm{E}(X_i) = \sum_{i=1}^{n} \Pr(S_i).$$

Rozważmy teraz i-tą urnę i potraktujmy każdy rzut kulą jako próbę Bernoulliego, gdzie sukcesem jest trafienie do tej urny. Mamy zatem n niezależnych prób Bernoulliego, każda z prawdopodobieństwem sukcesu $p = 1/n$. Aby i-ta urna pozostała pusta, nie możemy uzyskać żadnego sukcesu, a zatem korzystając z rozkładu dwumianowego, dostajemy

$$\Pr(S_i) = b(0;n,p) = \binom{n}{0}\left(\frac{1}{n}\right)^0\left(1-\frac{1}{n}\right)^n = \left(1-\frac{1}{n}\right)^n$$

oraz

$$\mathrm{E}(X) = \sum_{i=1}^{n}\left(1-\frac{1}{n}\right)^n = n\left(1-\frac{1}{n}\right)^n.$$

Wyznaczmy teraz oczekiwaną liczbę urn z dokładnie jedną kulą. W tym celu, podobnie jak poprzednio, dla $i = 1, 2, \ldots, n$ zdefiniujemy zdarzenie S_i, że i-ta urna po wykonaniu n rzutów zawiera dokładnie jedną kulę. Definicje zmiennych losowych X_i oraz X pozostają bez zmian i, tak jak poprzednio, zachodzi

$$\mathrm{E}(X) = \sum_{i=1}^{n} \Pr(S_i).$$

Dla analogicznej serii prób Bernoulliego stwierdzamy, że aby i-ta urna zawierała dokładnie jedną kulę, potrzebny jest 1 sukces i $n-1$ porażek, więc

$$\Pr(S_i) = b(1; n, p) = \binom{n}{1}\left(\frac{1}{n}\right)^1\left(1-\frac{1}{n}\right)^{n-1} = \left(1-\frac{1}{n}\right)^{n-1}$$

oraz

$$\mathrm{E}(X) = \sum_{i=1}^{n}\left(1-\frac{1}{n}\right)^{n-1} = n\left(1-\frac{1}{n}\right)^{n-1}.$$

5.4-7. W zadaniu przyjmujemy, że $n > 16$, ponieważ wtedy wyrażenie $\lg n - 2\lg\lg n$ jest dodatnie. Ponadto dla uproszczenia rachunków nie dbamy o to, aby niektóre liczby były całkowite.

Korzystając z przedstawionego w Podręczniku wyprowadzenia, mamy, że prawdopodobieństwo zdarzenia, że ciąg orłów długości co najmniej $\lg n - 2\lg\lg n$ rozpoczyna się na pozycji i, jest równe

$$\Pr(A_{i,\,\lg n - 2\lg\lg n}) = \frac{1}{2^{\lg n - 2\lg\lg n}} = \frac{2^{2\lg\lg n}}{2^{\lg n}} = \frac{\lg^2 n}{n},$$

a zatem prawdopodobieństwo, że ciąg orłów o długości co najmniej $\lg n - 2\lg\lg n$ nie rozpoczyna się na pozycji i, wynosi

$$1 - \frac{\lg^2 n}{n}.$$

Podzielmy ciąg n rzutów monetą na $n/(\lg n - 2\lg\lg n)$ grup po $\lg n - 2\lg\lg n$ kolejnych rzutów każda. Grupy te złożone są z różnych i wzajemnie niezależnych rzutów, a zatem prawdopodobieństwo, że żadna z nich nie będzie ciągiem orłów o długości $\lg n - 2\lg\lg n$, wynosi

$$\begin{aligned}\left(1-\frac{\lg^2 n}{n}\right)^{n/(\lg n - 2\lg\lg n)} &\leqslant \left(e^{-(\lg^2 n)/n}\right)^{n/(\lg n - 2\lg\lg n)} \\ &= e^{-(\lg^2 n)/(\lg n - 2\lg\lg n)} \\ &< e^{-\lg n} \\ &= 1/n.\end{aligned}$$

Skorzystaliśmy tutaj z nierówności (3.11) oraz z tego, że dla $n > 16$ zachodzi

$$\frac{\lg^2 n}{\lg n - 2\lg\lg n} > \lg n.$$

Problemy

5-1. Zliczanie probabilistyczne

(a) Zdefiniujmy X_j, dla $j = 1, 2, \ldots, n$, jako zmienną losową oznaczającą liczbę, o jaką zwiększy się wartość reprezentowana przez licznik po j-tym wykonaniu operacji INCREMENT. Ponadto

niech zmienna losowa X przyjmuje wartość reprezentowaną przez licznik po wykonaniu n operacji INCREMENT. Zachodzi $X = \sum_{j=1}^{n} X_j$ oraz, ze względu na liniowość wartości oczekiwanej,

$$\mathrm{E}(X) = \mathrm{E}\left(\sum_{j=1}^{n} X_j\right) = \sum_{j=1}^{n} \mathrm{E}(X_j).$$

Załóżmy teraz, że przed wykonaniem j-tej operacji INCREMENT licznik przechowuje wartość i, co stanowi reprezentację n_i. Jeśli inkrementacja powiedzie się, co zdarzy się z prawdopodobieństwem równym $1/(n_{i+1} - n_i)$, to wartość reprezentowana na liczniku zwiększy się o $n_{i+1} - n_i$. Dla każdego $j = 1, 2, \ldots, n$ mamy zatem

$$\mathrm{E}(X_j) = 0 \cdot \left(1 - \frac{1}{n_{i+1} - n_i}\right) + (n_{i+1} - n_i) \cdot \frac{1}{n_{i+1} - n_i} = 1,$$

a więc

$$\mathrm{E}(X) = \sum_{j=1}^{n} \mathrm{E}(X_j) = n,$$

co należało wykazać.

(b) Dla zmiennych losowych X_j oraz X zdefiniowanych w poprzednim punkcie mamy

$$\mathrm{Var}(X) = \mathrm{Var}\left(\sum_{j=1}^{n} X_j\right) = \sum_{j=1}^{n} \mathrm{Var}(X_j),$$

co zachodzi na mocy wzoru (C.28), ponieważ zmienne $X_1, X_2, \ldots, X_n$ są parami niezależne. Mamy $n_i = 100i$, a więc zwiększenie wartości reprezentowanej przez licznik o $n_{i+1} - n_i = 100$ odbędzie się z prawdopodobieństwem $1/(n_{i+1} - n_i) = 1/100$. Ze wzoru (C.26) otrzymujemy, że dla każdego $j = 1, 2, \ldots, n$ zachodzi

$$\mathrm{Var}(X_j) = \mathrm{E}(X_j^2) - \mathrm{E}^2(X_j) = 0^2 \cdot \left(1 - \frac{1}{100}\right) + 100^2 \cdot \frac{1}{100} - 1^2 = 99,$$

a stąd

$$\mathrm{Var}(X) = \sum_{j=1}^{n} \mathrm{Var}(X_j) = 99n.$$

5-2. Wyszukiwanie w nieposortowanej tablicy

(a) Oto procedura implementująca opisaną strategię:

```
RANDOM-SEARCH(A, x)
 1  n ← length[A]
 2  for k ← 1 to n
 3        do B[k] ← FALSE
 4  checked ← 0
 5  while checked < n
 6        do i ← RANDOM(1, n)
 7           if A[i] = x
 8              then return i
 9           if B[i] = FALSE
10              then B[i] ← TRUE
11                   checked ← checked + 1
12  return NIL
```

Algorytm korzysta z pomocniczej tablicy wartości logicznych $B[1\,..\,n]$, która na pozycji i przechowuje informację o tym, czy wybrana była już i-ta pozycja tablicy A. Ponadto zmienna *checked* przechowuje liczbę testowanych dotychczas komórek. W każdej iteracji pętli **while** w wierszach 5–11 algorytm sprawdza losowo wybrany indeks tablicy A. W przypadku odnalezienia x natychmiast zwracana jest jego pozycja. Jeśli jednak element x nie zostanie odnaleziony, a bieżąca komórka tablicy A nie była jeszcze wcześniej sprawdzana, to informacja ta zostaje odnotowana w tablicy B, a zmienna *checked* jest inkrementowana. Jeśli elementu x nie ma w tablicy A, to po sprawdzeniu wszystkich indeksów co najmniej raz, algorytm zwróci specjalną wartość NIL.

(b) Niech X będzie zmienną losową oznaczającą ilość wybranych indeksów tablicy A zanim odnaleziono x. Szukanie x realizowane przez procedurę RANDOM-SEARCH jest serią prób Bernoulliego, każda z prawdopodobieństwem sukcesu $p = 1/n$. Stosując wzór (C.31), otrzymujemy, że zostanie wybranych średnio $\mathrm{E}(X) = 1/p = n$ indeksów tablicy A.

(c) Rozważmy ponownie zmienną losową X i analogiczną serię prób Bernoulliego do tej z poprzedniego punktu. Jednak w tym przypadku sukces następuje z prawdopodobieństwem $p = k/n$, a zatem średnią liczbą wybranych indeksów przed odnalezieniem x jest $\mathrm{E}(X) = 1/p = n/k$.

(d) Ten przypadek wyszukiwania można sprowadzić do problemu kolekcjonera kuponów. Pozycje tablicy A reprezentują kupony, których skompletowanie (odpowiadające sprawdzeniu wszystkich pozycji tablicy) jest celem problemu. Zgodnie z uzasadnieniem podanym w Podręczniku, aby uzbierać pełny zestaw n kuponów pojawiających się losowo, należy zdobyć ich około $n \ln n$.

(e) Procedura DETERMINISTIC-SEARCH jest identyczna z algorytmem wyszukiwania liniowego opisanego w zad. 2.1-3. Z rozwiązania zad. 2.2-3 wynika zatem, że czas tego algorytmu – wyrażony jako liczba sprawdzanych indeksów tablicy – wynosi w średnim przypadku $(n + 1)/2$, a w pesymistycznym n.

(f) Oznaczmy przez X zmienną losową przyjmującą liczbę wybranych indeksów tablicy A przed odnalezieniem x. Zdarzenie $X = i$ zachodzi wtedy i tylko wtedy, gdy pierwsza z lewej wartość x zajmuje w A pozycję i. Pozostałe $k - 1$ elementów o wartości x można rozmieścić w obszarze $A[i + 1\,..\,n]$ na $\binom{n-i}{k-1}$ sposobów. Stąd $\Pr(X = i) = \binom{n-i}{k-1}/\binom{n}{k}$. Wartość oczekiwana X wynosi zatem

$$\mathrm{E}(X) = \sum_{i=1}^{n-k+1} i \Pr(X = i) = \frac{1}{\binom{n}{k}} \sum_{i=1}^{n-k+1} i \binom{n-i}{k-1}.$$

Pokażemy przez indukcję po n, że dla dowolnego $k = 1, 2, \ldots, n$ zachodzi $E(X) = \frac{n+1}{k+1}$, co na mocy wzoru (C.8) jest równoważne z udowodnieniem tożsamości

$$\binom{n+1}{k+1} = \sum_{i=1}^{n-k+1} i\binom{n-i}{k-1}.$$

Jeśli $k = n$, to po lewej stronie powyższego wzoru mamy $\binom{n+1}{n+1} = 1$, a po prawej stronie $\sum_{i=1}^{1} i\binom{n-i}{n-1} = \binom{n-1}{n-1} = 1$. A więc w tym przypadku wzór jest prawdziwy. Pokazaliśmy przy okazji, że spełniony jest pierwszy krok indukcji, gdy $n = 1$.

W drugim kroku zakładamy, że $n > 1$ i że dla każdego $k = 1, 2, \ldots, n$ zachodzi

$$\binom{n}{k} = \sum_{i=1}^{n-k+1} i\binom{n-1-i}{k-2}.$$

Korzystając dwukrotnie z zad. C.1-7, dla dowolnego $k = 1, 2, \ldots, n-1$ mamy

$$\begin{aligned}
\binom{n+1}{k+1} &= \binom{n}{k+1} + \binom{n}{k} \\
&= \sum_{i=1}^{n-k} i\binom{n-1-i}{k-1} + \sum_{i=1}^{n-k+1} i\binom{n-1-i}{k-2} \\
&= \sum_{i=1}^{n-k} i\left(\binom{n-1-i}{k-1} + \binom{n-1-i}{k-2}\right) + (n-k+1)\binom{k-2}{k-2} \\
&= \sum_{i=1}^{n-k} i\binom{n-i}{k-1} + (n-k+1)\binom{k-1}{k-1} \\
&= \sum_{i=1}^{n-k+1} i\binom{n-i}{k-1}.
\end{aligned}$$

Wzór jest zatem prawdziwy dla wszystkich n naturalnych i wszystkich $k = 1, 2, \ldots, n$.

Pesymistyczny przypadek dla algorytmu Deterministic-Search ma miejsce wtedy, gdy wszystkie egzemplarze x zajmują w tablicy k końcowych pozycji. Algorytm sprawdzi wówczas $n - k$ komórek tablicy, zanim odnajdzie pierwsze wystąpienie x.

(g) Przypadek średni i pesymistyczny są równoważne przy braku x w tablicy A, bowiem w obu tych przypadkach algorytm przegląda całą tablicę, co zajmuje czas n.

(h) Załóżmy, że do permutowania tablicy używany jest algorytm Randomize-In-Place, który generuje permutację losową zgodnie z rozkładem jednostajnym, wykonując przy tym n zamian elementów. Czas algorytmu Scramble-Search jest wtedy sumą n oraz liczby porównań wykonywanych podczas deterministycznego wyszukiwania liniowego. Wartości te – w zależności od przypadku – zostały wyznaczone w punktach (e), (f) i (g).

(i) W przypadku gdy tablica nie zawiera szukanego elementu, czasy działania algorytmów Deterministic-Search i Scramble-Search są asymptotycznie mniejsze od czasu działania Random-Search, a w pozostałych przypadkach są one tego samego rzędu (o ile traktujemy k jako stałą). Jest to wystarczający powód, aby odrzucić algorytm Random-Search z praktycznych zastosowań. Spośród pozostałych dwóch Deterministic-Search jest bardziej efektywny,

ponieważ nie wprowadza narzutu w postaci permutowania losowego tablicy i w efekcie działa szybciej.

Okazuje się więc, że w problemie wyszukiwania zastosowanie randomizacji nie jest szczególnie pomocne i zwykłe wyszukiwanie liniowe jest algorytmem optymalnym.

Część II

Sortowanie i statystyki pozycyjne

Rozdział **6**

Heapsort – sortowanie przez kopcowanie

6.1. Kopce

6.1-1. Korzystamy z faktu, że kopiec stanowi prawie pełne drzewo binarne, tzn. takie, w którym wszystkie poziomy, być może z wyjątkiem najniższego, zawierają komplet węzłów. Jeśli drzewo to ma wysokość h, to maksymalnie może mieć $2^{h+1} - 1$ węzłów (gdy jest drzewem pełnym), a minimalnie $(2^h - 1) + 1 = 2^h$ (gdy jego najniższy poziom składa się z tylko jednego węzła).

6.1-2. Niech h oznacza wysokość kopca. Z poprzedniego zadania mamy, że $2^h \leqslant n < 2^{h+1}$, skąd dostajemy $h \leqslant \lg n < h + 1$. Ponieważ h jest całkowite, to $h = \lfloor \lg n \rfloor$.

6.1-3. Wartość korzenia każdego poddrzewa w kopcu typu max jest równa lub większa od wartości obu synów tego korzenia (o ile istnieją). Dla dowolnego poddrzewa T można łatwo dowieść przez indukcję względem jego wysokości, że wartości węzłów wchodzących w skład ścieżek od liści w górę T, tworzą ciągi niemalejące. Ponieważ wszystkie takie ścieżki kończą się w korzeniu poddrzewa T, to musi on mieć największą wartość w T.

6.1-4. Analizując ścieżki od liści do korzenia kopca jak w poprzednim zadaniu, stwierdzamy, że najmniejsza wartość w każdej takiej ścieżce znajduje się w jej pierwszym elemencie. Ponieważ warunek kopca typu max nie narzuca żadnego ograniczenia w zbiorze liści, to każdy z nich może stanowić najmniejszą wartość kopca.

6.1-5. Powtarzając rozumowanie z zad. 6.1-3 dla kopców typu min, wnioskujemy, że korzeń takiego kopca stanowi jego najmniejszy element, czyli zajmuje pierwszą pozycję w posortowanej (niemalejąco) tablicy A. Dla każdego indeksu tablicy i z wyjątkiem pierwszego zachodzi $\textsc{Parent}(i) < i$, a więc $A[\textsc{Parent}(i)] \leqslant A[i]$. Własność kopca typu min jest zatem spełniona i tablica posortowana A stanowi taki kopiec.

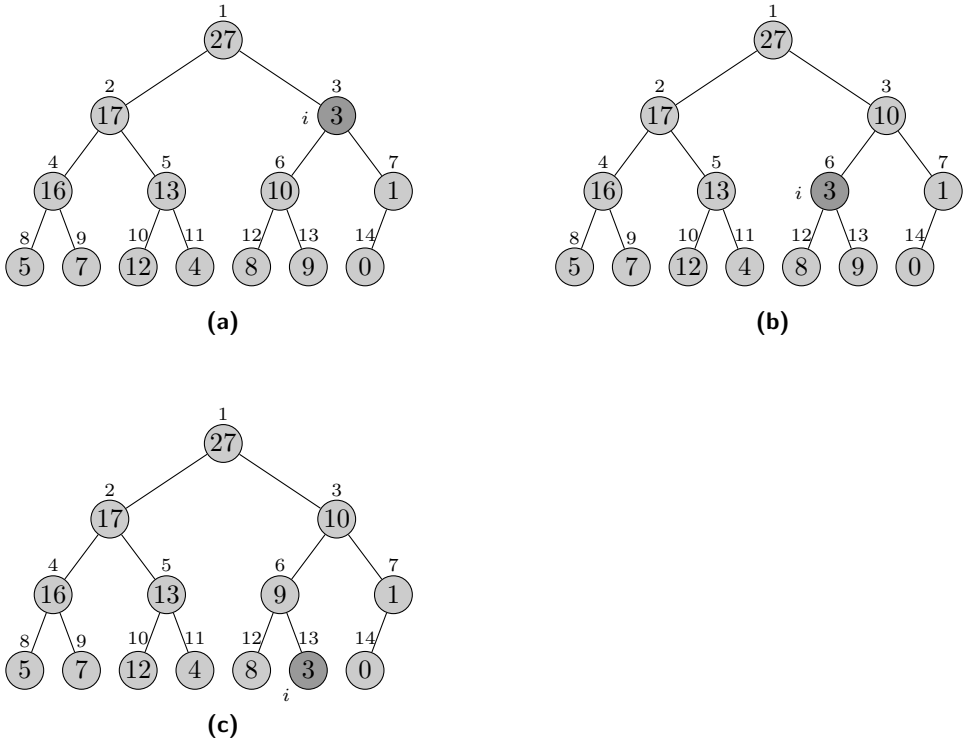
6.1-6. Potraktujmy ten ciąg jak tablicę A. Elementy na pozycjach $i = 9$ oraz $\textsc{Parent}(i) = 4$ nie spełniają własności $A[\textsc{Parent}(i)] \geqslant A[i]$, zatem tablica A nie jest kopcem typu max.

6.1-7. Element kopca na pozycji i nie jest liściem wtedy i tylko wtedy, gdy istnieje jego lewy syn. W kopcu o n elementach wierzchołki wewnętrzne znajdują się zatem na pozycjach i takich,

że LEFT$(i) \leqslant n$. Warunek ten sprowadza się do nierówności $2i \leqslant n$, skąd $i \leqslant \lfloor n/2 \rfloor$, bo i jest całkowite. Pozostałe wierzchołki są liśćmi i zajmują pozycje $\lfloor n/2 \rfloor + 1 \,..\, n$.

6.2. Przywracanie własności kopca

6.2-1. Rys. 8 przedstawia działanie procedury MAX-HEAPIFY$(A, 3)$.



Rysunek 8: Działanie procedury MAX-HEAPIFY$(A, 3)$ dla tablicy $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$. **(a)–(b)** Drzewo binarne reprezentujące A, w którym przywracana jest własność kopca typu max, odpowiednio, w węzłach $i = 3$ oraz $i = 6$. **(c)** Wynikowy kopiec z przywróconą własnością kopca.

6.2-2. Poniższy pseudokod prezentuje procedurę przywracania własności kopca typu min. Ponieważ jedyną modyfikacją w porównaniu z procedurą MAX-HEAPIFY jest zmiana znaków nierówności na przeciwne w warunkach w wierszach 3 i 6, to czas działania tej procedury jest identyczny z czasem działania MAX-HEAPIFY, czyli $\Theta(\lg n)$.

```
MIN-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ⩽ heap-size[A] i A[l] < A[i]
 4      then smallest ← l
 5      else smallest ← i
 6  if r ⩽ heap-size[A] i A[r] < A[smallest]
 7      then smallest ← r
 8  if smallest ≠ i
 9      then zamień A[i] ↔ A[smallest]
10           MIN-HEAPIFY(A, smallest)
```

6.2-3. Jeśli element $A[i]$ jest większy niż jego synowie, to *largest* jest ustawiane na i i warunek z wiersza 8 nie jest spełniony. Procedura zakończy więc działanie, nie dokonując żadnej zamiany elementów.

6.2-4. Z zad. 6.1-7 mamy, że element o indeksie $i > \textit{heap-size}[A]/2$ jest liściem kopca, czyli nie istnieją jego synowie. W dwóch pierwszych wierszach procedury MAX-HEAPIFY obliczone zostaną wartości przekraczające $\textit{heap-size}[A]$, więc zmienna *largest* przyjmie wartość i. Warunek w wierszu 8 będzie więc fałszywy i natychmiast po jego sprawdzeniu procedura zakończy działanie.

6.2-5. Iteracyjna wersja procedury MAX-HEAPIFY została przedstawiona poniżej.

```
ITERATIVE-MAX-HEAPIFY(A, i)
 1  while TRUE
 2        do l ← LEFT(i)
 3           r ← RIGHT(i)
 4           if l ⩽ heap-size[A] i A[l] > A[i]
 5              then largest ← l
 6              else largest ← i
 7           if r ⩽ heap-size[A] i A[r] > A[largest]
 8              then largest ← r
 9           if largest = i
10              then return
11           zamień A[i] ↔ A[largest]
12           i ← largest
```

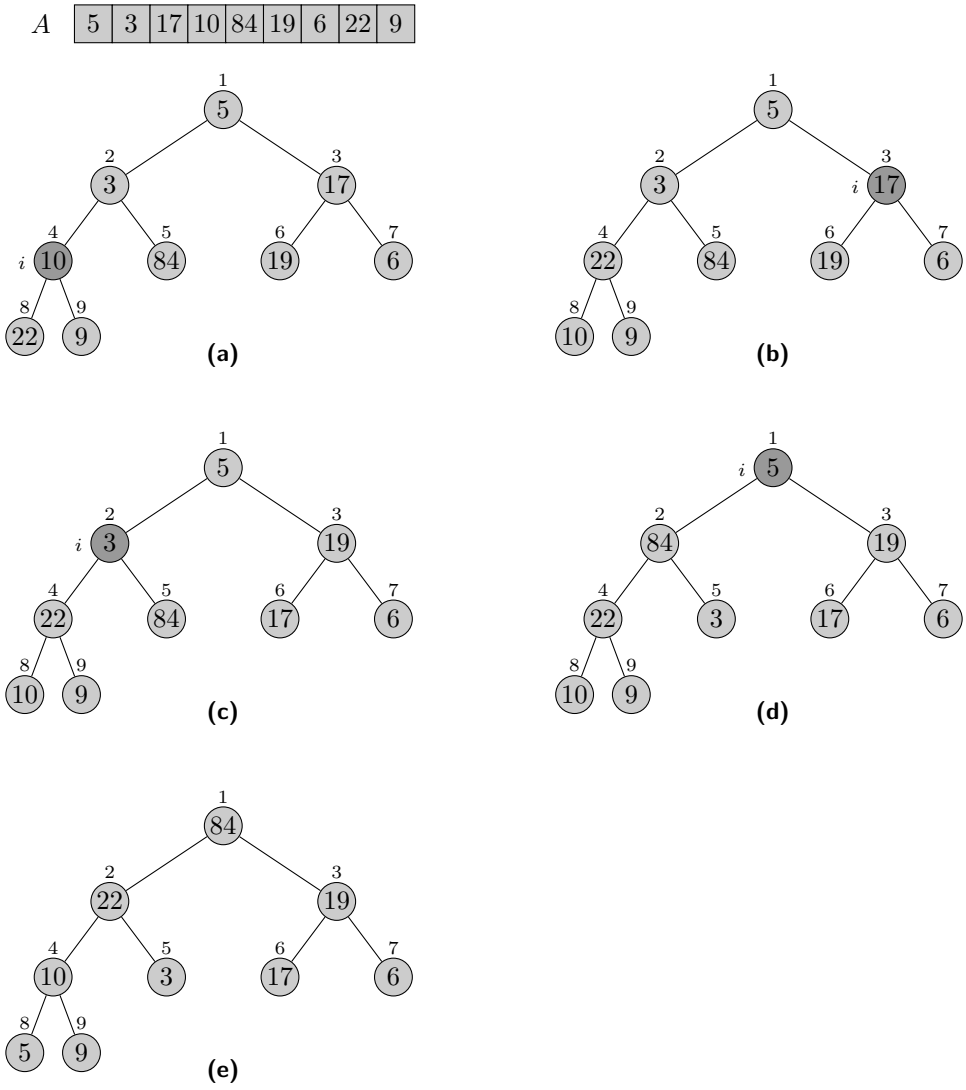
Działania wykonywane w wierszach 2–8 są identyczne jak w oryginalnej implementacji procedury. W zależności od wyniku testu z wiersza 9 procedura kończy działanie albo zamienia elementy $A[i]$ i $A[\textit{largest}]$, po czym symuluje wywołanie rekurencyjne, aktualizując wartość zmiennej i i wykonując kolejną iterację pętli **while**.

6.2-6. Najgorszy przypadek dla procedury MAX-HEAPIFY zachodzi wówczas, gdy zostanie ona wywołana dla korzenia kopca i schodzi rekurencyjnie aż do jego ostatniego poziomu. Najdłuższa ścieżka od korzenia do liścia składa się z $h = \lfloor \lg n \rfloor$ krawędzi (z zad. 6.1-2) i tyle będzie wywołań rekurencyjnych procedury w najgorszym przypadku. Koszt pracy wykonanej na każdym poziomie rekursji jest stały, a więc procedura MAX-HEAPIFY działa wtedy w czasie $\Omega(\lg n)$. Przykładowym

drzewem, dla którego procedura wykona opisane operacje, jest takie, w którym korzeń ma wartość 0, a każdy inny węzeł ma wartość 1.

6.3. Budowanie kopca

6.3-1. Ilustracja działania procedury BUILD-MAX-HEAP dla tablicy A znajduje się na rys. 9.



Rysunek 9: Działanie procedury BUILD-MAX-HEAP dla tablicy $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9\rangle$. **(a)** Tablica A i reprezentowane przez nią drzewo binarne przed pierwszym wywołaniem MAX-HEAPIFY z wiersza 3. **(b)–(d)** Drzewo przed każdym kolejnym wywołaniem MAX-HEAPIFY. **(e)** Wynikowy kopiec typu max.

6.3-2. Wywołując MAX-HEAPIFY(A, i), zakładamy, że drzewa o korzeniach LEFT(i) i RIGHT(i) (o ile istnieją) są kopcami typu max. Jeżeli podczas budowy kopca procedura MAX-HEAPIFY byłaby wywoływana dla węzłów o rosnących indeksach, to nie moglibyśmy zagwarantować, że założenie to jest spełnione, dlatego przetwarzanie odbywa się w kolejności malejących indeksów.

6.3-3. Oznaczmy kopiec przez T, a przez n_h – ilość węzłów kopca T znajdujących się na wysokości h. Udowodnimy fakt przez indukcję względem h.

W pierwszym kroku indukcji musimy pokazać, że $n_0 \leqslant \lceil n/2 \rceil$. W rzeczywistości udowodnimy, że $n_0 = \lceil n/2 \rceil$. Korzystając z zad. 6.1-7, mamy, że węzły znajdujące się w T na wysokości 0, czyli jego liście, zajmują pozycje $\lfloor n/2 \rfloor + 1 \,..\, n$. Jest ich zatem

$$n_0 = n - (\lfloor n/2 \rfloor + 1) + 1 = n - \lfloor n/2 \rfloor = \lceil n/2 \rceil.$$

A więc przypadek bazowy indukcji jest spełniony.

Załóżmy teraz, że $h > 0$ i że twierdzenie jest spełnione dla węzłów na wysokości $h-1$. Ponadto niech T' będzie kopcem powstałym z T po usunięciu z niego wszystkich jego liści. Nowy kopiec ma zatem $n' = n - n_0$ węzłów. Ponieważ w kroku bazowym pokazaliśmy, że $n_0 = \lceil n/2 \rceil$, to stąd $n' = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$. Węzły, które w kopcu T znajdują się na wysokości h, w T' zajmują wysokość $h-1$, więc jeśli oznaczymy przez n'_{h-1} liczbę węzłów na wysokości $h-1$ w kopcu T', to wówczas będzie $n_h = n'_{h-1}$. Wykorzystując założenie indukcyjne, dostajemy

$$n_h = n'_{h-1} \leqslant \lceil n'/2^h \rceil = \lceil \lfloor n/2 \rfloor / 2^h \rceil \leqslant \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil,$$

co kończy dowód.

6.4. Algorytm sortowania przez kopcowanie (heapsort)

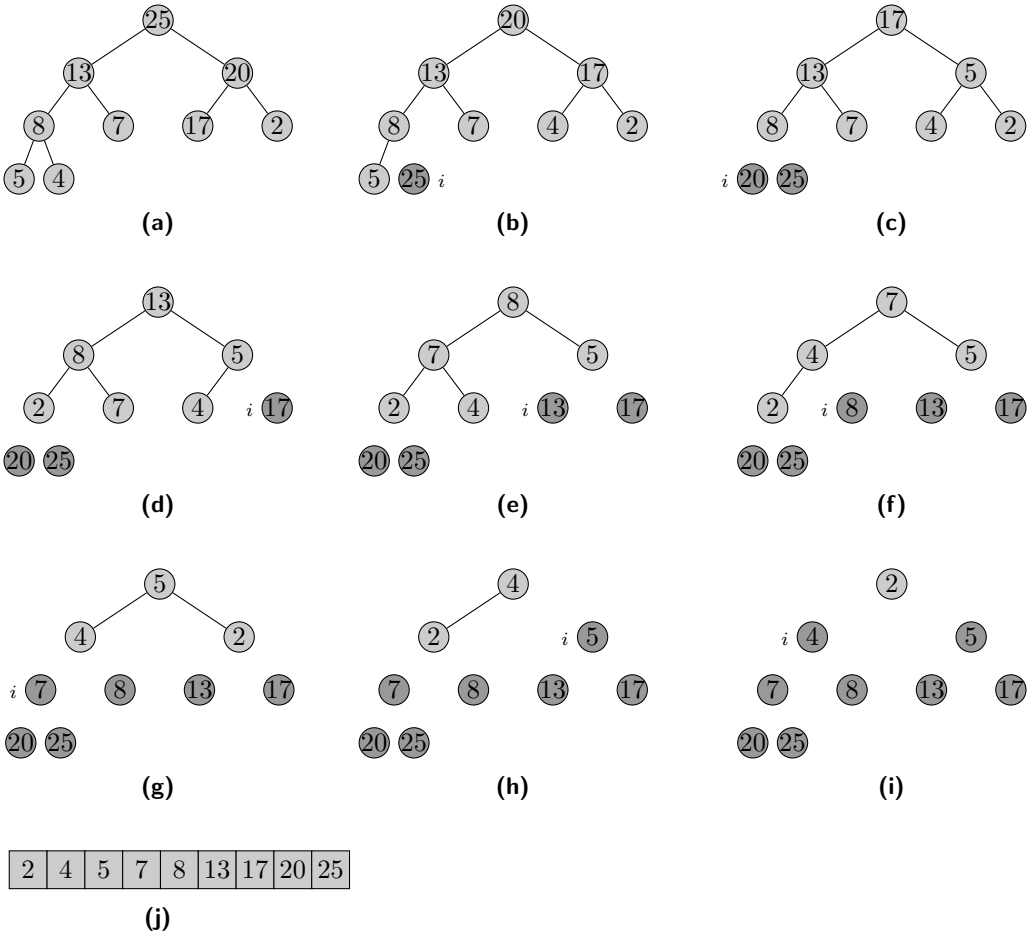
6.4-1. Na rys. 10 zilustrowano działanie sortowania przez kopcowanie dla tablicy A.

6.4-2. Pokażemy, że podany niezmiennik jest spełniony przed pierwszą iteracją pętli **for**, że każda iteracja tej pętli zachowuje niezmiennik i że po zakończeniu pętli z niezmiennika można wywnioskować poprawność procedury.

Inicjowanie: Przed pierwszą iteracją pętli mamy $i = \mathit{length}[A] = n$. Wówczas fragment $A[1 \,..\, i]$ jest całą tablicą A, która stanowi kopiec typu max, utworzony w wyniku działania procedury BUILD-MAX-HEAP, natomiast fragment $A[i+1 \,..\, n]$ jest pusty.

Utrzymanie: Załóżmy, że niezmiennik jest prawdziwy przed wykonaniem kolejnej iteracji pętli. Podtablica $A[1 \,..\, i]$ tworzy więc kopiec typu max, którego korzeniem jest $A[1]$, czyli największy element w tej podtablicy. Po wykonaniu wiersza 3 znajdzie się on na pozycji i. Dekrementacja $\mathit{heap\text{-}size}[A]$ powoduje, że element $A[i]$ nie wchodzi teraz w skład kopca, ale podtablica $A[i \,..\, n]$ zawiera teraz $n-i+1$ największych elementów z $A[1 \,..\, n]$ posortowanych niemalejąco, ponieważ element $A[i]$ jest równy lub mniejszy od wcześniej umieszczonych tam elementów. W tym momencie korzeń może naruszać własność kopca typu max, dlatego w wierszu 5 zostaje wywołana dla niego procedura MAX-HEAPIFY przywracająca tę własność. Uaktualnienie i powoduje odtworzenie niezmiennika.

Zakończenie: Po zakończeniu działania pętli jest $i = 1$, zatem podtablica $A[1 \,..\, i]$ składa się z jednego elementu, który jest najmniejszym elementem tablicy A. Ponadto $n-1$ pozostałych elementów jest uszeregowanych w kolejności niemalejącej w podtablicy $A[2 \,..\, n]$. Stąd mamy, że tablica A jest posortowana.



Rysunek 10: Działanie procedury HEAPSORT dla tablicy $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$. **(a)** Kopiec zaraz po jego zbudowaniu przez BUILD-MAX-HEAP. **(b)–(i)** Kopiec i elementy z niego usunięte po każdym wywołaniu MAX-HEAPIFY w wierszu 5. **(j)** Wynikowa posortowana tablica.

6.4-3. Na podstawie analizy zamieszczonej w Podręczniku czasem działania algorytmu heapsort dla tablicy o rozmiarze n jest $O(n \lg n)$. Z zad. 6.4-5 mamy, że jest to w rzeczywistości oszacowanie dokładne. A zatem w szczególności dla tablicy posortowanej rosnąco i tablicy posortowanej malejąco heapsort działa w czasie $\Theta(n \lg n)$.

6.4-4. Przypadek pesymistyczny algorytmu heapsort ma miejsce wtedy, gdy każde wywołanie MAX-HEAPIFY z wiersza 5 schodzi rekurencyjnie aż do ostatniego poziomu drzewa. Na mocy wyniku z zad. 6.2-6 oraz wzoru (3.18) wywołania te zabierają łącznie czas

$$\sum_{i=2}^{n} \Omega(\lg i) = \Omega(\lg(n!)) = \Omega(n \lg n)$$

i taki jest też czas działania algorytmu heapsort w pesymistycznym przypadku.

6.4-5. Dokonamy analizy liczby wykonywanych instrukcji z linii 9 procedury MAX-HEAPIFY podczas działania algorytmu sortowania przez kopcowanie w przypadku optymistycznym.

Załóżmy, że algorytm heapsort działa na tablicy A o rozmiarze $n = 2^{h+1} - 1$, gdzie h jest dodatnią liczbą całkowitą. A zatem kopiec zbudowany z A stanowi pełne drzewo binarne o wysokości h. Rozważanie tylko takich kopców nie powoduje zmniejszenia ogólności analizy. Przez j-ty etap działania algorytmu heapsort, gdzie $j = 0, 1, \ldots, h-1$, będziemy rozumieć działania wykonywane podczas iteracji pętli **for** z procedury HEAPSORT, w których $2^{h-j} \leqslant i \leqslant 2^{h-j+1} - 1$. Inaczej mówiąc, j-ty etap pozbawia kopiec $(h-j)$-tego poziomu.

Lemat. *Podczas j-tego etapu działania algorytmu heapsort na kopcu A o rozmiarze $n = 2^{h+1}-1$, $h \geqslant 5$, którego wszystkie elementy są różne, liczba wykonanych zamian elementów w linii 9 procedury* MAX-HEAPIFY *jest większa niż $(h-j-5)2^{h-j-3}$.*

Dowód. Oznaczmy przez m_j badaną liczbę zamian elementów wykonywanych w j-tym etapie. Niech $j = 0$ i bez utraty ogólności załóżmy, że $\langle A[1], A[2], \ldots, A[n]\rangle$ jest permutacją $\langle 1, 2, \ldots, n\rangle$. Liczbę k będziemy nazywać **dużą**, jeśli $k \geqslant (n+1)/2$. Niech S będzie zbiorem indeksów dużych elementów w kopcu A, które nie są liśćmi, czyli

$$S = \left\{ i \in \left\{1, 2, \ldots, \frac{n-1}{2}\right\} : A[i] \geqslant \frac{n+1}{2} \right\}.$$

Zauważmy, że wszystkie elementy, których pozycjami w A są indeksy ze zbioru S, zostaną usunięte z kopca w etapie $j = 0$. A zatem muszą wpierw znaleźć się w korzeniu kopca za sprawą wykonania pewnej liczby zamian z linii 9 procedury MAX-HEAPIFY. Stąd m_0 spełnia nierówność

$$m_0 \geqslant \sum_{i \in S} d_i,$$

gdzie d_i oznacza głębokość węzła o początkowej pozycji i w kopcu A.

Węzły o indeksach ze zbioru S tworzą w kopcu A poddrzewo T o korzeniu w $A[1]$. Jest tak dlatego, że jeśli węzeł $A[i]$ jest duży, to $A[\text{PARENT}(i)]$ również jest duży, a więc także wszystkie węzły na ścieżce od $A[i]$ do korzenia kopca, czyli $A[1]$. Jeśli zastąpimy każde puste poddrzewo w T pojedynczym węzłem, to dostaniemy regularne drzewo binarne, którego długość ścieżki wewnętrznej (patrz zad. B.5-5) wynosi m_0. W zbiorze wszystkich drzew binarnych o $|S|$ węzłach wewnętrznych najmniejsza możliwa długość ścieżki wewnętrznej jest osiągana dla pełnego drzewa binarnego (przy czym ostatni poziom tego drzewa może nie być wypełniony) i wynosi $\sum_{k=1}^{|S|} \lfloor \lg k \rfloor$. Korzystając ze wzoru (3.3) i zad. 8.1-2, mamy

$$m_0 \geqslant \sum_{k=1}^{|S|} \lfloor \lg k \rfloor > \sum_{k=1}^{|S|} (\lg k - 1) = \sum_{k=1}^{|S|} \lg k - |S| \geqslant \frac{|S|}{2} \lg \frac{|S|}{2} - |S| = \frac{|S|}{2} \lg |S| - \frac{3}{2}|S|.$$

Pokażemy teraz, że $|S| \geqslant 2^{h-2}$. Rozważmy w tym celu permutację π elementów kopca A na początku zerowego etapu w kolejności ich odwiedzania podczas przechodzenia kopca metodą inorder. Jeśli $\pi(i)$, gdzie $i \geqslant 2$, jest liściem kopca, to $\pi(i-1)$ nie może być liściem kopca. Jeśli w dodatku $\pi(i)$ jest dużym liściem, to $\pi(i-1)$ jest dużym węzłem wewnętrznym. Stąd indeks elementu $\pi(i-1)$ należy do S. Mamy więc, że l – liczba dużych liści – nie przekracza $|S|+1$, nawet jeśli $\pi(1)$ jest dużym liściem. Ponieważ liczba dużych elementów w kopcu wynosi $(n+1)/2 = 2^h$, to otrzymujemy, że $|S| = 2^h - l \geqslant 2^h - (|S| + 1)$, skąd $|S| \geqslant 2^{h-1} - 1/2 \geqslant 2^{h-2}$.

Powracając teraz do oszacowania na m_0, mamy

$$m_0 > \frac{|S|}{2} \lg |S| - \frac{3}{2}|S| = \frac{|S|}{2}(\lg |S| - 3) \geqslant 2^{h-3}(h-5),$$

czyli lemat jest prawdziwy, gdy $j = 0$.

Na początku j-tego etapu kopiec ma wysokość $h-j$, więc dowód lematu dla j-tego etapu, gdzie $1 \leqslant j \leqslant h-1$, sprowadza się do dowodu oszacowania m_0 dla kopca o rozmiarze $n = 2^{h-j+1}-1$. □

Załóżmy teraz, że $h \geqslant 5$ i wykorzystajmy oznaczenie m_j z powyższego dowodu. Sumaryczną liczbę zamian elementów podczas sortowania n liczb, na podstawie lematu, ograniczamy od dołu:

$$\sum_{j=0}^{h-1} m_j > \sum_{j=0}^{h-5} m_j > \sum_{j=0}^{h-5} (h-j-5)2^{h-j-3} = \sum_{j=0}^{h-5} j2^{j+2} = 4\sum_{j=0}^{h-5} j2^j.$$

Ostatnią sumę obliczamy poprzez skorzystanie ze wzoru (A.5):

$$\sum_{j=0}^{h-5} jx^j = x \cdot \frac{d}{dx}\left(\sum_{j=0}^{h-5} x^j\right) = x \cdot \frac{d}{dx}\left(\frac{x^{h-4}-1}{x-1}\right) = x\,\frac{(h-4)x^{h-5}(x-1)-(x^{h-4}-1)}{(x-1)^2}.$$

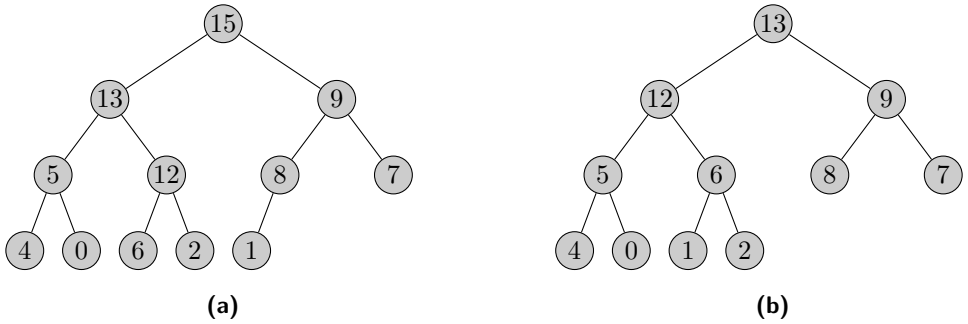
Przyjmując teraz $x = 2$ i korzystając z nierówności $2^h > n/2$ i $h > \lg n - 1$, mamy ostatecznie

$$4\sum_{j=0}^{h-5} j2^j = (h-4)2^{h-2} - 2^{h-1} + 8 > (h-6)2^{h-2} > \frac{1}{8}n\lg n - \frac{7}{8}n = \Omega(n\lg n).$$

Otrzymany wynik stanowi oszacowanie czasu działania algorytmu heapsort w przypadku optymistycznym.

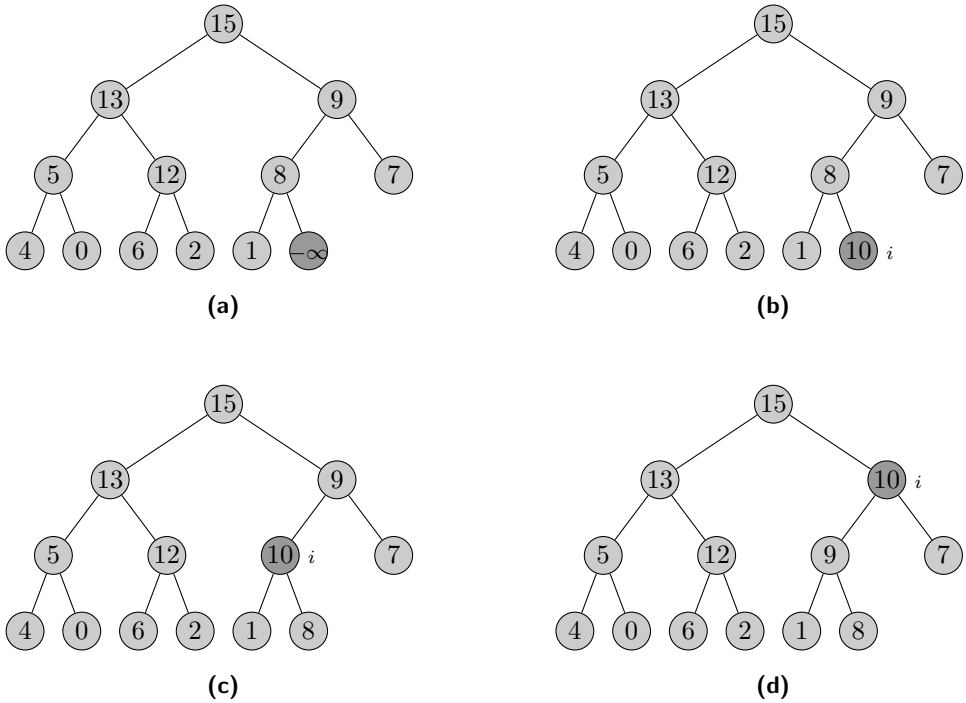
6.5. Kolejki priorytetowe

6.5-1. Na rys. 11 został przedstawiony kopiec wejściowy A i wynikowy kopiec otrzymany w wyniku działania procedury Heap-Extract-Max. W wierszu 3 zmiennej *max* przypisywana jest maksymalna wartość kopca, czyli 15. Następnie korzeń otrzymuje wartość 1 i rozmiar kopca jest pomniejszany o 1. Po przywróceniu własności kopca w linii 6 procedura zwraca wartość *max*.



Rysunek 11: Działanie procedury Heap-Extract-Max dla kopca $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\rangle$. **(a)** Kopiec wejściowy A. **(b)** Kopiec A po usunięciu maksymalnej wartości i przywróceniu własności kopca naruszonej przez korzeń, któremu wcześniej przypisano wartość 1.

6.5-2. Procedura Max-Heap-Insert rozpoczyna działanie od dodania do kopca nowego elementu o wartości $-\infty$. Wartość ta jest następnie odpowiednio modyfikowana i element jest umieszczany w odpowiednim miejscu w kopcu dzięki wywołaniu Heap-Increase-Key. Działanie procedury Max-Heap-Insert$(A, 10)$ zostało przedstawione na rys. 12.



Rysunek 12: Działanie procedury MAX-HEAP-INSERT$(A, 10)$ dla kopca $A = \langle 15,13,9,5,12,8,7,4,0,6,2,1\rangle$. **(a)** Kopiec po dodaniu nowego elementu o wartości początkowej $-\infty$. **(b)** Działa teraz procedura HEAP-INCREASE-KEY. Na rysunku pokazano wartość zmiennej i w tej procedurze. Wartość nowego elementu została zwiększona i wynosi teraz 10. **(c)** Po wykonaniu pierwszej iteracji pętli **while** procedury HEAP-INCREASE-KEY nowy element został zamieniony ze swoim ojcem. **(d)** Po drugiej iteracji pętli nastąpiła jeszcze jedna zamiana nowego elementu i jego aktualnego ojca, dzięki czemu A spełnia już własność kopca i procedura kończy działanie.

6.5-3. Zakładamy, że tablica A stanowi kopiec typu min. Poniższe procedury stanowią implementację kolejki priorytetowej typu min i działają analogicznie do odpowiadających im procedur dla kolejki priorytetowej typu max.

HEAP-MINIMUM(A)
1 **return** $A[1]$

HEAP-EXTRACT-MIN(A)
1 **if** *heap-size*$[A] < 1$
2 **then error** „kopiec pusty”
3 $min \leftarrow A[1]$
4 $A[1] \leftarrow A[$*heap-size*$[A]]$
5 *heap-size*$[A] \leftarrow$ *heap-size*$[A] - 1$
6 MIN-HEAPIFY$(A, 1)$
7 **return** min

```
HEAP-DECREASE-KEY(A, i, key)
1  if key > A[i]
2     then error „nowy klucz jest większy niż klucz aktualny"
3  A[i] ← key
4  while i > 1 i A[PARENT(i)] > A[i]
5        do zamień A[i] ↔ A[PARENT(i)]
6           i ← PARENT(i)
```

```
MIN-HEAP-INSERT(A, key)
1  heap-size[A] ← heap-size[A] + 1
2  A[heap-size[A]] ← ∞
3  HEAP-DECREASE-KEY(A, heap-size[A], key)
```

6.5-4. Po wykonaniu wiersza 1 procedury MAX-HEAP-INSERT wartość $A[\mathit{heap\text{-}size}[A]]$ pozostaje niezdefiniowana i może zawierać liczbę większą niż *key*. Wówczas jednak wywołanie HEAP-INCREASE-KEY zakończy się z błędem. Radzimy sobie z tym problemem poprzez nadanie elementowi wartości $-\infty$.

6.5-5. Dowodzimy, że niezmiennik spełniony jest przed każdą iteracją pętli **while** oraz że po zakończeniu tej pętli z niezmiennika wynika, że procedura jest poprawna.

Inicjowanie: Przed wykonaniem wiersza 3 tablica $A[1\,..\,\mathit{heap\text{-}size}[A]]$ jest kopcem typu max. Zwiększenie wartości $A[i]$ może naruszyć własność kopca tylko dla elementów $A[i]$ oraz $A[\textsc{Parent}(i)]$.

Utrzymanie: Dokonując zamiany elementów w wierszu 5 w bieżącej iteracji pętli, przywracamy własność kopca dla elementów $A[i]$ oraz $A[\textsc{Parent}(i)]$. Jednak operacja ta może wygenerować nową parę elementów niespełniających własności kopca: $A[\textsc{Parent}(i)]$ oraz $A[\textsc{Parent}(\textsc{Parent}(i))]$. Aktualizacja wartości i powoduje zachowanie niezmiennika, albowiem nowa para elementów jest jedyną, która może naruszać własność kopca.

Zakończenie: Pętla kończy działanie, gdy $i \leqslant 1$ lub $A[\textsc{Parent}(i)] \geqslant A[i]$. W pierwszym przypadku $\textsc{Parent}(i) \leqslant 0$, co jest niepoprawną wartością dla indeksów tablicy A. W drugim natomiast jedyna para, która mogłaby naruszać własność kopca, w rzeczywistości ją spełnia. A zatem po zakończeniu wykonywania pętli tablica $A[1\,..\,\mathit{heap\text{-}size}[A]]$ stanowi kopiec typu max.

Z prawdziwości niezmiennika pętli wynika, że procedura HEAP-INCREASE-KEY poprawnie zwiększa wartość węzła i, pozostawiając kopiec typu max.

6.5-6. Kolejkę FIFO implementujemy, wykorzystując do tego celu kolejkę priorytetową typu min. Przy inicjalizacji kolejki będziemy ustawiać wartość dodatkowej zmiennej *rank* na 1. Przed dodaniem nowego elementu do kolejki FIFO nadamy mu rangę, czyli powiążemy go z aktualną wartością zmiennej *rank*, po czym wstawimy element wraz z jego rangą do kolejki priorytetowej procedurą MIN-HEAP-INSERT. Warunek kolejki priorytetowej spełniany będzie tylko na podstawie wartości rang elementów. Po umieszczeniu obiektu w kolejce wartość zmiennej *rank* zostanie zwiększona o 1. Z kolei usuwanie elementów będzie odbywać się poprzez zwykłe wywołanie procedury HEAP-EXTRACT-MIN. Taka implementacja operacji na kolejce FIFO zapewnia, że w danym momencie w strukturze danych nie będzie dwóch różnych elementów z tą samą rangą i elementy pobierane będą w odpowiedniej kolejności.

Realizacja stosu jest podobna, ale używamy do tego celu kolejki priorytetowej typu max, w której porównań dokonujemy na rangach związanych z elementami. Podczas wstawiania elementów na stos korzystamy z procedury MAX-HEAP-INSERT i inkrementujemy zmienną *rank* (zainicjalizowaną na 1 w momencie utworzenia stosu). Usuwanie polega na odnalezieniu i pobraniu elementu z największą rangą, co realizowane jest za pomocą HEAP-EXTRACT-MAX. W wyniku tego elementy pobierane są w kolejności odwrotnej do tej, w której były wstawiane.

6.5-7. *Zmienimy nazwę operacji z sugerowanej w Podręczniku* HEAP-DELETE *na* MAX-HEAP-DELETE*, aby odróżnić ją od analogicznej procedury dla kopca typu min.*

Przedstawiona poniżej procedura MAX-HEAP-DELETE zamienia element $A[i]$ w n-elementowym kopcu A typu max z jego liściem $A[\textit{heap-size}[A]]$, po czym dekrementuje $\textit{heap-size}[A]$. Po tym kroku własność kopca może być naruszona przez węzeł $A[i]$ na dwa sposoby. W pierwszym przypadku mamy sytuację, w której $A[i] < A[\text{LEFT}(i)]$ lub $A[i] < A[\text{RIGHT}(i)]$ – przywracamy więc własność kopca za pomocą wywołania MAX-HEAPIFY. W drugim przypadku $A[i] > A[\text{PARENT}(i)]$, więc w celu odbudowy struktury kopca wystarczy wykonać podobne operacje, jak w procedurze HEAP-INCREASE-KEY. Ostatni krok procedury to zwrócenie elementu, który początkowo zajmował w kopcu pozycję i.

```
MAX-HEAP-DELETE(A, i)
1  zamień A[i] ↔ A[heap-size[A]]
2  heap-size[A] ← heap-size[A] − 1
3  MAX-HEAPIFY(A, i)
4  while i > 1 i A[PARENT(i)] < A[i]
5        do zamień A[i] ↔ A[PARENT(i)]
6           i ← PARENT(i)
7  return A[heap-size[A] + 1]
```

Zarówno wywołanie z wiersza 3, jak i pętla **while** w wierszach 4–6 zajmuje czas $O(\lg n)$, a więc czasem działania procedury MAX-HEAP-DELETE jest również $O(\lg n)$.

6.5-8. W algorytmie wykorzystamy kolejkę priorytetową typu min jako strukturę pomocniczą. Na początku do kolejki zostaną przeniesione elementy z głowy każdej listy. Jest oczywiste, że wśród tych elementów znajduje się najmniejszy element listy wynikowej. Aby go uzyskać, wystarczy wywołać na kolejce operację EXTRACT-MIN. Kolejnego elementu należy szukać wśród aktualnych węzłów kolejki lub w głowie listy, do której początkowo należało usunięte przed chwilą minimum. Głowę tej listy, o ile istnieje, przenosimy do kolejki. W kolejnych iteracjach powtarzamy te operacje – pobieramy najmniejszy element kolejki i wstawiamy na listę wynikową, po czym uzupełniamy kolejkę głową listy, do której należał pobrany element, o ile lista ta nie jest jeszcze pusta. Proces ten powtarzamy aż do opróżnienia kolejki, co następuje po przetworzeniu zawartości wszystkich list. Aby zachować kolejność niemalejącą na liście wynikowej, musimy wstawiać elementy na jej koniec, pamiętając wskaźnik do ogona tej listy.

Podczas działania algorytmu wykonamy n razy operację wstawienia węzła do kolejki zawierającej co najwyżej k elementów i tyleż samo operacji EXTRACT-MIN. Otrzymujemy zatem górne oszacowanie $O(n \lg k)$ na czas działania algorytmu, przy założeniu, że kolejka priorytetowa została zaimplementowana w oparciu o kopiec typu min.

Problemy

6-1. Budowa kopca przez wstawianie

(a) Procedury te nie zawsze generują identyczne kopce dla tej samej tablicy wejściowej. Jeśli na przykład rozważymy tablicę $A = \langle 1, 2, 3\rangle$, to kopce budowane przez obie procedury różnią się, jak to widać na rys. 13.

Rysunek 13: Porównanie kopców budowanych przez obie procedury dla tablicy $A = \langle 1, 2, 3\rangle$. **(a)** Wynik działania Build-Max-Heap. **(b)** Wynik działania Build-Max-Heap′.

(b) Najgorszym przypadkiem dla procedury Build-Max-Heap′ jest tablica uporządkowana rosnąco. W każdym z $n-1$ wywołań Max-Heap-Insert z wiersza 3 nowy węzeł transportowany jest wówczas aż do korzenia kopca, co wymaga $\Theta(\lg i)$ operacji przy i-elementowym kopcu. Stąd czas działania Build-Max-Heap′ w przypadku pesymistycznym wynosi

$$\sum_{i=1}^{n-1} \Theta(\lg i) = \Theta(\lg(n!)) = \Theta(n \lg n).$$

6-2. Analiza kopców rzędu d

(a) d-kopiec będziemy reprezentować w tablicy w następujący sposób. Podobnie jak w reprezentacji tablicowej kopców binarnych tablica A reprezentująca d-kopiec będzie mieć atrybuty *length*$[A]$ oraz *heap-size*$[A]$. Na pierwszej pozycji tablicy znajdzie się korzeń kopca, a pozycje $2\,..\,d+1$ będą zajmowane przez d synów korzenia. Synowie pierwszego z lewej syna korzenia zajmą pozycje $d+2\,..\,2d+1$, synowie drugiego od lewej syna korzenia – pozycje $2d+2\,..\,3d+1$ itd. Ogólnie, mając dany indeks węzła i, można wyznaczyć indeks jego ojca, korzystając ze wzoru $\lceil (i-1)/d \rceil$. Uogólnienie procedury Parent dla kopca rzędu d wygląda zatem następująco:

Multiary-Parent(d, i)
  return $\lceil (i-1)/d \rceil$

Łatwo pokazać, że indeks k-tego od lewej syna węzła o indeksie i, gdzie $k = 1, 2, \ldots, d$, jest opisany wzorem $d(i-1)+k+1$. Poniższa procedura stanowi uogólnienie procedur Left i Right dla kopca rzędu d – w porównaniu do nich przyjmuje dodatkowy parametr k oznaczający numer szukanego syna węzła i.

Multiary-Child(d, k, i)
  return $d(i-1)+k+1$

Można sprawdzić, że zachodzi Multiary-Parent$(d,$ Multiary-Child$(d, k, i)) = i$ dla każdego $k = 1, 2, \ldots, d$.

(b) Uogólnimy rozumowanie z zad. 6.1-1 na kopce rzędu d. Potraktujmy taki kopiec jak drzewo d-arne o wysokości h i n węzłach. Na i-tym poziomie tego drzewa, gdzie $i = 0, 1, \ldots, h-1$, znajduje się d^i węzłów. Najniższy, h-ty poziom, może zawierać od 1 do d^h węzłów. Mamy zatem

$$\sum_{i=0}^{h-1} d^i + 1 \leqslant n \leqslant \sum_{i=0}^{h} d^i.$$

Na podstawie punktu (c) problemu 3-1 sumę po lewej stronie można oszacować przez $\Theta(d^{h-1})$, a sumę po prawej – przez $\Theta(d^h)$. Oba te oszacowania dają w wyniku $h = \Theta(\log_d n)$.

(c) Przedstawimy najpierw implementację procedury MAX-HEAPIFY dla kopców rzędu d. Ogólny zarys jej działania pozostaje niezmieniony w porównaniu z oryginalną procedurą MAX-HEAPIFY. Na każdym poziomie rekursji musimy wyznaczyć maksimum z $d+1$ wartości – bieżącego węzła i jego d synów. W tym celu stosujemy pętlę przeglądającą wszystkich synów bieżącego węzła.

```
MULTIARY-MAX-HEAPIFY(A, d, i)
 1  largest ← i
 2  k ← 1
 3  child ← MULTIARY-CHILD(d, 1, i)
 4  while k ⩽ d i child ⩽ heap-size[A]
 5        do if A[child] > A[largest]
 6              then largest ← child
 7           k ← k + 1
 8           child ← MULTIARY-CHILD(d, k, i)
 9  if largest ≠ i
10     then zamień A[i] ↔ A[largest]
11          MULTIARY-MAX-HEAPIFY(A, d, largest)
```

Zauważmy, że podczas wykonywania pętli **while** w wierszach 4–8 zmienna k może przyjąć wartość $d+1$ i wówczas w wierszu 8 zostaje wyznaczony indeks nieistniejącego, $(d+1)$-szego syna węzła i. Jednak wartości tej nigdzie później nie wykorzystujemy, ponieważ następną operacją jest przerwanie pętli **while**.

Na każdym poziomie rekursji (z wyjątkiem być może ostatniego) wykonywanych jest $\Theta(d)$ operacji. Na mocy poprzedniego punktu mamy $\Theta(\log_d n)$ wywołań rekurencyjnych, a zatem czasem działania powyższej procedury jest $\Theta(d \log_d n)$.

Procedura MULTIARY-HEAP-EXTRACT-MAX, która implementuje operację EXTRACT-MAX dla d-kopca, przyjmuje jako parametry kopiec A oraz jego rząd d. Działa ona identyczne jak operacja EXTRACT-MAX dla kopca binarnego, jednak w wierszu 6 zamiast procedury MAX-HEAPIFY wywołuje procedurę MULTIARY-MAX-HEAPIFY przedstawioną powyżej. Czas działania tej operacji wynosi $\Theta(d \log_d n)$.

(d) Procedura MULTIARY-MAX-HEAP-INSERT implementująca operację INSERT dla d-kopca przyjmuje na wejściu kopiec A, rząd kopca d oraz wartość *key*, która będzie wstawiana do A. Jej działanie jest analogiczne do działania procedury wstawiania węzła do kopca binarnego. Jedyną różnicą jest wiersz 3, który zamiast HEAP-INCREASE-KEY zawiera analogiczne wywołanie procedury MULTIARY-HEAP-INCREASE-KEY zwiększającej wartość węzła w kopcu rzędu d.

MULTIARY-MAX-HEAP-INSERT(A, d, key)
1 $heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
2 $A[heap\text{-}size[A]] \leftarrow -\infty$
3 MULTIARY-HEAP-INCREASE-KEY$(A, d, heap\text{-}size[A], key)$

Czas działania operacji INSERT dla d-kopca jest tego samego rzędu co czas działania wywołania z wiersza 3. Implementacja wywoływanej procedury MULTIARY-HEAP-INCREASE-KEY i analiza jej czasu działania zostały opisane w następnym punkcie.

(e) Implementacja tej operacji dla d-kopca jest analogiczna do jej implementacji dla kopca binarnego. Jednak zamiast sprawdzania poprawności parametru k, do $A[i]$ przypisujemy natychmiast odpowiednią wartość.

MULTIARY-HEAP-INCREASE-KEY(A, d, i, k)
1 $A[i] \leftarrow \max(A[i], k)$
2 **while** $i > 1$ i $A[\text{MULTIARY-PARENT}(d, i)] < A[i]$
3 **do** zamień $A[i] \leftrightarrow A[\text{MULTIARY-PARENT}(d, i)]$
4 $i \leftarrow \text{MULTIARY-PARENT}(d, i)$

Po wykonaniu wiersza 1 wartość węzła i może być większa niż wartość jego ojca. W najgorszym przypadku, jeśli węzeł i jest liściem i jego nowa wartość jest największą wartością w kopcu, to zostanie on przetransportowany aż do korzenia, co zajmie czas proporcjonalny do wysokości kopca, czyli, na mocy punktu (b), $\Theta(\log_d n)$.

6-3. Tablice Younga

(a) Jedną z tablic Younga zawierających podane elementy jest

$$\begin{pmatrix} 2 & 3 & 14 & 16 \\ 4 & 8 & \infty & \infty \\ 5 & 12 & \infty & \infty \\ 9 & \infty & \infty & \infty \end{pmatrix}.$$

(b) Załóżmy, że $Y[1,1] = \infty$ i że tablica Y nie jest pusta, tzn. $Y[i,j] \neq \infty$ dla pewnych i, j takich, że $1 \leqslant i \leqslant m$ oraz $1 \leqslant j \leqslant n$. Ale z własności tablicy Younga otrzymujemy, że $Y[1,1] \leqslant Y[1,j] \leqslant Y[i,j]$, co prowadzi do sprzeczności z założeniem. A więc tablica Younga Y, w której $Y[1,1] = \infty$, jest pusta.

Dowód drugiej własności przebiega analogicznie. Przypuśćmy, że $Y[m,n] \neq \infty$ i że tablica Y nie jest pełna, tzn. $Y[i,j] = \infty$ dla pewnych i, j, gdzie $1 \leqslant i \leqslant m$ oraz $1 \leqslant j \leqslant n$. Wykorzystując własność tablicy Younga, dostajemy $Y[i,j] \leqslant Y[i,n] \leqslant Y[m,n]$, co jest sprzeczne z założeniem. Tablica Younga Y, w której $Y[m,n] \neq \infty$, jest pełna.

(c) Procedura ekstrakcji najmniejszego elementu tablicy Younga Y o rozmiarach $m \times n$ będzie opierać się o pomysł z MAX-HEAPIFY. Najmniejszym elementem tablicy Y jest $\mu = Y[1,1]$. Przetransportujemy go na ostatnią pozycję ostatniego wiersza tablicy, skąd będzie można bezpiecznie go usunąć przy jednoczesnym zachowaniu własności tablicy Younga. W tym celu porównajmy μ z elementem znajdującym się bezpośrednio na prawo i elementem bezpośrednio w dół od niego (o ile istnieją). Mniejszy z nich zamieniany jest następnie z μ, po czym procedura wywołuje się rekurencyjnie dla podtablicy Younga o rozmiarach $(m-1) \times n$ albo $m \times (n-1)$, w której μ stanowi pierwszy element pierwszej kolumny. Otrzymując w wyniku tego postępowania

tablicę o rozmiarach 1×1, można usunąć jej jedyny element będący najmniejszym elementem początkowej tablicy Younga (zastępując go wartością ∞) i zwrócić go jako wynik algorytmu.

Opisany sposób został zaimplementowany w poniższym pseudokodzie. Aby pobrać minimum z tablicy Younga Y o rozmiarach $m \times n$, należy wywołać YOUNG-EXTRACT-MIN$(Y, m, n, 1, 1)$.

```
YOUNG-EXTRACT-MIN(Y, m, n, i, j)
 1  if ⟨i, j⟩ = ⟨m, n⟩
 2     then min ← Y[i, j]
 3          Y[i, j] ← ∞
 4          return min
 5  ⟨i′, j′⟩ ← ⟨i, j + 1⟩
 6  if i < m
 7     then if j = n lub Y[i + 1, j] < Y[i, j + 1]
 8             then ⟨i′, j′⟩ ← ⟨i + 1, j⟩
 9  zamień Y[i, j] ↔ Y[i′, j′]
10  return YOUNG-EXTRACT-MIN(Y, m, n, i′, j′)
```

Niech $T(p)$ będzie maksymalnym czasem działania powyższego algorytmu dla tablicy Younga $m \times n$, gdzie $p = m + n$ jest łączną liczbą jej kolumn i wierszy. W każdym wywołaniu rekurencyjnym zmniejszamy p o 1, wykonując przy tym czas stały, skąd dostajemy

$$T(p) = \begin{cases} \Theta(1), & \text{jeśli } p = 2, \\ T(p-1) + \Theta(1), & \text{jeśli } p > 2. \end{cases}$$

Łatwo sprawdzić, że rozwiązaniem tej rekurencji jest $T(p) = O(p) = O(m + n)$.

(d) Podamy najpierw pomocniczą procedurę YOUNGIFY, która działa analogicznie do MAX-HEAPIFY i ma na celu przywrócenie własności tablicy Younga Y naruszoną przez $Y[i, j]$. Element ten wystarczy porównać z jego sąsiadem znajdującym się powyżej lub sąsiadem znajdującym się po lewej stronie w tablicy (o ile istnieją). W zależności od tego, który z tych trzech elementów jest największy, dokonywana jest odpowiednia zamiana i procedura wywoływana jest rekurencyjnie.

```
YOUNGIFY(Y, i, j)
1  ⟨i′, j′⟩ ← ⟨i, j⟩
2  if i > 1 i Y[i − 1, j] > Y[i′, j′]
3     then ⟨i′, j′⟩ ← ⟨i − 1, j⟩
4  if j > 1 i Y[i, j − 1] > Y[i′, j′]
5     then ⟨i′, j′⟩ ← ⟨i, j − 1⟩
6  if ⟨i′, j′⟩ ≠ ⟨i, j⟩
7     then zamień Y[i, j] ↔ Y[i′, j′]
8          YOUNGIFY(Y, i′, j′)
```

Ponieważ zakładamy, że tablica Younga Y nie jest pełna, to na mocy punktu (b) mamy $Y[m, n] = \infty$, czyli pozycja ta jest pusta i można wstawić na nią nowy element. Wówczas jednak własność tablicy Younga może być naruszona, dlatego korzystamy z procedury YOUNGIFY w celu przywrócenia tej własności.

```
YOUNG-INSERT(Y, m, n, key)
1  Y[m, n] ← key
2  YOUNGIFY(Y, m, n)
```

Analiza poprawności i czasu działania procedury YOUNGIFY opiera się na analizie procedury MAX-HEAPIFY. W każdym kolejnym wywołaniu rekurencyjnym jedna z liczb, i lub j, jest mniejsza o 1. Koniec działania następuje w najgorszym przypadku, gdy $i = j = 1$, po wykonaniu $O(m + n)$ operacji. A zatem czasem działania operacji YOUNG-INSERT jest również $O(m + n)$.

(e) Niech A będzie tablicą n^2 liczb, które należy posortować. Poniższy algorytm buduje tablicę Younga $n \times n$ z liczb tablicy A, wykonując na każdej z nich operację YOUNG-INSERT. Następnie liczby te są pobierane w kolejności niemalejącej dzięki n^2 wywołaniom YOUNG-EXTRACT-MIN.

YOUNG-SORT(A)
1 $n \leftarrow \sqrt{length[A]}$
2 **for** $i \leftarrow 1$ **to** n^2
3 **do** YOUNG-INSERT($Y, n, n, A[i]$)
4 **for** $i \leftarrow 1$ **to** n^2
5 **do** $A[i] \leftarrow$ YOUNG-EXTRACT-MIN($Y, n, n, 1, 1$)

Czas działania obu wywoływanych procedur wynosi $O(n)$, zatem powyższy algorytm działa w czasie $O(n^3)$. Jeśli mamy danych $m = n^2$ liczb, to jesteśmy w stanie posortować je przy użyciu tego algorytmu w czasie $O(m^{3/2})$. Jest to lepsza złożoność niż kwadratowa, ale gorsza od złożoności liniowo-logarytmicznej.

(f) Zbadajmy, jak szukana liczba v ma się do ostatniego elementu pierwszego wiersza tablicy Younga $m \times n$. Jeśli wartości te są równe, to oczywiście można zakończyć poszukiwania z rezultatem pozytywnym. W przeciwnym przypadku, w zależności od tego, która z liczb jest większa, odrzucamy z dalszych poszukiwań cały pierwszy wiersz lub całą ostatnią kolumnę i kontynuujemy szukanie v w otrzymanej podtablicy, która stanowi tablicę Younga $(m-1) \times n$ albo $m \times (n-1)$. W momencie uzyskania tablicy pustej wiadomo, że szukanej liczby nie ma w początkowej tablicy.

YOUNG-SEARCH(Y, m, n, v)
1 $i \leftarrow 1$
2 $j \leftarrow n$
3 **while** $i \leqslant m$ i $j \geqslant 1$
4 **do if** $v = Y[i, j]$
5 **then return** TRUE
6 **if** $v > Y[i, j]$
7 **then** $i \leftarrow i + 1$
8 **else** $j \leftarrow j - 1$
9 **return** FALSE

W każdym kroku pętli **while** zmniejszamy o 1 liczbę kolumn lub liczbę wierszy rozważanej tablicy, wykonując przy tym stałą liczbę operacji – jasne jest zatem, że czas działania algorytmu wynosi $O(m + n)$.

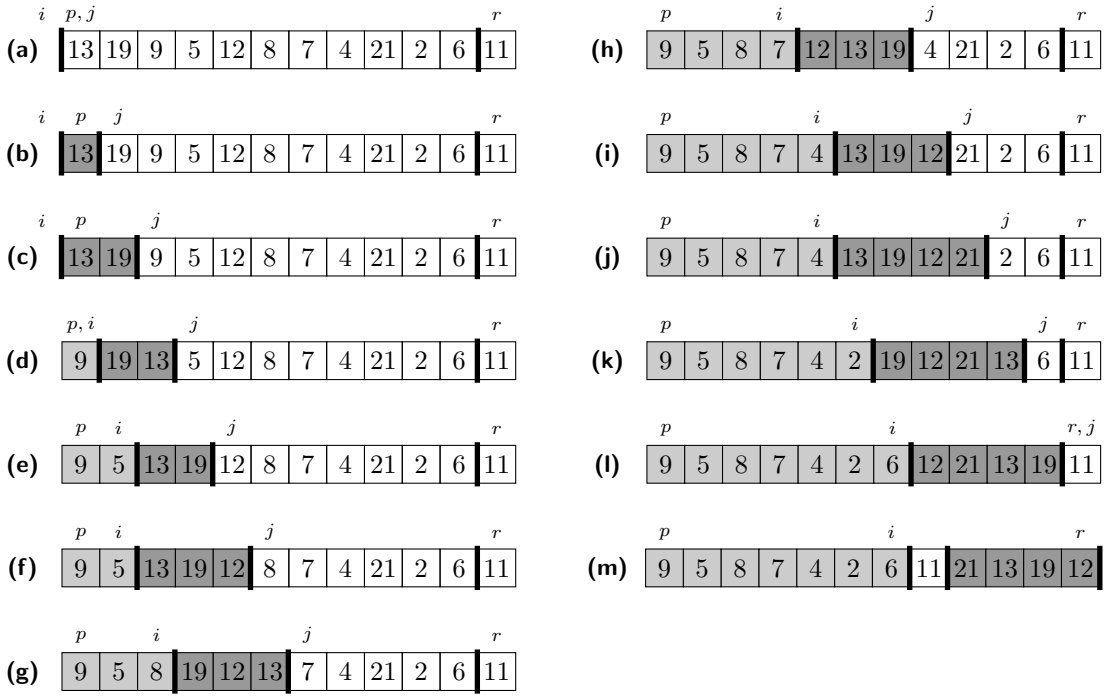
Rozdział **7**

Quicksort – sortowanie szybkie

7.1. Opis algorytmu

7.1-1. *Rozwiązanie dotyczy przykładu z tekstu oryginalnego.*

Rys. 14 przedstawia działanie procedury PARTITION dla tablicy A.



Rysunek 14: Działanie procedury PARTITION dla tablicy $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11\rangle$. **(a)** Tablica wejściowa z zaznaczonymi początkowymi wartościami zmiennych. **(b)–(l)** Kolejne iteracje pętli **for** w wierszach 3–6. **(m)** Wynikowa tablica A po wykonaniu zamiany z wiersza 7.

7.1-2. *Poprawną wartością dla q z treści zadania powinno być $\lfloor (p+r)/2 \rfloor$.*

Zauważmy, że jeśli wszystkie elementy podtablicy $A[p\,..\,r]$ mają taką samą wartość, to warunek z wiersza 4 procedury PARTITION jest spełniony w każdej iteracji pętli **for**. Oznacza to, że po wykonaniu tej pętli będzie $i = r - 1$ i procedura zwróci $q = r$.

Odpowiedniej modyfikacji procedury dokonujemy poprzez wprowadzenie licznika elementów równych elementowi rozdzielającemu w badanej podtablicy. W każdej iteracji pętli **for** sprawdzamy dodatkowo, czy $A[j] = x$ i jeśli tak, to licznik ten inkrementujemy. Jeśli na końcu procedury jego wartość jest równa $r - p + 1$, czyli rozmiarowi podtablicy, to zwracamy $q = \lfloor (p + r)/2 \rfloor$.

7.1-3. Podczas przetwarzania podtablicy $A[p\,..\,r]$ o rozmiarze $n = r - p + 1$ wykonywanych jest $n - 1$ iteracji pętli **for**, a każda z nich przeprowadza operacje zajmujące czas stały. Stąd czas działania procedury PARTITION wynosi $\Theta(n)$.

7.1-4. W warunku z wiersza 4 procedury PARTITION wystarczy zmienić znak nierówności na przeciwny.

7.2. Czas działania algorytmu quicksort

7.2-1. Niech c_1, $d_1 > 0$ będą stałymi. Korzystając z założenia, że $T(n-1) \leqslant c_1(n-1)^2$, dostajemy

$$T(n) = T(n-1) + \Theta(n) \leqslant c_1(n-1)^2 + d_1 n = c_1 n^2 + (d_1 - 2c_1)n + c_1 \leqslant c_1 n^2.$$

Ostatnia nierówność jest spełniona dla każdego $n \geqslant 1$, jeśli np. $c_1 = d_1 = 1$.

Weźmy teraz inne stałe c_2, $d_2 > 0$. Dolnego oszacowania na $T(n)$ dowodzimy analogicznie, wychodząc z założenia, że $T(n-1) \geqslant c_2(n-1)^2$:

$$T(n) = T(n-1) + \Theta(n) \geqslant c_2(n-1)^2 + d_2 n = c_2 n^2 + (d_2 - 2c_2)n + c_2 \geqslant c_2 n^2.$$

Przyjmujemy $c_2 = 1/2$, $d_2 = 2$, dzięki czemu ostatnia nierówność zachodzi dla wszystkich $n \geqslant 1$.

Przypadek brzegowy $n = 1$ można przyjąć za podstawę obu powyższych indukcji dla podanych wartości stałych, co kończy dowód, że $T(n) = \Theta(n^2)$.

7.2-2. Procedura PARTITION w takim przypadku zwraca wartość $q = r$ (zad. 7.1-2), a więc w następnych wywołaniach rekurencyjnych procedury QUICKSORT będą przetwarzane podtablice rozmiarów 0 i $n-1$. Napotykając w każdym wywołaniu rekurencyjnym przypadek pesymistyczny, algorytm będzie działać w czasie opisanym przez rekurencję z zad. 7.2-1, której rozwiązaniem jest $\Theta(n^2)$.

7.2-3. W tym przypadku podczas każdego wywołania rekurencyjnego procedury PARTITION elementem rozdzielającym jest najmniejszy element przetwarzanej podtablicy $A[p\,..\,r]$. W każdym wywołaniu jedyne elementy, które zostaną zamienione, to element rozdzielający i $A[p]$, po czym zwrócona zostanie wartość $q = r$. Przypadek ten jest zatem analogiczny do rozważanego w poprzednim zadaniu, a więc czasem działania procedury QUICKSORT dla tablicy posortowanej malejąco jest $\Theta(n^2)$.

7.2-4. Prawie posortowany ciąg jest niemal najgorszym przypadkiem dla algorytmu quicksort. W wielu wywołaniach rekurencyjnych na element rozdzielający wybierany jest bowiem największy element przetwarzanej podtablicy. Wykonywane są wówczas niezrównoważone podziały, przez co czas działania procedury QUICKSORT zbliża się do kwadratowego.

Z kolei dla sortowania przez wstawianie ciąg taki jest przypadkiem zbliżonym do optymistycznego, ponieważ czas działania procedury INSERTION-SORT jest tego samego rzędu, co ilość inwersji w ciągu wejściowym (punkt (c) problemu 2-4). W ciągu prawie posortowanym ilość inwersji jest niewielka, możemy zatem mówić o co najwyżej liniowym czasie działania sortowania przez wstawianie dla tego przypadku.

7.2-5. Rozważmy drzewo rekursji dla opisanego przypadku dokonywania podziałów. Najkrótsza gałąź w tym drzewie składa się z węzłów o wartościach $\alpha^i n$, gdzie i jest poziomem węzła, zaś najdłuższa gałąź na i-tym poziomie posiada węzeł o wartości $(1-\alpha)^i n$. Niech h będzie głębokością liścia najkrótszej gałęzi. Mamy wtedy $\alpha^h n = 1$, skąd

$$h = \log_\alpha \frac{1}{n} = -\frac{\lg n}{\lg \alpha}.$$

Jeśli przez H oznaczymy głębokość liścia na gałęzi najdłuższej, to mamy $(1-\alpha)^H n = 1$, a zatem

$$H = \log_{1-\alpha} \frac{1}{n} = -\frac{\lg n}{\lg(1-\alpha)}.$$

7.2-6. Załóżmy, że procedura PARTITION utworzyła podział w stosunku $1-\beta$ do β, gdzie $0 < \beta \leqslant 1/2$. Aby był on bardziej zrównoważony niż $1-\alpha$ do α, to musi być $\alpha < \beta$. Zdarzenie to można modelować za pomocą ciągłego rozkładu jednostajnego dla przedziału $(\alpha, 1/2]$ w przestrzeni $(0, 1/2]$. Wykorzystując definicję prawdopodobieństwa o ciągłym rozkładzie jednostajnym, dostajemy

$$\Pr((\alpha, 1/2]) = \frac{1/2-\alpha}{1/2-0} = 1-2\alpha.$$

7.3. Randomizowana wersja algorytmu quicksort

7.3-1. Randomizacja algorytmu sprawia, że żadne jego dane wejściowe nie stanowią przypadku pesymistycznego. Można więc przyjąć, że dla każdych danych wejściowych algorytm zachowuje się jak w przypadku średnim.

7.3-2. W przypadku pesymistycznym generator liczb losowych za każdym razem wybiera wartości, które tworzą najbardziej niezrównoważony podział podtablicy, czyli p lub r. W takiej sytuacji liczba wywołań generatora jest rzędu $\Theta(n)$.

W przypadku optymistycznym generator za każdym razem zwraca liczbę bliską $(p+r)/2$. Tworzone podziały są zatem zrównoważone i liczba wywołań generatora w tym przypadku wynosi $\Theta(\lg n)$.

7.4. Analiza algorytmu quicksort

7.4-1. Niech $n \geqslant 1$. Zgadujemy, że $T(q) \geqslant dq^2$ dla pewnej stałej $d > 0$ i wszystkich $q = 0, 1, \ldots, n-1$. Podstawiamy do wzoru na $T(n)$ i na podstawie zad. 7.4-3 otrzymujemy:

$$T(n) \geqslant \max_{0 \leqslant q \leqslant n-1} (dq^2 + d(n-q-1)^2) + \Theta(n)$$

$$\begin{aligned}
&= d \cdot \max_{0 \leqslant q \leqslant n-1} (q^2 + (n-q-1)^2) + \Theta(n) \\
&= dn^2 - d(2n-1) + \Theta(n) \\
&\geqslant dn^2.
\end{aligned}$$

Ostatnia nierówność zostaje spełniona, jeśli przyjmiemy d odpowiednio małe tak, aby składnik $\Theta(n)$ zdominował wyrażenie $d(2n-1)$. Przyjmijmy, że przypadkiem brzegowym rekurencji jest $T(0) = 1$. Dla dowolnego d zachodzi $T(0) \geqslant d \cdot 0^2 = 0$, a więc $n = 0$ przyjmujemy na podstawę indukcji, co kończy dowód, że $T(n) = \Omega(n^2)$.

7.4-2. Czas algorytmu quicksort w przypadku optymistycznym jest opisany przez rekurencję

$$T(n) = \min_{0 \leqslant q \leqslant n-1} (T(q) + T(n-q-1)) + \Theta(n).$$

Rozwiążemy ją metodą podstawiania. Niech $c > 0$ będzie stałą. Zgadujemy, że $T(q) \geqslant cq \lg q$ dla każdego $q = 0, 1, \ldots, n-1$ (przyjmujemy dla wygody, że $0 \lg 0 = 0$) i podstawiamy:

$$\begin{aligned}
T(n) &\geqslant \min_{0 \leqslant q \leqslant n-1} (cq \lg q + c(n-q-1) \lg(n-q-1)) + \Theta(n) \\
&= c \cdot \min_{0 \leqslant q \leqslant n-1} (q \lg q + (n-q-1) \lg(n-q-1)) + \Theta(n).
\end{aligned}$$

Obliczymy teraz minimum wyrażenia $q \lg q + (n-q-1) \lg(n-q-1)$, gdy $0 \leqslant q \leqslant n-1$. Jeśli $q = 0$ lub $q = n-1$, to wyrażenie ma wartość $(n-1) \lg(n-1)$. W pozostałych przypadkach potraktujmy je jako funkcję f zmiennej q. Pochodne f są postaci:

$$\begin{aligned}
\frac{df}{dq}(q) &= \frac{d}{dq}\left(\frac{q \ln q + (n-q-1) \ln(n-q-1)}{\ln 2}\right) = \frac{\ln q - \ln(n-q-1)}{\ln 2}, \\
\frac{d^2 f}{dq^2}(q) &= \frac{d}{dq}\left(\frac{\ln q - \ln(n-q-1)}{\ln 2}\right) = \frac{1}{\ln 2}\left(\frac{1}{q} + \frac{1}{n-q-1}\right).
\end{aligned}$$

Pierwsza pochodna zeruje się tylko wtedy, gdy $q = (n-1)/2$. Wyznaczamy drugą pochodną w tym punkcie:

$$\frac{d^2 f}{dq^2}\left(\frac{n-1}{2}\right) = \frac{1}{\ln 2}\left(\frac{2}{n-1} + \frac{2}{2n-(n-1)-2}\right) = \frac{4}{\ln 2 \cdot (n-1)}.$$

Liczba ta jest dodatnia, o ile $n > 1$. Mamy zatem, że minimum funkcji f znajduje się w punkcie $q = (n-1)/2$ i wynosi $(n-1) \lg((n-1)/2)$. Wartość ta stanowi tym samym minimum szukanego wyrażenia.

Powracamy do oszacowania rekurencji $T(n)$, przyjmując $n \geqslant 2$:

$$\begin{aligned}
T(n) &\geqslant c(n-1) \lg \frac{n-1}{2} + \Theta(n) \\
&= c(n-1) \lg(n-1) - c(n-1) + \Theta(n) \\
&= cn \lg(n-1) - c \lg(n-1) - c(n-1) + \Theta(n) \\
&\geqslant cn \lg(n/2) - c \lg(n-1) - c(n-1) + \Theta(n) \\
&= cn \lg n - c(2n + \lg(n-1) - 1) + \Theta(n) \\
&\geqslant cn \lg n.
\end{aligned}$$

Ostatnia nierówność zachodzi, o ile stała c jest na tyle mała, że wyrażenie $c(2n + \lg(n-1) - 1)$ jest zdominowane przez składnik $\Theta(n)$. Przyjmijmy, że przypadkiem brzegowym rekurencji jest $T(0) = d$, gdzie $d \geqslant 0$ jest pewną stałą. Na podstawę indukcji możemy więc przyjąć $n = 0$, dla którego $T(n)$ spełnia rozważane oszacowanie, a zatem $T(n) = \Omega(n \lg n)$.

7.4-3. Potraktujmy wyrażenie jako funkcję $f(q) = q^2 + (n-q-1)^2$, gdzie $0 \leqslant q \leqslant n-1$. W celu znalezienia maksimum globalnego tej funkcji obliczmy jej pierwszą i drugą pochodną:

$$\frac{df}{dq}(q) = 2q - 2(n-q-1),$$
$$\frac{d^2f}{dq^2}(q) = 4.$$

Ponieważ druga pochodna jest dodatnia, to funkcja f nie posiada maksimum lokalnego i jej największej wartości należy szukać w punktach brzegowych dziedziny. Mamy $f(0) = f(n-1) = (n-1)^2$, więc maksimum jest osiągane w obu tych punktach.

7.4-4. Przy wyznaczaniu dolnego oszacowania na oczekiwany czas działania algorytmu quicksort korzystamy z analizy przedstawionej w Podręczniku dla górnego oszacowania. Zauważmy, że lemat 7.1 pozostaje prawdziwy, gdyby zamiast notacji O zastosować Ω. Prowadząc rozumowanie analogicznie, dochodzimy w końcu do wartości oczekiwanej zmiennej losowej X, którą następnie ograniczamy od dołu:

$$\mathrm{E}(X) = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i}\frac{2}{k+1} \geqslant \sum_{i=1}^{n-1}\sum_{k=1}^{n-i}\frac{2}{2k} = \sum_{i=1}^{n-1} H_{n-i} = \sum_{i=1}^{n-1} H_i = \sum_{i=1}^{n-1}\Omega(\lg i) = \Omega(n\lg n).$$

Skorzystaliśmy tutaj z zad. A.2-3 oraz z punktu (b) problemu A-1 dla $s = 1$, skąd otrzymaliśmy ostatnie dwie równości.

7.4-5. W rozważanej modyfikacji drzewo rekursji w algorytmie quicksort ma około $\lg(n/k)$ poziomów, z których każdy wnosi koszt $O(n)$. W przypadku średnim czas wykonania tego kroku wynosi $O(n\lg(n/k))$. Liczba fragmentów o mniej niż k elementach, których nie uporządkowano w pierwszej fazie, jest rzędu $O(n/k)$. Drugi krok polega na posortowaniu ich przez wstawianie i zajmuje czas $O(n/k \cdot k^2) = O(nk)$. Stąd całkowitym oczekiwanym czasem działania algorytmu jest $O(nk + n\lg(n/k))$.

Teoretycznie parametr k powinien być rzędu co najwyżej $O(\lg n)$ – wtedy złożoność czasowa tego algorytmu nie przewyższa złożoności czasowej sortowania szybkiego. W praktyce jednak k powinno być tak dobrane, aby sortowanie przez wstawianie tablicy o długości k było wykonywane szybciej od sortowania takiej tablicy algorytmem quicksort.

7.4-6. Niech P będzie szukaną funkcją zmiennej $0 < \alpha < 1$. Zauważmy, że funkcja ta spełnia własność $P(\alpha) = P(1-\alpha)$, w rozwiązaniu będziemy więc zakładać, że $\alpha \leqslant 1/2$. Niech $i < j < k$ będą indeksami elementów losowo wybranych z A. Utworzenie w najgorszym przypadku podziału α do $1-\alpha$ jest równoważne temu, że $\alpha n \leqslant j \leqslant (1-\alpha)n$. Możliwe są następujące przypadki ze względu na wartości przyjmowane przez pozostałe indeksy:

(i) $i < \alpha n$, $k > (1-\alpha)n$, co zachodzi z prawdopodobieństwem około $6\alpha^2(1-2\alpha)$;

(ii) $i < \alpha n$, $\alpha n \leqslant k \leqslant (1-\alpha)n$, co zachodzi z prawdopodobieństwem około $3\alpha(1-2\alpha)^2$;

(iii) $\alpha n \leqslant i \leqslant (1-\alpha)n$, $k > (1-\alpha)n$, co zachodzi z prawdopodobieństwem około $3\alpha(1-2\alpha)^2$;

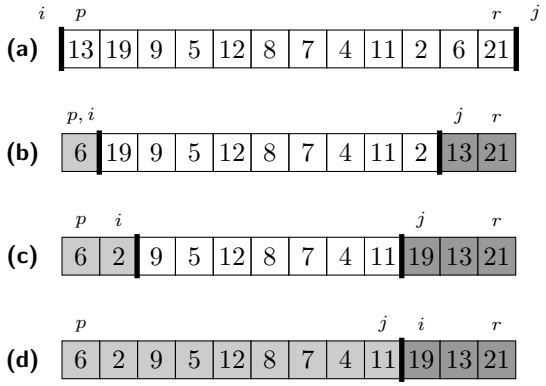
(iv) $\alpha n \leqslant i \leqslant (1-\alpha)n$, $\alpha n \leqslant k \leqslant (1-\alpha)n$, co zachodzi z prawdopodobieństwem około $(1-2\alpha)^3$.

Sumując prawdopodobieństwa z wszystkich przypadków, otrzymujemy $P(\alpha) \approx 4\alpha^3 - 6\alpha^2 + 1$.

Problemy

7-1. Poprawność algorytmu podziału Hoare'a

(a) Działanie procedury HOARE-PARTITION dla przykładowej tablicy A zostało przedstawione na rys. 15.



Rysunek 15: Działanie procedury HOARE-PARTITION dla tablicy $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$. Wszystkie elementy jasnoszare stanowią obszar złożony z wartości nie większych niż x. Ciemnoszare elementy tworzą obszar złożony z wartości nie mniejszych niż x. **(a)** Wejściowa tablica wraz z początkowym ustawieniem zmiennych. **(b)–(d)** Tablica i bieżące wartości zmiennych po wykonaniu, odpowiednio, jednej, dwóch i trzech iteracji pętli **while** w wierszach 4–11.

(b) W pierwszej iteracji pętli **while** zmienna j zatrzyma się na pewnym indeksie $q \geqslant p$, a zmienna i pozostanie na indeksie p, pod którym przechowywana jest wartość x. Jeśli $i = j$, to procedura kończy działanie, załóżmy więc, że $i < j$. Element $A[i] = A[p] = x$ zostanie zamieniony z $A[q]$. W kolejnych iteracjach pętli **while** zmienna i może dotrzeć najdalej na indeks q, natomiast j nie będzie nigdy mniejsze niż p, bo $A[p]$ zawiera teraz wartość mniejszą bądź równą x. Ponieważ $q > p$, to w pewnym momencie podczas działania pętli indeksy i i j miną się, czyli będzie $i \geqslant j$, co spowoduje przerwanie pętli w wierszu 11.

(c) *Zakładamy, że podtablica $A[p\,..\,r]$ składa się z co najmniej dwóch elementów.*

Fakt udowodniony w punkcie (b) mówi o tym, że $p \leqslant j \leqslant r$. Załóżmy, że $A[r] \leqslant x$, bowiem w przeciwnym przypadku $j < r$ już po pierwszej iteracji pętli **while**. W pierwszej iteracji element $A[p]$ jest zamieniany z $A[r]$, po czym w kolejnych iteracjach indeks j jest zmniejszany i również w tym przypadku mamy $j < r$.

(d) Wynika to z warunków stopu obu pętli **repeat** i testu z wiersza 9. Każda para elementów tablicy, która tworzyła inwersję, jest odwracana, dzięki czemu elementy, które naruszały warunek posortowania, po zamianie będą znajdować się w odpowiednich obszarach tablicy. Jak tylko indeksy i i j miną się, każdy element z podtablicy $A[p\,..\,j]$ będzie równy bądź mniejszy od każdego elementu z $A[j+1\,..\,r]$.

(e) Jedyną różnicą w porównaniu z procedurą QUICKSORT jest to, że element rozdzielający nie znajduje się w $A[q]$ po wykonaniu HOARE-PARTITION, musimy więc sortować rekurencyjnie

fragment $A[p\,..\,q]$ zamiast $A[p\,..\,q-1]$. Pseudokod procedury został przedstawiony poniżej.

HOARE-QUICKSORT(A, p, r)
1 **if** $p < r$
2 **then** $q \leftarrow$ HOARE-PARTITION(A, p, r)
3 HOARE-QUICKSORT(A, p, q)
4 HOARE-QUICKSORT$(A, q+1, r)$

7-2. Alternatywna analiza algorytmu quicksort

(a) W procedurze RANDOMIZED-PARTITION element $A[r]$ jest zamieniany z elementem losowo wybranym z tablicy A o rozmiarze n. Stąd szanse, że pewien ustalony element tablicy A zostanie elementem rozdzielającym, wynoszą $1/n$. Wobec tego wartość oczekiwana zmiennej X_i, gdzie $i = 1, 2, \ldots, n$, wynosi $\mathrm{E}(X_i) = \Pr(X_i = 1) = 1/n$.

(b) Wywołanie procedury RANDOMIZED-QUICKSORT dla tablicy wejściowej A o rozmiarze n potrzebuje czasu $\Theta(n)$ na podział tablicy. Zostaje wyznaczony pewien indeks $1 \leqslant q \leqslant n$, po czym procedura jest wywoływana rekurencyjnie dla podtablic rozmiarów $q-1$ i $n-q$. Zmienna losowa X_i zdefiniowana w punkcie (a) przyjmuje wartość 1, gdy $i = q$, a w pozostałych przypadkach przyjmuje wartość 0. Stąd czas potrzebny na posortowanie n-elementowej tablicy wyraża się wzorem

$$T(n) = T(q-1) + T(n-q) + \Theta(n) = \sum_{i=1}^{n} X_i(T(i-1) + T(n-i) + \Theta(n)).$$

Biorąc wartości oczekiwane skrajnych wyrażeń, otrzymujemy wzór (7.5) na oczekiwany czas działania algorytmu quicksort.

(c) *Przyjmiemy, że we wzorze (7.6) sumowanie przebiega od $q = 2$ do $q = n-1$.*

Wykorzystując części (a) i (b) oraz liniowość wartości oczekiwanej, otrzymujemy

$$\begin{aligned}
\mathrm{E}(T(n)) &= \sum_{q=1}^{n} \mathrm{E}\big(X_q(T(q-1) + T(n-q) + \Theta(n))\big) \\
&= \sum_{q=1}^{n} \mathrm{E}(X_q)\big(\mathrm{E}(T(q-1)) + \mathrm{E}(T(n-q)) + \mathrm{E}(\Theta(n))\big) \\
&= \sum_{q=1}^{n} \frac{1}{n}\big(\mathrm{E}(T(q-1)) + \mathrm{E}(T(n-q)) + \Theta(n)\big) \\
&= \frac{1}{n}\sum_{q=0}^{n-1} 2\,\mathrm{E}(T(q)) + \frac{1}{n}\sum_{q=1}^{n} \Theta(n) \\
&= \frac{2}{n}\sum_{q=2}^{n-1} \mathrm{E}(T(q)) + \frac{2}{n}\big(\mathrm{E}(T(0)) + \mathrm{E}(T(1))\big) + \Theta(n) \\
&= \frac{2}{n}\sum_{q=2}^{n-1} \mathrm{E}(T(q)) + \Theta(n).
\end{aligned}$$

Skorzystaliśmy z faktu, że $\mathrm{E}(T(0))$ i $\mathrm{E}(T(1))$ są stałymi i mogą zostać wchłonięte przez $\Theta(n)$.

(d) Rozdzielamy sumę na dwie części:

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k,$$

po czym zauważamy, że czynnik $\lg k$ w pierwszej sumie po prawej stronie znaku równości możemy ograniczyć z góry przez $\lg(n/2) = \lg n - 1$, a czynnik $\lg k$ w drugiej sumie – przez $\lg n$. Stąd

$$\begin{aligned}
\sum_{k=1}^{n-1} k \lg k &\leqslant (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\
&= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \\
&\leqslant \frac{n(n-1)\lg n}{2} - \frac{n}{4}\left(\frac{n}{2} - 1\right) \\
&= \frac{n^2 \lg n}{2} - \frac{n \lg n}{2} - \frac{n^2}{8} + \frac{n}{4} \\
&\leqslant \frac{n^2 \lg n}{2} - \frac{n^2}{8},
\end{aligned}$$

przy czym ostatnia nierówność zachodzi, gdy $n \geqslant \sqrt{2}$.

(e) Załóżmy, że $n > 2$ i przyjmijmy założenie indukcyjne $\mathrm{E}(T(q)) \leqslant aq \lg q$ dla każdego $q = 2, 3, \ldots, n-1$ oraz pewnej stałej $a > 0$. Z punktu (c) mamy

$$\mathrm{E}(T(n)) = \frac{2}{n} \sum_{q=2}^{n-1} \mathrm{E}(T(q)) + \Theta(n) \leqslant \frac{2}{n} \sum_{q=2}^{n-1} aq \lg q + \Theta(n) = \frac{2a}{n} \sum_{q=1}^{n-1} q \lg q + \Theta(n).$$

Wykorzystując teraz wynik z punktu (d), dostajemy

$$\mathrm{E}(T(n)) \leqslant \frac{2a}{n}\left(\frac{n^2 \lg n}{2} - \frac{n^2}{8}\right) + \Theta(n) = an \lg n - \frac{an}{4} + \Theta(n) \leqslant an \lg n,$$

gdyż możemy wybrać a wystarczająco duże, by wyrażenie $an/4$ dominowało nad składnikiem $\Theta(n)$.

Jeśli przyjmiemy, że $\mathrm{E}(T(2)) = 1$, to wówczas będziemy mieć $1 \leqslant a \cdot 2 \cdot \lg 2 = 2a$, skąd $a \geqslant 1/2$. A zatem można przyjąć $n = 2$ na podstawę indukcji i dowód faktu, że $\mathrm{E}(T(n)) = O(n \lg n)$ jest zakończony. Łącząc ten wynik z oszacowaniem $\Omega(n \lg n)$ dla przypadku optymistycznego, które zostało wyznaczone w zad. 7.4-2, dostajemy, że oczekiwanym czasem działania algorytmu quicksort jest $\Theta(n \lg n)$.

7-3. Nieefektywne sortowanie

(a) Udowodnimy poprawność algorytmu przez indukcję względem rozmiaru tablicy n. Łatwo sprawdzić, że algorytm działa poprawnie w przypadku, gdy $n \leqslant 2$. Załóżmy więc, że $n > 2$ i że poprawnie sortowane są tablice o rozmiarach mniejszych niż n. W wyniku wykonania wiersza 6 na pozycjach $\lfloor n/3 \rfloor + 1 \,..\, n - \lfloor n/3 \rfloor$ znajdują się elementy nie mniejsze od tych z pozycji

$1\,..\,\lfloor n/3 \rfloor$. A zatem, po wykonaniu wiersza 7, $\lfloor n/3 \rfloor$ największych elementów tablicy A znajduje się w obszarze $A[n-\lfloor n/3 \rfloor+1\,..\,n]$. Aby zakończyć sortowanie tablicy A, wystarczy uporządkować fragment $A[1\,..\,n-\lfloor n/3 \rfloor]$, co realizuje wiersz 8.

(b) Procedura jest wywoływana rekurencyjnie 3 razy, zaś każde wywołanie otrzymuje tablicę o rozmiarze około 2/3 rozmiaru oryginalnej tablicy. Ponadto praca poza wywołaniami rekurencyjnymi jest wykonywana w czasie stałym. Stąd dostajemy równanie rekurencyjne

$$T(n) = \begin{cases} \Theta(1), & \text{jeśli } n \leqslant 2, \\ 3T(2n/3) + \Theta(1), & \text{jeśli } n > 2, \end{cases}$$

które rozwiązujemy przy użyciu twierdzenia o rekurencji uniwersalnej, otrzymując wynik $T(n) = \Theta(n^{\log_{3/2} 3}) \approx \Theta(n^{2,71})$.

(c) Pesymistyczny czas działania algorytmu Stooge-Sort jest wyższego rzędu nie tylko od pesymistycznego czasu działania algorytmów sortowania przez scalanie, kopcowanie czy sortowania szybkiego, ale również od mniej efektywnego sortowania przez wstawianie. Metoda sortowania profesorów jest więc poprawna, ale bardzo powolna.

7-4. Głębokość stosu w algorytmie quicksort

(a) Quicksort′ wykonuje te same operacje na tablicy A co oryginalny algorytm quicksort. Różnica jest tylko w przetwarzaniu podtablicy $A[q+1\,..\,r]$. Zamiast drugiego wywołania rekurencyjnego procedura wykonuje przypisanie w wierszu 5, po czym następuje kolejna iteracja pętli **while**, co działa identycznie jak wywołanie Quicksort′$(A, q+1, r)$, ale bez zwiększania stosu rekurencji. Poprawność algorytmu wynika zatem z poprawności oryginalnego algorytmu quicksort.

(b) Stos rekurencji może urosnąć do rozmiaru $\Theta(n)$ w sytuacji, gdy będzie $\Theta(n)$ wywołań rekurencyjnych Quicksort′. Dzieje się tak wtedy, gdy na każdym poziomie rekursji procedura Partition zwraca $q = r$. Do wywołania rekurencyjnego przekazywana jest wtedy podtablica o 1 mniejsza w porównaniu z początkową tablicą. Algorytm zachowuje się w ten sposób, jeśli na wejście dostanie tablicę posortowaną niemalejąco.

(c) Wykorzystamy pomysł, aby wywoływać procedurę rekurencyjnie dla mniejszej podtablicy, natomiast większą przetwarzać w bieżącym wywołaniu. Dzięki temu na kolejnym poziomie rekursji rozważany będzie problem mniejszy co najmniej o połowę, więc w najgorszym przypadku głębokość stosu rekurencji wyniesie $\Theta(\lg n)$. Tworzone podziały będą takie same jak przed dokonaniem usprawnienia, zatem oczekiwany czas działania algorytmu nie zmieni się. Zmodyfikowany kod procedury Quicksort′ został przedstawiony poniżej.

```
Quicksort''(A, p, r)
1  while p < r
2        do ▷ Dziel i sortuj mniejszą podtablicę.
3           q ← Partition(A, p, r)
4           if q − p < r − q
5              then Quicksort''(A, p, q − 1)
6                   p ← q + 1
7              else Quicksort''(A, q + 1, r)
8                   r ← q − 1
```

7-5. Podział względem mediany trzech wartości

(a) Element rozdzielający x znajdzie się na pozycji i w tablicy A', jeżeli drugi z wybranych elementów będzie na lewo od i w tej tablicy, a trzeci wybrany element – na prawo od i. Liczby możliwych pozycji, jakie mogą zająć drugi i trzeci element, wynoszą, odpowiednio, $i-1$ i $n-i$. Otrzymujemy zatem

$$p_i = \frac{(i-1)(n-i)}{\binom{n}{3}}.$$

(b) W zwykłej implementacji szanse wyboru mediany tablicy $A[1\,..\,n]$ na element rozdzielający są równe $1/n$ (z części (a) problemu 7-2), natomiast w metodzie mediany trzech wartości wynoszą one $p_{\lfloor (n+1)/2 \rfloor}$. Jeśli n jest parzyste, to

$$p_{\lfloor (n+1)/2 \rfloor} = p_{n/2} = \frac{\left(\frac{n}{2}-1\right)\left(n-\frac{n}{2}\right)}{\binom{n}{3}} = \frac{3}{2(n-1)}.$$

Stosunek prawdopodobieństw z obu metod dąży do

$$\lim_{n\to\infty} \frac{p_{\lfloor (n+1)/2 \rfloor}}{1/n} = \lim_{n\to\infty} \frac{3n}{2(n-2)} = \frac{3}{2}.$$

Dla n nieparzystego mamy

$$p_{\lfloor (n+1)/2 \rfloor} = p_{(n+1)/2} = \frac{\left(\frac{n+1}{2}-1\right)\left(n-\frac{n+1}{2}\right)}{\binom{n}{3}} = \frac{3(n-1)}{2n(n-2)},$$

więc granica stosunku wynosi

$$\lim_{n\to\infty} \frac{p_{\lfloor (n+1)/2 \rfloor}}{1/n} = \lim_{n\to\infty} \frac{3(n-1)}{2(n-2)} = \frac{3}{2}.$$

A zatem niezależnie od parzystości n szanse na to, że mediana tablicy $A[1\,..\,n]$ zostanie wybrana na element rozdzielający, są większe o 50% w metodzie mediany trzech wartości dla dostatecznie dużych n.

(c) W tradycyjnym podejściu szansa na uzyskanie dobrego podziału jest bliska $1/3$. Szanse na dobry podział w metodzie mediany trzech wartości wynoszą

$$\sum_{i=\lceil n/3 \rceil}^{\lfloor 2n/3 \rfloor} p_i \approx 1 - 2\sum_{i=1}^{n/3} \frac{(i-1)(n-i)}{\binom{n}{3}} = 1 - \frac{12}{n(n-1)(n-2)} \sum_{i=1}^{n/3} (i-1)(n-i).$$

Ostatnią sumę przybliżamy za pomocą całki $\int_1^{n/3} (x-1)(n-x)\,dx$, podstawiając $t = x-1$:

$$\begin{aligned}
\int_0^{n/3-1} t(n-t-1)\,dt &= \left[\frac{(n-1)t^2}{2} - \frac{t^3}{3}\right]_0^{n/3-1} \\
&= \frac{(n-1)(n/3-1)^2}{2} - \frac{(n/3-1)^3}{3} \\
&= \frac{(n-1)(n-3)^2}{18} - \frac{(n-3)^3}{81}.
\end{aligned}$$

Wracając do oszacowania prawdopodobieństwa uzyskania dobrego podziału, otrzymujemy

$$\sum_{i=\lceil n/3\rceil}^{\lfloor 2n/3\rfloor} p_i \approx 1 - \frac{2(n-3)^2}{3n(n-2)} + \frac{4(n-3)^3}{27n(n-1)(n-2)}.$$

Dzięki zastosowaniu nowej strategii wyboru elementu rozdzielającego szanse na utworzenie dobrego podziału wzrastają o czynnik

$$\lim_{n\to\infty} \frac{\sum_{i=\lceil n/3\rceil}^{\lfloor 2n/3\rfloor} p_i}{1/3} \approx \frac{1 - 2/3 + 4/27}{1/3} = \frac{13}{9}.$$

(d) Nowy sposób wyboru elementu rozdzielającego zwiększa jedynie szanse na uzyskanie podziału zrównoważonego, co z kolei obniża prawdopodobieństwo, że algorytm quicksort będzie działał w czasie kwadratowym. Jednakże dolne oszacowanie na czas działania algorytmu pozostaje bez zmian i wynosi nadal $\Omega(n \lg n)$ – można sobie wyobrazić sytuację, w której oryginalny sposób wyboru elementu rozdzielającego generuje za każdym razem najbardziej zrównoważony podział.

7-6. Rozmyte sortowanie przedziałów

(a) Niech A będzie tablicą wejściową, przy czym $A[i] = [a_i, b_i]$ dla $i = 1, 2, \ldots, n$. Zauważmy, że jeśli $[a_i, b_i] \cap [a_j, b_j] \neq \emptyset$, czyli przedziały $A[i]$ i $A[j]$ nachodzą na siebie, to mogą wystąpić w tablicy wynikowej w dowolnej kolejności. Algorytm działa podobnie jak quicksort, ale wykorzystuje tę obserwację, znajdując zbiór przedziałów nachodzących na przedział stanowiący element rozdzielający i pomijając wywołanie rekurencyjne dla podtablicy utworzonej przez ten zbiór przedziałów.

Procedura Fuzzy-Sort implementuje rozmyte sortowanie przedziałów. Aby posortować całą tablicę A, należy wywołać Fuzzy-Sort$(A, 1, \mathit{length}[A])$.

```
FUZZY-SORT(A, p, r)
1  if p < r
2     then ⟨q1, q2⟩ ← FUZZY-PARTITION(A, p, r)
3          FUZZY-SORT(A, p, q1 − 1)
4          FUZZY-SORT(A, q2 + 1, r)
```

Pomocnicza procedura Fuzzy-Partition dokonuje podziału tablicy $A[p\,..\,r]$ na 3 podtablice: $A[q_1\,..\,q_2]$, która nie musi być dalej sortowana, oraz $A[p\,..\,q_1 - 1]$ i $A[q_2 + 1\,..\,r]$, które następnie sortowane są w wywołaniach rekurencyjnych w wierszach 3 i 4. Pseudokod tej procedury pomocniczej został przedstawiony poniżej.

```
FUZZY-PARTITION(A, p, r)
 1  zamień A[r] ↔ A[RANDOM(p, r)]
 2  x ← a_r
 3  i ← p − 1
 4  for j ← p to r − 1
 5        do if a_j ⩽ x
 6              then i ← i + 1
 7                   zamień A[i] ↔ A[j]
 8  zamień A[i + 1] ↔ A[r]
 9  q ← i + 1
10  for k ← i downto p
11        do if b_k ⩾ x
12              then q ← q − 1
13                   zamień A[q] ↔ A[k]
14  return ⟨q, i + 1⟩
```

Lewy koniec przedziału stanowiącego element rozdzielający, wybrany losowo spośród wszystkich elementów tablicy wejściowej, zostaje przypisany do zmiennej x. Po wykonaniu wiersza 8 tablica A jest podzielona na dwie podtablice według lewych końców przedziałów względem x. Ta część jest więc analogiczna do zwykłej procedury PARTITION, w wyniku czego dostajemy dwa obszary tablicy rozdzielone elementem x. Następnie, w wierszach 9–13, wszystkie przedziały z podtablicy $A[p\,..\,i]$, które nachodzą na element rozdzielający znajdujący się teraz w $A[i+1]$, zostają przeniesione na koniec tej podtablicy. W rezultacie z przedziałów tych utworzony zostaje trzeci obszar tablicy, niewymagający dalszego sortowania. Na końcu zwracane są indeksy początku i końca tego obszaru.

(b) Algorytm został oparty o randomizowaną wersję quicksorta, więc jego czas działania dla tablicy n-elementowej w przypadku średnim wynosi $\Theta(n \lg n)$. Jeśli jednak wszystkie przedziały na siebie nachodzą, to lewa część podtablicy podzielonej w wyniku działania procedury FUZZY-PARTITION będzie za każdym razem pusta. Na każdym poziomie rekursji będzie więc sortowany tylko jeden obszar. Randomizacja zapewnia, że oczekiwaną pozycją elementu rozdzielającego jest środek podtablicy $A[p\,..\,r]$ (patrz zad. C.3-2) i w kolejnym wywołaniu rekurencyjnym sortowany fragment jest o połowę mniejszy. Oczekiwany czas działania algorytmu w tym przypadku jest więc opisany przez rekurencję $T(n) = T(n/2) + \Theta(n)$, której rozwiązaniem jest $\Theta(n)$.

Rozdział **8**

Sortowanie w czasie liniowym

8.1. Dolne ograniczenia dla problemu sortowania

8.1-1. Łatwo zauważyć, że do znalezienia uporządkowania n elementów ciągu wejściowego potrzebnych jest w najlepszym przypadku $n-1$ porównań. Sytuacja zachodzi np. dla ciągu niemalejącego. A zatem najmniejszą głębokością liścia w drzewie decyzyjnym jest $n-1$.

8.1-2. Górne oszacowanie znajdujemy w prosty sposób:

$$\lg(n!) = \sum_{k=1}^{n} \lg k \leqslant \sum_{k=1}^{n} \lg n = n \lg n = O(n \lg n).$$

Aby uzyskać oszacowanie dolne, rozdzielamy sumę, korzystając z faktu, że $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$:

$$\lg(n!) = \sum_{k=1}^{n} \lg k = \sum_{k=1}^{\lfloor n/2 \rfloor} \lg k + \sum_{k=\lceil n/2 \rceil}^{n} \lg k.$$

W pierwszej sumie po prawej ograniczamy $\lg k$ od dołu przez $\lg 1$, a w drugiej – przez $\lg\lceil n/2 \rceil$:

$$\lg(n!) \geqslant \sum_{k=1}^{\lfloor n/2 \rfloor} \lg 1 + \sum_{k=\lceil n/2 \rceil}^{n} \lg\lceil n/2 \rceil \geqslant 0 + (n/2) \lg\lceil n/2 \rceil \geqslant (n/2) \lg(n/2) = \Omega(n \lg n).$$

8.1-3. Jeśli sortowanie działa w czasie $\Theta(n)$ dla l permutacji wejściowych długości n, to drzewo decyzyjne składające się z gałęzi odpowiadających tylko tym permutacjom, ma wysokość $h = \Theta(n)$. Powtarzając rozumowanie z dowodu tw. 8.1, dostajemy nierówność $2^h \geqslant l$, a stąd $h \geqslant \lg l$. Pozostaje więc sprawdzić, czy h jest asymptotycznie ograniczone funkcją liniową względem n dla poszczególnych wartości przyjmowanych przez l:

- jeśli $l = n!/2$, to $h \geqslant \lg l = \lg(n!/2) = \lg(n!) - 1 = \Omega(n \lg n)$;
- jeśli $l = n!/n$, to $h \geqslant \lg l = \lg(n!/n) = \lg(n!) - \lg n = \Omega(n \lg n)$;
- jeśli $l = n!/2^n$, to $h \geqslant \lg l = \lg(n!/2^n) = \lg(n!) - n = \Omega(n \lg n)$.

W każdym badanym przypadku $h = \Omega(n \lg n)$, zatem nie istnieje algorytm sortujący za pomocą porównań i działający w czasie liniowym dla poszczególnych ilości permutacji wejściowych.

8.1-4. Rozważmy drzewo decyzyjne reprezentujące sortowanie takiego częściowo uporządkowanego ciągu. Każdy podciąg może wystąpić na wyjściu jako jedna z $k!$ możliwych permutacji. Ponieważ jest n/k podciągów, to w drzewie decyzyjnym znajduje się co najmniej $(k!)^{n/k}$ osiągalnych liści. Na mocy faktu, że drzewo decyzyjne o wysokości h nie może mieć więcej niż 2^h liści, dostajemy

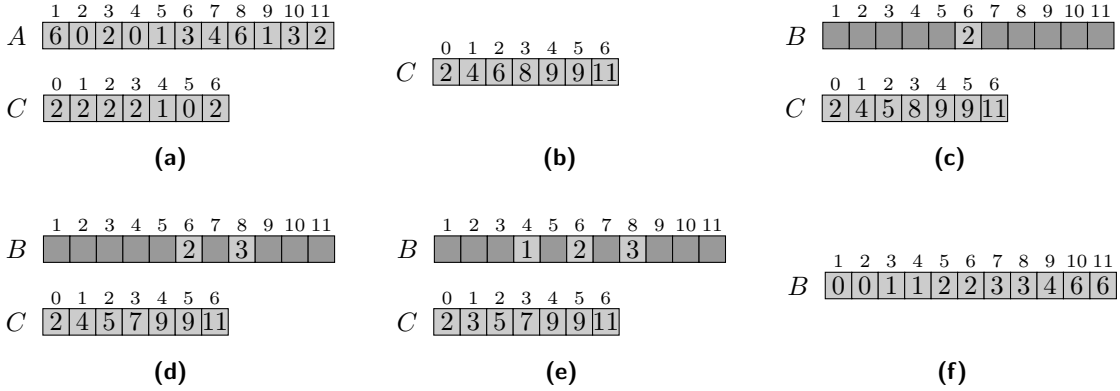
$$2^h \geqslant (k!)^{n/k},$$

skąd uzyskujemy dolne oszacowanie na ilość porównań wykonywanych w tym algorytmie:

$$h \geqslant \lg\big((k!)^{n/k}\big) = (n/k)\lg(k!) = (n/k)\cdot\Omega(k\lg k) = \Omega(n\lg k).$$

8.2. Sortowanie przez zliczanie

8.2-1. Rys. 16 przedstawia działanie procedury COUNTING-SORT dla tablicy A.



Rysunek 16: Działanie procedury COUNTING-SORT dla tablicy $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2\rangle$, której każdy element jest nieujemną liczbą całkowitą nie większą niż $k = 6$. **(a)** Tablica A wraz z tablicą pomocniczą C po wykonaniu pętli w wierszach 3–4. **(b)** Tablica C po wykonaniu pętli w wierszach 6–7. **(c)–(e)** Tablica wynikowa B oraz tablica pomocnicza C po wykonaniu, odpowiednio, jednej, dwóch i trzech iteracji pętli **for** w wierszach 9–11. **(f)** Wynikowa posortowana tablica B.

8.2-2. Elementy tablicy wejściowej przetwarzane są od końca, a wartości w tablicy C stanowiące indeksy tablicy wynikowej, na które trafią wejściowe elementy, są systematycznie zmniejszane. A zatem równe sobie elementy będą umieszczane na coraz niższych pozycjach w tablicy wynikowej, dzięki czemu ich początkowa kolejność zostanie zachowana.

8.2-3. Po wprowadzeniu takiej modyfikacji tablica A będzie przeglądana od lewej do prawej, a równe elementy będą umieszczane w odpowiedniej części tablicy wynikowej na coraz niższych pozycjach. Oczywiście kolejność przetwarzania elementów tablicy A nie wpływa na poprawność algorytmu COUNTING-SORT, jednak zastosowanie opisanej zmiany powoduje, że sortowanie nie jest stabilne.

8.2-4. Algorytm będzie zliczać elementy z wejściowej tablicy w czasie $\Theta(n + k)$, zapamiętując wyniki w pomocniczej tablicy $C[0\,..\,k]$. Następnie, zapytany o liczbę elementów z przedziału $a\,..\,b$

na podstawie danych z tablicy pomocniczej, zwróci liczbę elementów z zakresu $0 \,..\, b$ pomniejszoną o liczbę elementów z zakresu $0 \,..\, a-1$. Dokładniej, jego pierwsza faza jest równoważna wierszom 1–8 procedury COUNTING-SORT, natomiast w drugiej fazie zwracany jest odpowiedni wynik w zależności od przypadku:

(i) $C[b] - C[a-1]$, jeśli $0 < a \leqslant b \leqslant k$;

(ii) $C[k] - C[a-1]$, jeśli $0 < a \leqslant k < b$;

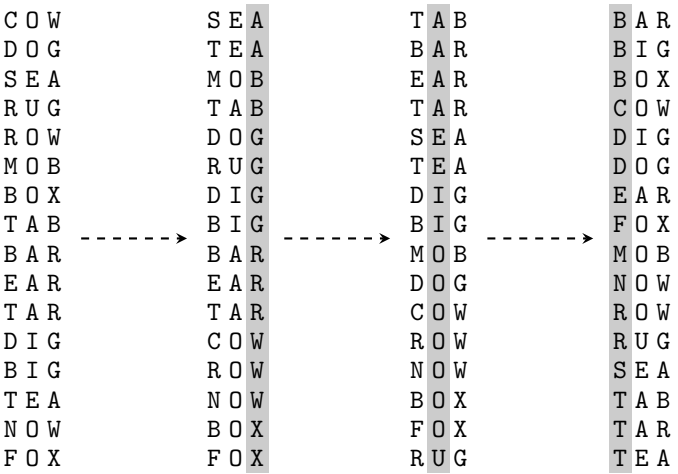
(iii) $C[b]$, jeśli $a \leqslant 0 \leqslant b \leqslant k$;

(iv) $C[k]$, jeśli $a \leqslant 0 \leqslant k < b$;

(v) 0, jeśli $a > k$ lub $b < 0$.

8.3. Sortowanie pozycyjne

8.3-1. Przebieg działania procedury RADIX-SORT dla podanej listy słów został przedstawiony na rys. 17.



Rysunek 17: Działanie procedury RADIX-SORT dla zbioru słów trzyliterowych.

8.3-2. Algorytmami stabilnymi są sortowanie przez wstawianie i sortowanie przez scalanie. W pierwszym z nich, wstawiając element $A[j]$ do podtablicy $A[1 \,..\, j-1]$, zatrzymujemy się na pierwszym elemencie z tej podtablicy, który jest mniejszy lub równy od $A[j]$. Zatem elementy równe sobie nie zostaną wymieszane. Podobnie w procedurze MERGE, jeśli porównywane elementy będą sobie równe, to do wynikowej tablicy zostanie wstawiony najpierw element z tablicy L i dopiero potem ten z tablicy R. Stabilność algorytmu wynika na podstawie indukcji po scalanych fragmentach. Natomiast zarówno heapsort, jak i quicksort nie sortuje stabilnie – przykładem danych wejściowych, które ilustrują ten fakt w obu tych algorytmach, jest ciąg $A = \langle 2, 2, 1 \rangle$.

Aby dowolny algorytm sortowania za pomocą porównań uczynić stabilnym, można zapamiętywać z każdym elementem jego początkową pozycję w tablicy wejściowej i przy każdym teście dającym odpowiedź, że elementy są sobie równe, porządkować je za pomocą ich początkowych

pozycji. Do realizacji takiego podejścia wymagana jest dodatkowa pamięć rzędu $\Theta(n)$, gdzie n jest długością sortowanej tablicy. Nie zwiększa się natomiast asymptotyczne oszacowanie na czas działania zmodyfikowanego sortowania, ponieważ przeprowadzenie dodatkowego testu odbywa się w czasie stałym.

8.3-3. Przeprowadzimy dowód przez indukcję względem liczby cyfr d elementów wejściowych. Dla $d = 1$ algorytm RADIX-SORT sprowadza się do wywołania pomocniczej procedury do posortowania pojedynczych cyfr, a zatem poprawność algorytmu wynika z poprawności pomocniczego sortowania.

Załóżmy teraz, że $d > 1$ i że wywołanie algorytmu RADIX-SORT po $d-1$ fazach zwróciło tablicę elementów, które, obcięte do $d - 1$ ostatnich pozycji, wyznaczają porządek niemalejący. Teraz elementy sortowane są względem pozycji d. Niech a i b będą cyframi porównywanymi podczas tej fazy. Jeśli $a \neq b$, to niezależnie od pozostałych cyfr elementów, których częściami są a i b, elementy te zostaną ustawione w odpowiedniej kolejności. Jeśli jednak $a = b$, to elementy nie zostaną zamienione miejscami, bo pomocnicza procedura sortuje stabilnie. Elementy pozostają jednak we właściwym porządku, bo jest on wyznaczony przez ich $d - 1$ ostatnich pozycji, a te, na mocy założenia indukcyjnego, zostały poprawnie uporządkowane w poprzednich fazach algorytmu.

8.3-4. Każdą liczbę z zakresu $0 \,..\, n^2 - 1$ można potraktować jak liczbę dwucyfrową w systemie o podstawie n. Liczby w takiej postaci sortujemy, wywołując algorytm RADIX-SORT o dwóch fazach. W lemacie 8.3 mamy $d = 2$ oraz $k = n$, zatem opisane sortowanie działa w czasie $\Theta(2(n + n)) = \Theta(n)$.

8.3-5. Rozważmy sortowanie pozycyjne w wersji intuicyjnej dla liczb trzycyfrowych. W pierwszej fazie sortujemy po najbardziej znaczącej cyfrze wejściowego ciągu liczb. Podczas drugiej fazy dokonujemy sortowania w obrębie fragmentów tablicy zawierających liczby o tej samej najbardziej znaczącej cyfrze. W najgorszym przypadku po pierwszej fazie dostaniemy 10 takich fragmentów, każdy o innej najbardziej znaczącej cyfrze, a zatem środkowe cyfry sortowane będą w kolejnych 10 fazach. Sytuacja w najgorszym przypadku powtórzy się – z każdego sortowanego fragmentu utworzy się po 10 jeszcze mniejszych fragmentów o poszczególnych kombinacjach dwóch pierwszych cyfr. Będzie zatem maksymalnie 100 takich podtablic, których posortowanie będzie wymagało 100 kolejnych faz. Łącznie algorytm wykona zatem 111 faz.

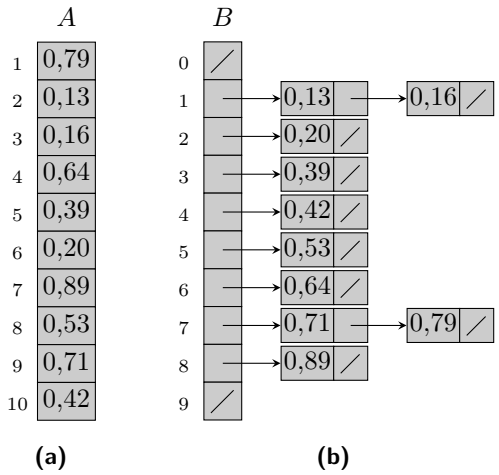
Najwięcej tymczasowych podtablic pozostawionych do późniejszego przetworzenia istnieje wówczas, gdy rozpoczyna się sortowanie ostatnich cyfr. W przypadku sortowania liczb trzycyfrowych jest łącznie 27 takich podtablic – 9 powstałych po posortowaniu najbardziej znaczących cyfr, 9 kolejnych powstałych po posortowaniu środkowych cyfr w obrębie pierwszego fragmentu i 9 takich, które czekają na posortowanie po najmniej znaczących cyfrach.

W ogólności do posortowania liczb d-cyfrowych tym algorytmem wymaganych jest co najwyżej $\sum_{i=0}^{d-1} 10^i = (10^d - 1)/9$ faz. Najwięcej tymczasowych fragmentów istnieje w momencie rozpoczęcia sortowania po najmniej znaczącej cyfrze – jest ich wtedy maksymalnie $9d$.

8.4. Sortowanie kubełkowe

8.4-1. Na rys. 18 zostało przedstawione działanie sortowania kubełkowego dla tablicy A.

8.4-2. Pesymistyczny przypadek dla algorytmu sortowania kubełkowego zachodzi, gdy wszystkie elementy z wejściowej tablicy trafią do tego samego kubełka. Mamy wtedy do posortowania



Rysunek 18: Działanie procedury BUCKET-SORT dla tablicy $A = \langle 0{,}79,\ 0{,}13,\ 0{,}16,\ 0{,}64,\ 0{,}39,\ 0{,}20,\ 0{,}89,\ 0{,}53,\ 0{,}71,\ 0{,}42\rangle$. **(a)** Wejściowa tablica A. **(b)** Tablica B zawierająca posortowane listy (kubełki) po wykonaniu pętli w wierszach 4–5.

pojedynczą listę o długości n, co zajmuje czas $O(n^2)$.

W celu uporządkowania kubełków, zamiast sortowania przez wstawianie, można użyć algorytmu sortującego, który wykonuje $O(n \lg n)$ operacji w przypadku pesymistycznym, np. sortowanie przez scalanie. Aby obliczyć średni czas działania sortowania kubełkowego w takim wariancie, podstawiamy do wzoru (8.1) ograniczenie na czas działania sortowania przez scalanie:

$$\mathrm{E}(T(n)) = \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}(n_i \lg n_i)) \leqslant O(n) + \sum_{i=0}^{n-1} O(\mathrm{E}(n_i^2)) = O(n).$$

Ponadto pętla **for** w wierszach 2–3 zajmuje czas liniowy, skąd mamy $\mathrm{E}(T(n)) = \Omega(n)$. A zatem oczekiwany czas działania sortowania kubełkowego po modyfikacji pozostaje liniowy.

8.4-3. Szanse nieuzyskania orła w dwóch rzutach monetą wynoszą $1/4$, co jest równe szansie uzyskania dwóch orłów w dwóch rzutach. Dokładnie jednego orła można uzyskać z prawdopodobieństwem równym $1/2$. Na podstawie tych wartości obliczamy:

$$\mathrm{E}(X^2) = 0^2 \cdot \Pr(X = 0) + 1^2 \cdot \Pr(X = 1) + 2^2 \cdot \Pr(X = 2) = 3/2,$$
$$\mathrm{E}^2(X) = \big(0 \cdot \Pr(X = 0) + 1 \cdot \Pr(X = 1) + 2 \cdot \Pr(X = 2)\big)^2 = 1.$$

8.4-4. Algorytm będzie opierał się na pomyśle z sortowania kubełkowego z n kubełkami. Podzielimy koło jednostkowe na n obszarów o równych polach reprezentujących przedziały odległości od środka koła i z każdym takim obszarem skojarzymy inny kubełek. Dzięki temu punkty będą umieszczane w poszczególnych kubełkach z jednakowym prawdopodobieństwem. Łatwo zauważyć, że obszary te będą pierścieniami kołowymi (z wyjątkiem jednego, który będzie kołem) o jednakowych powierzchniach równych π/n. Pozostaje tylko wyznaczyć ich promienie.

Ponumerujmy obszary (kubełki) kolejnymi liczbami całkowitymi od 0 do $n - 1$ w kolejności od środka do brzegu koła jednostkowego i oznaczmy przez r_j długość promienia wewnętrznego pierścienia kołowego stanowiącego j-ty obszar. Obszar zerowy jest kołem o polu π/n, więc $r_1 =$

$\sqrt{1/n}$. Ponieważ suma tego koła z pierścieniem do niego przylegającym jest kołem o polu $2\pi/n$, to stąd mamy $r_2 = \sqrt{2/n}$. Rozumowanie przeprowadzamy dla pozostałych pierścieni, otrzymując $r_j = \sqrt{j/n}$ dla każdego $j = 1, 2, \ldots, n-1$. Przyjmijmy ponadto $r_0 = 0$ oraz $r_n = 1$.

Punkty sortowane są względem ich odległości od środka koła jednostkowego przy wykorzystaniu opisanych kubełków. Dla każdego punktu $p_i = \langle x_i, y_i \rangle$ obliczane jest $d_i = \sqrt{x_i^2 + y_i^2}$ i jeśli zachodzi $r_j < d_i \leqslant r_{j+1}$, to punkt umieszczany jest w kubełku o numerze j.

8.4-5. Niech X będzie zmienną losową o ciągłej dystrybuancie P_X. Definiujemy nową zmienną losową $Y = P_X(X)$ o dystrybuancie P_Y. Wówczas, dla $0 < y < 1$, mamy

$$P_Y(y) = \Pr(Y \leqslant y) = \Pr(P_X(X) \leqslant y) = \Pr(X \leqslant P_X^{-1}(y)) = P_X(P_X^{-1}(y)) = y.$$

W uzasadnieniu korzystamy z funkcji odwrotnej do dystrybuanty P_X, jednak ta ostatnia może nie być ściśle rosnąca i funkcja do niej odwrotna może nie być poprawnie zdefiniowana. Dlatego funkcję P_X^{-1} definiujemy tutaj jako

$$P_X^{-1}(y) = \inf\{\, x \in \mathbb{R} : P_X(x) \geqslant y \,\} \quad \text{dla } 0 < y < 1.$$

Pokazaliśmy, że P_Y jest identycznością na $(0, 1)$, a zatem Y jest zmienną losową o ciągłym rozkładzie jednostajnym w tym przedziale.

W naszym problemie dla każdej zmiennej losowej X_i o dystrybuancie P, gdzie $i = 1, 2, \ldots, n$, wyznaczamy nową zmienną losową $Y_i = P(X_i)$. Zgodnie z powyższym opisem każda nowa zmienna ma rozkład jednostajny w przedziale $(0, 1)$, możemy zatem posortować je, stosując sortowanie kubełkowe o n kubełkach. Jako wynik dostaniemy pewną permutację $\langle Y_{\pi(1)}, Y_{\pi(2)}, \ldots, Y_{\pi(n)} \rangle$. Rozwiązanie problemu stanowi wówczas ciąg $\langle X_{\pi(1)}, X_{\pi(2)}, \ldots, X_{\pi(n)} \rangle$.

Wyznaczenie nowego ciągu zmiennych losowych zabiera czas $\Theta(n)$, gdyż zakładamy, że wartości dystrybuanty P można obliczać w czasie stałym. Również sortowanie kubełkowe n elementów działa średnio w czasie $\Theta(n)$, zatem w średnim przypadku cała procedura zajmuje czas liniowy.

Problemy

8-1. Dolne ograniczenia na średni czas działania sortowania za pomocą porównań

(a) Podczas sortowania algorytmem A żadne dwie różne permutacje wejściowe nie prowadzą do tego samego liścia w drzewie T_A – jest w nim zatem co najmniej $n!$ liści. Ponieważ algorytm A jest deterministyczny, to dla każdej permutacji wejściowej osiągany jest zawsze ten sam liść, a więc w T_A jest co najwyżej $n!$ osiągalnych liści. Wynika stąd, że algorytm A może dotrzeć do dokładnie $n!$ liści. Ponieważ każda permutacja ma szanse pojawić się na wejściu z równym prawdopodobieństwem, to szanse na dotarcie do dowolnego osiągalnego liścia wynoszą $1/n!$. Pozostałe liście nigdy nie zostaną odwiedzone podczas działania algorytmu A.

(b) Oznaczmy przez $L(T)$ zbiór liści drzewa T, a przez $d_T(x)$ – głębokość węzła x w drzewie T. Ponieważ $k > 1$, to korzeń drzewa T nie jest jego liściem, więc $L(T) = L(LT) \cup L(RT)$. Głębokość każdego liścia w poddrzewie LT jest o 1 mniejsza niż głębokość tego samego liścia w drzewie T i analogicznie dla liści w poddrzewie RT. Mamy zatem

$$D(T) = \sum_{x \in L(T)} d_T(x)$$

$$
\begin{aligned}
&= \sum_{x \in L(LT)} d_T(x) + \sum_{x \in L(RT)} d_T(x) \\
&= \sum_{x \in L(LT)} (d_{LT}(x) + 1) + \sum_{x \in L(RT)} (d_{RT}(x) + 1) \\
&= \sum_{x \in L(LT)} d_{LT}(x) + \sum_{x \in L(RT)} d_{RT}(x) + \sum_{x \in L(T)} 1 \\
&= D(LT) + D(RT) + k.
\end{aligned}
$$

(c) *W tekście zadania wykorzystywane jest* $d(1)$ *mimo braku jego definicji – przyjmujemy zatem* $d(1) = 0$.

W celu udowodnienia wzoru z treści zadania pokażemy, że zachodzą nierówności

$$d(k) \leqslant \min_{1 \leqslant i \leqslant k-1} (d(i) + d(k-i) + k) \quad \text{oraz} \quad d(k) \geqslant \min_{1 \leqslant i \leqslant k-1} (d(i) + d(k-i) + k).$$

Ustalmy pewne i ze zbioru $\{1, 2, \ldots, k-1\}$. Niech LT będzie takim drzewem binarnym o i liściach, że $d(i) = D(LT)$ i niech RT będzie takim drzewem binarnym o $k-i$ liściach, że $d(k-i) = D(RT)$. Niech T będzie drzewem binarnym, w którym LT jest jego lewym poddrzewem, a RT – jego prawym poddrzewem. Korzystając z definicji $d(k)$ i wzoru z części (b), mamy

$$d(k) \leqslant D(T) = D(LT) + D(RT) + k = d(i) + d(k-i) + k.$$

Ponieważ otrzymana nierówność zachodzi dla każdego $i = 1, 2, \ldots, k-1$, to możemy napisać $d(k) \leqslant \min_{1 \leqslant i \leqslant k-1}(d(i) + d(k-i) + k)$.

Pokażemy teraz, że zachodzi także nierówność przeciwna. Ustalmy w tym celu drzewo binarne T o k liściach, dla którego $d(k) = D(T)$. Niech LT i RT będą poddrzewami drzewa T, odpowiednio, lewym i prawym. Niech i_0, gdzie $1 \leqslant i_0 \leqslant k-1$, będzie liczbą liści w drzewie LT, a $k - i_0$ – liczbą liści w drzewie RT. Wówczas

$$d(k) = D(T) = D(LT) + D(RT) + k \geqslant d(i_0) + d(k - i_0) + k \geqslant \min_{1 \leqslant i \leqslant k-1} (d(i) + d(k-i) + k).$$

Założyliśmy tutaj, że ani LT, ani RT nie są puste. Gdyby tak jednak było, to drugie poddrzewo T zawierałoby wszystkie k liści drzewa T i wówczas, na mocy punktu (b), funkcja D dla niego przyjęłaby wartość $D(T) - k$. To jednak przeczyłoby wyborowi T jako drzewa o k liściach z minimalną wartością funkcji D.

(d) W zad. 7.4-2 analizowana była funkcja $f(q) = q \lg q + (n-q-1)\lg(n-q-1)$. Zostało tam pokazane, że w przedziale $(0, n-1)$ funkcja f osiąga minimum dla $q = (n-1)/2$. Wystarczy zatem przyjąć $q = i$ oraz $n = k+1$, aby pokazać stwierdzenie z treści zadania. Minimalna wartość badanej funkcji wynosi $k \lg k - k$.

Pokażemy oszacowanie na $d(k)$ przez indukcję, zakładając w tym celu, że $d(i) \geqslant i \lg i$ dla każdego $i = 1, 2, \ldots, k-1$ i korzystając z powyższego rezultatu oraz z punktu (c):

$$
\begin{aligned}
d(k) &= \min_{1 \leqslant i \leqslant k-1} (d(i) + d(k-i) + k) \\
&\geqslant \min_{1 \leqslant i \leqslant k-1} (i \lg i + (k-i)\lg(k-i)) + k \\
&\geqslant k \lg k - k + k \\
&= k \lg k.
\end{aligned}
$$

Podstawa indukcji zachodzi trywialnie, bo $d(1) = 0 \geqslant 1 \lg 1$, a zatem $d(k) = \Omega(k \lg k)$.

(e) Zauważmy, że możemy usunąć wszystkie nieosiągalne węzły z drzewa T_A, ponieważ nie wpływają one na czas działania algorytmu A. A zatem, na mocy punktu (a), po ich usunięciu T_A zawiera dokładnie $n!$ liści. Wykorzystując definicję $d(k)$ oraz wynik z poprzedniej części, mamy

$$D(T_A) \geqslant d(n!) = \Omega(n! \lg(n!)).$$

$D(T_A)$ stanowi sumę głębokości liści drzewa decyzyjnego T_A, a głębokość każdego liścia jest proporcjonalna do czasu działania algorytmu A dla permutacji reprezentowanej przez ten liść. Ponieważ prawdopodobieństwo osiągnięcia dowolnego liścia drzewa T_A jest równe $1/n!$, to oczekiwany czas algorytmu A wynosi

$$\frac{D(T_A)}{n!} = \frac{\Omega(n! \lg(n!))}{n!} = \Omega(\lg(n!)) = \Omega(n \lg n).$$

(f) Niech B będzie dowolnym randomizowanym algorytmem sortującym za pomocą porównań, a T_B – jego drzewem decyzyjnym. Istnieje wówczas deterministyczny algorytm A sortujący za pomocą porównań, którego drzewo decyzyjne T_A powstaje z drzewa T_B poprzez zastąpienie każdego węzła zrandomizowanego minimalnym poddrzewem o korzeniu będącym synem tegoż węzła zrandomizowanego. Przez minimalne poddrzewo rozumiemy tutaj takie, które posiada minimalną średnią odległość od swojego korzenia od liścia. Algorytm A wybiera zatem najlepszą możliwość zawsze wtedy, gdy algorytm B dokonuje losowego wyboru spośród r możliwości. Dlatego w średnim przypadku A wykonuje co najwyżej tyle porównań, co B.

8-2. Sortowanie w miejscu w czasie liniowym

(a) Sortowanie przez zliczanie dla $k = 1$.

(b) Poniższy pseudokod implementuje algorytm o zadanych własnościach:

```
BITWISE-SORT(A)
1  n ← length[A]
2  i ← 1
3  j ← n
4  while i < j
5        do zamień A[i] ↔ A[j]
6           while i ⩽ n i A[i] = 0
7                 do i ← i + 1
8           while j ⩾ 1 i A[j] = 1
9                 do j ← j − 1
```

(c) Sortowanie przez wstawianie.

(d) Do posortowania n rekordów b-bitowych w czasie $O(bn)$ za pomocą algorytmu sortowania pozycyjnego, potrzebna jest pomocnicza procedura, która sortuje stabilnie i działa w czasie $O(n)$. Takie warunki spełnia jedynie algorytm z części (a). W każdej kolejnej fazie sortowania rekordy powinny być porządkowane względem coraz bardziej znaczących bitów.

(e) W algorytmie wykorzystamy tablicę pomocniczą $C[0\,..\,k]$ tworzoną identycznie jak w wierszach 1–7 procedury COUNTING-SORT. Ideą algorytmu jest przejrzenie tablicy wejściowej w poszukiwaniu elementów, które należą do niewłaściwych obszarów tablicy, a następnie umieszczenie

ich na właściwych pozycjach, które znajdujemy, korzystając z informacji zebranych w tablicy C na początku działania algorytmu.

COUNTING-SORT-IN-PLACE(A, k)

```
1  utwórz tablicę C[0..k] jak w wierszach 1–7 procedury COUNTING-SORT
2  skopiuj zawartość tablicy C do tablicy C'
3  i ← 1
4  while i ⩽ length[A] − 1
5        do key ← A[i]
6           if C'[key − 1] < i ⩽ C'[key]
7              then i ← i + 1
8              else zamień A[i] ↔ A[C[key]]
9                   C[key] ← C[key] − 1
```

Początkowy stan tablicy C zostaje zapamiętany w tablicy C'. Następnie pętla **while** w wierszach 4–9 przegląda tablicę wejściową w poszukiwaniu elementów, które nie powinny znajdować się w posortowanej tablicy na aktualnej pozycji. Do sprawdzenia, czy element znajduje się we właściwym miejscu, używany jest test z wiersza 6. W przypadku negatywnego wyniku tego testu element z aktualnej pozycji zostaje przeniesiony na swoją docelową pozycję po zamianie z elementem, który tam się znajduje. Ten ostatni będzie sprawdzany w następnej iteracji pętli **while**.

Algorytm sortuje tablicę wejściową w miejscu (niekoniecznie stabilnie), wykorzystując $O(k)$ dodatkowej pamięci. Czas potrzebny na utworzenie tablic C i C' wynosi $O(n + k)$. Zauważmy, że element, który zostanie umieszczony na swojej docelowej pozycji, nie zmieni jej aż do zakończenia działania algorytmu – zostanie więc wykonanych co najwyżej $n - 1$ zamian w wierszu 8. W iteracjach pętli **while**, w których zamiany nie są dokonywane, jest inkrementowana zmienna i – sumaryczna liczba iteracji jest więc ograniczona przez $2(n - 1)$. Stąd czas działania algorytmu wynosi $O(n + k)$.

8-3. Sortowanie obiektów zmiennej długości

(a) Opiszemy sposób sortowania tylko dla liczb nieujemnych. Jeśli w tablicy znajdują się liczby ujemne, to wystarczy posortować je osobno analogicznym sposobem do opisanego poniżej, a następnie scalić z uporządkowaną tablicą liczb nieujemnych.

Sortowanie odbywa się w dwóch fazach. Najpierw liczby porządkowane są według ilości cyfr, z których się składają – im większa ilość cyfr, tym liczba jest większa. Wykorzystywany jest w tym celu algorytm sortowania przez zliczanie, w którym $k = n$. Następnie liczby o tej samej ilości cyfr sortowane są pozycyjnie.

W tablicy wejściowej może znaleźć się maksymalnie n liczb, zatem pierwsza część algorytmu zajmuje czas $O(n)$. Wyznaczmy teraz czas działania drugiej części. W tym celu oznaczmy przez m_i ilość liczb o i cyfrach w tablicy. Zachodzi wówczas $\sum_{i=1}^{n} im_i = n$. Sortowanie pozycyjne podtablicy liczb o i cyfrach zajmuje czas $\Theta(im_i)$, a więc całkowity czas wymagany przez drugą fazę algorytmu wynosi

$$\sum_{i=1}^{n} \Theta(im_i) = \Theta\left(\sum_{i=1}^{n} im_i\right) = \Theta(n).$$

Stąd oczywiście wynika, że algorytm działa w czasie $O(n)$.

(b) Skorzystajmy z obserwacji, że jeśli pierwsza litera napisu x jest leksykograficznie mniejsza od pierwszej litery napisu y, to x wystąpi w wynikowej tablicy przed y. Można zatem sortować

napisy przez zliczanie według ich pierwszych liter, a następnie, w obrębie każdej grupy napisów o takiej samej literze początkowej, sortować napisy według ich drugiej litery itd. Pamiętajmy jednak, że napisy mają różną długość, dlatego po ich posortowaniu względem i-tych liter, należy wyłączyć z kolejnej fazy słowa o dokładnie i literach, umieszczając je w tablicy wynikowej przed grupą napisów, które będą sortowane w kolejnej fazie.

Zauważmy, że napis x składający się z i liter będzie brał udział w co najwyżej i fazach sortowania. Niech m_i oznacza liczbę napisów na wejściu posiadających dokładnie i liter. Wówczas operacje odpowiedzialne za sortowanie napisów i-literowych wymagają czasu $O(im_i)$. Korzystając z faktu, że $\sum_{i=1}^{n} im_i = n$, otrzymujemy, że czasem działania algorytmu jest

$$\sum_{i=1}^{n} O(im_i) = O\left(\sum_{i=1}^{n} im_i\right) = O(n).$$

8-4. Dzbanki

(a) Dla każdego czerwonego dzbanka wystarczy przejrzeć wszystkie dotychczas niesparowane niebieskie dzbanki. Po znalezieniu pasującego możemy wyłączyć go z poszukiwań dla kolejnych czerwonych dzbanków. Łatwo sprawdzić, że ta metoda pozwala na pogrupowanie wszystkich dzbanków w pary za pomocą $\Theta(n^2)$ porównań.

(b) Ponumerujmy czerwone dzbanki kolejnymi liczbami całkowitymi od 1 do n i analogicznie dzbanki niebieskie. Problem sprowadza się do znalezienia takiej permutacji π niebieskich dzbanków, w której i-ty dzbanek czerwony jest tej samej pojemności, co dzbanek niebieski o numerze $\pi(i)$.

Zilustrujemy za pomocą drzewa decyzyjnego działanie algorytmu wyznaczającego tę permutację. Każdy węzeł w tym drzewie będzie odpowiadać porównaniu pewnego dzbanka czerwonego z pewnym dzbankiem niebieskim. Wynikiem każdego takiego porównania jest jedna z trzech możliwości: dzbanek czerwony ma mniejszą pojemność niż dzbanek niebieski, dzbanek czerwony ma większą pojemność niż dzbanek niebieski albo oba dzbanki mają tę samą pojemność. Z tego powodu drzewo decyzyjne jest drzewem 3-arnym o $n!$ osiągalnych liściach, bo tyle jest permutacji niebieskich dzbanków. Wysokość h tego drzewa oznacza pesymistyczną liczbę porównań wykonywanych przez algorytm. Każdy węzeł w tym drzewie ma co najwyżej 3 synów, prawdziwa jest zatem nierówność $3^h \geqslant n!$, skąd

$$h \geqslant \log_3(n!) = \frac{\lg(n!)}{\lg 3} = \Omega(n \lg n).$$

A zatem dowolny algorytm rozwiązujący ten problem wykonuje co najmniej $\Omega(n \lg n)$ porównań.

(c) Rozwiązanie tego problemu będzie opierać się na idei algorytmu quicksort. Zauważmy, że posortowanie obu ciągów dzbanków względem ich pojemności natychmiast prowadzi do rozwiązania problemu – grupujemy wówczas w pary dzbanek czerwony z dzbankiem niebieskim stojące na tych samych pozycjach w swoich ciągach. Niech R będzie tablicą dzbanków czerwonych, a B tablicą dzbanków niebieskich. Poniższa procedura stanowi adaptację algorytmu quicksort do omawianego problemu.

```
JUGS-MATCH(R, B, p, r)
1  if p < r
2     then q ← JUGS-PARTITION(R, B, p, r)
3          JUGS-MATCH(R, B, p, q − 1)
4          JUGS-MATCH(R, B, q + 1, r)
```

Poniżej znajduje się zmodyfikowana procedura RANDOMIZED-PARTITION. Przyjmuje ona dwie tablice R i B, z których jedna jest permutacją drugiej i w każdej z nich wszystkie elementy są parami różne. Przyjmujemy, że operacja porównania dwóch dzbanków jest w rzeczywistości porównaniem ich pojemności. Procedura dokonuje podziału obu tablic na podstawie losowo wybranego elementu rozdzielającego, przy czym nie porównuje elementów z tej samej tablicy. Zwracanym wynikiem jest pozycja elementu rozdzielającego (która jest identyczna w obu wynikowych tablicach).

```
JUGS-PARTITION(R, B, p, r)
 1  zamień R[r] ↔ R[RANDOM(p, r)]
 2  x ← R[r]
 3  i ← p
 4  while B[i] ≠ x
 5        do i ← i + 1
 6  zamień B[i] ↔ B[r]
 7  i ← p − 1
 8  for j ← p to r − 1
 9        do if B[j] < x
10              then i ← i + 1
11                   zamień B[i] ↔ B[j]
12  zamień B[i + 1] ↔ B[r]
13  x ← B[i + 1]
14  powtórz kroki z wierszy 7–12 dla tablicy R
15  return i + 1
```

Procedura wybiera najpierw losowo element rozdzielający x z tablicy $R[p\,..\,r]$. Elementy równe x są dla uproszczenia późniejszych kroków przenoszone na ostatnie pozycje swoich tablic. Następnie pętla **for** z wierszy 8–11 dokonuje podziału tablicy $B[p\,..\,r]$ względem x. Po zakończeniu pętli $B[r] = x$, podtablica $B[p\,..\,i]$ zawiera elementy mniejsze niż x, a podtablica $B[i+1\,..\,r-1]$ – elementy większe niż x. Wystarczy jeszcze zamienić element rozdzielający z $B[i+1]$, aby zakończyć podział tablicy B. W celu przeprowadzenia podziału tablicy R algorytm wykonuje analogiczne kroki z wierszy 7–12, uprzednio nadając zmiennej x wartość dotychczasowego elementu rozdzielającego, ale pobranego z tablicy B.

Udowodnimy teraz, że oczekiwana liczba porównań wykonywanych przez algorytm JUGS-MATCH wynosi $O(n \lg n)$. Nasza analiza będzie opierać się na analizie średniego przypadku algorytmu quicksort. Niech $\langle r_1, r_2, \ldots, r_n \rangle$ będzie ciągiem dzbanków czerwonych uporządkowanym rosnąco, a $\langle b_1, b_2, \ldots, b_n \rangle$ – rosnącym ciągiem dzbanków niebieskich. Definiujemy zmienną losową wskaźnikową

$$X_{ij} = \mathrm{I}(r_i \text{ jest porównywane z } b_j).$$

Zauważmy, że dane elementy r_i i b_j są porównywane ze sobą co najwyżej raz. Po porównaniu r_i z elementami fragmentu B (w tym być może z b_j) r_i jest umieszczane na właściwym miejscu w tablicy R i nie uczestniczy w późniejszych wywołaniach procedury JUGS-PARTITION. Analogiczne rozumowanie stosuje się do b_j.

Z powyższej obserwacji wynika, że całkowita liczba porównań wykonywanych w algorytmie wynosi

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

Biorąc wartości oczekiwane obu stron, dostajemy

$$\mathrm{E}(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(r_i \text{ jest porównywane z } b_j).$$

Pozostaje teraz tylko obliczyć $\Pr(r_i$ jest porównywane z $b_j)$. Jeśli jako element rozdzielający wybrane zostanie x takie, że $r_i < x < b_j$, to r_i zostanie umieszczone w lewej części podziału podtablicy R, a b_j – w prawej części podziału podtablicy B. A więc elementy r_i i b_j nigdy więcej nie będą porównywane. Oznaczmy przez R_{ij} zbiór $\{r_i, r_{i+1}, \ldots, r_j\}$. Wówczas elementy r_i i b_j są porównywane wtedy i tylko wtedy, gdy pierwszym elementem wybranym jako rozdzielający ze zbioru R_{ij} jest r_i albo r_j. To, że zostanie nim dowolny ustalony element zbioru R_{ij}, zachodzi z jednakowym prawdopodobieństwem i na mocy faktu, że $|R_{ij}| = j - i + 1$, szanse na zajście tego zdarzenia wynoszą $1/(j - i + 1)$. Mamy zatem

$$\begin{aligned}
\Pr(r_i \text{ jest porównywane z } b_j) &= \Pr(r_i \text{ lub } r_j \text{ jest pierwszym elementem rozdzielającym z } R_{ij}) \\
&= \Pr(r_i \text{ jest pierwszym elementem rozdzielającym z } R_{ij}) \\
&\quad + \Pr(r_j \text{ jest pierwszym elementem rozdzielającym z } R_{ij}) \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1}.
\end{aligned}$$

Wstawiając obliczony wynik do wzoru na $\mathrm{E}(X)$ i ograniczając tę wartość od góry, dostajemy $\mathrm{E}(X) = O(n \lg n)$, co stanowi oczekiwaną liczbę porównań wykonywaną przez algorytm JugsMatch.

Pesymistyczny przypadek zachodzi wtedy, gdy na każdym poziomie rekursji elementy rozdzielające są wybierane tak, że tworzą się podziały najbardziej niezrównoważone, czyli z tablicy o n elementach zostaje utworzona w wyniku podziału podtablica o $n-1$ elementach i podtablica pusta. Wówczas algorytm wykonuje $O(n^2)$ porównań.

8-5. Sortowanie względem średnich

(a) Zgodnie z definicją tablica $A[1\,..\,n]$ jest 1-posortowana, jeśli dla każdego $i = 1, 2, \ldots, n-1$ zachodzi $A[i] \leqslant A[i+1]$, a to jest równoważne temu, że tablica A jest posortowana niemalejąco.

(b) Jedną z takich permutacji jest $\langle 2, 1, 5, 3, 7, 4, 8, 6, 10, 9\rangle$.

(c) Aby udowodnić ten fakt, wystarczy nierówność

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leqslant \frac{\sum_{j=i+1}^{i+k} A[j]}{k}$$

pomnożyć przez k i zredukować te same składniki po obu stronach znaku nierówności.

(d) Jednym ze sposobów k-posortowania tablicy $A[1\,..\,n]$ jest zastosowanie algorytmu quicksort, którego rekursja zatrzymuje się dla podtablic o rozmiarach nieprzekraczających k. Dokładniej, należy zamienić warunek z wiersza 1 procedury QUICKSORT na następujący:

1 **if** $p + k - 1 < r$

Po zakończeniu działania algorytmu mamy, że $A[i] \leqslant A[i + k]$ dla każdego $i = 1, 2, \ldots, n - k$. Warunek ten, na podstawie części (c), jest równoważny temu, że tablica A jest k-posortowana.

W zad. 7.4-5 pokazaliśmy, że quicksort po takiej modyfikacji działa w oczekiwanym czasie $O(n \lg(n/k))$. Aby było to oszacowanie na czas w przypadku pesymistycznym, musimy zagwarantować najlepszy przypadek wyboru elementów rozdzielających. Można to zrealizować poprzez wybieranie do tej roli mediany bieżącej podtablicy, jak to opisano w zad. 9.3-3.

(e) W k-posortowanej tablicy każdy z ciągów postaci $\langle A[j], A[j+k], A[j+2k], \ldots, A[j+m_j k]\rangle$, gdzie $j = 1, 2 \ldots, k$ i $j + m_j k \leqslant n$, jest uporządkowany niemalejąco i wszystkie one zawierają łącznie n elementów. Wystarczy więc scalić je w jedną posortowaną tablicę, co na podstawie zad. 6.5-8 można wykonać w czasie $O(n \lg k)$.

(f) Przyjmiemy dla wygody, że n jest podzielne przez k – nie spowoduje to zmniejszenia ogólności naszej analizy.

W dowodzie dolnego ograniczenia czasowego tego problemu wykorzystamy model drzew decyzyjnych. Niech T będzie drzewem decyzyjnym pewnego algorytmu k-sortowania za pomocą porównań tablicy o n elementach. Drzewo T jest podobne do drzewa decyzyjnego algorytmu zwykłego sortowania za pomocą porównań – każdy jego liść odpowiada pewnej permutacji tablicy wejściowej. Jednak liczba osiągalnych liści jest na ogół mniejsza niż w drzewie zwykłego sortowania. Wynikowa tablica składa się z n/k podtablic k-elementowych, w których uporządkowanie elementów nie ma znaczenia. Osiągalnych liści w tym drzewie jest zatem

$$\frac{n!}{(k!)^{n/k}}.$$

Niech h oznacza wysokość drzewa T. Liczba liści w T nie przekracza 2^h, więc

$$h \geqslant \lg \frac{n!}{(k!)^{n/k}} = \lg(n!) - (n/k)\lg(k!) = \Omega(n \lg n),$$

ponieważ traktujemy k jako stałą. Wyznaczone oszacowanie na h stanowi dolne ograniczenie na czas działania dowolnego algorytmu k-sortowania za pomocą porównań tablicy o n elementach.

8-6. Dolna granica dla scalania posortowanych list

(a) Spośród $2n$ różnych liczb możemy wybrać n z nich do pierwszej listy, a pozostałe n – do drugiej listy. Ponieważ każdy taki podział jednoznacznie determinuje parę posortowanych list i na odwrót, to liczba sposobów utworzenia tej pary list wynosi $\binom{2n}{n}$.

(b) Działanie algorytmu scalania list można przedstawić, korzystając z drzewa decyzyjnego, w którym każdy węzeł odpowiada jednemu porównaniu elementów list. Wszystkie elementy są różne, mamy więc tylko dwa możliwe wyniki każdego z porównań, a zatem drzewo decyzyjne jest drzewem binarnym. Na podstawie poprzedniego punktu mamy, że liczba możliwych układów dwóch n-elementowych list wejściowych, z połączenia których powstaje wynikowa lista o $2n$

elementach, wynosi $\binom{2n}{n}$. W drzewie decyzyjnym jest więc tyleż osiągalnych liści. Wyznaczając oszacowanie wysokości h tego drzewa, znajdziemy ograniczenie dla maksymalnej liczby porównań wykonanych przez algorytm scalania list. Mamy $2^h \geqslant \binom{2n}{n}$ i korzystając z zad. C.1-13, dostajemy

$$h \geqslant \lg \binom{2n}{n} = \lg\left(\frac{2^{2n}}{\sqrt{\pi n}}(1 + O(1/n))\right) \geqslant \lg \frac{2^{2n}}{\sqrt{\pi n}} = 2n - \frac{\lg \pi}{2} - \frac{\lg n}{2} = 2n - o(n).$$

(c) Niech $\langle a_1, a_2, \ldots, a_n\rangle$ oraz $\langle b_1, b_2, \ldots, b_n\rangle$ będą listami wejściowymi. Załóżmy, że w liście wynikowej element a_i sąsiaduje z elementem b_j dla $1 \leqslant i, j \leqslant n$. Jeśli algorytm, scalając obie listy wejściowe, porównywałby a_i z elementem b_k, gdzie $1 \leqslant k \leqslant j-1$, to wynikiem porównania byłoby $b_k < a_i$, ale stąd nie wynikałoby ani $b_j < a_i$, ani $a_i < b_j$. Analogicznie przy porównaniu a_i z elementem b_k, gdzie $j+1 \leqslant k \leqslant n$. A zatem, aby rozstrzygnąć, w jakiej kolejności powinny wystąpić a_i i b_j w liście wynikowej, algorytm musi porównać te dwa elementy ze sobą.

(d) Jeśli do wynikowej listy będą trafiać elementy z pierwszej listy na przemian z elementami z drugiej, to każde dwa sąsiednie elementy w wynikowej liście będą pochodzić z dwóch różnych list wejściowych. Takich par sąsiadujących elementów będzie w sumie $2n-1$. Na mocy poprzedniego punktu wnioskujemy, że w tym przypadku algorytm scalania list wykona tyleż porównań.

Rozdział **9**

Mediany i statystyki pozycyjne

9.1. Minimum i maksimum

9.1-1. Wyznaczmy najpierw μ – najmniejszą spośród n liczb – w następujący sposób. Łączymy liczby w pary i odrzucamy te, które są większe w swoich parach, po czym wykonujemy te operacje rekurencyjnie dla zbioru pozostawionych liczb, aż do uzyskania jednej liczby, którą oczywiście będzie μ. Przyjmujemy, że w razie nieparzystej liczby elementów na danym poziomie rekurencji, element bez pary nie jest porównywany z żadnym innym i zwyczajnie przechodzi do kolejnego etapu. Ponieważ na każdym poziomie z k liczb zostaje $\lceil k/2 \rceil$, to będzie $\lceil \lg n \rceil$ wywołań rekurencyjnych tej procedury i co najwyżej tylu testom będzie poddawany element μ. Zauważmy, że każdy test odrzuca jedną liczbę, wykonamy zatem dokładnie $n-1$ porównań.

Zastanówmy się teraz, która z pozostałych liczb może być drugą najmniejszą w zbiorze. Liczba ta została odrzucona po porównaniu jej z elementem μ, więc problem sprowadza się do wyznaczenia minimum zbioru tych liczb, które były testowane z μ. Na mocy wcześniejszej obserwacji mamy, że zbiór ten składa się z $\lceil \lg n \rceil$ elementów, więc wystarczy $\lceil \lg n \rceil - 1$ porównań do wyznaczenia jego minimum.

Ostatecznie dostajemy, że drugą najmniejszą spośród n liczb można wyznaczyć, wykonując $n + \lceil \lg n \rceil - 2$ porównań.

9.1-2. Zadanie rozwiążemy prostszą metodą, niż sugeruje nam to wskazówka.

Jeśli n jest parzyste, to zgodnie z podaną w Podręczniku informacją, wykonywanych jest $3n/2 - 2$ porównań. Ale dla parzystego n zachodzi $3n/2 = \lceil 3n/2 \rceil$, więc wzór na liczbę potrzebnych porównań przyjmuje postać $\lceil 3n/2 \rceil - 2$. Niech teraz n będzie liczbą nieparzystą, czyli $n = 2k+1$ dla pewnego całkowitego k. Chcemy wykazać, że koniecznych jest $\lceil 3n/2 \rceil - 2$ porównań, czyli $\lceil 3k + 3/2 \rceil - 2 = 3k + 2 - 2 = 3k$. Ale wynik ten zgadza się z opisanym w Podręczniku dolnym oszacowaniem na liczbę porównań dla nieparzystego n, bo $3\lfloor n/2 \rfloor = 3\lfloor k + 1/2 \rfloor = 3k$.

9.2. Wybór w oczekiwanym czasie liniowym

9.2-1. Zakładamy, że parametr i jest liczbą całkowitą spełniającą nierówności $1 \leqslant i \leqslant r - p + 1$. Wywołanie procedury RANDOMIZED-PARTITION w wierszu 3 zwraca liczbę całkowitą q taką, że $p \leqslant q \leqslant r$. Dla liczby k wyznaczonej w kolejnym wierszu zachodzi więc $1 \leqslant k \leqslant r - p + 1$. Wywołanie rekurencyjne w wierszu 8 nastąpi dla tablicy długości 0, jeśli $i < k$ i $q = p$, ale wówczas $k = 1$, co jest sprzeczne z założeniem o parametrze i. Podobnie w wierszu 9 funkcja zostanie wywołana rekurencyjnie dla pustej tablicy, o ile $i > k$ i $q = r$, lecz wtedy $k = r - p + 1$ i również w tym przypadku dochodzimy do sprzeczności.

9.2-2. Czas działania procedury RANDOMIZED-PARTITION dla tablicy o mniej niż n elementach jest niezależny od tego, jak została podzielona tablica o n elementach w poprzednim wywołaniu rekurencyjnym. Jest tak między innymi dlatego, że żaden poziom rekursji nie przekazuje do następnego poziomu informacji o tym, na jakim fragmencie tablicy działa.

9.2-3. Po dokonaniu oczywistych zmian w oryginalnej procedurze, otrzymujemy następujący pseudokod:

ITERATIVE-RANDOMIZED-SELECT(A, p, r, i)

```
1  while p < r
2        do q ← RANDOMIZED-PARTITION(A, p, r)
3           k ← q − p + 1
4           if i = k
5              then return A[q]
6           if i < k
7              then r ← q − 1
8              else  p ← q + 1
9                    i ← i − k
10 return A[p]
```

9.2-4. W przypadku szukania elementu najmniejszego pesymistyczny przypadek dzielenia podtablicy występuje, gdy na element rozdzielający wybierany jest za każdym razem jej największy element. Kolejne wywołania rekurencyjne zmniejszają wówczas obszar poszukiwań o 1, jednocześnie umieszczając na końcu tablicy elementy w kolejności rosnącej, w wyniku czego, jako efekt uboczny, tablica zostaje posortowana.

9.3. Wybór w pesymistycznym czasie liniowym

9.3-1. Dokonajmy analizy algorytmu SELECT w przypadku, gdy elementy będą dzielone na grupy 7-elementowe. Wówczas w co najmniej połowie spośród $\lceil n/7 \rceil$ grup są po 4 elementy większe od x, oprócz jednej grupy o mniej niż 7 elementach, jeśli n nie jest podzielne przez 7, i jednej grupy zawierającej sam element x. Odliczając te dwie grupy, wnioskujemy, że liczba elementów większych od x wynosi co najmniej

$$4\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{7}\right\rceil \right\rceil - 2\right) \geqslant \frac{2n}{7} - 8.$$

Podobnie wykazuje się, że liczba elementów mniejszych od x wynosi co najmniej $2n/7 - 8$. Stąd procedura wywoła się rekurencyjnie dla zbioru co najwyżej $(5n/7+8)$-elementowego. Rekurencja przyjmuje więc postać

$$T(n) \leqslant \begin{cases} \Theta(1), & \text{jeśli } n < d, \\ T(\lceil n/7 \rceil) + T(5n/7 + 8) + O(n), & \text{jeśli } n \geqslant d, \end{cases}$$

przy czym $d > 0$ jest pewną stałą, którą wyznaczymy później.

Wykażemy metodą przez podstawianie, że $T(n) = O(n)$. Zachodzi oczywiście $T(n) \leqslant cn$ dla pewnej stałej $c > 0$ oraz wszystkich $n < d$. Załóżmy teraz, że $n \geqslant d$ i że $T(k) \leqslant ck$ dla pewnej stałej $c > 0$ i wszystkich $k < n$. Dla pewnej stałej $a > 0$ zachodzi wówczas

$$T(n) \leqslant c\lceil n/7 \rceil + c(5n/7 + 8) + an$$

$$\begin{aligned}
&\leqslant cn/7 + c + 5cn/7 + 8c + an \\
&= 6cn/7 + 9c + an \\
&= cn + (-cn/7 + 9c + an) \\
&\leqslant cn,
\end{aligned}$$

o ile składnik $-cn/7 + 9c + an$ jest niedodatni. Dla $n > 63$ warunek ten jest spełniony, kiedy $c \geqslant 7a(n/(n-63))$. Jeśli z kolei $n \geqslant 126$, to $n/(n-63) \leqslant 2$ i wtedy musi być $c \geqslant 14a$. Można zatem przyjąć za d wartość 126, co kończy dowód.

Pokażemy teraz, że jeśli podział będzie dokonywany na grupy 3-elementowe, to czas działania tak zmodyfikowanego algorytmu SELECT będzie wyższy od liniowego. Rozważmy w szczególności przypadek, kiedy jest dokładnie $\left\lceil \frac{1}{2}\left\lceil \frac{n}{3} \right\rceil \right\rceil$ grup o medianach większych lub równych x, w tym ostatnia, niepełna grupa, która zawiera 2 elementy większe niż x. Stąd całkowita liczba elementów większych od x wynosi

$$2\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{3} \right\rceil \right\rceil - 1\right) + 1 = 2\left\lceil \frac{n}{6} \right\rceil - 1.$$

Procedura jest wywoływana rekurencyjne dla co najmniej $n-(2\lceil n/6\rceil-1) \geqslant n-(2(n/6+1)-1) = 2n/3-1$ elementów nieprzekraczających x. Na mocy faktu, że sortowanie elementów w kroku 2 algorytmu SELECT zabiera czas dokładnie $\Theta(n)$, otrzymujemy rekurencję opisującą czas działania algorytmu w tym przypadku:

$$T(n) \geqslant \begin{cases} \Theta(1), & \text{jeśli } n < d, \\ T(\lceil n/3 \rceil) + T(2n/3 - 1) + \Theta(n), & \text{jeśli } n \geqslant d, \end{cases}$$

gdzie $d > 0$ jest pewną stałą. Można wykazać, stosując metodę przez podstawianie, że rozwiązaniem powyższej rekurencji jest $T(n) = \Omega(n \lg n)$, co oznacza, że algorytm w tym wariancie nie działa w czasie liniowym.

9.3-2. Z analizy algorytmu SELECT wynika, że zarówno liczba elementów większych od x, jak i liczba elementów mniejszych od x, wynosi co najmniej $3n/10-6$. Na mocy założenia, że $n \geqslant 140$, mamy

$$\frac{3n}{10} - 6 - \left\lceil \frac{n}{4} \right\rceil \geqslant \frac{3n}{10} - 6 - \left(\frac{n}{4} + 1\right) = \frac{n - 140}{20} \geqslant 0,$$

skąd wnioskujemy, że obie badane liczby są nie mniejsze niż $\lceil n/4 \rceil$.

9.3-3. *W rozwiązaniu przyjmujemy założenie, że elementy tablicy wejściowej są parami różne.*

Wykorzystamy algorytm SELECT do znalezienia mediany x elementów z tablicy wejściowej. Przy okazji tablica wejściowa zostanie podzielona względem x. Można więc w algorytmie quicksort zastąpić wywołanie procedury PARTITION wywołaniem SELECT szukającym mediany. W rezultacie wykonywane będą najbardziej zrównoważone podziały i pesymistyczny czas algorytmu quicksort po takiej modyfikacji sprowadzi się do rekurencji $T(n) = 2T(\lfloor n/2 \rfloor) + O(n)$, której rozwiązaniem jest $T(n) = O(n \lg n)$.

9.3-4. Jedynym źródłem informacji o elementach we wspomnianym algorytmie wyznaczającym i-tą najmniejszą wartość są porównania. Ponadto tablica wejściowa zawierająca te elementy nie jest modyfikowana – po zakończeniu działania algorytmu ma ona identyczną zawartość jak na początku. Zakładamy, że wynik każdego porównania liczb (tzn. ustalenie relacji $\leqslant$ albo $>$) jest

zapamiętywany, więc przy następnej próbie sprawdzenia tych samych liczb wystarczy odczytać wynik z pamięci. W związku z tym pozostaje nam udowodnić, że porównania, które przeprowadzono i zapamiętano podczas działania algorytmu, aby jednoznacznie stwierdzić, który element jest i-tym najmniejszym, są wystarczające, by jednoznacznie wskazać $i-1$ najmniejszych oraz $n-i$ największych elementów.

Oznaczmy przez u element zwrócony przez algorytm. W rozwiązaniu będziemy korzystać z przechodniości relacji porządku liczb. Jeśli algorytm stwierdził, że $x \leqslant y$ oraz $y \leqslant z$, to wnioskujemy, że $x \leqslant z$. Wykażemy, że dzięki tej obserwacji możemy uzyskać informacje o wzajemnym porządku w każdej parze elementów ze zbioru wejściowego. Załóżmy nie-wprost, że istnieje para $\langle x, y\rangle$, przy czym nie potrafimy stwierdzić, czy $x \leqslant y$, czy $x > y$. Wówczas nie potrafimy także wskazać relacji między x i każdym elementem, z którym jesteśmy w stanie porównać y, i na odwrót. W szczególności istnieje element v, którego relacja z u nie jest nam znana. Ale to leży w sprzeczności z faktem, że u zostało jednoznacznie wskazane jako i-ty najmniejszy ze zbioru wejściowego – v może bowiem leżeć po jednej albo po drugiej stronie u, a to prowadzi do niejednoznaczności przy określaniu pozycji u w zbiorze wejściowym.

Mając informacje na temat każdej pary, możemy przejrzeć wyniki porównań elementu u z każdym innym i na tej podstawie klasyfikować elementy do jednego z dwóch zbiorów wynikowych.

9.3-5. Zmodyfikujemy procedurę RANDOMIZED-SELECT, dokonując zmiany w wierszu 3. Zamiast wywoływać RANDOMIZED-PARTITION, znajdziemy medianę x elementów z tablicy wejściowej za pomocą danej „czarnej skrzynki", po czym podzielimy tablicę względem x. W każdym wywołaniu rekurencyjnym będzie wówczas dokonywany najbardziej zrównoważony podział, więc czas działania algorytmu wyboru w przypadku pesymistycznym przyjmie postać $T(n) = T(\lfloor n/2\rfloor) + O(n)$. Rozwiązaniem tej rekurencji jest $T(n) = O(n)$.

9.3-6. Oznaczmy przez S badany zbiór, a przez $Q_k(S)$ – zbiór jego kwantyli rzędu k. Dopuszczalnymi wartościami parametru k są liczby całkowite od 1 do $n+1$, gdzie $n = |S|$. Zbiór kwantyli nie jest jednoznacznie zdefiniowany, opiszemy jednak jeden z poprawnych sposobów jego konstrukcji. Oczywiście $Q_1(S) = \emptyset$. Jeśli k jest parzyste, to $Q_k(S)$ składa się z mediany m zbioru S oraz kwantyli rzędu $k/2$ zbiorów $\{\, x \in S : x < m \,\}$ i $\{\, x \in S : x > m \,\}$. W przypadku, gdy k jest liczbą nieparzystą i większą od 1, musimy wyznaczyć te kwantyle rzędu k, które znajdują się najbliżej mediany m i są od niej różne. Oznaczmy je przez m_1 i m_2. Wówczas

$$Q_k(S) = Q_{(k-1)/2}(\{\, x \in S : x < m_1 \,\}) \cup \{m_1, m_2\} \cup Q_{(k-1)/2}(\{\, x \in S : x > m_2 \,\}).$$

Zbiór S reprezentujemy jako tablicę A parami różnych elementów. Na podstawie powyższego opisu otrzymujemy następujący algorytm oparty o metodę „dziel i zwyciężaj":

```
QUANTILES(A, p, r, k)
 1  if k = 1
 2     then return ∅
 3  n ← r − p + 1
 4  q1 ← p + ⌊⌊k/2⌋(n/k)⌋     ▷ pozycje zajmowane przez ⌊k/2⌋-ty i ⌈k/2⌉-ty kwantyl rzędu k
 5  q2 ← p + ⌊⌈k/2⌉(n/k)⌋        podtablicy A[p..r], gdyby została ona uporządkowana
 6  SELECT(A, p, r, q1 − p + 1)
 7  if q1 ≠ q2
 8     then SELECT(A, q1 + 1, r, q2 − q1)
 9  L ← QUANTILES(A, p, q1 − 1, ⌊k/2⌋)
10  R ← QUANTILES(A, q2 + 1, r, ⌊k/2⌋)
11  return L ∪ {A[q1], A[q2]} ∪ R
```

Aby wyznaczyć $Q_k(S)$, należy wywołać QUANTILES$(A, 1, n, k)$, gdzie $n = |S|$.

W pseudokodzie nie rozdzielamy przypadków explicite ze względu na parzystość k. Po wyznaczeniu długości przetwarzanego fragmentu $A[p\,..\,r]$ obliczane są pozycje q_1 i q_2 zajmowane przez kwantyle m_1 i m_2 (jeśli k jest nieparzyste) lub medianę m (jeśli k jest parzyste – wówczas $q_1 = q_2$), gdyby uporządkować podtablicę $A[p\,..\,r]$. Następnie korzystamy z algorytmu SELECT, aby podzielić tę podtablicę względem elementu m_1 lub m (w zależności od parzystości k). Przyjmujemy, że parametrami procedury SELECT są kolejno: tablica wejściowa, indeks początku przetwarzanego fragmentu tej tablicy, indeks końca tego fragmentu oraz numer statystyki pozycyjnej, którą zamierzamy odnaleźć w tym fragmencie. Jeśli k jest nieparzyste (czyli $q_1 \neq q_2$), to w wierszu 8 fragment tablicy A zawierający elementy większe niż m_1 dzielimy względem m_2. W tym momencie fragment $A[p\,..\,q_1 - 1]$ składa się z elementów mniejszych niż $A[q_1] = m_1$, fragment $A[q_2 + 1\,..\,r]$ – z elementów większych niż $A[q_2] = m_2$, natomiast elementy z $A[q_1 + 1\,..\,q_2 - 1]$ (o ile istnieją) są pomiędzy m_1 a m_2. Teraz wystarczy znaleźć kwantyle rzędu $\lfloor k/2 \rfloor$ w pierwszym i drugim fragmencie, wykorzystując wywołania rekurencyjne, po czym zwrócić wynikowy zbiór.

Nierekurencyjna część algorytmu zajmuje czas $O(n)$, zatem rekursja opisująca całkowity czas działania przyjmuje postać $T(n, k) \leqslant 2T(\lfloor n/2 \rfloor, \lfloor k/2 \rfloor) + O(n)$. Jej rozwiązaniem jest $T(n, k) = O(n \lg k)$, o czym można się przekonać, przeprowadzając analizę z wykorzystaniem metody drzewa rekursji.

9.3-7. Wyznaczmy medianę x zbioru S, a następnie dla każdego elementu z tego zbioru obliczmy jego odległość od x, czyli wartość bezwzględną z ich różnicy. Problem sprowadza się w tym momencie do znalezienia k najmniejszych odległości, co można zrealizować, szukając wśród nich k-tej statystyki pozycyjnej y, a następnie zwracając elementy odległe od x o nie więcej niż y.

Pojawiają się jednak dwie subtelności. O ile elementy wejściowe z założenia są parami różne, to odległości od mediany mogą się powtarzać – dokładniej, każda odległość może występować w co najwyżej dwóch egzemplarzach. Analiza czasu działania algorytmu SELECT przeprowadzona w Podręczniku pozostaje jednak prawdziwa, gdy rozszerzymy ją na przypadek powtarzających się elementów w tablicy wejściowej, dlatego oszacowanie na czas działania tego algorytmu nie zmienia się. Może się także zdarzyć, że zbiór elementów o odległościach nieprzekraczających y będzie mieć $k + 1$ elementów. Jest tak wówczas, gdy wśród wyznaczonych odległości są dwie kopie y. Przed zwróceniem wynikowego zbioru wystarczy więc usunąć z niego element $x + y$ albo $x - y$.

Poniższy algorytm implementuje opisane podejście. Przyjmuje on na wejściu n-elementową tablicę A zawierającą wszystkie elementy zbioru S oraz liczbę k.

```
MEDIAN-PROXIMITY(A, k)
 1  n ← length[A]
 2  x ← SELECT(A, 1, n, ⌊(n + 1)/2⌋)
 3  for i ← 1 to n
 4        do dist[i] ← |A[i] − x|
 5  y ← SELECT(dist, 1, n, k)
 6  M ← ∅
 7  for i ← 1 to n
 8        do if |A[i] − x| ⩽ y
 9              then M ← M ∪ {A[i]}
10  if |M| = k + 1
11     then M ← M \ {x + y}
12  return M
```

W czasie $O(n)$ wyznaczana jest mediana x zbioru S, jak również tablica *dist* zawierająca odległości poszczególnych elementów S od x oraz k-ta najmniejsza wartość y tej tablicy. W pozostałej części algorytmu budowany jest zbiór M złożony z k elementów najbliższych x. Zakładamy, że operacje inicjalizacji zbioru, pobierania jego rozmiaru i usuwania z niego pojedynczego elementu działają w czasie co najwyżej proporcjonalnym do rozmiaru tego zbioru oraz że dodanie do niego pojedynczego elementu zajmuje czas stały. Założenia te można łatwo spełnić, implementując zbiór M np. jako listę dwukierunkową (patrz podrozdział 10.2). Wówczas czasem działania algorytmu jest $O(n)$.

9.3-8. Niech m_X będzie medianą elementów z tablicy X i niech m_Y będzie medianą elementów z tablicy Y. Jeśli $m_X = m_Y$, to wśród wszystkich $2n$ elementów obu tablic co najmniej n jest większych (bądź równych) od obu median i co najmniej n jest od nich mniejszych (lub równych). Wartość m_X jest więc szukaną medianą wszystkich $2n$ liczb. Załóżmy teraz, że $m_X \neq m_Y$ i bez utraty ogólności, niech $m_X < m_Y$. Pomijając teraz około $n/2$ elementów X mniejszych lub równych m_X i około $n/2$ elementów Y większych lub równych m_Y, sprowadzamy problem do identycznego, ale o około połowę mniejszego, ponieważ wiadomo, że poszukiwana mediana znajduje się wśród pozostawionych elementów. Dokładniej, podproblem będzie operował na tablicach o rozmiarach $\lfloor n/2 \rfloor + 1$.

W celu wyznaczenia mediany $2n$ liczb nasz algorytm będzie wykorzystywał rekurencję o przypadku brzegowym, gdy $n \leqslant 2$. Jeśli $n = 1$, to jako wynik algorytmu wystarczy zwrócić mniejszy z dwóch wejściowych elementów (zgodnie z konwencją, że interesuje nas mediana dolna). W przypadku zaś, gdy $n = 2$, wyznaczamy większy z elementów znajdujących się w pierwszych komórkach tablic oraz mniejszy z elementów zajmujących drugie komórki. Łatwo sprawdzić, że jeden z nich to mediana dolna, a drugi to mediana górna czterech liczb wejściowych. Jako wynik podajemy zatem minimum z obu tych liczb.

Poniższy pseudokod implementuje opisany algorytm. Po sprawdzeniu warunku brzegowego obliczane są indeksy median dolnych elementów z każdej tablicy, jak również indeksy ich median górnych. Te ostatnie przekazywane są w wywołaniach rekurencyjnych w celu zapewnienia jednakowych rozmiarów obu tablic na kolejnym poziomie rekursji. Aby odnaleźć medianę $2n$ elementów z tablic $X[1\,..\,n]$ i $Y[1\,..\,n]$, wywołujemy TWO-ARRAYS-MEDIAN$(X, 1, n, Y, 1, n)$.

```
TWO-ARRAYS-MEDIAN(X, p_X, r_X, Y, p_Y, r_Y)
 1  if r_X − p_X ⩽ 1
 2     then return min(max(X[p_X], Y[p_Y]), min(X[r_X], Y[r_Y]))
 3  q_X ← ⌊(p_X + r_X)/2⌋
 4  q'_X ← ⌈(p_X + r_X)/2⌉
 5  q_Y ← ⌊(p_Y + r_Y)/2⌋
 6  q'_Y ← ⌈(p_Y + r_Y)/2⌉
 7  if X[q_X] = Y[q_Y]
 8     then return X[q_X]
 9  if X[q_X] < Y[q_Y]
10     then return TWO-ARRAYS-MEDIAN(X, q_X, r_X, Y, p_Y, q'_Y)
11     else return TWO-ARRAYS-MEDIAN(X, p_X, q'_X, Y, q_Y, r_Y)
```

Czas działania podanego algorytmu w przypadku pesymistycznym jest opisany przez rekurencję $T(n) = T(\lfloor n/2 \rfloor + 1) + \Theta(1)$, gdzie $n = r_X - p_X + 1$. Jej rozwiązaniem jest oczywiście $T(n) = O(\lg n)$.

9.3-9. Oznaczmy przez y_1 i y_2, odpowiednio, medianę dolną i górną współrzędnych y wszystkich wież. Umieścimy główny rurociąg na współrzędnej y równej y_r takiej, że $y_1 \leqslant y_r \leqslant y_2$ i zbadamy, jak będzie zmieniać się suma długości odnóg północ-południe s, gdy będziemy zmieniać jego położenie.

Wybierzmy y'_r, spełniające nierówności $y_1 \leqslant y'_r \leqslant y_2$, na współrzędną y nowego położenia głównego rurociągu. Niech $d = |y_r - y'_r|$. Jeśli $y_1 = y_2$, to $y'_r = y_r$, rozważmy więc przypadek przeciwny. Wtedy zarówno na północ, jak i na południe od głównego rurociągu znajduje się $n/2$ wież wiertniczych. Do jednych zbliżyliśmy się o odległość d, ale od pozostałych o tyle samo się oddaliliśmy. Widać stąd, że wybór dowolnego y'_r znajdującego się pomiędzy medianami nie zmienia sumy długości odnóg północ-południe.

Niech teraz y'_r będzie współrzędną y głównego rurociągu taką, że $y'_r < y_1$ lub $y'_r > y_2$. Podobnie jak poprzednio, niech $d = |y_r - y'_r|$. Gdy n jest parzyste, to po jednej stronie rurociągu mamy co najmniej $n/2+1$ wież, a po drugiej – co najwyżej $n/2-1$. Stąd otrzymujemy oszacowanie na nową sumę długości odnóg:

$$s' \geqslant s + d(n/2+1) - d(n/2-1) = s + 2d > s.$$

W przypadku, gdy n jest liczbą nieparzystą, rurociąg w nowym położeniu po jednej stronie ma co najmniej $(n+1)/2$ wież wiertniczych, a po drugiej stronie co najwyżej $(n-1)/2$. Nową sumę długości odnóg można więc ograniczyć od dołu:

$$s' \geqslant s + d(n+1)/2 - d(n-1)/2 = s + d > s.$$

A zatem, niezależnie od parzystości n, przesunięcie rurociągu poza obszar wyznaczony przez mediany y_1 i y_2, wiąże się z powiększeniem sumy długości odnóg północ-południe – dowolna wartość y_r pomiędzy tymi medianami, łącznie z nimi, jest optymalna. Problem sprowadza się zatem do wyznaczenia mediany współrzędnych y wież wiertniczych i można go rozwiązać w czasie liniowym względem n mimo braku założenia o tym, że elementy, spośród których wyznaczamy medianę, są parami różne (co zauważyliśmy w zad. 9.3-7).

Problemy

9-1. Sortowanie największych i elementów

(a) Liczby można posortować algorytmem sortowania przez scalanie, który w najgorszym przypadku potrzebuje czasu $\Theta(n \lg n)$. Wypisanie i największych liczb otrzymanej tablicy, poprzez zwyczajne przeglądnięcie jej i ostatnich elementów, zajmuje czas $\Theta(i)$. Całkowity czas algorytmu w przypadku pesymistycznym wynosi zatem $\Theta(n \lg n + i)$.

(b) Zbudowanie kopca typu max dla kolejki priorytetowej algorytmem Build-Max-Heap wymaga czasu $\Theta(n)$. Wykonanie kolejno i operacji Extract-Max na kopcu o co najwyżej n elementach wymaga czasu $i \cdot O(\lg n) = O(i \lg n)$. Z kolei połowa tych operacji będzie wykonywana na kopcu o co najmniej $n/2$ elementach. Zajmą one czas $(i/2) \cdot \Omega(\lg(n/2)) = \Omega(i \lg n)$ w najgorszym przypadku. Stąd mamy, że wszystkie operacje ekstrakcji zostaną wykonane w czasie $\Theta(i \lg n)$ w przypadku pesymistycznym, a zatem całkowitym czasem algorytmu jest $\Theta(n + i \lg n)$.

(c) Aby osiągnąć najlepszy czas w przypadku pesymistycznym, użyjemy procedury Select do znalezienia i-tej statystyki pozycyjnej. Jednocześnie tablica elementów wejściowych zostanie podzielona względem tejże statystyki. Sortowanie i największych liczb można wykonać algorytmem

sortowania przez scalanie, które w najgorszym przypadku zajmuje czas $\Theta(i \lg i)$. Stąd całkowity czas działania algorytmu wynosi $\Theta(n + i \lg i)$, co czyni go najbardziej efektywnym spośród wszystkich rozważanych algorytmów w niniejszym problemie.

9-2. Mediana ważona

Rozwiązanie korzysta z definicji mediany ważonej (dolnej) z tekstu oryginalnego. Jedyną różnicą w porównaniu z definicją z tłumaczenia jest to, że w oryginale suma wag elementów mniejszych niż mediana ważona x_k jest ostro mniejsza niż $1/2$.

(a) Dla tak przyjętych wag elementów mamy

$$\sum_{x_i<x_k} w_i = \frac{k-1}{n} \quad \text{oraz} \quad \sum_{x_i>x_k} w_i = \frac{n-k}{n}.$$

Jeśli ograniczymy obie sumy od góry przez $1/2$ – pierwszą ostro, drugą nieostro – to dostaniemy, że medianą ważoną elementów $x_1, x_2, \ldots, x_n$ jest x_k, gdzie $n/2 \leqslant k < n/2 + 1$. Ponieważ k jest całkowite, to musi być $k = \lfloor (n+1)/2 \rfloor$. A zatem x_k jest również zwykłą medianą (dolną) elementów $x_1, x_2, \ldots, x_n$.

(b) Po posortowaniu elementów wyznaczenie mediany ważonej odbywa się w prosty sposób – należy przeglądać elementy w kolejności rosnącej i sumować ich wagi aż do momentu, kiedy suma wag osiągnie lub przekroczy $1/2$. Wówczas wystarczy zwrócić ostatnio przeglądany element.

Złożoność tej procedury zależy od efektywności sortowania, ale używając odpowiedniego algorytmu, jesteśmy w stanie osiągnąć czas $O(n \lg n)$ w pesymistycznym przypadku.

(c) Rozwiążemy problem metodą „dziel i zwyciężaj". Przyjmijmy, że elementy przechowywane są w tablicy $A[1..n]$, a ich wagi w tablicy $w[1..n]$, przy czym $w[i]$ jest wagą elementu $A[i]$ dla $i = 1, 2, \ldots, n$. Najpierw dzielimy elementy względem ich mediany algorytmem SELECT, w którym dodatkowo przy każdej zmianie pozycji elementu $A[i]$ na indeks j w tablicy A, będziemy także zmieniać pozycję wagi $w[i]$ na indeks j w tablicy w. Następnie wyznaczamy W_L i W_R – sumy wag elementów, odpowiednio, mniejszych i większych od znalezionej mediany. Jeśli $W_L < 1/2$ i $W_R < 1/2$, to medianą ważoną jest zwykła mediana. W przeciwnym przypadku obszar tablicy wejściowej, który zawiera medianę wraz z elementami o większej sumie wag, przekazujemy do następnego wywołania rekurencyjnego.

```
WEIGHTED-MEDIAN(A, w, p, r)
 1  if r − p + 1 ⩽ 2
 2     then if w[p] ⩾ w[r]
 3             then return A[p]
 4             else  return A[r]
 5  podziel elementy w A[p..r] i ich wagi względem mediany tablicy A[p..r]
 6  q ← ⌊(p + r)/2⌋
 7  W_L ← 0
 8  for i ← p to q − 1
 9        do W_L ← W_L + w[i]
10  W_R ← 1 − W_L − w[q]
11  if W_L < 1/2 i W_R < 1/2
12     then return A[q]
13  if W_L ⩾ 1/2
14     then w[q] ← w[q] + W_R
15          return WEIGHTED-MEDIAN(A, w, p, q)
16     else  w[q] ← w[q] + W_L
17          return WEIGHTED-MEDIAN(A, w, q, r)
```

W powyższym pseudokodzie rozważana jest tablica $A[p\,..\,r]$ o $n = r - p + 1$ elementach. Pierwszym krokiem jest sprawdzenie przypadków brzegowych – jeśli $n = 1$, to wystarczy zwrócić jedyny element wejściowy, a jeśli $n = 2$, to zwracany jest element o większej wadze. Działanie to implementujemy za pomocą pojedynczej instrukcji **if** w wierszach 1–4. Następnie elementy wraz z ich wagami są dzielone względem mediany (która po wykonaniu wiersza 5 znajduje się w A na pozycji $q = \lfloor (p+r)/2 \rfloor$) i obliczane są sumy wag obu części tego podziału. W zależności od tego, czy jedna z tych wartości wynosi co najmniej $1/2$, zwracana jest wyznaczona wcześniej mediana, bądź algorytm zostaje wywoływany rekurencyjnie dla odpowiedniej części podziału. Aby zachować warunek mówiący o tym, że suma wag wszystkich elementów tablicy wejściowej wynosi 1, przed kolejnym wywołaniem algorytmu zwiększamy wagę mediany do odpowiedniej wartości – $w[q]$ „kumuluje" więc wagi tej części tablicy, którą odrzucamy.

Przy założeniu, że wiersz 5 w najgorszym przypadku zajmuje czas $\Theta(n)$, pesymistyczny czas działania opisanego algorytmu spełnia równanie rekurencyjne $T(n) = T(\lfloor n/2 \rfloor + 1) + \Theta(n)$, którego rozwiązaniem jest $T(n) = \Theta(n)$.

(d) Niech p_k będzie medianą ważoną danych punktów $p_1, p_2, \ldots, p_n$ (które są liczbami rzeczywistymi) i ich wag $w_1, w_2, \ldots, w_n$. Dla dowolnego punktu p niech $f(p) = \sum_{i=1}^{n} w_i |p - p_i|$. Szukamy takiego punktu p spośród danych, dla którego $f(p)$ przyjmuje możliwie najmniejszą wartość. Pokażemy, że szukanym punktem jest p_k.

Niech p_l będzie dowolnym punktem różnym od p_k. Zbadamy znak różnicy

$$f(p_l) - f(p_k) = \sum_{i=1}^{n} w_i |p_l - p_i| - \sum_{i=1}^{n} w_i |p_k - p_i| = \sum_{i=1}^{n} w_i (|p_l - p_i| - |p_k - p_i|).$$

Niech $p_l < p_k$. Zauważmy, że w przypadku, gdy $p_l \leqslant p_i < p_k$, zachodzi $|p_l - p_i| - |p_k - p_i| = p_i - p_l - p_k + p_i \geqslant 2p_l - p_l - p_k = p_l - p_k$. Mamy zatem

$$\begin{aligned} f(p_l) - f(p_k) = &\sum_{p_i < p_l} w_i (p_l - p_i - p_k + p_i) \\ &+ \sum_{p_l \leqslant p_i < p_k} w_i (p_i - p_l - p_k + p_i) \end{aligned}$$

$$+ \sum_{p_k \leqslant p_i} w_i(p_i - p_l - p_i + p_k)$$
$$\geqslant (p_k - p_l)\Bigg(\sum_{p_k \leqslant p_i} w_i - \sum_{p_i < p_k} w_i\Bigg).$$

Pierwszy czynnik ostatniego iloczynu jest dodatni. Na mocy faktu, że wszystkie wagi sumują się do 1 oraz z definicji mediany ważonej, mamy

$$\sum_{p_k \leqslant p_i} w_i - \sum_{p_i < p_k} w_i = 1 - 2\sum_{p_i < p_k} w_i > 1 - 2 \cdot 1/2 = 0,$$

czyli drugi z czynników także jest dodatni. Pokazaliśmy tym samym, że dla dowolnego $p_l < p_k$ zachodzi $f(p_l) > f(p_k)$, co oznacza, że mediana ważona p_k minimalizuje wartość funkcji f.

Podobnie wnioskujemy, gdy $p_l > p_k$, otrzymując

$$f(p_l) - f(p_k) \geqslant (p_k - p_l)\Bigg(\sum_{p_k < p_i} w_i - \sum_{p_i \leqslant p_k} w_i\Bigg) = (p_k - p_l)\Bigg(2\sum_{p_k < p_i} w_i - 1\Bigg).$$

W powyższym iloczynie pierwszy z czynników jest ujemny, a drugi można ograniczyć od góry (tym razem nieostro) przez $2 \cdot 1/2 - 1 = 0$. A zatem także w tym przypadku mamy $f(p_l) > f(p_k)$, co kończy dowód twierdzenia.

(e) Niech $p_i = \langle x_i, y_i \rangle$, gdzie $i = 1, 2, \ldots, n$, będą punktami wejściowymi w 2-wymiarowym problemie lokalizacji urzędu pocztowego z metryką miejską. Jego rozwiązaniem jest taki punkt $p = \langle x_p, y_p \rangle$, dla którego suma $\sum_{i=1}^{n} w_i(|x_p - x_i| + |y_p - y_i|)$ przyjmuje najmniejszą możliwą wartość. Zauważmy, że

$$\min_{\langle x_p, y_p \rangle \in \mathbb{R} \times \mathbb{R}} \Bigg(\sum_{i=1}^{n} w_i(|x_p - x_i| + |y_p - y_i|)\Bigg) = \min_{x_p \in \mathbb{R}} \Bigg(\sum_{i=1}^{n} w_i|x_p - x_i|\Bigg) + \min_{y_p \in \mathbb{R}} \Bigg(\sum_{i=1}^{n} w_i|y_p - y_i|\Bigg).$$

A zatem w celu wyznaczenia optymalnego punktu p wystarczy znaleźć jego współrzędne niezależnie jako rozwiązania 1-wymiarowej wersji problemu. Na podstawie poprzedniego punktu zadanie to sprowadza się do wyznaczenia mediany ważonej x_p spośród elementów $x_1, x_2, \ldots, x_n$ o wagach $w_1, w_2, \ldots, w_n$ i analogicznie mediany ważonej y_p spośród elementów $y_1, y_2, \ldots, y_n$ o tych samych wagach, a następnie zwróceniu pary $\langle x_p, y_p \rangle$.

Zauważmy, że oba ciągi współrzędnych, które przeszukujemy celem znalezienia ich median ważonych, mogą zawierać powtarzające się elementy – w szczególności wszystkie punkty mogą mieć równe pierwsze (lub drugie) współrzędne. Jedynym miejscem w procedurze WEIGHTED-MEDIAN z części (c), gdzie mają znaczenie wartości elementów, jest linia 5, w której dokonywany jest podział elementów algorytmem opartym o SELECT. Okazuje się jednak, o czym łatwo się przekonać, że algorytm ten działa poprawnie nawet wówczas, gdy elementy w tablicy wejściowej powtarzają się – zmienić się może jedynie jego czas działania, który w przypadku pesymistycznym wzrasta do kwadratowego. A zatem procedura z punktu (c) może posłużyć jako narzędzie do szukania median ważonych współrzędnych punktów w algorytmie rozwiązującym rozważany wariant problemu lokalizacji urzędu pocztowego.

9-3. Małe statystyki pozycyjne

(a) *W treści problemu występuje błąd w sformułowaniu rekurencji $U_i(n)$. Wartość $T(n)$ powinna być zwracana dla $i \geqslant n/2$, a część rekurencyjna – dla pozostałych wartości i.*

Jeśli $i \geqslant n/2$, to w celu znalezienia i-tej statystyki pozycyjnej tablicy wejściowej wywołujemy algorytm SELECT. Stąd $U_i(n) = T(n)$.

Załóżmy teraz, że $i < n/2$ i że tablicą wejściową jest $A[p\,..\,r]$, gdzie $n = r - p + 1$. Niech $m = \lfloor n/2 \rfloor$. Dzielimy tablicę wejściową na dwie podtablice $A[p\,..\,p + m - 1]$ oraz $A[p + m\,..\,r]$. Dla każdego $j = 0, 1, \ldots, m - 1$ porównamy elementy $A[p + j]$ oraz $A[p + m + j]$ i ewentualnie zamienimy je, aby zachodziło $A[p + j] > A[p + m + j]$. Wywołujemy następnie nasz algorytm rekurencyjnie w celu znalezienia i-tej statystyki pozycyjnej podtablicy $A[p+m\,..\,r]$, czego efektem ubocznym jest podział tej podtablicy względem szukanego elementu. Po powrocie z wywołania rekurencyjnego podtablica $A[p\,..\,p+m-1]$ zostanie podzielona w analogiczny sposób. Dokładniej, dla każdej pary elementów $A[j]$ i $A[k]$ zamienionych w wywołaniu rekurencyjnym, gdzie $p + m \leqslant j < k \leqslant p + 2m - 1$, zamienione zostaną również elementy $A[j - m]$ i $A[k - m]$. Dzięki temu, po wyznaczeniu i-tego najmniejszego elementu podtablicy $A[p+m\,..\,r]$, fragmenty $A[p\,..\,p+i-1]$ oraz $A[p+m\,..\,p+m+i-1]$ będą zawierać po i najmniejszych elementów ze swoich podtablic. A zatem i-ty najmniejszy element tablicy wejściowej znajduje się w jednym z tych dwóch fragmentów. Wystarczy więc połączyć je w tablicę $B[1\,..\,2i]$, a następnie zwrócić SELECT$(B, 1, 2i, i)$.

Omówimy teraz szczegóły implementacyjne. Aby zrealizować równoczesny podział tablicy, jak również utrzymywać elementy w parach tak, że większy element pary należy do lewej części tablicy, a mniejszy do prawej, będziemy rejestrować każdą zamianę w tablicy wejściowej. Można tego dokonać poprzez utworzenie dodatkowej tablicy *permutation* wypełnionej kolejno liczbami od 1 do n i aktualizowanie jej odpowiednio przy każdej zamianie elementów w tablicy wejściowej. Wynikowa permutacja liczb od 1 do n w tablicy *permutation* jednoznacznie określi permutację elementów tablicy wejściowej w porównaniu z jej stanem początkowym na danym poziomie rekurencji i będzie wykorzystywana do przeprowadzenia podziału drugiej części tablicy po powrocie z wywołania rekurencyjnego, jak również aktualizacji tablicy *permutation* na wyższym poziomie. Zauważmy także, że tablica B, którą tworzymy pod koniec działania algorytmu, może w rzeczywistości stanowić część tablicy A – wystarczy bowiem, aby był to spójny fragment, co da się uzyskać poprzez przeniesienie fragmentu $A[p + m\,..\,p + m + i - 1]$ tuż za koniec fragmentu $A[p\,..\,p + i - 1]$.

Sprawdzimy teraz, ile wynosi $U_i(n)$ w przypadku, gdy $i < n/2$. W pierwszej fazie algorytm wykonuje $\lfloor n/2 \rfloor$ porównań elementów pochodzących z dwóch części tablicy wejściowej. Wywołanie rekurencyjne wprowadza składnik $U_i(\lceil n/2 \rceil)$, zaś procedura SELECT wywołana na tablicy B wykonuje $T(2i)$ porównań. Stąd otrzymujemy

$$U_i(n) = \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i).$$

(b) Stosując metodę podstawiania, wykażemy, że jeśli $i < n/2$, to

$$U_i(n) \leqslant n + cT(2i) \lg(n/i),$$

gdzie $c > 0$ jest pewną stałą.

Jako przypadek bazowy indukcji będziemy rozważać wyrazy $U_i(n)$, w których $2i < n \leqslant 4i$, skąd $n/4 \leqslant i < n/2$. Wykażemy, że spełniają one podane oszacowanie. Mamy:

$$U_i(n) = \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) < n + T(\lceil n/2 \rceil) + T(2i).$$

Jeśli teraz ograniczymy powyższe wyrażenie od góry przez $n + cT(2i) \lg(n/i)$ i rozwiążemy ze względu na c, to otrzymamy

$$c \geqslant \frac{1}{\lg(n/i)} \left(\frac{T(\lceil n/2 \rceil)}{T(2i)} + 1 \right).$$

Pierwszy czynnik po prawej stronie powyższej nierówności można ograniczyć od góry przez $1/\lg 2 = 1$. W celu oszacowania drugiego czynnika zauważmy, że rekurencja $T(n)$ jest funkcją dodatnią klasy $\Theta(n)$, zatem istnieją stałe d_1, $d_2 > 0$, że dla każdego $n \geqslant 1$ spełnione są nierówności $d_1 n \leqslant T(n) \leqslant d_2 n$. Mamy zatem

$$\frac{T(\lceil n/2 \rceil)}{T(2i)} \leqslant \frac{d_2 \lceil n/2 \rceil}{2 d_1 i} < \frac{d_2 n}{d_1 n/2} = \frac{2 d_2}{d_1},$$

a stąd wynika, że $c \geqslant 2d_2/d_1 + 1$. Pokazaliśmy, że c istotnie może być stałą, a zatem podstawa indukcji zachodzi.

W drugim kroku indukcyjnym mamy $n > 4i$ i przyjmujemy założenie

$$U_i(\lceil n/2 \rceil) \leqslant \lceil n/2 \rceil + cT(2i) \lg(\lceil n/2 \rceil / i).$$

Wówczas:

$$\begin{aligned} U_i(n) &= \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) \\ &\leqslant \lfloor n/2 \rfloor + \lceil n/2 \rceil + cT(2i) \lg(\lceil n/2 \rceil / i) + T(2i) \\ &= n + cT(2i) \lg \lceil n/2 \rceil - cT(2i) \lg i + T(2i) \\ &\leqslant n + cT(2i) \lg n - cT(2i) \lg i. \end{aligned}$$

Ostatnia nierówność jest spełniona, jeżeli $cT(2i) \lg \lceil n/2 \rceil + T(2i) \leqslant cT(2i) \lg n$, co po uproszczeniu sprowadza się do warunku $c \lg \frac{n}{\lceil n/2 \rceil} \geqslant 1$. Łatwo zauważyć, że aby go spełnić dla $n \geqslant 5$, wystarczy przyjąć $c \geqslant \log_{5/3} 2$, co kończy dowód oszacowania na $U_i(n)$.

(c) Ponieważ i jest stałe, to $T(2i)$ również można potraktować jako wartość stałą. Na mocy poprzedniego punktu mamy

$$U_i(n) = n + O(T(2i) \lg(n/i)) = n + O(\lg n - \lg i) = n + O(\lg n).$$

(d) Dla $k > 2$ oszacowanie wynika natychmiast z części (b) po podstawieniu $i = n/k$, bo wtedy oczywiście $i < n/2$.

Jeśli $k = 2$, to $i = n/2$ i rozwiązaniem rekurencji $U_i(n)$ jest $U_i(n) = T(n) = O(n)$. Teza w tym przypadku przyjmuje postać $U_i(n) = n + O(T(n))$, co oczywiście jest spełnione, ponieważ $n + O(T(n)) = O(n)$.

Część **III**

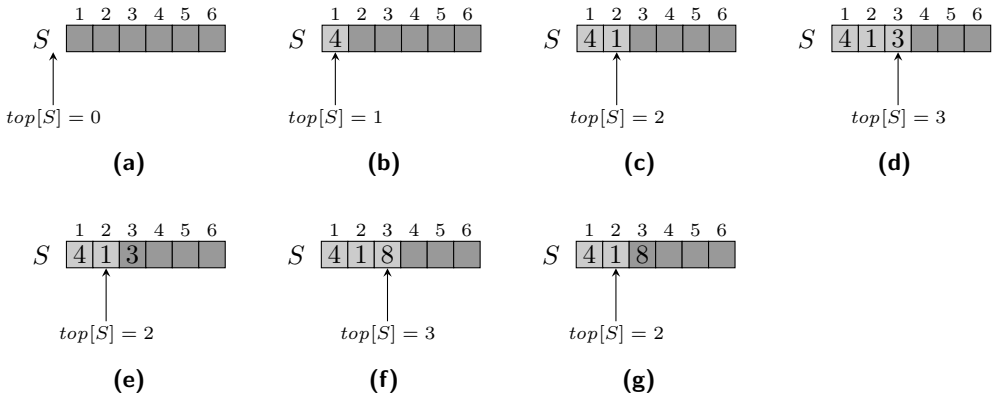
Struktury danych

Rozdział **10**

Elementarne struktury danych

10.1. Stosy i kolejki

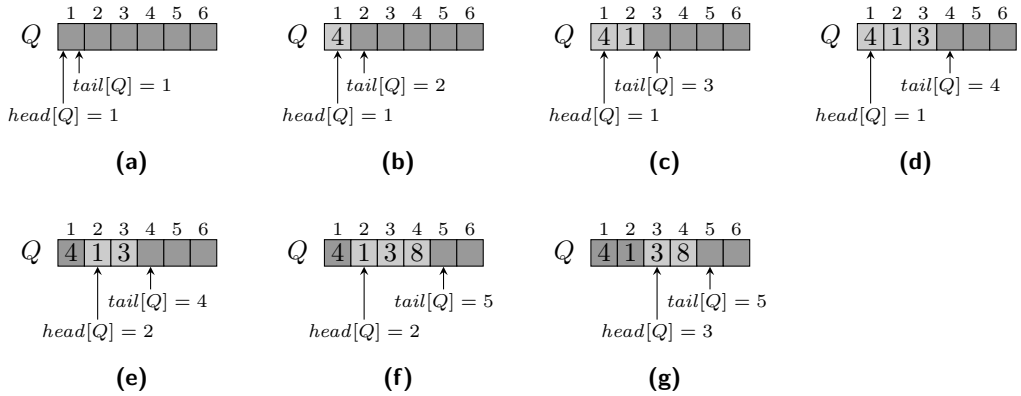
10.1-1. Ciąg operacji na stosie S został przedstawiony na rys. 19.



Rysunek 19: Operacje wstawiania i usuwania elementów na stosie S. **(a)** Pusty stos S reprezentowany jako tablica $S[1\,..\,6]$. **(b)–(d)** Stos S po wykonaniu na nim kolejnych operacji PUSH. **(e)** Po usunięciu elementu ze stosu S zmieniana jest jedynie pozycja atrybutu $top[S]$, natomiast sam element pozostaje w tablicy. **(f)** Wstawienie nowego elementu na stos S nadpisuje stary element, który został usunięty w poprzednim kroku. **(g)** Stos S po wykonaniu ostatniej operacji POP.

10.1-2. Z tablicą A związujemy atrybuty *left-top*$[A]$ i *right-top*$[A]$, które będą wskaźnikami na pozycje wierzchołków, odpowiednio, pierwszego i drugiego stosu. Pierwszy stos będzie składał się z elementów $A[1\,..\,$*left-top*$[A]]$, a drugi – z elementów $A[$*right-top*$[A]\,..\,n]$. Początkowo *left-top*$[A] = 0$ i *right-top*$[A] = n + 1$. Dodawanie i usuwanie elementów w pierwszym stosie działa identycznie jak dla pojedynczego stosu znajdującego się w tablicy A, którego wierzchołek zajmuje pozycję *left-top*$[A]$. Drugi stos zachowuje się symetrycznie do pierwszego – podczas dodawania do niego nowego elementu atrybut *right-top*$[A]$ będzie dekrementowany, a podczas usuwania – inkrementowany. Operacje dodawania i usuwania elementów działają oczywiście w czasie stałym. Jeśli *left-top*$[A] = 0$, to pierwszy stos jest pusty, a jeśli *right-top*$[A] = n+1$, to drugi stos jest pusty. Przepełnienie ma miejsce tylko wtedy, gdy *left-top*$[A] =$ *right-top*$[A] - 1$ i usiłujemy dodać nowy element do któregokolwiek stosu, czyli wówczas, gdy łączna liczba elementów na obu stosach wynosi n.

10.1-3. Ciąg operacji na kolejce Q ilustruje rys. 20.



Rysunek 20: Operacje wstawiania i usuwania elementów na kolejce Q. **(a)** Pusta kolejka Q reprezentowana jako tablica $Q[1\,..\,6]$. **(b)–(d)** Kolejka Q po wykonaniu na niej kolejnych operacji ENQUEUE. **(e)** Po usunięciu elementu z kolejki Q zmieniana jest jedynie pozycja atrybutu $head[Q]$, natomiast sam element pozostaje w tablicy. **(f)** Wstawienie nowego elementu do kolejki Q nadpisuje stary element, który został usunięty w poprzednim kroku. **(g)** Kolejka Q po wykonaniu ostatniej operacji DEQUEUE.

10.1-4. Poniżej znajduje się pseudokod operacji analogicznej do STACK-EMPTY, ale testującej pustość kolejki. Wykorzystujemy ją podczas wykrywania błędów niedomiaru.

```
QUEUE-EMPTY(Q)
1  if head[Q] = tail[Q]
2      then return TRUE
3      else  return FALSE
```

Następujące wiersze należy dodać na początek procedury ENQUEUE:

```
if head[Q] = tail[Q] + 1
    then error „nadmiar”
if head[Q] = 1 i tail[Q] = length[Q]
    then error „nadmiar”
```

Z kolei poniższy fragment kodu umieszczamy na początku procedury DEQUEUE:

```
if QUEUE-EMPTY(Q)
    then error „niedomiar”
```

10.1-5. Kolejkę dwustronną implementujemy za pomocą tablicy $D[1\,..\,n]$. Podobnie jak w zwykłej kolejce atrybut $head[D]$ wskazuje na początek kolejki, natomiast atrybut $tail[D]$ wyznacza następną wolną pozycję, na którą można wstawić nowy element. Procedury HEAD-ENQUEUE oraz HEAD-DEQUEUE mają na celu, odpowiednio, dodanie nowego elementu na początek kolejki i usunięcie elementu z początku kolejki. Z kolei procedury TAIL-ENQUEUE oraz TAIL-DEQUEUE implementują dodawanie i usuwanie elementów na końcu kolejki. Dla skrócenia zapisu pominięto sprawdzanie błędów niedomiaru i przepełnienia.

```
HEAD-ENQUEUE(D, x)
1  if head[D] = 1
2     then head[D] ← length[D]
3     else head[D] ← head[D] − 1
4  D[head[D]] ← x
```

```
HEAD-DEQUEUE(D)
1  x ← D[head[D]]
2  if head[D] = length[D]
3     then head[D] ← 1
4     else head[D] ← head[D] + 1
5  return x
```

```
TAIL-ENQUEUE(D, x)
1  D[tail[D]] ← x
2  if tail[D] = length[D]
3     then tail[D] ← 1
4     else tail[D] ← tail[D] + 1
```

```
TAIL-DEQUEUE(D, x)
1  if tail[D] = 1
2     then tail[D] ← length[D]
3     else tail[D] ← tail[D] − 1
4  return D[tail[D]]
```

Wszystkie powyższe operacje działają w czasie $\Theta(1)$.

10.1-6. Wszystkie elementy kolejki będą trzymane na jednym stosie – drugi stos będzie pełnił funkcję pomocniczą. Dodawanie nowego elementu do kolejki oraz sprawdzanie czy kolejka jest pusta, nie różnią się od analogicznych operacji wykonywanych na pierwszym stosie. Usuwanie elementu z kolejki jest już operacją nieco bardziej skomplikowaną, gdyż należy pobrać element znajdujący się najgłębiej na tym stosie. Ściągamy wpierw z niego wszystkie elementy za pomocą operacji POP i umieszczamy kolejno na drugim stosie, wywołując ciąg operacji PUSH. W rezultacie drugi stos będzie zawierał wszystkie elementy kolejki w odwrotnej kolejności. Teraz pobieramy i zapamiętujemy element ze szczytu drugiego stosu, gdyż za chwilę zwrócimy go jako wynik procedury. Wcześniej trzeba bowiem przenieść pozostałą zawartość drugiego stosu z powrotem do pierwszego, co przy okazji przywróci początkową kolejność elementów.

Łatwo sprawdzić, że testowanie pustości kolejki oraz dodawanie do niej nowego elementu, są wykonywane w czasie stałym, natomiast usuwanie wymaga czasu proporcjonalnego do liczby elementów kolejki.

10.1-7. Rozwiązanie jest analogiczne do rozwiązania poprzedniego zadania. Sprawdzanie czy stos jest pusty, jak również dodawanie nowego elementu, to identyczne operacje wywoływane na pierwszej kolejce. Usuwanie elementu ze stosu odbywa się poprzez pobranie wszystkich elementów z wyjątkiem ostatniego z pierwszej kolejki i dodanie tych elementów do drugiej. Ostatni z nich także usuwamy z kolejki, zapamiętawszy go w celu późniejszego zwrócenia jako wyniku operacji. Ostatnim krokiem jest przeniesienie reszty elementów z powrotem do pierwszej kolejki.

Podobnie jak w poprzednim zadaniu operacja odpowiedzialna za sprawdzenie czy stos jest pusty oraz dodawanie elementu działają w czasie stałym, a operacja usuwania – w czasie liniowym względem liczby elementów na stosie.

10.2. Listy

10.2-1. Operację INSERT, dodającą nowy element x na początek listy jednokierunkowej L, można zaimplementować w czasie stałym – wystarczy ustawić pole $next[x]$ na głowę listy L (albo na NIL, jeśli lista L jest pusta) i uaktualnić wskaźnik $head[L]$. Operacja DELETE, czyli usuwanie z listy jednokierunkowej elementu wskazywanego przez x, wymaga jednak czasu wyższego niż stały. Jest tak dlatego, że jedynym sposobem na dotarcie do elementu poprzedzającego x na liście L w celu aktualizacji jego pola $next$, jest przejście tej listy od głowy aż do tegoż elementu. Czynność ta w najgorszym przypadku zajmuje czas proporcjonalny do liczby elementów listy L.

Poniżej zamieszczamy implementacje obu tych operacji – będziemy z nich korzystać w późniejszych zadaniach.

```
SINGLY-LINKED-LIST-INSERT(L, x)
1  next[x] ← head[L]
2  head[L] ← x
```

```
SINGLY-LINKED-LIST-DELETE(L, x)
1  if x = head[L]
2      then head[L] ← next[head[L]]
3      else y ← head[L]
4           while next[y] ≠ x
5                do y ← next[y]
6           next[y] ← next[x]
```

10.2-2. Wszystkie elementy implementowanego stosu będziemy trzymać na liście jednokierunkowej L w kolejności od szczytu w głowie listy do dna stosu w ogonie. Dzięki takiemu rozwiązaniu operacje dodawania i usuwania elementów będą działać w czasie $\Theta(1)$. Sprawdzenie pustości stosu sprowadza się do sprawdzenia pustości listy L. Pseudokody operacji PUSH i POP prezentujemy poniżej.

```
SINGLY-LINKED-LIST-PUSH(L, k)
1  key[x] ← k
2  SINGLY-LINKED-LIST-INSERT(L, x)
```

```
SINGLY-LINKED-LIST-POP(L)
1  if head[L] = NIL
2      then error „niedomiar”
3  x ← head[L]
4  head[L] ← next[x]
5  return key[x]
```

10.2-3. Elementy kolejki będziemy przechowywać na liście jednokierunkowej L w kolejności od głowy kolejki do jej ogona. Aby jednak operacja dodawania była wykonalna w czasie $\Theta(1)$,

wymagane jest związanie z listą L dodatkowego atrybutu $tail[L]$, który będzie wskazywał na ogon listy L albo na NIL, jeżeli lista L jest pusta. Oczywiście testowanie pustości kolejki polega na sprawdzeniu, czy pusta jest lista L. Implementacje operacji ENQUEUE i DEQUEUE znajdują się poniżej.

SINGLY-LINKED-LIST-ENQUEUE(L, k)

```
1  next[x] ← NIL
2  key[x] ← k
3  if tail[L] ≠ NIL
4      then next[tail[L]] ← x
5      else head[L] ← x
6  tail[L] ← x
```

SINGLY-LINKED-LIST-DEQUEUE(L)

```
1  if head[L] = NIL
2      then error „niedomiar”
3  x ← head[L]
4  head[L] ← next[x]
5  if tail[L] = x
6      then tail[L] ← NIL
7  return key[x]
```

10.2-4. Zauważmy, że pole $key[nil[L]]$ jest niewykorzystywane w implementacji listy z wartownikami. Możemy zatem na początku działania procedury LIST-SEARCH$'$ przypisać mu wartość k. Jeśli na liście L nie będzie elementu o wartości k, to pętla tej procedury zatrzyma się na elemencie $nil[L]$, po czym zostanie on zwrócony.

10.2-5. Operacja wstawiania nowego elementu na jednokierunkową listę cykliczną L umieszcza go jako następnik głowy listy L. Podczas operacji usuwania znajdowany jest poprzednik y usuwanego elementu x, po czym $next[y]$ zostaje uaktualnione. Jeśli x jest głową listy, to atrybut $head[L]$ zostaje ustawiony na następnik x albo na NIL, w przypadku gdy x stanowi jedyny element listy L. Wreszcie operacja wyszukiwania przechodzi całą listę L, począwszy od jej głowy, zatrzymując się w momencie odnalezienia elementu o szukanym kluczu bądź w chwili ponownego dotarcia do głowy listy L.

Poniższe procedury stanowią implementacje tych trzech operacji słownikowych.

CIRCULAR-LIST-INSERT(L, x)

```
1  if head[L] = NIL
2      then head[L] ← next[x] ← x
3      else next[x] ← next[head[L]]
4           next[head[L]] ← x
```

```
CIRCULAR-LIST-DELETE(L, x)
1  y ← head[L]
2  while next[y] ≠ x
3        do y ← next[y]
4  next[y] ← next[x]
5  if head[L] = x
6     then if next[x] = x
7             then head[L] ← NIL
8             else  head[L] ← next[x]
```

```
CIRCULAR-LIST-SEARCH(L, k)
 1  if head[L] = NIL
 2     then return NIL
 3  if key[head[L]] = k
 4     then return head[L]
 5  x ← next[head[L]]
 6  while x ≠ head[L]
 7        do if key[x] = k
 8              then return x
 9           x ← next[x]
10  return NIL
```

Łatwo sprawdzić, że procedura implementująca operację INSERT działa w czasie stałym, natomiast pozostałe dwie procedury dla listy o n elementach w pesymistycznym przypadku potrzebują czasu $\Theta(n)$.

10.2-6. Zbiory można zaimplementować jako jednokierunkowe listy cykliczne. Zbiory S_1 i S_2 są rozłączne, więc w reprezentacji sumy $S_1 \cup S_2$ nie pojawią się powtarzające się elementy. Operacja UNION może więc tylko „sklejać" listy reprezentujące zbiory S_1 i S_2. Podczas działania tej operacji następnikiem głowy pierwszej listy staje się następnik głowy drugiej listy, a następnikiem głowy drugiej staje się początkowy następnik głowy pierwszej. Oczywiście działania te są wykonywane tylko wtedy, gdy obie listy są niepuste. Wykonując stałą liczbę kroków, otrzymujemy w wyniku tej operacji jednokierunkową listę cykliczną (której głową może być dowolny jej element) reprezentującą zbiór $S_1 \cup S_2$.

10.2-7. W algorytmie wykorzystamy pomocniczą listę jednokierunkową L', na którą będziemy umieszczać kolejno usuwane elementy z głowy listy L. Łatwo zauważyć, że pod koniec operacji przeniesione elementy będą znajdować się w L' w porządku odwrotnym względem początkowego ustawienia na liście L. Nadanie atrybutowi $head[L]$ wartości $head[L']$ wystarczy, aby przenieść całą zawartość listy L' do listy L.

```
SINGLY-LINKED-LIST-REVERSE(L)
1  head[L'] ← NIL
2  while head[L] ≠ NIL
3        do x ← head[L]
4           SINGLY-LINKED-LIST-DELETE(L, head[L])
5           SINGLY-LINKED-LIST-INSERT(L', x)
6  head[L] ← head[L']
```

Procedury SINGLY-LINKED-LIST-INSERT i SINGLY-LINKED-LIST-DELETE zostały przedstawione w zad. 10.2-1. Ich wywołania w powyższym algorytmie zajmują czas stały, stąd czasem działania algorytmu jest $\Theta(n)$.

10.2-8. Zgodnie z opisem w treści zadania przyjmijmy, że każdy element listy zamiast wskaźników na poprzednik i następnik posiada atrybut np, będący alternatywą wykluczającą tych dwóch wskaźników. Załóżmy, że znamy adres elementu x listy L i elementu y będącego poprzednikiem x na liście L. Niech z będzie następnikiem x na tej liście. Wówczas $np[x] = z \operatorname{xor} y$, zatem na podstawie własności operacji xor, aby dostać się do z, wystarczy obliczyć wartość

$$z = z \operatorname{xor} 0 = z \operatorname{xor} (y \operatorname{xor} y) = (z \operatorname{xor} y) \operatorname{xor} y = np[x] \operatorname{xor} y.$$

Podobnie ma się rzecz podczas wyznaczania y na podstawie x i z – wówczas $y = np[x] \operatorname{xor} z$. Zauważmy, że jeśli element x jest ogonem listy, to $np[x] = \text{NIL} \operatorname{xor} y = 0 \operatorname{xor} y = y$, skąd mamy $z = np[x] \operatorname{xor} y = 0 = \text{NIL}$ i analogicznie dla głowy listy, z faktu, że $np[x] = z \operatorname{xor} 0 = z$ wynika $y = \text{NIL}$. Podobnie jak w zwykłych listach przyjmujemy, że atrybut $head[L]$ przechowuje wskaźnik do głowy listy L. Ponadto z listą związujemy atrybut $tail[L]$, który będzie wskazywał na jej ogon – jest on konieczny do tego, aby operacja odwracania kolejności elementów na liście działała w czasie stałym.

Poniżej przedstawiono procedury implementujące operacje SEARCH, INSERT i DELETE dla takiej listy.

```
XOR-LINKED-LIST-SEARCH(L, k)
1  x ← head[L]
2  y ← NIL
3  while x ≠ NIL i key[x] ≠ k
4        do z ← np[x] xor y
5           y ← x
6           x ← z
7  return x
```

```
XOR-LINKED-LIST-INSERT(L, x)
1  np[x] ← head[L]
2  if head[L] ≠ NIL
3     then np[head[L]] ← np[head[L]] xor x
4  head[L] ← x
5  if tail[L] = NIL
6     then tail[L] ← x
```

```
XOR-LINKED-LIST-DELETE(L, x)
 1  x' ← head[L]
 2  y ← NIL
 3  while x' ≠ x
 4        do z ← np[x'] xor y
 5           y ← x'
 6           x' ← z
 7  z ← np[x] xor y
 8  if x = head[L]
 9     then head[L] ← z
10     else np[y] ← np[y] xor x xor z
11  if x = tail[L]
12     then tail[L] ← y
13     else np[z] ← np[z] xor x xor y
```

Procedury XOR-LINKED-LIST-SEARCH i XOR-LINKED-LIST-INSERT są analogiczne do swoich odpowiedników dla zwykłej listy dwukierunkowej i pesymistyczny czas ich działania wynosi odpowiednio $\Theta(n)$ oraz $\Theta(1)$. W procedurze XOR-LINKED-LIST-DELETE należy odszukać na liście poprzednika i następnika elementu x, aby uaktualnić ich pola np, co w pesymistycznym przypadku zajmuje czas $\Theta(n)$. Odwracanie kolejności elementów – zarówno w zwykłej liście dwukierunkowej, jak i w tak zmodyfikowanej liście – jest możliwe w czasie $\Theta(1)$. W tym celu wystarczy zamienić ze sobą wartości atrybutów $head[L]$ i $tail[L]$.

10.3. Reprezentowanie struktur wskaźnikowych za pomocą tablic

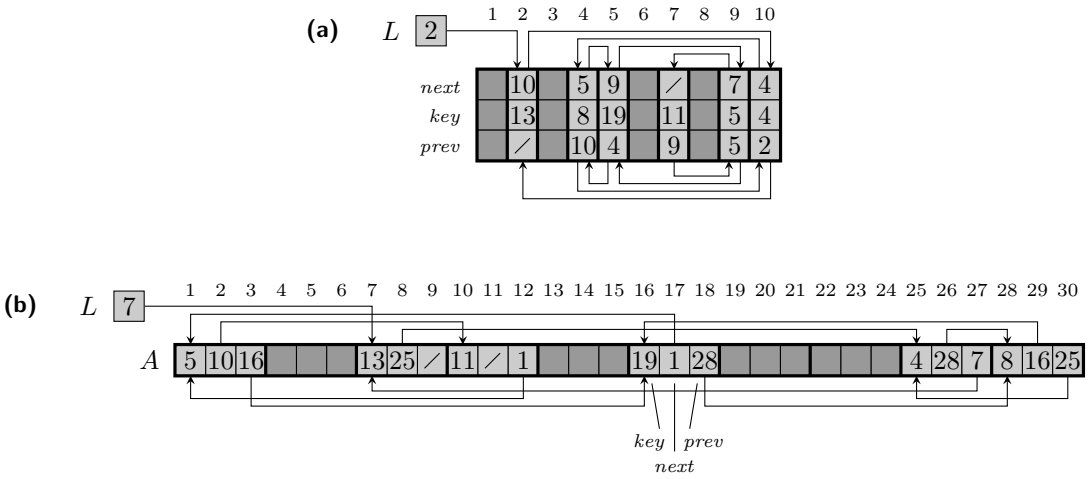
10.3-1. *Oryginalna treść drugiej części zadania żąda, aby podany ciąg zilustrować jako listę dwukierunkową w reprezentacji jednotablicowej.*

Rys. 21 przedstawia przykładowe rozmieszczenie elementów ciągu jako listy dwukierunkowej w reprezentacji wielotablicowej i jednotablicowej.

10.3-2. Załóżmy, że właściwa lista oraz lista wolnych pozycji znajdują się w pojedynczej tablicy A. Poniższe procedury są adaptacjami operacji przydzielania i zwalniania pamięci dla list dwukierunkowych w reprezentacji jednotablicowej. Wykorzystujemy tutaj fakt, że polu *next* odpowiada przesunięcie 1 względem początku fragmentu tablicy A przechowującego dany element listy.

```
SINGLE-ARRAY-ALLOCATE-OBJECT()
1  if free = NIL
2     then error „brak pamięci"
3  x ← free
4  free ← A[x + 1]
5  return x
```

```
SINGLE-ARRAY-FREE-OBJECT(x)
1  A[x + 1] ← free
2  free ← x
```



Rysunek 21: Ciąg $\langle 13, 4, 8, 19, 5, 11\rangle$ w postaci listy dwukierunkowej w reprezentacji tablicowej. **(a)** Lista reprezentowana za pomocą tablic *key*, *next* oraz *prev*. **(b)** Ta sama lista reprezentowana w pojedynczej tablicy A.

10.3-3. Lista wolnych pozycji jest listą jednokierunkową – nie są więc wykorzystywane w niej pola *prev*.

10.3-4. Korzystając ze wskazówki z treści zadania, zaimplementujemy listę wolnych pozycji jako stos. Będzie on zajmował wszystkie pozycje na prawo od tej wskazywanej przez *free*, na której znajdzie się jego wierzchołek. Stos jest pusty wtedy i tylko wtedy, gdy $free = \text{NIL}$. Przydzielanie pamięci polega na wywołaniu na stosie wolnych pozycji operacji POP, a więc procedura COMPACT-LIST-ALLOCATE-OBJECT jest identyczna z oryginalną ALLOCATE-OBJECT. Z kolei zwalnianie pamięci to wywołanie na tym stosie operacji PUSH. Wcześniej jednak należy zamienić zwalniany element z tym, który znajduje się bezpośrednio na lewo od wierzchołka stosu. Poniższy pseudokod stanowi implementację tej operacji.

COMPACT-LIST-FREE-OBJECT(x)

```
 1  if free = NIL
 2     then y ← length[key]
 3     else  y ← free − 1
 4  if next[x] ≠ NIL
 5     then prev[next[x]] ← prev[x]
 6  if prev[x] ≠ NIL
 7     then next[prev[x]] ← next[x]
 8  if x ≠ y
 9     then if next[y] ≠ NIL
10             then prev[next[y]] ← x
11          if prev[y] ≠ NIL
12             then next[prev[y]] ← x
13  skopiuj wartości pól key, next i prev elementu y do pól elementu x
14  if L = y
15     then L ← x
16  FREE-OBJECT(y)
```

10.3-5. Ogólna idea procedury COMPACTIFY-LIST wygląda następująco. Podczas przechodzenia po liście L wyznaczane są te elementy, które zajmują pozycje na prawo od m. Wszystkie one muszą być zamienione z elementami listy F znajdującymi się na pozycjach do m włącznie. Aby zachować czas $\Theta(m)$, procedura nie może przechodzić po liście F, która składa się z $n - m$ elementów. Zamiast tego będzie ona przeszukiwać liniowo tablice implementujące listy i wyznaczać rekordy należące do listy wolnych pozycji. Jednym ze sposobów rozróżniania elementów list L i F jest wykorzystanie wskaźników *prev*. Na początku działania procedury do pól *prev* wszystkich elementów listy L zostanie wpisana specjalna wartość, nie będąca poprawnym indeksem tablicy, np. -1. Tuż przed zakończeniem działania wskaźnikom *prev* zostaną nadane właściwe wartości, ale wpierw pola te posłużą do identyfikacji elementów listy.

```
COMPACTIFY-LIST(L, F)
 1  n ← length[key]
 2  wpisz −1 do pól prev elementów listy L i wyznacz liczbę jej elementów m
 3  x ← L
 4  x′ ← NIL
 5  y ← 1
 6  while x ≠ NIL
 7        do if x ⩽ m
 8              then x′ ← x
 9                   x ← next[x]
10              else  while prev[y] = −1
11                          do y ← y + 1
12                    zamień wartości pól key, next i prev elementu x z polami elementu y
13                    if next[x] ≠ NIL
14                       then prev[next[x]] ← x
15                    if prev[x] ≠ NIL
16                       then next[prev[x]] ← x
17                       else  F ← x
18                    if x′ ≠ NIL
19                       then next[x′] ← y
20                       else  L ← y
21                    x′ ← y
22                    x ← next[y]
23  przywróć poprawne wartości w polach prev elementów listy L
```

Procedura przechodzi przez listę L trzy razy. Najpierw w wierszu 2 tablice przechowujące listy L i F są przygotowywane do właściwego przetwarzania poprzez ustawienie pól *prev* wszystkich elementów listy L na -1. W tym samym kroku wyznaczana jest wartość m – rozmiar listy L.

Kolejne przejście po liście to właściwe kompaktowanie. Jeśli dany element x listy L zajmuje w tablicach pozycję wyższą niż m, to zostaje zamieniony z elementem y listy F znajdującym się najbardziej na lewo. Pętla **while** w wierszach 10–11 wyznacza y poprzez liniowe przeglądanie tablicy *prev*, po czym w wierszu 12 elementy x i y zamieniają swoje pozycje. Należy jeszcze uaktualnić pola *prev* i *next* elementów sąsiednich na liście F (wiersze 14 i 16), pole *next* poprzednika x na liście L (wiersz 19), jak również wskaźniki L i F, w przypadku gdy wśród zamienianych elementów była głowa którejś z list (wiersze 17 oraz 20).

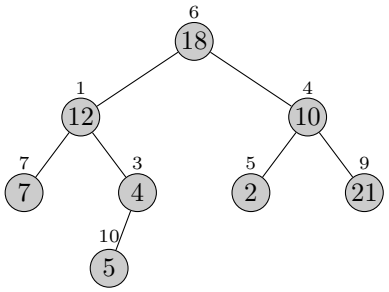
Ostatnim krokiem procedury jest ponowne przejście przez listę L w wierszu 23 i przywrócenie odpowiednich wartości w polach *prev* wszystkich jej elementów.

Po wykonaniu procedury elementy z listy wolnych pozycji nie zawsze będą umieszczone w tablicach według ich kolejności na tej liście. Z tego powodu na liście przetworzonej proce-

durą COMPACTIFY-LIST nie możemy korzystać z procedur COMPACT-LIST-ALLOCATE-OBJECT i COMPACT-LIST-FREE-OBJECT z zad. 10.3-4, które zakładają, że reprezentacja tablicowa listy wolnych pozycji stanowi stos.

10.4. Reprezentowanie drzew (ukorzenionych)

10.4-1. Drzewo binarne, którego reprezentacją są podane tablice, przedstawiono na rys. 22. Węzły o kluczach 14 i 15 nie należą do drzewa.



Rysunek 22: Drzewo binarne o korzeniu o indeksie 6 reprezentowane przez tablice *key*, *left* i *right*.

10.4-2. Szukany algorytm został opisany w Podręczniku w podrozdziale 12.1 jako procedura INORDER-TREE-WALK. Czas działania tego algorytmu dla drzewa o n węzłach wynosi $\Theta(n)$ – mówi o tym tw. 12.1 z Podręcznika.

10.4-3. Przedstawiona poniżej procedura stanowi nierekurencyjną implementację algorytmu przechodzenia drzewa metodą preorder (patrz podrozdział 12.1). Do emulowania rekursji wykorzystywany jest stos. Każdy węzeł drzewa jest dokładnie raz wstawiany na stos i dokładnie raz z niego usuwany, stąd czas działania tej procedury dla drzewa o n węzłach wynosi $\Theta(n)$.

ITERATIVE-PREORDER-TREE-WALK(T)

```
 1  if root[T] = NIL
 2     then return
 3  PUSH(S, root[T])
 4  while STACK-EMPTY(S) = FALSE
 5        do x ← POP(S)
 6           wypisz key[x]
 7           if right[x] ≠ NIL
 8              then PUSH(S, right[x])
 9           if left[x] ≠ NIL
10              then PUSH(S, left[x])
```

10.4-4. Nasza procedura będzie przyjmować węzeł x drzewa w reprezentacji „na lewo syn, na prawo brat" i wypisywać wszystkie klucze z poddrzewa o korzeniu w x. Jeśli $x \neq$ NIL, to zostanie wypisany klucz węzła x i procedura zostanie wywołana rekurencyjnie najpierw dla najbardziej lewego syna x, a następnie dla kolejnego brata x.

```
TREE-WALK(x)
1  if x ≠ NIL
2     then wypisz key[x]
3          TREE-WALK(left-child[x])
4          TREE-WALK(right-sibling[x])
```

Aby wypisać wszystkie klucze drzewa T w reprezentacji „na lewo syn, na prawo brat", należy wywołać TREE-WALK($root[T]$).

Każdy węzeł drzewa jest wypisywany w dokładnie jednym wywołaniu rekurencyjnym. Procedura wywoływana jest także dla wszystkich pustych najbardziej lewych synów i dla wszystkich pustych braci znajdujących się w drzewie. Stąd wnioskujemy, że procedura wypisze wszystkie n kluczy drzewa w czasie $\Theta(n)$.

10.4-5. W algorytmie będziemy symulować przechodzenie drzewa w porządku inorder, używając trzech wskaźników – *curr* będący aktualnie odwiedzanym węzłem oraz *prev* i *next* – odpowiednio, poprzednio przetwarzanym i kolejnym do przetworzenia węzłem. Odwiedzenie danego węzła x będzie realizowane przez pomocniczą procedurę STACKLESS-INORDER-VISIT(x). Polega ono na wypisaniu klucza x oraz wyznacza kolejny węzeł do przetworzenia.

```
STACKLESS-INORDER-VISIT(x)
1  wypisz key[x]
2  if right[x] ≠ NIL
3     then return right[x]
4     else return p[x]
```

Algorytm zapisujemy w postaci pseudokodu:

```
STACKLESS-INORDER-TREE-WALK(T)
 1  prev ← NIL
 2  curr ← root[T]
 3  while curr ≠ NIL
 4        do if prev = p[curr]
 5              then if left[curr] ≠ NIL
 6                      then next ← left[curr]
 7                      else next ← STACKLESS-INORDER-VISIT(curr)
 8           elseif prev = left[curr]
 9              then next ← STACKLESS-INORDER-VISIT(curr)
10           else next ← p[curr]
11           prev ← curr
12           curr ← next
```

Kolejne węzły, przez które przechodzimy w algorytmie, wyznaczane są na podstawie wzajemnej relacji między węzłami *prev* i *curr*. Gdy *prev* jest ojcem *curr*, to aktualnie schodzimy w dół drzewa po lewych synach (wiersz 6). W przypadku osiągnięcia liścia, odwiedzamy go, po czym przechodzimy do jego prawego poddrzewa albo zawracamy w kierunku korzenia (wiersz 7). Gdy *prev* jest lewym synem *curr*, to wracamy w górę drzewa, odwiedzając napotykane węzły i przechodząc ich prawe poddrzewa (wiersz 9). Jeśli wreszcie *prev* jest prawym synem węzła *curr*, to wracamy w górę drzewa z właśnie odwiedzonego prawego poddrzewa (wiersz 10). W wierszach 11 i 12 następuje aktualizacja wskaźników *prev* i *curr*, po czym algorytm kontynuuje swoje działanie, aż *curr* = NIL.

Każdą krawędzią drzewa przejdziemy dokładnie 2 razy – poruszając się najpierw w dół, a później w górę drzewa. Odwiedzenie węzła x następuje w dwóch przypadkach – gdy docieramy do niego od jego ojca i x nie ma lewego syna lub wtedy, gdy wracamy w górę drzewa z lewego syna x. Algorytm odwiedzi zatem każdy węzeł dokładnie raz, działając w czasie $\Theta(n)$.

10.4-6. *W oryginalnej treści zadania szukana jest taka reprezentacja drzewa, która umożliwi wyznaczanie i uzyskiwanie dostępu do ojca danego węzła* ***lub*** *wszystkich jego synów w czasie proporcjonalnym do liczby synów.*

Opiszemy modyfikacje, jakie należy wprowadzić w reprezentacji „na lewo syn, na prawo brat", aby spełnić wymaganie z treści zadania. Ponieważ nie wymagamy dostępu do ojca danego węzła w stałym czasie, to możemy zrezygnować z atrybutu p. Zauważmy ponadto, że w reprezentacji „na lewo syn, na prawo brat", jeśli węzeł x jest korzeniem drzewa albo najbardziej na prawo położonym synem swojego ojca, to $\textit{right-sibling}[x] = \text{NIL}$. Wykorzystamy ten wskaźnik w węźle x, pokazując nim na ojca x (albo NIL, jeśli x jest korzeniem). Aby móc jednoznacznie określać, czy węzeł wskazywany przez ten atrybut jest bratem, czy ojcem x, wykorzystamy dodatkowe pole – zmienną boolowską. Nazwijmy następująco atrybuty każdego węzła x w nowej reprezentacji:

- $\textit{left-child}[x]$ – wskazuje na najbardziej lewego syna x albo NIL, jeśli x jest liściem (identyczny z $\textit{left-child}[x]$ z reprezentacji „na lewo syn, na prawo brat");
- $\textit{next}[x]$ – wskazuje na kolejnego brata x albo na ojca x, jeśli x jest korzeniem drzewa lub najbardziej na prawo wysuniętym synem swojego ojca;
- $\textit{last-sibling}[x]$ – zmienna boolowska przyjmująca wartość TRUE, jeśli węzeł x jest korzeniem lub najbardziej na prawo wysuniętym synem swojego ojca i FALSE w przeciwnym przypadku.

Dzięki tak zdefiniowanym atrybutom, dla danego węzła x możemy wyznaczyć wszystkich jego synów poprzez przejście do węzła $\textit{left-child}[x]$, a następnie poruszając się po wskaźnikach *next* kolejnych synów x. Każdy napotykany węzeł y pokazuje wskaźnikiem $\textit{next}[y]$ na swojego brata po prawej stronie, o ile $\textit{last-sibling}[y] = \text{FALSE}$. W momencie dotarcia do węzła y, dla którego $\textit{last-sibling}[y] = \text{TRUE}$, odwiedzimy wszystkich synów węzła x. Podobnie, aby ustalić ojca x, przechodzimy po wskaźnikach *next* braci x znajdujących się po jego prawej stronie, aż do napotkania węzła z, dla którego $\textit{last-sibling}[z] = \text{TRUE}$. Warunek ten oznacza, że węzeł $\textit{next}[z]$, o ile istnieje, jest ojcem zarówno z, jak i x. Jeśli z kolei $\textit{next}[z] = \text{NIL}$, to x jest korzeniem drzewa.

Podana implementacja pozwala na wyznaczenie wszystkich synów danego węzła w czasie proporcjonalnym do ich liczby oraz jego ojca w czasie proporcjonalnym do liczby braci danego węzła.

Problemy

10-1. Porównanie list

Tabela 4 zawiera pesymistyczne czasy poszczególnych operacji słownikowych dla danych czterech typów list. Przyjmujemy, że operacje wykonywane są na listach o rozmiarach n.

Jeśli w implementacjach list będziemy dodatkowo utrzymywać atrybut *tail* wskazujący na ogon listy, to operację MAXIMUM dla list posortowanych możemy wykonywać w czasie stałym.

	Nieposortowana jedno-kierunkowa	Posortowana jedno-kierunkowa	Nieposortowana dwu-kierunkowa	Posortowana dwu-kierunkowa
SEARCH(L, k)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT(L, x)	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
DELETE(L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
SUCCESSOR(L, x)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
PREDECESSOR(L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
MINIMUM(L)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
MAXIMUM(L)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Tabela 4: Porównanie pesymistycznych złożoności operacji słownikowych dla różnych typów list.

10-2. Listowa reprezentacja kopców złączalnych

(a) Kopiec zaimplementujemy jako posortowaną listę jednokierunkową. Operacja MAKE-HEAP tworzy pustą listę, co zajmuje oczywiście czas stały. Dodanie elementu do kopca polega na dodaniu go do listy. Aby zachować jej uporządkowanie, musimy odnaleźć miejsce, które zajmie nowy element, co w pesymistycznym przypadku wymaga czasu $\Theta(n)$. Dzięki uporządkowaniu listy można zaimplementować operacje MINIMUM i EXTRACT-MIN działające w czasie $\Theta(1)$. Z kolei stosując algorytm opisany w zad. 6.5-8 w wersji dla list jednokierunkowych, a następnie przechodząc po scalonej liście w celu usunięcia powtarzających się elementów, jesteśmy w stanie zaimplementować operację UNION w czasie $\Theta(n)$, gdzie n jest liczbą elementów na wynikowej liście.

(b) W tym przypadku także wystarczy nam lista jednokierunkowa. Zarówno utworzenie pustego kopca, jak i dodanie do niego nowego elementu odbywa się w czasie stałym, ale odszukanie oraz usunięcie minimalnego elementu wiąże się z przeszukaniem całej listy, co zajmuje czas liniowy względem rozmiaru listy. Możemy jednak wprowadzić usprawnienie, dzięki któremu operacja MINIMUM będzie działać w czasie stałym. Będziemy mianowicie przechowywać minimalny element listy w jej głowie. Podczas dodawania nowego elementu wystarczy porównać ten element z głową listy (o ile istnieje) i jeśli stanowi on nowe minimum, umieścić go w głowie listy albo, w przeciwnym przypadku, tuż za nią. Operacja EXTRACT-MIN po usunięciu głowy nadal jednak musi przeszukać całą listę w celu odszukania aktualnego minimum i umieszczenia go w głowie listy.

Podczas operacji UNION, musimy pamiętać o tym, że łączone listy nie zawsze reprezentują rozłączne zbiory, a także o tym, że w głowie wynikowej listy powinien znaleźć się najmniejszy element z obu łączonych list. Efektywna implementacja spełniająca oba warunki będzie polegać na wywołaniu operacji UNION z punktu (a) po uprzednim posortowaniu obu list, co zrealizujemy za pomocą algorytmu sortowania przez scalanie w wersji dla list jednokierunkowych. Do zaimplementowania pomocniczej procedury MERGE możemy użyć algorytmu przedstawionego w zad. 6.5-8. Ostatecznie łączenie list jesteśmy w stanie zaimplementować w czasie $\Theta(n \lg n)$, gdzie n jest rozmiarem wynikowej listy.

(c) Przypadek jest identyczny z poprzednim, ale tym razem nie jest wymagane wykrywanie powtórzeń podczas operacji UNION. Łączenie list możemy więc zaimplementować jako ich konkatenację, to znaczy ustawienie głowy jednej listy jako elementu następującego po ogonie drugiej.

To, w jakiej kolejności skleimy listy, będzie zależeć od tego, której głowa przechowuje mniejszy element, co ma na celu spełnienie usprawnienia opisanego w poprzednim punkcie. Operację łączenia możemy zrealizować w czasie stałym, jeśli dla listy w tej reprezentacji kopca będziemy pamiętać wskaźnik na jej ogon.

Zestawienie czasów działania poszczególnych operacji w przypadkach pesymistycznych dla omawianych reprezentacji listowych kopców złączalnych przedstawiono w tabeli 5.

	Listy posortowane	Listy nieposortowane	Listy nieposortowane, rozłączne zbiory w UNION
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
UNION	$\Theta(n)$	$\Theta(n \lg n)$	$\Theta(1)$

Tabela 5: Porównanie pesymistycznych czasów operacji słownikowych dla reprezentacji listowych kopców złączalnych. Dla operacji UNION n oznacza rozmiar zbioru po połączeniu.

10-3. Wyszukiwanie na posortowanej liście zajmującej spójny obszar pamięci (liście upakowanej)

W pseudokodzie procedury COMPACT-LIST-SEARCH *jest błąd – w linijce 4 zamiast testowania, czy* $key[j] < k$*, powinno być sprawdzenie, czy* $key[j] \leqslant k$*. Ponadto w wywołaniach procedur* COMPACT-LIST-SEARCH *i* COMPACT-LIST-SEARCH′ *na listach ich argumentów brakuje* n *jako drugiego parametru.*

(a) W żadnej z procedur nigdy nie zostanie wykonany skok na losowo wybraną pozycję bliżej głowy listy ani na pozycję zawierającą element o kluczu większym niż k – gwarantują to warunki w wierszach 4 obu algorytmów. Ponadto w każdej iteracji w pętlach **while** obu procedur następuje przesunięcie indeksu i o jedną pozycję do przodu. Skoki te wykonywane są, dopóki indeks i nie dotrze na pozycję elementu o kluczu większym lub równym k, bądź nie osiągnie końca listy L. A zatem oba wywołania zwrócą ten sam wynik.

Zauważmy, że podczas ustalonej iteracji pętli **while** w procedurze COMPACT-LIST-SEARCH indeks i nie jest bliżej głowy listy L niż ten sam indeks podczas tej samej iteracji pętli **for** w procedurze COMPACT-LIST-SEARCH′. Pętla pierwszego algorytmu nie zakończy się więc później niż pętla drugiego algorytmu, która to wykona dokładnie t iteracji. A zatem łączna liczba iteracji pętli **for** i **while** w procedurze COMPACT-LIST-SEARCH′ wynosi co najmniej t.

(b) W wywołaniu COMPACT-LIST-SEARCH′(L, n, k, t) zostanie wykonanych nie więcej niż t iteracji pętli **for**, czyli etap ten wymaga czasu $O(t)$. Na podstawie definicji zmiennej losowej X_t otrzymujemy z kolei, że średnia liczba wykonanych iteracji pętli **while** wynosi $\mathrm{E}(X_t)$. A zatem oczekiwanym czasem działania procedury COMPACT-LIST-SEARCH′(L, n, k, t) jest $O(t + \mathrm{E}(X_t))$.

(c) Oznaczmy przez s indeks klucza k na liście L, a przez $j_1, j_2, \ldots, j_t$ – ciąg liczb całkowitych wyznaczonych przez t wywołań RANDOM$(1, n)$. Ponadto niech π będzie funkcją przypisującą

indeksowi elementu na liście L jego rzeczywistą pozycję na tej liście wyznaczoną na podstawie łańcucha wskaźników *next*. Po t iteracjach pętli **for** odległość między szukanym kluczem a elementem o indeksie i (mierzona długością łańcucha wskaźników *next*) będzie większa lub równa r, jeśli dla każdego $q = 1, 2, \ldots, t$ spełnione będzie $\pi(j_q) \leqslant \pi(s) - r$ lub $\pi(j_q) > \pi(s)$. Mamy więc

$$\Pr(X_t \geqslant r) = \prod_{q=1}^{t} \Pr(\pi(j_q) \leqslant \pi(s) - r \text{ lub } \pi(j_q) > \pi(s)) = \prod_{q=1}^{t} \frac{n-r}{n} = \left(1 - \frac{r}{n}\right)^t.$$

Korzystając z tożsamości (C.24) i z powyższego oszacowania, otrzymujemy

$$\mathrm{E}(X_t) = \sum_{r=1}^{\infty} \Pr(X_t \geqslant r) = \sum_{r=1}^{n} \Pr(X_t \geqslant r) = \sum_{r=1}^{n} \left(1 - \frac{r}{n}\right)^t.$$

(d) Sumę ograniczamy całką, otrzymując:

$$\sum_{r=0}^{n-1} r^t \leqslant \int_0^n x^t \, dx = \left[\frac{x^{t+1}}{t+1}\right]_0^n = \frac{n^{t+1}}{t+1}.$$

(e) Na podstawie rozwiązań punktów (c) i (d) mamy:

$$\mathrm{E}(X_t) = \sum_{r=1}^{n} \left(1 - \frac{r}{n}\right)^t = \sum_{r=1}^{n-1} \left(\frac{n-r}{n}\right)^t = \sum_{r=1}^{n-1} \left(\frac{r}{n}\right)^t = \frac{\sum_{r=0}^{n-1} r^t}{n^t} \leqslant \frac{\frac{n^{t+1}}{t+1}}{n^t} = \frac{n}{t+1}.$$

(f) Na mocy punktów (b) i (e) dostajemy, że oczekiwany czas działania procedury Compact-List-Search$'(L, n, k, t)$ wynosi $O(t + \mathrm{E}(X_t)) = O(t + n/(t+1)) = O(t + n/t)$.

(g) Oznaczmy przez p pozycję listy L, na którą został wykonany ostatni skok w wierszu 5 procedury Compact-List-Search. Zauważmy, że w procedurze Compact-List-Search po wykonaniu tego skoku zostanie wykonanych co najwyżej t operacji $i \leftarrow next[i]$. Z kolei pozycja na liście L, będąca celem ostatniego skoku z wiersza 5 w Compact-List-Search$'$, nie może znajdować się bliżej głowy listy niż pozycja p. Na tej podstawie wnioskujemy, że liczba wykonanych operacji $i \leftarrow next[i]$ w pętli **while** algorytmu Compact-List-Search$'$ również nie przekroczy t. Oznacza to, że etap, który na podstawie poprzedniego punktu zajmuje czas $O(n/t)$, będzie w rzeczywistości działać w czasie $O(t)$. Stąd $O(n/t) = O(t)$, czyli $t = O(\sqrt{n})$, a więc czas działania procedury Compact-List-Search wynosi $O(\sqrt{n})$.

Wynik ten jest prawdziwy także w przypadku, gdy elementu o kluczu k nie ma na liście. Wystarczy bowiem zdefiniować zmienną losową X_t jako odległość na liście (mierzoną długością łańcucha wskaźników *next*) od pozycji i do pozycji elementu o kluczu większym niż k albo do pozycji bezpośrednio za ogonem listy, w przypadku, gdy taki element nie istnieje. Analizę w punktach (b)–(g) dla nowej definicji zmiennej X_t przeprowadza się analogicznie.

(h) Jeśli dopuścimy, aby elementy na liście powtarzały się, to może dojść do sytuacji, w której procedura próbuje wykonać skok na pozycję j bliższą szukanemu elementowi niż pozycja i, ale nie wykonuje go, gdyż stwierdza w linii 4, że $key[i] = key[j]$. Jeśli każdy losowy skok będzie eliminowany na tej podstawie, to działanie procedury sprowadzi się do działania zwykłego algorytmu wyszukiwania na posortowanej liście.

Rozdział **11**

Tablice z haszowaniem

11.1. Tablice z adresowaniem bezpośrednim

11.1-1. Procedura wyszukująca największy element zbioru S będzie przeglądać liniowo tablicę T od końca, to znaczy od indeksu $m-1$ ku jej początkowi, zwracając element z pierwszej niepustej pozycji. Jeśli zbiór S jest pusty, to na wszystkich pozycjach tablicy T znajduje się wartość NIL i wówczas procedura przeglądnie całą tablicę, po czym zwróci NIL. Jest to przypadek pesymistyczny, który wymaga czasu $\Theta(m)$.

11.1-2. Elementy składają się tylko z kluczy, wystarczy więc pamiętać tylko, czy dany klucz należy do zbioru, czy nie. Na pozycji k wektora bitowego o długości m będzie znajdować się jedynka, jeśli element o kluczu k należy do zbioru i zero w przeciwnym przypadku. Wyszukiwanie elementu o kluczu k polega na odczytaniu k-tej pozycji wektora, dodanie tego elementu do zbioru – na wpisaniu na tę pozycję jedynki, a usunięcie – na wpisaniu tam zera.

11.1-3. W naszej implementacji elementy przechowywane w tablicy T, oprócz klucza i dodatkowych danych, zawierać będą wskaźniki *prev* i *next*, dzięki którym elementy będą mogły zostać uszeregowane w listy dwukierunkowe. Komórka o indeksie k w tablicy T będzie zawierać głowę listy elementów o kluczu k znajdujących się aktualnie w tablicy albo NIL, jeżeli lista ta będzie pusta.

Dodanie nowego elementu x będzie polegać na dodaniu go do listy znajdującej się w $T[key[x]]$. Ponieważ operacja DELETE przyjmuje wskaźnik do elementu x, a nie jego klucz, możliwe jest zrealizowanie jej poprzez natychmiastowe usunięcie elementu x z listy w $T[key[x]]$. Wreszcie działanie operacji wyszukującej element o kluczu k będzie polegać na zwróceniu wartości $T[k]$. Każda z opisanych operacji działa w czasie $\Theta(1)$.

Opisaną strukturę danych można traktować jak tablicę z haszowaniem z identycznością jako funkcją haszującą.

11.1-4. *Dodatkową strukturą danych, jaką należy użyć według wskazówki, powinna być tablica traktowana jak stos.*

Przez T oznaczymy ogromną tablicę, a przez S – korzystając ze wskazówki z treści zadania – dodatkową tablicę, za pomocą której będziemy odwoływać się do odpowiednich pozycji w T. Tablicę S będziemy traktować jak stos elementów przechowywanych w T i będziemy używać atrybutu $top[S]$ wskazującego na ostatnią zajętą komórkę w S. Rozmiar tablicy S, jaki ustalimy podczas jej tworzenia, określa pojemność ogromnej tablicy – próba umieszczenia w tej struk-

turze większej ilości elementów zakończy się błędem nadmiaru wykrywanym przez odpowiednią implementację operacji PUSH.

Tablice S i T weryfikują się wzajemnie. To znaczy, jeśli klucz k znajduje się w tablicy T, to $T[k]$ przechowuje poprawny indeks j tablicy S, a $S[j]$ zawiera element o kluczu k. Innymi słowy, zachodzą zależności: $1 \leqslant T[k] \leqslant top[S]$, $key[S[T[k]]] = k$ oraz $T[key[S[j]]] = j$.

Inicjowanie tablicy T sprowadza się do utworzenia tablicy S. Aby wyszukać element o kluczu k, należy sprawdzić, czy $1 \leqslant T[k] \leqslant top[S]$ i $key[S[T[k]]] = k$. Jeśli warunki te są spełnione, to zwrócony zostanie element $S[T[k]]$, w przeciwnym przypadku zaś NIL. W celu wstawienia elementu x (przy założeniu, że nie ma go jeszcze w tablicy) należy umieścić x na stosie S, po czym zaktualizować $T[key[x]]$:

HUGE-ARRAY-INSERT(S, T, x)
1 PUSH(S, x)
2 $T[key[x]] \leftarrow top[S]$

Usuwanie elementu x o kluczu k polega na przeniesieniu elementu usuniętego ze szczytu stosu S na pozycję $T[k]$ w tym stosie oraz aktualizacji odpowiedniej wartości w tablicy T. Dokładniej, operacja ta sprowadza się do poniższego pseudokodu:

HUGE-ARRAY-DELETE(S, T, x)
1 $k \leftarrow key[x]$
2 $y \leftarrow$ POP(S)
3 $S[T[k]] \leftarrow y$
4 $T[key[y]] \leftarrow T[k]$

Wszystkie omówione operacje działają w czasie $\Theta(1)$, a każdy element znajdujący się w ogromnej tablicy (nie licząc dodatkowych danych z nim związanych) zajmuje $\Theta(1)$ pamięci.

11.2. Tablice z haszowaniem

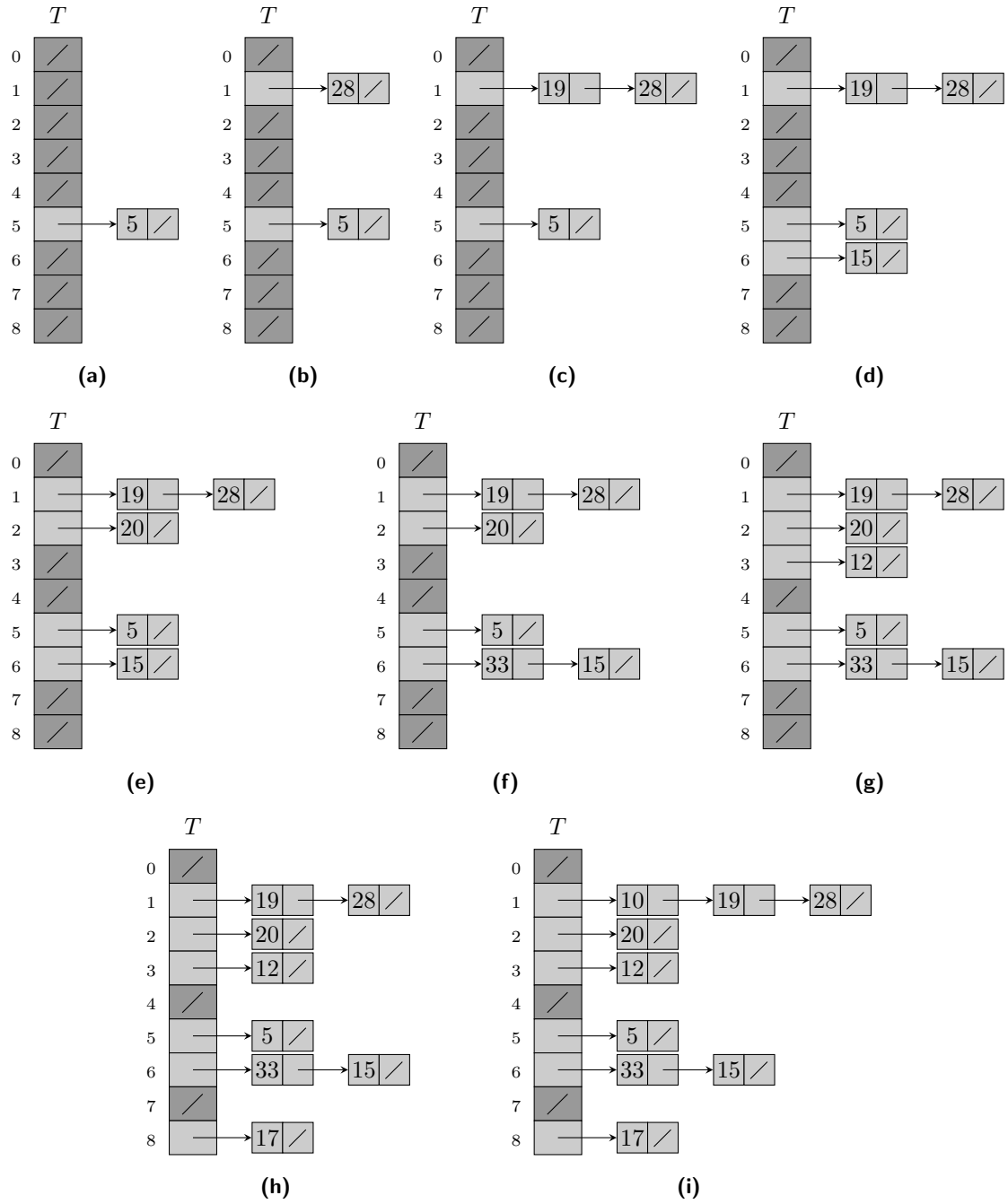
11.2-1. Dla kluczy k, l $(k \neq l)$, definiujemy zmienną losową $X_{kl} = \mathrm{I}(h(k) = h(l))$. Przy założeniu o prostym równomiernym haszowaniu mamy $\Pr(h(k) = h(l)) = 1/m$ i na podstawie lematu 5.1 otrzymujemy $\mathrm{E}(X_{kl}) = 1/m$. Niech X będzie zmienną losową oznaczającą liczbę kolizji w tablicy T, czyli

$$X = \sum_{k=1}^{n-1} \sum_{l=k+1}^{n} X_{kl}.$$

Oczekiwana liczba kolizji wynosi

$$\begin{aligned} \mathrm{E}(X) &= \mathrm{E}\left(\sum_{k=1}^{n-1} \sum_{l=k+1}^{n} X_{kl}\right) = \sum_{k=1}^{n-1} \sum_{l=k+1}^{n} \mathrm{E}(X_{kl}) \\ &= \sum_{k=1}^{n-1} \sum_{l=k+1}^{n} \frac{1}{m} = \frac{1}{m} \sum_{k=1}^{n-1} (n-k) = \frac{1}{m} \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2m}. \end{aligned}$$

11.2-2. Ciąg wstawień elementów do tablicy z haszowaniem został zobrazowany na rys. 23.



Rysunek 23: Ilustracja wstawiania do tablicy z haszowaniem T elementów o kluczach 5, 28, 19, 15, 20, 33, 12, 17, 10. Do rozwiązywania kolizji używana jest metoda łańcuchowa.

11.2-3. Zakładamy, że listy są dwukierunkowe i wartości funkcji haszującej można wyznaczać w czasie stałym. Na podstawie wyników z problemu 10-1 usuwanie elementu z tablicy z haszowa-

niem będzie działać w czasie $\Theta(1)$. Aby wstawić nowy element do tablicy, należy umieścić go na liście wskazanej przez funkcję haszującą, znajdując wpierw odpowiednie miejsce na tej liście, aby dodanie elementu nie zaburzyło jej uporządkowania. W pesymistycznym przypadku dla tablicy o n elementach zajmie to czas $\Theta(n)$, podobnie jak wyszukiwanie elementu, gdyż operacja ta także polega na liniowym przejrzeniu jednej z list.

11.2-4. *Treść zadania podaje opis tablicy z haszowaniem jako struktury heterogenicznej, czyli przechowującej obiekty różnego typu w zależności od zajętości pozycji danego obiektu. W rozwiązaniu podajemy zaś implementację tej struktury w postaci tablicy homogenicznej.*

Każda pozycja tablicy T w takiej reprezentacji będzie zawierać znacznik przechowujący informację o tym, czy pozycja ta jest wolna. Na zajętych pozycjach oprócz elementu będzie przechowywany wskaźnik (który oznaczymy przez *next*) do innej pozycji tablicy T lub wskaźnik pusty, dzięki czemu komórki te będą mogły formować listy jednokierunkowe. Wolne pozycje natomiast tworzyć będą listę dwukierunkową F, do zrealizowania której wykorzystane zostaną dwa wskaźniki na każdej wolnej pozycji – *prev* na poprzednik i *next* na następnik. Do zapamiętania, gdzie znajduje się głowa listy F, zostanie użyty dodatkowy wskaźnik będący atrybutem tablicy T. Wskaźnik ten oraz *prev* i *next* są tutaj tak naprawdę liczbami całkowitymi określającymi indeksy tablicy T, przy czym pusty wskaźnik (odpowiednik NIL) będziemy reprezentować wartością -1.

W celu zachowania homogeniczności tablicy T musimy sprawić, by pozycje wolne i zajęte składały się z tego samego zestawu atrybutów. Niech więc pole odpowiedzialne za przechowywanie elementu będzie wskaźnikiem na ten element. Dopuścimy obecność tego wskaźnika również na wolnych pozycjach – przyjmie on wtedy wartość NIL. Z kolei pole *prev* możemy wykorzystać w dwóch celach – do przechowywania indeksu poprzedniej pozycji na liście F, jak również jako znacznik określający zajętość aktualnej pozycji. Jeśli tablica T ma długość m, to na wolnych pozycjach pole *prev* może przyjąć wartości od 0 do $m-1$ oznaczające indeksy tablicy T oraz wartość -1 jako reprezentację wskaźnika pustego. Przyjmiemy natomiast, że na zajętych pozycjach pole to będzie miało wartość m i będzie to warunek rozstrzygający o zajętości danej pozycji.

Aby dodać do tablicy $T[0\,..\,m-1]$ element x o kluczu k, wpierw sprawdzamy, czy pozycja $h(k)$ w T jest wolna. Jeśli tak, to usuwamy ją z listy F, umieszczamy na niej element x, ustawiamy $prev[T[h(k)]]$ na m, a $next[T[h(k)]]$ – na -1. W przeciwnym razie na pozycji $h(k)$ znajduje się już inny element y o kluczu l. Wówczas usuwamy głowę listy F, przenosimy na nią element y wraz ze wskaźnikiem $next[T[h(k)]]$ i ustawiamy znacznik *prev* tej pozycji na m. Następnie do komórki $h(k)$ wstawiamy element x.

Pozostaje uaktualnić wskaźnik $next[T[h(k)]]$. W tym celu oznaczmy przez j indeks nowej pozycji elementu y i rozważmy dwie sytuacje. W pierwszej z nich na podstawie funkcji haszującej elementowi y odpowiada komórka $h(k)$, tzn. $h(l) = h(k)$. Wówczas wskaźnikiem $next[T[h(k)]]$ należy pokazać na pozycję j, gdyż x i y powinny należeć do tej samej listy zajętych pozycji. W drugim przypadku element y znajdował się na liście zajętych pozycji rozpoczynającej się na indeksie $h(l) \neq h(k)$. Musimy zatem przejść po tej liście i zmodyfikować ją tak, aby wskaźnik *next* poprzednika elementu y pokazywał teraz na pozycję j. Element x będzie wówczas jedyny na swojej liście – wystarczy więc wpisać do $next[T[h(k)]]$ wartość -1.

Ponieważ traktujemy zajęte pozycje tablicy T jak węzły list jednokierunkowych, to usuwanie elementu x o kluczu k polega na usunięciu pozycji, która go zawiera, z listy o głowie w $h(k)$. Należy pamiętać jeszcze o tym, aby wstawić zwolnioną pozycję na początek listy F i przestawić znacznik *prev* na -1. Z kolei wyszukiwanie elementu o kluczu k sprowadza się do przeglądnięcia listy o głowie w $h(k)$ i zwrócenia wskaźnika do takiego elementu (o ile istnieje) albo wartości NIL.

Zauważmy, że przechowywanie list zajętych pozycji bezpośrednio w tablicy nie zmienia czasów

działania operacji słownikowych w porównaniu z zastosowaniem metody łańcuchowej, w której listy te znajdują się poza tablicą. Ponadto w opisanej reprezentacji mamy zawsze $\alpha \leqslant 1$. Operacje INSERT, DELETE i SEARCH działają więc tutaj w oczekiwanym czasie $\Theta(1+\alpha) = \Theta(1)$. Aby go uzyskać, musieliśmy założyć, że lista wolnych pozycji F jest listą dwukierunkową – w przeciwnym przypadku bowiem nie można byłoby usuwać z niej w czasie stałym.

11.2-5. Funkcja haszująca przyjmuje m różnych wartości, a elementów zbioru U jest więcej niż nm. Z tego powodu istnieje taka wartość, która została przyporządkowana więcej niż n elementom ze zbioru U. Fakt ten wynika z nieco zmodyfikowanej wersji **zasady szufladkowej Dirichleta** [12] zwanej także **zasadą gołębnika**.

11.3. Funkcje haszujące

11.3-1. W celu odnalezienia elementu o kluczu k obliczamy $h(k)$ i przeglądamy listę, porównując jedynie wartości funkcji haszującej, co jest znacznie szybsze od porównywania kluczy (długich ciągów znaków). Gdy znajdziemy element, dla którego wartość funkcji haszującej jest równa $h(k)$, to dopiero wówczas sprawdzamy, czy kluczem tego elementu jest k.

11.3-2. Reprezentacją napisu $x = x_0x_1 \ldots x_{r-1}$ jest liczba $\sum_{i=0}^{r-1} x_i 128^i$ (dla uproszczenia zapisu utożsamiamy znak z odpowiadającą mu wartością w kodzie ASCII). Obliczenie $h(x)$ polega na wyznaczeniu reszty z dzielenia reprezentacji napisu x przez m. W tym celu przedstawimy $h(x)$ w postaci

$$h(x) = x_0 + 128(x_1 + 128(x_2 + \cdots + 128(x_{r-2} + 128x_{r-1}) \cdots)),$$

przy czym dodawanie i mnożenie są tutaj działaniami w zbiorze $\{0, 1, \ldots, m-1\}$. Następnie stosujemy schemat Hornera (patrz problem 2-3). Ciągłe obliczanie reszt z dzielenia przez m pozwala utrzymać pośrednie wyniki w co najwyżej dwóch słowach maszynowych.

11.3-3. Jeśli napis $x = x_0x_1 \ldots x_{r-1}$ jest permutacją napisu $y = y_0y_1 \ldots y_{r-1}$, to x możemy uzyskać z y, zamieniając ze sobą znaki występujące na tych samych pozycjach w obu napisach, dopóki oba napisy są różne. Można pokazać, że zawsze istnieje skończony ciąg pozycji, na których należy zamieniać znaki w celu przekształcenia jednego napisu w drugi. Założymy zatem, że napis x można uzyskać z y, dokonując tylko jednej takiej zamiany, tzn. $x_a = y_b$ oraz $y_a = x_b$ dla pewnych $0 \leqslant a < b \leqslant r-1$ oraz $x_i = y_i$ dla $i \neq a$ i $i \neq b$. Łatwo pokazać przez indukcję, że jeśli występuje kolizja dla takiej pary napisów, to ma ona miejsce również dla par napisów różniących się więcej niż jedną parą znaków.

Podobnie jak w poprzednim zadaniu będziemy traktować zamiennie znak oraz jego reprezentację w kodzie ASCII. Pokażemy, że $h(x) = h(y)$, czyli że różnica

$$h(x) - h(y) = \left(\sum_{i=0}^{r-1} x_i 2^{ip}\right) \bmod (2^p - 1) - \left(\sum_{i=0}^{r-1} y_i 2^{ip}\right) \bmod (2^p - 1)$$

jest zerem. Mamy:

$$\begin{aligned} h(x) - h(y) &= \big((x_a 2^{ap} + x_b 2^{bp}) - (y_a 2^{ap} + y_b 2^{bp})\big) \bmod (2^p - 1) \\ &= \big((y_b 2^{ap} + y_a 2^{bp}) - (y_a 2^{ap} + y_b 2^{bp})\big) \bmod (2^p - 1) \\ &= \big((y_a - y_b)2^{bp} - (y_a - y_b)2^{ap}\big) \bmod (2^p - 1) \end{aligned}$$

$$= \big((y_a - y_b)(2^{bp} - 2^{ap})\big) \bmod (2^p - 1)$$
$$= \big((y_a - y_b)2^{ap}(2^{(b-a)p} - 1)\big) \bmod (2^p - 1).$$

Ze wzoru (A.5) zachodzi

$$\sum_{i=0}^{b-a-1} 2^{pi} = \frac{2^{(b-a)p} - 1}{2^p - 1},$$

skąd

$$(2^p - 1) \sum_{i=0}^{b-a-1} 2^{pi} = 2^{(b-a)p} - 1$$

i ostatecznie

$$h(x) - h(y) = \left((y_a - y_b)2^{ap}(2^p - 1) \sum_{i=0}^{b-a-1} 2^{pi}\right) \bmod (2^p - 1) = 0,$$

ponieważ jeden z czynników jest równy $2^p - 1$.

Jeśli napisami będą np. sekwencje DNA, to użycie h jako funkcji haszującej może doprowadzić do powstania dużej ilości kolizji. Każda sekwencja DNA stanowi bowiem ciąg nukleotydów czterech typów (reprezentowanych jako symbole `A`, `C`, `G` i `T`), zatem prawdopodobieństwo, że jakaś sekwencja jest permutacją innej, jest stosunkowo duże.

11.3-4. Największą potęgą 2 nieprzekraczającą $m = 1000$ jest $2^9 = 512$, przyjmujemy zatem $p = 9$. Założymy, że słowo maszynowe ma długość $w = 32$ bity i przybliżymy stałą A wartością $s/2^w = 2654435769/2^{32}$. Wówczas:

- jeśli $k = 61$, to $k \cdot s = 161920581909 = 37 \cdot 2^{32} + 3006791957$, więc $h(61) = 358$;
- jeśli $k = 62$, to $k \cdot s = 164575017678 = 38 \cdot 2^{32} + 1366260430$, więc $h(62) = 162$;
- jeśli $k = 63$, to $k \cdot s = 167229453447 = 38 \cdot 2^{32} + 4020696199$, więc $h(63) = 479$;
- jeśli $k = 64$, to $k \cdot s = 169883889216 = 39 \cdot 2^{32} + 2380164672$, więc $h(64) = 283$;
- jeśli $k = 65$, to $k \cdot s = 172538324985 = 40 \cdot 2^{32} + 739633145$, więc $h(65) = 88$.

11.3-5. Wprowadźmy oznaczenia $u = |U|$ oraz $b = |B|$. Dla ustalonej funkcji haszującej $h \in \mathcal{H}$ i dla każdego elementu $j \in B$ niech u_j będzie liczbą elementów z U, które zostały odwzorowane na j przez funkcję h. Wówczas liczba kolizji powstałych na danym elemencie $j \in B$ wynosi $\binom{u_j}{2} = u_j(u_j - 1)/2$.

Pokażemy, że sumaryczna liczba kolizji generowanych przez funkcję h jest minimalna, gdy $u_j = u/b$ dla każdego $j \in B$. Załóżmy, że dla pewnych dwóch różnych i, $k \in B$ zachodzi $0 < u_i \leqslant u/b \leqslant u_k$. Łączna liczba kolizji na elementach i i k wynosi $S = u_i(u_i - 1)/2 + u_k(u_k - 1)/2$. Niech teraz h' będzie funkcją identyczną z h z jednym wyjątkiem – dla dokładnie jednego dowolnie wybranego elementu $x \in U$ takiego, że $h(x) = i$, niech zachodzi $h'(x) = k$. Funkcja h' tworzy $S' = (u_i - 1)(u_i - 2)/2 + (u_k + 1)u_k/2$ kolizji na elementach i i k. Zbadajmy znak różnicy $S' - S$:

$$\begin{aligned} S' - S &= (u_i - 1)(u_i - 2)/2 + (u_k + 1)u_k/2 - u_i(u_i - 1)/2 - u_k(u_k - 1)/2 \\ &= (u_i^2 - 3u_i + 2 + u_k^2 + u_k - u_i^2 + u_i - u_k^2 + u_k)/2 \\ &= (2u_k - 2u_i + 2)/2 \end{aligned}$$

$$\begin{aligned} &= u_k - u_i + 1 \\ &> 0. \end{aligned}$$

Otrzymaliśmy, że $S' > S$, czyli że sumaryczna liczba kolizji zwiększa się, kiedy wartości u_j odbiegają od u/b. Stąd wniosek, że liczba kolizji jest możliwie najmniejsza, gdy dla wszystkich elementów $j \in B$ zachodzi $u_j = u/b$.

Na podstawie powyższego faktu dostajemy, że dowolnie wybrana funkcja haszująca z $\mathcal{H}$ generuje łącznie co najmniej $b(u/b)(u/b-1)/2$ kolizji. Stąd prawdopodobieństwo $p_{\mathcal{H}}$ wystąpienia kolizji przy losowo wybranej funkcji z $\mathcal{H}$ można ograniczyć od dołu przez

$$p_{\mathcal{H}} \geqslant \frac{b(u/b)(u/b-1)/2}{u(u-1)/2} = \frac{u/b-1}{u-1} > \frac{u/b-1}{u} = \frac{1}{b} - \frac{1}{u}.$$

Jeśli rodzina $\mathcal{H}$ jest ϵ-uniwersalna, to $p_{\mathcal{H}} \leqslant \epsilon$, a stąd dostajemy $\epsilon > 1/b - 1/u = 1/|B| - 1/|U|$.

11.3-6. Niech $x = \langle x_0, x_1, \ldots, x_{n-1}\rangle$, $y = \langle y_0, y_1, \ldots, y_{n-1}\rangle$ będą dwiema różnymi n-tkami o wartościach z $\mathbb{Z}_p$. Dla losowo wybranej funkcji $h_b \in \mathcal{H}$ mamy:

$$h_b(x) - h_b(y) = \sum_{j=0}^{n-1} x_j b^j - \sum_{j=0}^{n-1} y_j b^j = \sum_{j=0}^{n-1} (x_j - y_j) b^j.$$

Kolizja między x a y wystąpi wówczas, gdy powyższa różnica będzie zerem. Wyrażenie po prawej stronie jest wielomianem stopnia $n-1$ zmiennej b o współczynnikach z $\mathbb{Z}_p$, a z zad. **??** mamy, że taki wielomian posiada co najwyżej $n-1$ różnych pierwiastków modulo p. A zatem co najwyżej $n-1$ spośród p funkcji z rodziny $\mathcal{H}$ spowoduje kolizję między x a y. Prawdopodobieństwo kolizji jest więc równe $(n-1)/p$, czyli rodzina $\mathcal{H}$ jest $((n-1)/p)$-uniwersalna według definicji z zad. 11.3-5.

11.4. Adresowanie otwarte

11.4-1. *Funkcja h' nie jest pierwotną funkcją haszującą, tylko pomocniczą funkcją haszującą. Zakładamy, że pierwszą pomocniczą funkcją haszującą w haszowaniu dwukrotnym jest $h_1 = h'$.*

Tabela 6 przedstawia pozycje tablicy, które są obliczane dla poszczególnych kluczy przy zastosowaniu różnych sposobów obliczania ciągów kontrolnych.

11.4-2. Pseudokod procedury HASH-DELETE przedstawiono poniżej:

```
HASH-DELETE(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3          if T[j] = k
4             then T[j] ← DELETED
5                  return
6          i ← i + 1
7     until T[j] = NIL lub i = m
```

W procedurze HASH-INSERT wystarczy zamienić warunek z wiersza 3 na następujący:

```
3  if T[j] = NIL lub T[j] = DELETED
```

	adresowanie liniowe	adresowanie kwadratowe	haszowanie dwukrotne
10	10	10	10
22	0	0	0
31	9	9	9
4	4	4	4
15	4, 5	4, 8	4, 10, 5
28	6	6	6
17	6, 7	6, 10, 9, 3	6, 3
88	0, 1	0, 4, 3, 8, 8, 3, 4, 0, 2	0, 9, 7
59	4, 5, 6, 7, 8	4, 8, 7	4, 3, 2

Tabela 6: Pozycje obliczane dla podanego ciągu kluczy w różnych metodach adresowania otwartego. Dany klucz trafia ostatecznie na pierwszą wolną pozycję ze swojego ciągu kontrolnego.

11.4-3. Dzięki skorzystaniu z tw. 31.20 otrzymujemy, że rzędem grupy generowanej przez $h_2(k)$ jest m/d, co oznacza, że w $\mathbb{Z}_m$ wyrażenie $ih_2(k)$ dla $i = 0, 1, \ldots, m-1$ przyjmuje m/d różnych wartości. Podobną własność wykazuje także suma $h_1(k) + ih_2(k)$. A zatem stosując funkcję haszującą h, podczas wyszukiwania klucza k, które zakończy się porażką, zostanie sprawdzonych m/d różnych komórek tablicy, co stanowi $(1/d)$-tą część tej tablicy.

11.4-4. Wykorzystując tw. 11.8, otrzymujemy, że dla współczynnika zapełnienia $\alpha = 1/2$ oczekiwana liczba porównań nie przekracza $2\ln 2 \approx 1{,}386$. Dla $\alpha = 3/4$ oszacowanie to wynosi $(4/3)\ln 4 \approx 1{,}848$, a dla $\alpha = 7/8$ jest ono równe $(8/7)\ln 8 \approx 2{,}377$.

11.4-5. Z tw. 11.6 i 11.8 dostajemy, że szukane $0 < \alpha < 1$ spełnia równanie

$$\frac{1}{1-\alpha} = \frac{2}{\alpha}\ln\frac{1}{1-\alpha}.$$

Niech $\beta = 1/(1-\alpha)$. Wówczas $\alpha = 1 - 1/\beta$ i powyższy wzór przyjmuje postać

$$\beta = \frac{2\beta}{\beta - 1}\ln\beta,$$

co jest równoważne

$$\beta - 2\ln\beta = 1.$$

Mamy dalej:

$$\begin{aligned} e^{\beta - 2\ln\beta} &= e, \\ \beta^{-2}e^{\beta} &= e, \\ \beta e^{-\beta/2} &= e^{-1/2}, \\ (-\beta/2)e^{-\beta/2} &= -e^{-1/2}/2. \end{aligned}$$

Skorzystamy teraz z **funkcji** W **Lamberta** [1] zdefiniowanej jako wielowartościowe odwzorowanie odwrotne do funkcji $f(x) = xe^x$. Innymi słowy, dla każdej liczby rzeczywistej $x \geqslant -1/e$ spełniony jest wzór

$$x = W(x)e^{W(x)}.$$

Nasze równanie sprowadza się zatem do

$$W(-e^{-1/2}/2) = -\beta/2,$$

skąd

$$\beta = -2W(-e^{-1/2}/2).$$

Dla argumentu $-e^{-1/2}/2$ odwzorowanie W przyjmuje dwie wartości. Jedną z nich jest oczywiście $-1/2$. Ale wówczas $\beta = 1$ i $\alpha = 0$, co jest sprzeczne z założeniem. Drugą wartość $W(-e^{-1/2}/2)$ można uzyskać, stosując metody numeryczne – wynosi ona w przybliżeniu $-1{,}756$, skąd $\beta \approx 3{,}513$ i $\alpha \approx 0{,}715$.

11.5. Haszowanie doskonałe

11.5-1. *W treści zadania zamiast haszowania uniwersalnego powinno być haszowanie równomierne.*

Rozważmy prawdopodobieństwo $q(n, m)$, że wystąpi przynajmniej jedna kolizja. Jest $\binom{n}{2}$ par kluczy, które mogą tworzyć kolizję z prawdopodobieństwem $1/m$ każda. Mamy więc

$$q(n, m) = \binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m}$$

i teraz, dzięki skorzystaniu ze wzoru (3.11), otrzymujemy

$$p(n, m) = 1 - q(n, m) = 1 - \frac{n(n-1)}{2m} \leqslant e^{-n(n-1)/2m}.$$

W drugiej części zadania rozważymy funkcję $f(x) = e^{-x(x-1)/2m}$ zmiennej rzeczywistej x, traktując m jak stałą dodatnią. Jak łatwo zauważyć, granicą tej funkcji w ∞ jest 0. Wyznaczając jej pierwszą i drugą pochodną, mamy

$$\frac{df}{dx}(x) = e^{-x(x-1)/2m}\,\frac{1-2x}{2m}$$

oraz

$$\frac{d^2f}{dx^2}(x) = e^{-x(x-1)/2m}\left(\frac{1-2x}{2m}\right)^2 + e^{-x(x-1)/2m}\cdot\frac{-1}{m} = e^{-x(x-1)/2m}\,\frac{4x^2-4x+1-4m}{4m^2}.$$

Czynnik $e^{-x(x-1)/2m}$ jest dodatni niezależnie od wartości x i m. Jeśli $x > \sqrt{m}$, to natychmiast widać, że pierwsza pochodna funkcji f jest ujemna. Do tego samego wniosku dochodzimy dla drugiej pochodnej, ograniczając jej drugi czynnik:

$$\frac{4x^2-4x+1-4m}{4m^2} < \frac{4m-4\sqrt{m}+1-4m}{4m^2} = \frac{-4\sqrt{m}+1}{4m^2} < 0.$$

A zatem dla $x > \sqrt{m}$ funkcja f jest malejąca i wklęsła. Oznacza to, że prawdopodobieństwo $p(n, m)$, które wynosi co najwyżej $f(n)$, maleje gwałtownie do zera, gdy n przekracza $\sqrt{m}$.

Problemy

11-1. Szacowanie najdłuższego ciągu odwołań do tablicy z haszowaniem

W punkcie (b) należy wykazać, że prawdopodobieństwo opisanego tam zdarzenia wynosi $O(1/n^2)$. *W treści (także oryginalnej) brakuje założenia o równomiernym haszowaniu wymaganego do pokazania tego oszacowania. W paragrafie między częścią (b) a (c) powinno być* $\Pr(X_i > 2\lg n) = O(1/n^2)$. *W punkcie (c) należy wykazać, że* $\Pr(X > 2\lg n) = O(1/n)$. *Ponadto użyto błędnych liter do oznaczenia punktów (c) i (d).*

(a) Niech X_i będzie zmienną losową oznaczającą liczbę odwołań do tablicy wykonywanych podczas wstawiania i-tego elementu. Operacja ta jest poprzedzona wyszukiwaniem tego elementu w tablicy z negatywnym skutkiem. Dzięki założeniu o równomiernym haszowaniu możemy zatem skorzystać z dowodu tw. 11.6, z którego wynika, że $\Pr(X_i > k) = \Pr(X_i \geqslant k+1) \leqslant \alpha^k$. Ponieważ $n \leqslant m/2$, to $\alpha \leqslant 1/2$, a stąd $\Pr(X_i > k) \leqslant (1/2)^k = 2^{-k}$.

(b) Wynik otrzymujemy natychmiast po podstawieniu $k = 2\lg n$ do oszacowania z poprzedniego punktu.

(c) Oznaczmy przez A zdarzenie, że $X > 2\lg n$, a przez A_i, dla $i = 1, 2, \ldots, n$, zdarzenie, że $X_i > 2\lg n$. Zauważmy, że $A = \bigcup_{i=1}^{n} A_i$. Wykorzystując nierówność Boole'a (zad. C.2-1) oraz wynik z punktu (b), w którym pokazaliśmy, że $\Pr(A_i) = O(1/n^2)$, otrzymujemy

$$\Pr(A) = \Pr\left(\bigcup_{i=1}^{n} A_i\right) \leqslant \sum_{i=1}^{n} \Pr(A_i) = n \cdot O(1/n^2) = O(1/n).$$

(d) Na podstawie oszacowania z poprzedniej części otrzymujemy

$$\begin{aligned}
\mathrm{E}(X) &= \sum_{k=1}^{n} k \Pr(X = k) \\
&= \sum_{k=1}^{\lfloor 2\lg n \rfloor} k \Pr(X = k) + \sum_{k=\lfloor 2\lg n \rfloor + 1}^{n} k \Pr(X = k) \\
&\leqslant \sum_{k=1}^{\lfloor 2\lg n \rfloor} \lfloor 2\lg n \rfloor \Pr(X = k) + \sum_{k=\lfloor 2\lg n \rfloor + 1}^{n} n \Pr(X = k) \\
&= \lfloor 2\lg n \rfloor \Pr(X \leqslant 2\lg n) + n \Pr(X > 2\lg n) \\
&= \lfloor 2\lg n \rfloor \cdot 1 + n \cdot O(1/n) \\
&= \lfloor 2\lg n \rfloor + O(1) \\
&= O(\lg n).
\end{aligned}$$

11-2. Długość listy w metodzie łańcuchowej

(a) Potraktujmy odwzorowywanie kluczy jako ciąg prób Bernoulliego, w których sukcesem jest odwzorowanie klucza na ustaloną pozycję tablicy. Prawdopodobieństwo sukcesu każdej takiej próby jest równe $1/n$. Z rozkładu dwumianowego wynika, że uzyskanie dokładnie k sukcesów

w tej serii prób jest równe

$$Q_k = b(k; n, 1/n) = \binom{n}{k}\left(\frac{1}{n}\right)^k\left(1-\frac{1}{n}\right)^{n-k}.$$

(b) Niech A_i, dla $i = 1, 2, \ldots, n$, oznacza zdarzenie, że liczba elementów odwzorowanych na i-tą pozycję tablicy wynosi k. Wówczas zdarzenie A, że $M = k$, spełnia inkluzję $A \subseteq \bigcup_{i=1}^{n} A_i$. Z własności prawdopodobieństwa i z punktu (a) mamy

$$P_k = \Pr(A) \leqslant \Pr\left(\bigcup_{i=1}^{n} A_i\right) \leqslant \sum_{i=1}^{n} \Pr(A_i) = \sum_{i=1}^{n} Q_k = nQ_k.$$

(c) Zauważmy, że dla całkowitych n, k spełniających $0 < k \leqslant n$ zachodzi

$$(n-k+1)(n-k+2)\ldots n(n-1)^{n-k} < n^n,$$

gdyż po lewej stronie jest iloczyn n czynników nieprzekraczających n. Mnożąc tę nierówność obustronnie przez $(n-k)!$, dostajemy

$$n!\,(n-1)^{n-k} < (n-k)!\,n^n.$$

Ponadto ze wzoru Stirlinga dla całkowitego $k > 0$ wynika

$$\frac{1}{k!} = \frac{e^k}{k^k\sqrt{2\pi k}\,(1+\Theta(1/k))} < \frac{e^k}{k^k}.$$

Jeśli $k = 0$, to oczywiście

$$Q_0 = \left(1-\frac{1}{n}\right)^n < 1 = \frac{e^0}{0^0}.$$

Załóżmy teraz, że $k > 0$. Wykorzystując nierówności z poprzedniego paragrafu, mamy

$$Q_k = \frac{n!}{k!\,(n-k)!}\left(\frac{1}{n}\right)^k\left(1-\frac{1}{n}\right)^{n-k} = \frac{n!\,(n-1)^{n-k}}{k!\,(n-k)!\,n^n} < \frac{1}{k!} < \frac{e^k}{k^k}.$$

(d) *W drugiej części zadania należy wywnioskować, że* $P_k < 1/n^2$ *dla* $k \geqslant k_0 = c\lg n/\lg\lg n$.

Badając wyrażenie $\lg Q_{k_0}$, wyznaczymy odpowiednie c tak, aby zachodziło $Q_{k_0} < 1/n^3$, gdzie $k_0 = c\lg n/\lg\lg n$. Z poprzedniego punktu mamy, że $Q_{k_0} < e^{k_0}/{k_0}^{k_0}$, a więc

$$\begin{aligned}
\lg Q_{k_0} &< k_0\lg e - k_0\lg k_0 \\
&= \frac{c\lg n\lg e}{\lg\lg n} - \frac{c\lg n}{\lg\lg n}\lg\frac{c\lg n}{\lg\lg n} \\
&= \frac{c\lg n\lg e}{\lg\lg n} - \frac{c\lg n(\lg c + \lg\lg n - \lg\lg\lg n)}{\lg\lg n} \\
&= \frac{c\lg n(\lg e - \lg c)}{\lg\lg n} + \frac{c\lg n\lg\lg\lg n}{\lg\lg n} - c\lg n.
\end{aligned}$$

Niech $c \geqslant 3 + c'$ dla pewnej nowej stałej $c' \geqslant 0$. Wówczas pierwszy składnik powyższej sumy jest ujemny. Ponadto, jeśli c' jest wystarczająco duże, to dla każdego n zachodzi

$$\frac{(3+c')\lg\lg\lg n}{\lg\lg n} - c' \leqslant 0.$$

Dla uproszczenia zapisu przyjmijmy $m = \lg\lg\lg n$. Aby powyższa nierówność była prawdziwa, musi zachodzić $(3 + c')m \leqslant c'2^m$, skąd

$$c' \geqslant \frac{3m}{2^m - m}.$$

Można pokazać, że wyrażenie po prawej stronie jest mniejsze niż 4 niezależnie od wartości m, możemy zatem przyjąć $c' \geqslant 4$.

Powracając teraz do głównego oszacowania, mamy

$$\lg Q_{k_0} < \left(\frac{(3 + c')\lg\lg\lg n}{\lg\lg n} - c' - 3\right)\lg n \leqslant -3\lg n,$$

skąd $Q_{k_0} < 2^{-3\lg n} = 1/n^3$, o ile wybierzemy $c \geqslant 7$.

Aby udowodnić, że $P_k < 1/n^2$ dla wszystkich $k \geqslant k_0$, wykorzystamy część (b), w której pokazaliśmy, że $P_k \leqslant nQ_k$ dla dowolnego k. Dla $k = k_0$ mamy $P_{k_0} \leqslant nQ_{k_0} = n \cdot 1/n^3 = 1/n^2$. Pokażemy teraz, że dobierając odpowiednią stałą c, możemy spełnić nierówność $Q_k < 1/n^3$ dla każdego $k \geqslant k_0$ i na tej podstawie wywnioskujemy, że $P_k < 1/n^2$ dla każdego $k \geqslant k_0$. Wybierzmy wystarczająco dużą stałą c tak, aby $k_0 > e$. Wówczas $e/k < 1$ dla wszystkich $k \geqslant k_0$ i wyrażenie $(e/k)^k$ maleje wraz ze wzrostem k. Mamy

$$Q_k < e^k/k^k \leqslant e^{k_0}/k_0^{k_0} < 1/n^3,$$

a zatem badane oszacowanie jest spełnione.

(e) Niech $k_0 = c\lg n/\lg\lg n$. Wówczas

$$\begin{aligned}\mathrm{E}(M) &= \sum_{k=0}^{n} kP_k = \sum_{k=0}^{k_0} kP_k + \sum_{k=k_0+1}^{n} kP_k \\ &\leqslant k_0 \sum_{k=0}^{k_0} P_k + n \sum_{k=k_0+1}^{n} P_k = k_0 \Pr(M \leqslant k_0) + n \Pr(M > k_0).\end{aligned}$$

Aby pokazać, że $\mathrm{E}(M) = O(\lg n/\lg\lg n)$, wykorzystamy fakt, który udowodniliśmy w punkcie (d), że $P_k < 1/n^2$ dla $k \geqslant k_0$. Mamy

$$\begin{aligned}\mathrm{E}(M) &\leqslant k_0 \Pr(M \leqslant k_0) + n \Pr(M > k_0) = k_0 \Pr(M \leqslant k_0) + n \sum_{k=k_0+1}^{n} P_k \\ &< k_0 \cdot 1 + n \sum_{k=k_0+1}^{n} 1/n^2 < k_0 \cdot 1 + n^2 \cdot 1/n^2 = k_0 + 1 = O(\lg n/\lg\lg n).\end{aligned}$$

11-3. Adresowanie kwadratowe

Krok 3 opisanego algorytmu wyszukiwania powinien mieć następującą treść:

3. *Wykonaj $j \leftarrow j+1$. Jeśli $j = m$, to tablica jest pełna, więc zakończ wyszukiwanie. W przeciwnym przypadku wykonaj $i \leftarrow (i + j) \bmod m$, a następnie wróć do kroku 2.*

(a) Dla klucza k algorytm ten, o ile wcześniej nie zostanie przerwany, odwołuje się kolejno do pozycji $h(k)$, $(h(k)+1) \bmod m$, $(h(k)+1+2) \bmod m$, ..., $\left(h(k)+\sum_{j=1}^{m-1} j\right) \bmod m$. Stąd mamy, że i-tą sprawdzaną pozycją tablicy ($i = 0, 1, \ldots, m-1$) jest

$$\left(h(k)+\sum_{j=0}^{i} j\right) \bmod m = \left(h(k)+\frac{i(i+1)}{2}\right) \bmod m = \left(h(k)+\frac{i}{2}+\frac{i^2}{2}\right) \bmod m.$$

Jest to zatem przykład adresowania kwadratowego, w którym $c_1 = c_2 = 1/2$.

(b) Niech $h'(k,i) = (h(k) + i/2 + i^2/2) \bmod m$. Dowód sprowadza się do pokazania, że wyrazy ciągu $\langle h'(k,0), h'(k,1), \ldots, h'(k,m-1)\rangle$ dla dowolnego klucza k są parami różne.

Załóżmy nie-wprost, że istnieje klucz k oraz liczby całkowite i, j spełniające $0 \leqslant i < j < m$ takie, że $h'(k,i) = h'(k,j)$. Wówczas

$$h(k) + i(i+1)/2 \equiv h(k) + j(j+1)/2 \pmod m,$$

co daje

$$j(j+1)/2 - i(i+1)/2 \equiv 0 \pmod m.$$

Na mocy tożsamości $j(j+1)/2 - i(i+1)/2 = (j-i)(j+i+1)/2$ dostajemy

$$(j-i)(j+i+1)/2 \equiv 0 \pmod m.$$

Z ostatniego wzoru wynika, że istnieje całkowite r, dla którego zachodzi $(j-i)(j+i+1) = 2rm$. Przy założeniu, że m jest potęgą 2, $m = 2^p$, sprowadza się to do postaci $(j-i)(j+i+1) = r2^{p+1}$. Nietrudno zauważyć, że tylko jeden z czynników, $j-i$ albo $j+i+1$, jest parzysty, zatem 2^{p+1} dzieli tylko jeden z nich. Nie może nim być $j-i$, gdyż $j-i < m < 2^{p+1}$. Ale czynnik $j+i+1$ również nie dzieli się przez 2^{p+1}, bo $j+i+1 \leqslant (m-1)+(m-2)+1 = 2m-2 < 2^{p+1}$. Otrzymana sprzeczność prowadzi do wniosku, że $h'(k,i) \neq h'(k,j)$.

11-4. Haszowanie k-uniwersalne i uwierzytelnianie

Poprawki wprowadzone w angielskiej treści tego problemu okazały się na tyle znaczące, że problem został napisany od nowa. Poniżej prezentujemy polskie tłumaczenie jego nowej wersji.

Niech $\mathcal{H}$ będzie rodziną funkcji haszujących, które odwzorowują uniwersum kluczy U w zbiór $\{0, 1, \ldots, m-1\}$. Powiemy, że $\mathcal{H}$ jest **k-uniwersalna**, jeśli dla każdego ustalonego ciągu k różnych kluczy $\langle x^{(1)}, x^{(2)}, \ldots, x^{(k)}\rangle$ oraz funkcji h wybranej losowo z $\mathcal{H}$ ciąg $\langle h(x^{(1)}), h(x^{(2)}), \ldots, h(x^{(k)})\rangle$ jest z jednakowym prawdopodobieństwem równy dowolnemu spośród m^k ciągów k elementów ze zbioru $\{0, 1, \ldots, m-1\}$.

(a) Wykaż, że jeśli rodzina funkcji haszujących $\mathcal{H}$ jest 2-uniwersalna, to jest uniwersalna.

(b) Załóżmy, że U jest zbiorem n-tek o wartościach z $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, gdzie p jest liczbą pierwszą. Rozważmy element $x = \langle x_0, x_1, \ldots, x_{n-1}\rangle \in U$. Dla każdej n-tki $a = \langle a_0, a_1, \ldots, a_{n-1}\rangle \in U$ definiujemy funkcję haszującą h_a jako

$$h_a(x) = \left(\sum_{j=0}^{n-1} a_j x_j\right) \bmod p.$$

Udowodnij, że rodzina $\mathcal{H} = \{h_a\}$ jest uniwersalna, ale nie 2-uniwersalna. (*Wskazówka:* Znajdź klucz, dla którego wszystkie funkcje z $\mathcal{H}$ przyjmują tę samą wartość.)

(c) Załóżmy, że zmodyfikowaliśmy nieco rodzinę $\mathcal{H}$ z punktu (b): dla każdego $a \in U$ i każdego $b \in \mathbb{Z}_p$ definiujemy

$$h'_{a,b}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b\right) \bmod p$$

oraz $\mathcal{H}' = \{h'_{a,b}\}$. Udowodnij, że rodzina $\mathcal{H}'$ jest 2-uniwersalna. (*Wskazówka:* Rozważ ustalone $x \in U$ i $y \in U$ spełniające $x_i \neq y_i$ dla pewnego i. Co dzieje się z $h'_{a,b}(x)$ i $h'_{a,b}(y)$, gdy a_i i b przyjmują poszczególne wartości z $\mathbb{Z}_p$?)

(d) Przypuśćmy, że Alicja i Bob uzgodnili w sekrecie funkcję haszującą h z 2-uniwersalnej rodziny funkcji haszujących $\mathcal{H}$. Każda funkcja $h \in \mathcal{H}$ odwzorowuje uniwersum kluczy U w $\mathbb{Z}_p$, gdzie p jest liczbą pierwszą. Następnie Alicja przesyła do Boba przez Internet komunikat $m \in U$. Alicja uwierzytelnia komunikat, przesyłając dodatkowo znacznik $t = h(m)$, a Bob sprawdza, czy para $\langle m, t\rangle$, którą otrzymuje, faktycznie spełnia $t = h(m)$. Załóżmy, że przeciwnik przechwytuje przesyłaną parę $\langle m, t\rangle$ i próbuje oszukać Boba, zamieniając ją na inną parę $\langle m', t'\rangle$. Wykaż, że prawdopodobieństwo, iż przeciwnikowi uda się oszukać Boba i że zaakceptuje on parę $\langle m', t'\rangle$, wynosi co najwyżej $1/p$, niezależnie od tego, jak wielką mocą obliczeniową przeciwnik dysponuje, i nawet wówczas, gdy przeciwnik zna rodzinę funkcji haszujących $\mathcal{H}$.

Poniżej znajduje się rozwiązanie nowej wersji problemu.

(a) Niech $\mathcal{H}$ będzie 2-uniwersalną rodziną funkcji haszujących oraz niech $\langle x, y\rangle$ będzie parą różnych kluczy z U. Wówczas, dla losowo wybranej funkcji haszującej $h_a \in \mathcal{H}$, para $\langle h(x), h(y)\rangle$ jest z jednakowym prawdopodobieństwem dowolną spośród m^2 par o elementach ze zbioru $\{0, 1, \ldots, m-1\}$. A zatem kolizja, czyli zdarzenie, że $h(x) = h(y)$, wystąpi z prawdopodobieństwem $1/m$. Rodzina $\mathcal{H}$ jest więc uniwersalna.

(b) Aby zbadać 2-uniwersalność, wykorzystamy wskazówkę. Dla $x = \langle 0, 0, \ldots, 0\rangle$ wszystkie funkcje haszujące z $\mathcal{H}$ dają w wyniku 0, więc dla dowolnej funkcji $h \in \mathcal{H}$ i dowolnej pary $\langle x, y\rangle$ różnych kluczy z U nigdy nie otrzymamy pary $\langle h_a(x), h_a(y)\rangle$, której pierwszym elementem jest liczba różna od zera. To wyklucza 2-uniwersalność rodziny $\mathcal{H}$.

Udowodnimy teraz, że rodzina $\mathcal{H}$ jest uniwersalna. Wybierzmy w tym celu dowolną parę $\langle x, y\rangle$ różnych kluczy z U i pewną funkcję h_a z $\mathcal{H}$. Bez utraty ogólności przyjmijmy, że $x_0 \neq y_0$. Kolizja $h_a(x) = h_a(y)$ wystąpi tylko wtedy, gdy suma $\sum_{j=0}^{n-1} a_j x_j$ będzie dawać taką samą resztę z dzielenia przez p, co suma $\sum_{j=0}^{n-1} a_j y_j$, lub równoważnie, kiedy spełniony będzie poniższy wzór:

$$\sum_{j=0}^{n-1} a_j (x_j - y_j) \equiv 0 \pmod{p}.$$

Niech $S = \sum_{j=1}^{n-1} a_j (x_j - y_j)$. Wówczas wzór przyjmuje postać

$$a_0 (x_0 - y_0) \equiv -S \pmod{p},$$

którą potraktujemy jak modularne równanie liniowe zmiennej a_0. Ponieważ $x_0 \neq y_0$, a p jest liczbą pierwszą, to $\gcd(x_0 - y_0, p) = 1$ i na podstawie wniosku 31.25 a_0 jest wyznaczone jednoznacznie modulo p. A zatem dla ustalonych $a_1, a_2, \ldots, a_{n-1}$ istnieje dokładnie jedno a_0, dla którego funkcja h_a, gdzie $a = \langle a_0, a_1, \ldots, a_{n-1}\rangle$, generuje kolizję między x a y. To oznacza, że dokładnie p^{n-1} spośród p^n funkcji w $\mathcal{H}$ doprowadzi do kolizji. Prawdopodobieństwo tego zdarzenia, przy założeniu, że funkcja h_a jest wybrana losowo z $\mathcal{H}$, wynosi $1/p$, co kończy dowód.

(c) Ustalmy parę $\langle x, y\rangle$ różnych kluczy z U i wybierzmy pewną funkcję $h'_{a,b}$ z $\mathcal{H}'$. Bez utraty ogólności przyjmijmy, że $x_0 \neq y_0$. Wprowadźmy oznaczenia $\alpha = h'_{a,b}(x)$, $\beta = h'_{a,b}(y)$ oraz $X = \sum_{j=1}^{n-1} a_j x_j$, $Y = \sum_{j=1}^{n-1} a_j y_j$. Zachodzi $\alpha = (a_0 x_0 + X + b) \bmod p$ oraz $\beta = (a_0 y_0 + Y + b) \bmod p$. Zauważmy, że aby wygenerować każdą możliwą parę $\langle \alpha, \beta\rangle$, wystarczy abyśmy byli w stanie wygenerować dowolne $\alpha - \beta$ i dowolne β. Mamy $\alpha - \beta = (a_0(x_0 - y_0) + X - Y) \bmod p$, skąd

$$a_0(x_0 - y_0) \equiv \alpha - \beta - X + Y \pmod{p}.$$

Ustalmy dowolną wartość wyrażenia $\alpha - \beta$ i potraktujmy powyższy wzór jak modularne równanie liniowe zmiennej a_0. Oczywiście $\gcd(x_0 - y_0, p) = 1$, więc na podstawie wniosku 31.25 a_0 jest wyznaczone jednoznacznie modulo p. Istnieje zatem jednoznaczna odpowiedniość między wartością a_0 a wartością $\alpha - \beta$. Mając ustalone a_0 i dobierając różne wartości dla b, możemy z kolei wygenerować każde β. Jest dokładnie p^2 możliwych par $\langle \alpha, \beta\rangle$ i tyleż samo możliwości wyboru a_0 i b. Stąd wnioskujemy, że każda para $\langle \alpha, \beta\rangle$ jest jednoznacznie generowana przez odpowiednie a_0 i b. Istnieje zatem p^{n-1} funkcji $h'_{a,b} \in \mathcal{H}'$, które generują zadaną parę $\langle \alpha, \beta\rangle$. Wnioskujemy stąd, że uzyskanie każdej takiej pary jest jednakowo prawdopodobne, gdy funkcja $h'_{a,b}$ jest wybrana losowo z $\mathcal{H}'$, a to oznacza, że rodzina $\mathcal{H}'$ jest 2-uniwersalna.

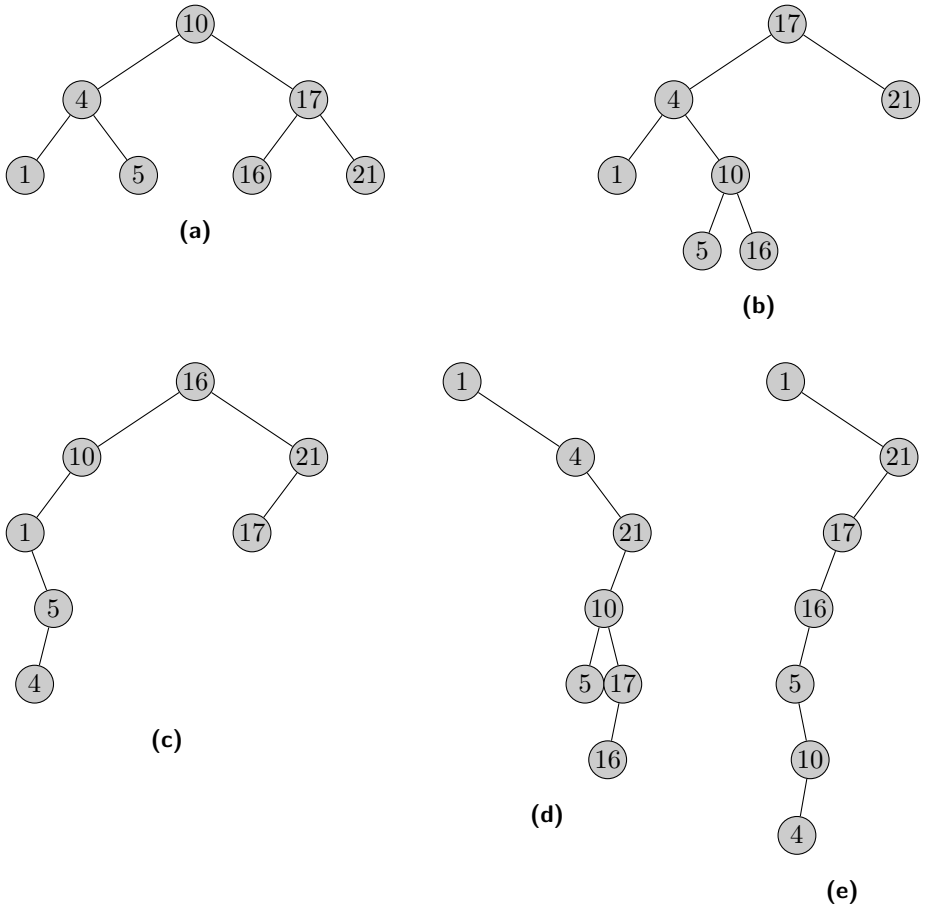
(d) Ponieważ rodzina $\mathcal{H}$ jest 2-uniwersalna, to dla każdej pary kluczy $\langle m, m'\rangle$, w której $m \neq m'$, uzyskanie dowolnej pary wartości $\langle h(m), h(m')\rangle$ jest jednakowo prawdopodobne, gdy funkcja h zostanie wybrana losowo z $\mathcal{H}$. W szczególności każda z p par postaci $\langle t, h(m')\rangle$ ma jednakowe szanse wystąpienia. Dlatego przeciwnik, nawet jeśli dysponuje pełną wiedzą na temat rodziny $\mathcal{H}$ i przechwyci parę $\langle m, t\rangle$, to nie zyskuje żadnej informacji o wartości $h(m')$, którą powinien przesłać jako t' celem oszukania Boba. Przeciwnik może więc tylko zgadywać, a szansa, że wybierze właściwą spośród p wartości, jest równa $1/p$.

Rozdział **12**

Drzewa wyszukiwań binarnych

12.1. Co to jest drzewo wyszukiwań binarnych?

12.1-1. Przykładowe drzewa BST przedstawiono na rys. 24.



Rysunek 24: Drzewa BST o różnych wysokościach zawierające zbiór kluczy $\{1, 4, 5, 10, 16, 17, 21\}$.

12.1-2. *Wyrażenie „właściwa kolejność" użyte w treści zadania oznacza kolejność niemalejącą.*

Niech x będzie węzłem drzewa T, a y, z, odpowiednio, lewym i prawym synem x. Jeśli drzewo T byłoby drzewem BST, czyli spełniałoby własność drzewa BST, to wówczas prawdziwe byłyby nierówności $key[y] \leqslant key[x] \leqslant key[z]$. Jeśli z kolei drzewo T stanowiłoby kopiec typu min, czyli spełniałoby własność kopca typu min, to zachodziłoby wtedy $key[x] \leqslant key[y]$ oraz $key[x] \leqslant key[z]$.

Ta różnica sprawia, że podczas przechodzenia kopca typu min w kolejności niemalejących węzłów, po odwiedzeniu węzła x nie jest wiadomo, które jego poddrzewo należałoby następnie odwiedzić, ponieważ nie jest znana relacja między y a z. Oznacza to, że wypisanie wszystkich n węzłów kopca typu min w kolejności niemalejącej zajmuje czas wyższy niż $O(n)$. W przeciwnym przypadku moglibyśmy zaimplementować algorytm heapsort w czasie liniowym.

12.1-3. Nierekurencyjna wersja algorytmu przechodzenia drzewa metodą inorder, która wykorzystuje stos, została przedstawiona w zad. 10.4-3. Nierekurencyjny algorytm o identycznej funkcjonalności, ale niewykorzystujący stosu, został opisany w zad. 10.4-5.

12.1-4. Algorytmy przechodzenia drzewa metodami preorder i postorder przedstawiono poniżej.

```
PREORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then wypisz key[x]
3          PREORDER-TREE-WALK(left[x])
4          PREORDER-TREE-WALK(right[x])
```

```
POSTORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then POSTORDER-TREE-WALK(left[x])
3          POSTORDER-TREE-WALK(right[x])
4          wypisz key[x]
```

Dla każdego z tych algorytmów można udowodnić twierdzenie analogiczne do tw. 12.1, a więc każdy z nich działa w czasie $\Theta(n)$ na drzewie n-wierzchołkowym.

12.1-5. Załóżmy nie-wprost, że istnieje algorytm konstruowania drzewa BST z dowolnej listy n elementów za pomocą porównań, którego pesymistyczny czas działania wynosi $o(n \lg n)$. Na drzewie BST utworzonym za pomocą tego algorytmu moglibyśmy następnie uruchomić algorytm przechodzenia drzewa metodą inorder i wypisać wszystkie jego klucze w kolejności niemalejącej w czasie $\Theta(n)$. Posortowanie wejściowej listy n elementów można byłoby wtedy przeprowadzić w czasie niższym niż liniowo-logarytmiczny, co przeczyłoby tw. 8.1.

12.2. Wyszukiwanie w drzewie wyszukiwań binarnych

12.2-1. Ciągi z przykładów (a), (b) i (d) są możliwe do uzyskania podczas wyszukiwania klucza 363 w drzewie BST. W przykładzie (c) po odwiedzeniu węzła o kluczu 911 przechodzimy do lewego poddrzewa, w którym znajdują się węzły o kluczach nie większych niż 911. Jednakże później napotykamy 912, a to nie jest możliwe w drzewie BST. W przykładzie (e) występuje podobna sytuacja – po odwiedzeniu węzła o kluczu 347 przetwarzane jest prawe poddrzewo.

Klucz 299 napotykany później jest jednak mniejszy niż 347, co w drzewie BST nie może mieć miejsca.

12.2-2. Rekurencyjne wersje procedur zostały przedstawione poniżej.

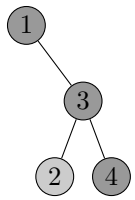
```
RECURSIVE-TREE-MINIMUM(x)
1  if left[x] ≠ NIL
2     then return RECURSIVE-TREE-MINIMUM(left[x])
3     else  return x
```

```
RECURSIVE-TREE-MAXIMUM(x)
1  if right[x] ≠ NIL
2     then return RECURSIVE-TREE-MAXIMUM(right[x])
3     else  return x
```

12.2-3. Procedura TREE-PREDECESSOR jest symetryczna do TREE-SUCCESSOR. Jej pseudokod znajduje się poniżej.

```
TREE-PREDECESSOR(x)
1  if left[x] ≠ NIL
2     then return TREE-MAXIMUM(left[x])
3  y ← p[x]
4  while y ≠ NIL i x = left[y]
5        do x ← y
6           y ← p[y]
7  return y
```

12.2-4. Rys. 25 przedstawia najmniejsze drzewo BST, które obala postawioną hipotezę.



Rysunek 25: Najmniejszy kontrprzykład dla rozumowania profesora Bunyana. W drzewie tym szukano klucza $k = 4$. Jasnym kolorem zaznaczono jedyny węzeł o kluczu należącym do zbioru A, a ciemnym kolorem – węzły o kluczach ze zbioru B. Zbiór C w tym przykładzie jest pusty. Klucze 1 i 2 nie spełniają nierówności postawionej w hipotezie.

12.2-5. Niech x będzie węzłem o dwóch synach w drzewie BST, a U_x zbiorem składającym się ze wszystkich przodków x, których lewe poddrzewa zawierają x. Węzły o kluczach większych niż $key[x]$ znajdują się w prawym poddrzewie x, a także w zbiorze U_x i w prawych poddrzewach węzłów z U_x. Dla każdego $u \in U_x$, dla każdego węzła v z prawego poddrzewa u i dla każdego węzła w z prawego poddrzewa x spełnione są nierówności $key[x] < key[w] < key[u] < key[v]$, a to oznacza, że następnik x znajduje się w jego prawym poddrzewie.

Niech y będzie następnikiem węzła x. Jeśli węzeł y miałby lewego syna z, to na podstawie faktu, że y należy do prawego poddrzewa x, zachodziłoby $key[x] < key[z] < key[y]$, a to z kolei przeczyłoby faktowi, że następnikiem x jest węzeł y. Stąd następnik x nie ma lewego syna.

Analogiczne rozumowanie przeprowadza się dla poprzednika węzła x.

12.2-6. Niech U_x będzie zbiorem zdefiniowanym jak w zad. 12.2-5. Na podstawie rozumowania z tamtego zadania mamy, że dla każdego $u \in U_x$ i dla każdego węzła v z prawego poddrzewa u zachodzą nierówności $key[x] < key[u] < key[v]$. Następnik y węzła x jest więc elementem zbioru U_x o minimalnym kluczu.

Niech u_1, $u_2 \in U_x$. Węzeł x należy do lewego poddrzewa zarówno u_1, jak i u_2. Węzły te nie mogą znajdować się w drzewie T na tej samej głębokości, bo wtedy ich lewe poddrzewa są rozłączne. Załóżmy więc bez utraty ogólności, że u_2 leży w T głębiej niż u_1. Wówczas węzeł u_2 oraz jego lewe poddrzewo znajdują się w lewym poddrzewie węzła u_1. Wynika stąd, że węzły ze zbioru U_x na większych głębokościach w drzewie T mają mniejsze klucze, a to oznacza, że następnikiem węzła x jest najgłębszy węzeł z U_x, czyli najniższy przodek x, którego lewy syn jest także przodkiem x.

12.2-7. Wywołanie TREE-MINIMUM, po którym następuje $n-1$ wywołań TREE-SUCCESSOR, spowoduje oczywiście wypisanie kluczy wszystkich n węzłów drzewa w tej samej kolejności, co wywołanie na tym drzewie INORDER-TREE-WALK. Jest w sumie n wywołań procedur – algorytm wymaga więc czasu $\Omega(n)$. Pokażemy jeszcze, że każda krawędź drzewa jest pokonywana co najwyżej dwukrotnie, skąd wynika górne oszacowanie $O(n)$ na czas działania algorytmu.

Niech u będzie węzłem drzewa, a v jego lewym synem. W trakcie działania algorytmu przed wypisaniem klucza u wpierw wypisane zostaną klucze wszystkich węzłów z jego lewego poddrzewa, którego korzeniem jest v, co zostanie zapoczątkowane przez przejście krawędzią $\langle u, v\rangle$. Następnie w celu wypisania klucza u procedura TREE-SUCCESSOR przejdzie w górę drzewa ścieżką do u od maksymalnego elementu poddrzewa o korzeniu w v. Na tej ścieżce oczywiście znajduje się krawędź $\langle u, v\rangle$. Po pokonaniu ścieżki krawędź ta nie zostanie ponownie wykorzystana, ponieważ wszystkie węzły w lewym poddrzewie u zostały już odwiedzone.

Załóżmy teraz, że v jest prawym synem u. Tuż po wypisaniu klucza u wywoływane jest TREE-SUCCESSOR(u). Aby dostać się do węzła o najmniejszym kluczu w prawym poddrzewie węzła u, algorytm musi zejść krawędzią $\langle u, v\rangle$. Przejście nią w drodze powrotnej nastąpi po odwiedzeniu wszystkich węzłów z prawego poddrzewa u, o ile będą wówczas jeszcze nieodwiedzone węzły w drzewie. Będzie to miało miejsce podczas wywołania TREE-SUCCESSOR dla węzła o maksymalnym kluczu w prawym poddrzewie u, kiedy to algorytm będzie przechodził w górę drzewa do węzła następującego po u w porządku inorder. Krawędź $\langle u, v\rangle$ należy do tej ścieżki i z racji tego, że wszystkie węzły prawego poddrzewa u zostały odwiedzone, nie zostanie już użyta ponownie.

12.2-8. Na podstawie zad. 12.2-6, jeśli węzeł początkowy u nie ma prawego syna, to w wyniku wywołania TREE-SUCCESSOR(u), następnym odwiedzonym węzłem będzie najniższy przodek u, którego lewy syn też jest przodkiem u. Oznaczmy ten węzeł przez v. Jeśli z kolei prawy syn u istnieje, to zanim algorytm dotrze do v, odwiedzi on wpierw prawe poddrzewo u. O ile nie zakończył swojego działania podczas tego kroku, to w obu przypadkach opisane operacje zostają następnie powtórzone dla węzła v. Widać zatem, że algorytm porusza się w górę drzewa po ścieżce, której długość ograniczona jest przez wysokość drzewa, odwiedzając prawe poddrzewa węzłów z tej ścieżki. Ponieważ mamy k wywołań TREE-SUCCESSOR, to sumaryczna liczba węzłów w tych poddrzewach wynosi co najwyżej $O(k)$. Algorytm działa więc w czasie $O(k+h)$.

12.2-9. Dowód sprowadza się do pokazania, że y jest albo następnikiem albo poprzednikiem węzła x.

Załóżmy, że x jest lewym synem węzła y. Zauważmy, że w drzewie T istnieje następnik węzła x, ponieważ x nie jest skrajnie prawym węzłem drzewa T. Możemy więc skorzystać z zad. 12.2-6, żeby pokazać, że następnikiem x jest w rzeczywistości y. Symetryczne twierdzenie do tego z zad. 12.2-6 charakteryzujące poprzednika węzła o pustym lewym poddrzewie, można dowieść, korzystając z analogicznego rozumowania z rozwiązania tamtego zadania. W przypadku, gdy x jest prawym synem y, na podstawie tego twierdzenia pokazujemy, że y jest poprzednikiem x.

12.3. Wstawianie i usuwanie

Linia 16 procedury TREE-DELETE *mówi o kopiowaniu wartości wszystkich pól y do pól z, podczas gdy kopiowane powinny być jedynie dodatkowe dane, jakie są przechowywane w węźle y.*

12.3-1. Rekurencyjna wersja operacji wstawiania będzie przyjmować na wejściu korzeń x poddrzewa, do którego odbywa się wstawianie, oraz nowy węzeł z. Jeśli $x = \text{NIL}$, to procedura natychmiast zakończy działanie, zwracając z. W przeciwnym przypadku będzie schodzić rekurencyjnie w dół poddrzewa o korzeniu w x odpowiednią ścieżką tak, aby na końcu tego procesu uczynić z synem liścia znajdującego się na tej ścieżce. Następnie, po aktualizacji lewego lub prawego syna x na wynik poprzedniego wywołania rekurencyjnego, zwrócone zostanie x.

```
RECURSIVE-TREE-INSERT(x, z)
1  if x = NIL
2     then return z
3  if key[z] < key[x]
4     then left[x] ← RECURSIVE-TREE-INSERT(left[x], z)
5          p[left[x]] ← x
6     else right[x] ← RECURSIVE-TREE-INSERT(right[x], z)
7          p[right[x]] ← x
8  return x
```

Procedura ta nie powinna być używana bezpośrednio, gdyż nie aktualizuje ona pola *root* drzewa T, do którego wstawiany jest nowy węzeł z. Czynność tę wykonuje poniższy pseudokod, który powinien być wywoływany, aby skorzystać z procedury RECURSIVE-TREE-INSERT.

```
RECURSIVE-TREE-INSERT-WRAPPER(T, z)
1  root[T] ← RECURSIVE-TREE-INSERT(root[T], z)
```

12.3-2. Procedura TREE-INSERT wywołana dla nowego węzła z o kluczu k, będzie schodzić ścieżką od korzenia w dół drzewa w celu odnalezienia liścia, który stanie się ojcem węzła z. Z kolei procedura TREE-SEARCH wyszukująca w drzewie węzeł o kluczu k zejdzie podobną ścieżką w celu odnalezienia tego węzła. Porównajmy instrukcje odpowiedzialne za poruszanie się po tych ścieżkach – w pierwszej procedurze są to wiersze 3–7, a w drugiej – wiersze 1–4. W TREE-INSERT obecność linii 4 nie wpływa na wybór ścieżki, a $key[z] = k$, więc $key[x]$ jest w obu procedurach porównywane z tą samą wartością. Jeszcze jedną różnicą jest występowanie w TREE-SEARCH dodatkowego warunku $k \neq key[x]$ w pętli **while**. Z założenia o unikalności kluczy w drzewie wnioskujemy, że warunek ten będzie fałszywy dla każdego węzła na ścieżce, zanim osiągnięty zostanie szukany węzeł. Ścieżki pokonywane w obu procedurach są zatem identyczne, przy czym

procedura TREE-SEARCH porówna jeszcze poszukiwany klucz z węzłem o tym kluczu na końcu tej ścieżki.

12.3-3. Pesymistyczny przypadek dla takiego sposobu sortowania n liczb zachodzi wtedy, gdy tablica jest uporządkowana rosnąco bądź malejąco. Wówczas drzewo powstałe po serii n operacji TREE-INSERT ma wysokość $n-1$. Budowa takiego drzewa zajmuje $O(n^2)$ i taki też jest czas sortowania w tym przypadku.

Z kolei przypadek optymistyczny ma miejsce, gdy zbudowane drzewo ma minimalną wysokość. Z zad. B.5-4 wiemy, że drzewo zawierające n węzłów ma wysokość co najmniej $\lfloor \lg n \rfloor$. Najmniejszy możliwy czas działania algorytmu sortowania wynosi zatem $O(n \lg n)$, bo to jest czas potrzebny na zbudowanie takiego drzewa.

12.3-4. Procedura TREE-DELETE wywołana dla węzła o dwóch synach w rzeczywistości usunie następnik tego węzła. Problem pojawia się, gdy pewna struktura danych przechowuje wskaźnik do węzła y i wywołane zostanie TREE-DELETE celem usunięcia węzła z, będącego poprzednikiem y. Struktura ta może wciąż zakładać, że z należy do drzewa. Tak jednak nie jest, ponieważ to węzeł wskazywany przez y został ostatecznie usunięty, a jego wszystkie atrybuty skopiowane zostały do węzła wskazywanego przez z.

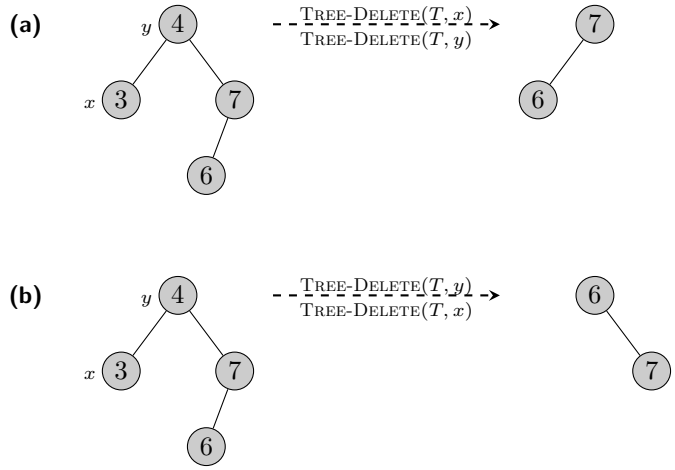
Rozwiązanie tego problemu polega na podmianie węzła z przez y tuż przed zakończeniem operacji usuwania w przypadku, gdy węzeł z początkowo miał dwóch synów. Wówczas węzłem efektywnie usuwanym byłby za każdym razem ten wskazywany przez z, dlatego w bezpiecznej wersji procedury usuwania, której pseudokod prezentujemy poniżej, możemy zrezygnować ze zwracania jakiejkolwiek wartości.

```
SAFE-TREE-DELETE(T, z)
1  y ← TREE-DELETE(T, z)
2  if y ≠ z
3     then p[left[z]] ← p[right[z]] ← y
4          if p[z] ≠ NIL
5             then if z = left[p[z]]
6                       then left[p[z]] ← y
7                       else  right[p[z]] ← y
8          skopiuj zawartość wszystkich pól z z do y
```

Powyższa procedura deleguje operację usuwania węzła z do oryginalnej procedury TREE-DELETE. Jeśli zwrócony przez tę ostatnią węzeł y jest różny od z, to w wierszach 3–8 następuje przepięcie synów i ojca z na węzeł y i przepisanie wszystkich pól z z do y. Po wykonaniu tej procedury struktura danych korzystająca z drzewa może bezpiecznie założyć, że usunięty został dokładnie ten węzeł, który stanowił argument operacji usuwania.

12.3-5. Operacja usuwania z drzewa wyszukiwań binarnych nie jest przemienna. Kontrprzykład został zilustrowany na rys. 26.

12.3-6. Wybór między poprzednikiem i następnikiem uzależnimy od wyniku rzutu monetą, który będziemy symulować wywołaniem RANDOM$(0,1)$. Wiersz 3 w procedurze TREE-DELETE zastąpimy więc następującym fragmentem:



Rysunek 26: Kontrprzykład dla przemienności operacji usuwania węzła z drzewa BST. **(a)** W przykładowym drzewie T najpierw usuwany jest węzeł o kluczu 3, a następnie węzeł o kluczu 4. **(b)** To samo drzewo T, z którego usuwane są te same węzły, ale w odwrotnej kolejności. Wynikowe drzewa są różne.

if RANDOM$(0, 1) = 0$
then $y \leftarrow$ TREE-PREDECESSOR(z)
else $y \leftarrow$ TREE-SUCCESSOR(z)

12.4. Losowo skonstruowane drzewa wyszukiwań binarnych

12.4-1. Wzór udowodnimy przez indukcję względem n. Jeśli $n = 1$, to po obu stronach znaku równości jest 1. Załóżmy teraz, że $n > 1$ i że spełnione jest założenie indukcyjne

$$\sum_{i=0}^{n-2} \binom{i+3}{3} = \binom{n+2}{4}.$$

Mamy

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \sum_{i=0}^{n-2} \binom{i+3}{3} + \binom{n+2}{3} = \binom{n+2}{4} + \binom{n+2}{3} = \binom{n+3}{4},$$

przy czym w ostatniej równości wykorzystaliśmy zad. C.1-7.

12.4-2. Rozważmy drzewo binarne w którym początkowe poziomy stanowią pełne drzewo binarne składające się z $n - \sqrt{n \lg n}$ węzłów, natomiast pozostałe $\sqrt{n \lg n}$ węzłów znajduje się na ścieżce odchodzącej od tegoż pełnego drzewa w dół na coraz to niższe poziomy. Drzewo takie ma wysokość

$$\Theta\big(\lg\big(n - \sqrt{n \lg n}\big)\big) + \sqrt{n \lg n} = \Theta\big(\sqrt{n \lg n}\big) = \omega(\lg n).$$

O tym, że średnia głębokość węzła w tym drzewie wynosi $\Theta(\lg n)$, przekonamy się, wyprowadzając najpierw górne, a potem dolne oszacowanie tej wartości. W oszacowaniu górnym użyjemy

$O(\lg n)$ jako ograniczenia na głębokość każdego z $n - \sqrt{n \lg n}$ węzłów z części stanowiącej pełne drzewo binarne oraz $O\big(\lg n + \sqrt{n \lg n}\big)$ jako ograniczenia na głębokość każdego z $\sqrt{n \lg n}$ węzłów ze ścieżki odchodzącej od pełnego drzewa. Średnia głębokość węzła wynosi więc co najwyżej

$$\frac{1}{n} \cdot O\Big(\big(n - \sqrt{n \lg n}\big) \lg n + \sqrt{n \lg n}\,\big(\lg n + \sqrt{n \lg n}\big)\Big) = \frac{1}{n} \cdot O(n \lg n) = O(\lg n).$$

Z kolei zauważmy, że najniższy poziom pełnego drzewa binarnego składa się z $\Theta\big(n - \sqrt{n \lg n}\big)$ węzłów, z których każdy ma głębokość $\Theta(\lg n)$. Stąd średnia głębokość węzła ograniczona jest z dołu przez

$$\frac{1}{n} \cdot \Theta\Big(\big(n - \sqrt{n \lg n}\big) \lg n\Big) = \frac{1}{n} \cdot \Omega(n \lg n) = \Omega(\lg n).$$

W drugiej części zadania udowodnimy, że wysokość drzewa binarnego o n węzłach i średniej głębokości węzła $\Theta(\lg n)$ wynosi $O\big(\sqrt{n \lg n}\big)$. Niech dane będzie drzewo binarne o n węzłach i wysokości h ze średnią głębokością węzła $\Theta(\lg n)$. Istnieje więc ścieżka od korzenia w dół tego drzewa, na której głębokościami węzłów są kolejno 0, 1, …, h. Oznaczmy przez U zbiór węzłów na tej ścieżce, przez U' zbiór wszystkich pozostałych węzłów drzewa, a przez $d(x)$ głębokość węzła x. Wówczas średnia głębokość węzła w tym drzewie wynosi

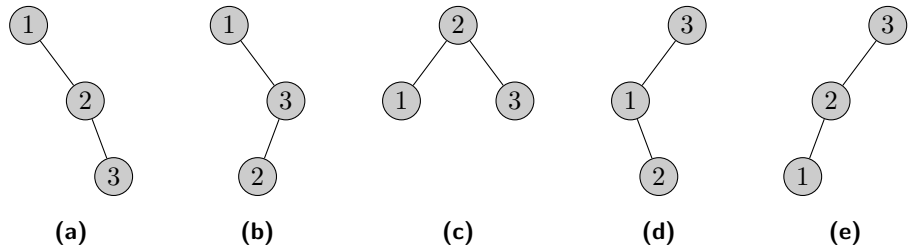
$$\frac{1}{n}\left(\sum_{x \in U} d(x) + \sum_{x \in U'} d(x)\right) \geqslant \frac{1}{n} \sum_{x \in U} d(x) = \frac{1}{n} \sum_{i=0}^{h} i = \frac{1}{n} \cdot \Theta(h^2).$$

Jeśli byłoby $h = \omega\big(\sqrt{n \lg n}\big)$, to wtedy mielibyśmy

$$\frac{1}{n} \cdot \Theta(h^2) = \frac{1}{n} \cdot \omega(n \lg n) = \omega(\lg n),$$

co stoi w sprzeczności z założeniem, że średnią głębokością węzła jest $\Theta(\lg n)$. A zatem wysokość drzewa musi być ograniczona przez $O\big(\sqrt{n \lg n}\big)$.

12.4-3. Istnieje 5 różnych drzew BST o $n = 3$ węzłach, podczas gdy jest $3! = 6$ permutacji trójelementowych. Jest więc jasne, że któreś z tych drzew można zbudować z więcej niż jednej permutacji. Rys. 27 ilustruje każde drzewo, podając permutacje, z których one powstają.



Rysunek 27: Drzewa BST uzyskane z permutacji **(a)** $\langle 1, 2, 3\rangle$, **(b)** $\langle 1, 3, 2\rangle$, **(c)** $\langle 2, 1, 3\rangle$ oraz $\langle 2, 3, 1\rangle$, **(d)** $\langle 3, 1, 2\rangle$, **(e)** $\langle 3, 2, 1\rangle$.

12.4-4. Udowodnimy silniejsze twierdzenie, że każda funkcja $f(x) = c^x$, gdzie c jest stałą dodatnią, jest wypukła. Z definicji wypukłości funkcji z dodatku C musimy pokazać, że dla każdych x, y i dla każdego $0 \leqslant \lambda \leqslant 1$ prawdziwa jest nierówność

$$c^{\lambda x + (1-\lambda)y} \leqslant \lambda c^x + (1 - \lambda) c^y.$$

Lemat. *Dla dowolnych liczb rzeczywistych a, b i dla dowolnej dodatniej liczby rzeczywistej c zachodzi*

$$c^a \geqslant c^b + (a-b)c^b \ln c.$$

Dowód. Na podstawie wzoru (3.11), $e^x \geqslant 1+x$ dla dowolnego x. Jeśli przyjmiemy $x = r\ln c$, to $e^x = e^{r\ln c} = (e^{\ln c})^r = c^r$ i nierówność sprowadza się do postaci $c^r \geqslant 1 + r\ln c$. Po podstawieniu $r = a-b$ dostajemy $c^{a-b} \geqslant 1+(a-b)\ln c$ i aby otrzymać żądaną nierówność, wystarczy pomnożyć obie strony przez c^b. □

Niech $z = \lambda x + (1-\lambda)y$. Skorzystajmy z powyższego lematu dwukrotnie, najpierw podstawiając $a = x$ i $b = z$ i otrzymując nierówność $c^x \geqslant c^z + (x-z)c^z \ln c$, a następnie przyjmując $a = y$ i $b = z$, skąd dostajemy $c^y \geqslant c^z + (y-z)c^z \ln c$. Pierwszą otrzymaną nierówność pomnożymy przez λ, a drugą przez $1-\lambda$, a następnie dodamy do siebie:

$$\begin{aligned}
\lambda c^x + (1-\lambda)c^y &\geqslant \lambda(c^z + (x-z)c^z \ln c) + (1-\lambda)(c^z + (y-z)c^z \ln c) \\
&= \lambda c^z + \lambda x c^z \ln c - \lambda z c^z \ln c + (1-\lambda)c^z + (1-\lambda)y c^z \ln c - (1-\lambda)z c^z \ln c \\
&= (\lambda + (1-\lambda))c^z + (\lambda x + (1-\lambda)y)c^z \ln c - (\lambda + (1-\lambda))z c^z \ln c \\
&= c^z + z c^z \ln c - z c^z \ln c \\
&= c^z \\
&= c^{\lambda x + (1-\lambda)y}.
\end{aligned}$$

Otrzymany wynik dowodzi wypukłości funkcji $f(x) = c^x$, a więc w szczególności $f(x) = 2^x$.

12.4-5. *Wymagane jest założenie, że elementy tablicy wejściowej algorytmu* Randomized-Quicksort *są parami różne.*

Problemy

12-1. Drzewa wyszukiwań binarnych z powtarzającymi się kluczami

(a) Pierwsze wywołanie Tree-Insert umieszcza nowy węzeł w korzeniu drzewa. W pozostałych wywołaniach warunki w liniach 5 i 11 są fałszywe, co oznacza, że każdy nowy węzeł wstawiony zostanie jako prawy syn poprzedniego węzła i po n wywołaniach Tree-Insert drzewo będzie mieć postać ścieżki o n elementach. Każde kolejne wywołanie działa na drzewie o wysokości o 1 większej niż w poprzednim wywołaniu – sumarycznie więc pokonają ścieżkę o łącznej długości równej $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, co zajmie czas $\Theta(n^2)$.

(b) Zauważmy, że drzewo tworzone w tej strategii w dowolnym momencie spełnia następującą własność: dla każdego jego węzła v liczba węzłów w lewym poddrzewie v jest równa lub o 1 większa od liczby węzłów w prawym poddrzewie v. Wynika z tego, że drzewo jest wypełnione na każdym poziomie, być może z wyjątkiem ostatniego. Podczas pierwszego wywołania Tree-Insert drzewo jest puste, a w każdym kolejnym wywołaniu jego wysokość wynosi $\Theta(\lg i)$. Ciąg n operacji Tree-Insert działa zatem w czasie

$$\sum_{i=2}^{n} \Theta(\lg i) = \Theta\left(\sum_{i=2}^{n} \lg i\right) = \Theta(\lg(n!)) = \Theta(n \lg n)$$

na podstawie wzoru (3.18).

(c) Dodanie węzła do listy węzłów o tym samym kluczu odbywa się w czasie stałym. Ciąg n wywołań TREE-INSERT wykona się więc w czasie $\Theta(n)$.

(d) W pesymistycznym przypadku x jest zawsze ustawiane na $left[x]$ (albo zawsze na $right[x]$) i efektywność działania ciągu operacji TREE-INSERT w tej strategii nie różni się od efektywności ciągu wywołań jej oryginalnej wersji, gdyż w czasie $\Theta(n^2)$ tworzone jest drzewo będące ścieżką n-elementową.

W przypadku średnim x z jednakowym prawdopodobieństwem przyjmie $left[x]$ albo $right[x]$, co oznacza, że są równe szanse na to, że wstawiany węzeł trafi do lewego bądź do prawego poddrzewa węzła, z którym jest aktualnie testowany. Wejściowy ciąg kluczy jest więc nieodróżnialny od pewnej losowej permutacji n różnych kluczy i powstające drzewo można potraktować jak losowo skonstruowane drzewo wyszukiwań binarnych o n węzłach. Konstrukcja ta wymaga czasu proporcjonalnego do sumy głębokości węzłów drzewa, a na podstawie problemu 12-3 w średnim przypadku wartość ta jest ograniczona przez $O(n \lg n)$.

12-2. Drzewa pozycyjne

S jest zbiorem różnych ciągów bitowych, których długości sumują się do n.

Zbiór S ciągów bitowych możemy posortować poprzez zbudowanie z jego elementów drzewa pozycyjnego, a następnie wypisanie wszystkich kluczy tego drzewa należących do S w porządku preorder. Uzasadnimy teraz to stwierdzenie i znajdziemy czas działania tego sortowania.

Zbadajmy drzewo pozycyjne zbudowane z elementów zbioru S. Rozważmy węzeł x o kluczu $s \in S$ z poziomu i tego drzewa. Ciąg s jest i-bitowym prefiksem wszystkich ciągów z lewego i z prawego poddrzewa węzła x. Ponadto w każdym ciągu z lewego poddrzewa na $(i+1)$-szej pozycji jest 0, a w każdym ciągu z prawego poddrzewa na tej samej pozycji jest 1. A zatem ciąg s leksykograficznie poprzedza ciągi po lewej stronie x, które z kolei leksykograficznie poprzedzają ciągi po prawej stronie x. W wynikowej posortowanej permutacji bezpośrednio po s znajdą się więc wszystkie klucze z S należące do lewego poddrzewa węzła x, po czym wszystkie klucze z S należące do jego prawego poddrzewa. Klucze w takiej kolejności możemy wypisać, przechodząc poddrzewo o korzeniu w x metodą preorder. Zbiór S posortujemy więc poprzez wywołanie PREORDER-TREE-WALK na całym drzewie pozycyjnym, przy czym wypisywane będą tylko klucze należące do zbioru S.

Wstawienie węzła o kluczu s do drzewa pozycyjnego zajmuje czas $\Theta(|s|)$, gdzie $|s|$ oznacza długość ciągu s, gdyż węzeł ten zostanie umieszczony na poziomie $|s|$. Budowa drzewa pozycyjnego z elementów zbioru S odbywa się więc w czasie

$$\sum_{s \in S} \Theta(|s|) = \Theta\left(\sum_{s \in S} |s|\right) = \Theta(n).$$

Podczas wstawiania węzła o kluczu s tworzonych jest co najwyżej $|s|+1$ nowych węzłów, zatem drzewo zbudowane ze zbioru S posiada nie więcej niż

$$\sum_{s \in S} (|s|+1) \leqslant (|\varepsilon|+1) + \sum_{s \in S \setminus \{\varepsilon\}} 2|s| = 1 + 2n = O(n)$$

węzłów (ε oznacza ciąg pusty). Przechodzenie drzewa w porządku preorder działa w czasie proporcjonalnym do liczby jego węzłów. Dodając do tego czas spędzony na budowaniu drzewa, otrzymujemy, że opisane sortowanie odbywa się w czasie $\Theta(n)$.

12-3. Średnia głębokość węzła w losowo zbudowanym drzewie wyszukiwań binarnych

W treści problemu – także w wersji oryginalnej – brakuje założenia, że n jest liczbą węzłów w T. Co więcej, nie zdefiniowano notacji $x \in T$ oznaczającej przynależność węzła x do drzewa T.

(a) Wzór wynika wprost z definicji średniej głębokości węzła w drzewie T i z definicji $P(T)$.

(b) Zauważmy, że dla każdego $x \in T_L$ zachodzi $d(x, T_L) = d(x, T) - 1$ i podobnie, dla każdego $x \in T_R$ zachodzi $d(x, T_R) = d(x, T) - 1$. Stąd, jeśli r jest korzeniem drzewa T, to

$$\begin{aligned} P(T) &= \sum_{x \in T} d(x, T) \\ &= d(r, T) + \sum_{x \in T_L} d(x, T) + \sum_{x \in T_R} d(x, T) \\ &= \sum_{x \in T_L} (d(x, T_L) + 1) + \sum_{x \in T_R} (d(x, T_R) + 1) \\ &= \sum_{x \in T_L} d(x, T_L) + \sum_{x \in T_R} d(x, T_R) + n - 1 \\ &= P(T_L) + P(T_R) + n - 1. \end{aligned}$$

(c) Jeśli T jest losowo skonstruowanym drzewem wyszukiwań binarnych, to klucz pierwszego wstawianego do T węzła jest z równymi szansami dowolnym w kolejności rosnącej w ciągu n kluczy wstawianych węzłów. Pierwszy węzeł staje się korzeniem drzewa T, a więc po zbudowaniu T jego lewe poddrzewo T_L może z jednakowym prawdopodobieństwem być drzewem o i węzłach, gdzie $i = 0, 1, \ldots, n-1$. Dla ustalonego i prawe poddrzewo T_R ma wówczas rozmiar $n - i - 1$. Z definicji $P(n)$ jako średniej wartości $P(T)$ oraz z punktu (b) otrzymujemy

$$P(n) = \frac{1}{n} \sum_{i=0}^{n-1} (P(i) + P(n - i - 1) + n - 1).$$

(d) Na podstawie poprzedniego punktu mamy:

$$\begin{aligned} P(n) &= \frac{1}{n} \sum_{k=0}^{n-1} (P(k) + P(n - k - 1) + n - 1) \\ &= \frac{1}{n} \left(\sum_{k=0}^{n-1} P(k) + \sum_{k=0}^{n-1} P(n - k - 1) + \sum_{k=0}^{n-1} (n - 1) \right) \\ &= \frac{1}{n} \left(2 \sum_{k=0}^{n-1} P(k) + n(n - 1) \right) \\ &= \frac{2}{n} \sum_{k=1}^{n-1} P(k) + \Theta(n). \end{aligned}$$

W ostatnim przekształceniu skorzystaliśmy z tego, że $P(0) = 0$ i opuściliśmy ten składnik sumy.

(e) Podobnie jak w problemie 7-2 pokażemy, że $P(n) \leqslant an \lg n$ dla pewnej dodatniej stałej a. Podstawą indukcji niech będzie $n = 1$. Oczywiście $P(1) = 0$ oraz $a \cdot 1 \cdot \lg 1 = 0$, więc podstawa

jest spełniona dla jakiegokolwiek a. Niech teraz $n > 1$. Przyjmijmy założenie, że dla dowolnego $k = 1, 2, \ldots, n-1$ zachodzi $P(k) \leqslant ak \lg k$. Wówczas:

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n) \leqslant \frac{2}{n}\sum_{k=1}^{n-1} ak \lg k + \Theta(n) = \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + \Theta(n).$$

Posługując się nierównością (7.7) udowodnioną w punkcie (d) problemu 7-2 i dobierając odpowiednio duże a tak, aby wyrażenie $an/4$ nie było mniejsze od składnika $\Theta(n)$, mamy

$$P(n) \leqslant \frac{2a}{n}\left(\frac{n^2 \lg n}{2} - \frac{n^2}{8}\right) + \Theta(n) = an \lg n - \frac{an}{4} + \Theta(n) \leqslant an \lg n = O(n \lg n).$$

(f) Zauważmy, że po tym, jak węzeł x zostaje wybrany jako korzeń drzewa T, wszystkie węzły wstawiane do T po x są porównywane z x. Analogicznie, po tym, jak element y zostaje wybrany jako element rozdzielający w podtablicy S, wszystkie elementy wstawiane do S po y są porównywane z y. Podczas budowy losowo skonstruowanego drzewa wyszukiwań binarnych korzeniem staje się zawsze pierwszy węzeł z permutacji wejściowej. Implementacja quicksorta o identycznych porównaniach elementów powinna zatem wybierać na element rozdzielający pierwszy element tablicy wejściowej.

12-4. Zliczanie różnych drzew binarnych

(a) Oczywiście jest tylko jedno drzewo binarne niezawierające żadnego węzła, zatem $b_0 = 1$.

Rozważmy drzewo binarne o $n \geqslant 1$ węzłach. Jeśli jego lewe poddrzewo ma k węzłów, gdzie $k = 0, 1, \ldots, n-1$, to jego prawe poddrzewo ma $n-1-k$ węzłów. Wynika stąd, że liczba drzew binarnych o n węzłach i o ustalonej liczbie k węzłów w lewym poddrzewie, jest równa $b_k b_{n-1-k}$. Liczba wszystkich drzew binarnych o n węzłach wynosi zatem

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}.$$

(b) Wyznaczmy $B^2(x)$. Kwadrat sumy wyrazów ciągu jest sumą iloczynów wyrazów tego ciągu na zasadzie „każdy z każdym”. Dzięki odpowiedniemu pogrupowaniu iloczynów możemy napisać:

$$B^2(x) = \left(\sum_{n=0}^{\infty} b_n x^n\right)^2 = \sum_{n=0}^{\infty}\sum_{k=0}^{n} b_k x^k b_{n-k} x^{n-k} = \sum_{n=0}^{\infty}\sum_{k=0}^{n} b_k b_{n-k} x^n.$$

Wewnętrzną sumę na podstawie punktu (a) zamieniamy na b_{n+1} i wówczas:

$$B^2(x) = \sum_{n=0}^{\infty} b_{n+1} x^n = \frac{1}{x}\sum_{n=0}^{\infty} b_{n+1} x^{n+1} = \frac{1}{x}\sum_{n=1}^{\infty} b_n x^n = \frac{1}{x}\left(\sum_{n=0}^{\infty} b_n x^n - 1\right) = \frac{B(x)-1}{x}.$$

Stąd $B(x) = xB^2(x) + 1$ i jednym ze sposobów wyrażenia $B(x)$ w postaci jawnej jest

$$B(x) = \frac{1 - \sqrt{1-4x}}{2x}.$$

(c) *Rozwinięciem Taylora funkcji $f(x)$ wokół punktu $x = a$ jest* $f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k$.

Aby skorzystać z rozwinięcia Taylora funkcji $f(x)$, musimy wyznaczyć jej kolejne pochodne.

Lemat. *Jeśli $f(x) = \sqrt{1-4x}$, to dla każdego całkowitego $k \geqslant 1$ zachodzi*

$$f^{(k)}(x) = -\frac{2(2k-2)!}{(k-1)!}\,(1-4x)^{1/2-k}.$$

Dowód. Wzór udowodnimy przez indukcję. Jeśli $k = 1$, to $f'(x) = \frac{-2}{\sqrt{1-4x}}$ i łatwo sprawdzić, że wzór zachodzi. Niech teraz $k \geqslant 2$. Wykorzystując założenie indukcyjne, wyznaczmy k-tą pochodną funkcji $f(x)$:

$$\begin{aligned}
f^{(k)}(x) &= \left(f^{(k-1)}(x)\right)' \\
&= \left(-\frac{2(2k-4)!}{(k-2)!}\,(1-4x)^{3/2-k}\right)' \\
&= -\frac{2(2k-4)!}{(k-2)!} \cdot \left(\frac{3}{2}-k\right) \cdot (-4) \cdot (1-4x)^{1/2-k} \\
&= -\frac{2(2k-4)!}{(k-2)!} \cdot \frac{(2k-3)(2k-2)}{(k-1)} \cdot (1-4x)^{1/2-k} \\
&= -\frac{2(2k-2)!}{(k-1)!}\,(1-4x)^{1/2-k}.
\end{aligned}$$

□

Rozwinięcie Taylora funkcji $f(x) = \sqrt{1-4x}$ wokół punktu $x = 0$ ma postać:

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!}\,x^k = 1 - 2\sum_{k=1}^{\infty} \frac{(2k-2)!}{(k-1)!\,k!}\,x^k = 1 - 2\sum_{k=1}^{\infty} \binom{2k-2}{k-1}\frac{x^k}{k} = 1 - 2\sum_{k=0}^{\infty} \binom{2k}{k}\frac{x^{k+1}}{k+1}.$$

Podstawiając powyższy wynik do postaci $B(x)$ wyprowadzonej w części (b), dostajemy

$$B(x) = \frac{1-\sqrt{1-4x}}{2x} = \sum_{k=0}^{\infty} \binom{2k}{k}\frac{x^k}{k+1}.$$

co po przyrównaniu do wzoru z definicji funkcji $B(x)$, daje

$$b_n = \frac{1}{n+1}\binom{2n}{n}.$$

(d) Na mocy poprzedniego punktu i z zad. C.1-13 mamy

$$\begin{aligned}
b_n &= \frac{1}{n+1}\binom{2n}{n} \\
&= \frac{1}{n+1} \cdot \frac{2^{2n}}{\sqrt{\pi n}}\,(1+O(1/n)) \\
&= \frac{n}{n+1} \cdot \frac{4^n}{\sqrt{\pi}\,n^{3/2}}\,(1+O(1/n))
\end{aligned}$$

$$= \left(1 - \frac{1}{n+1}\right) \frac{4^n}{\sqrt{\pi}\, n^{3/2}} \left(1 + O(1/n)\right)$$
$$= \frac{4^n}{\sqrt{\pi}\, n^{3/2}} \left(1 + O(1/n)\right).$$

Ostatnia równość zachodzi, bo

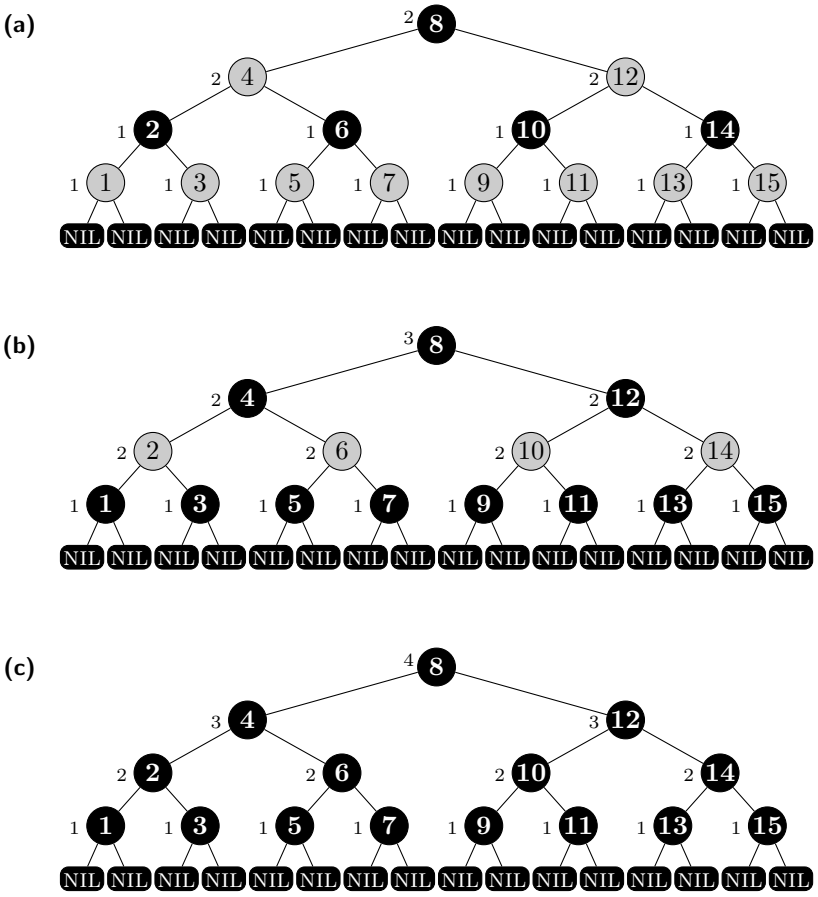
$$\left(1 - \frac{1}{n+1}\right)(1 + O(1/n)) = 1 - \frac{1}{n+1} + O(1/n) - O(1/n^2) = 1 + O(1/n).$$

Rozdział **13**

Drzewa czerwono-czarne

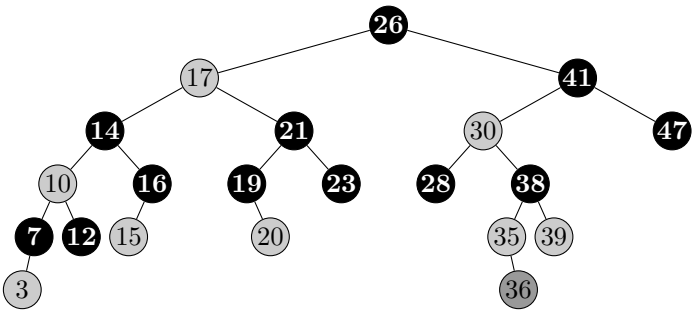
13.1. Własności drzew czerwono-czarnych

13.1-1. Drzewa zostały przedstawione na rys. 28.



Rysunek 28: Drzewa czerwono-czarne o wysokości 3 zawierające zbiór kluczy $\{1, 2, \ldots, 15\}$ z dodanymi liśćmi NIL. Czarna wysokość tych drzew wynosi, odpowiednio, **(a)** 2, **(b)** 3 oraz **(c)** 4.

13.1-2. Rys. 29 przedstawia drzewo po dodaniu nowego węzła o nieustalonym kolorze. Pokolorowanie go na czerwono naruszy jednak własność 4 drzewa czerwono-czarnego. Natomiast nadanie mu koloru czarnego naruszy własność 5 – na prostej ścieżce od korzenia drzewa do liścia NIL, będącego synem nowego węzła, znajdzie się 5 czarnych węzłów, a na prostej ścieżce od korzenia do dowolnego innego liścia NIL będą 4 czarne węzły. A zatem, niezależnie od wyboru koloru dla nowego węzła, wynikowe drzewo nie będzie drzewem czerwono-czarnym.



Rysunek 29: Drzewo z rys. 13.1 z Podręcznika po dodaniu węzła o kluczu 36 procedurą TREE-INSERT.

13.1-3. Po zmianie koloru korzenia na czarny w uproszczonym drzewie czerwono-czarnym własności 1–4 drzewa czerwono-czarnego oczywiście będą spełnione. Liczba czarnych węzłów na każdej prostej ścieżce od korzenia do liścia zwiększy się o 1, a liczba czarnych węzłów na prostych ścieżkach od dowolnego innego węzła do ich potomnych liści nie zmieni się. To oznacza, że także własność 5 nie będzie naruszona i otrzymane drzewo będzie drzewem czerwono-czarnym.

13.1-4. Niech x będzie czarnym węzłem w drzewie czerwono-czarnym. Jeśli x nie ma czerwonego syna, to w wyniku pochłaniania czerwonych węzłów jego stopień nie zmieni się i będzie nadal wynosił 0 dla $x = \text{NIL}$ albo 2 dla $x \neq \text{NIL}$. Jeśli x ma dokładnie jednego czerwonego syna y, to pochłaniając go, przyjmuje czarnych synów y, zwiększając swój stopień do 3. Gdy obaj synowie y_1, y_2 węzła x są czerwoni, to po ich pochłonięciu x będzie ojcem 4 czarnych węzłów, które początkowo były synami węzłów y_1 i y_2.

Po pochłonięciu wszystkich czerwonych węzłów dowolna prosta ścieżka od korzenia do liścia będzie składać się z tej samej liczby węzłów, skąd wynika, że głębokość każdego liścia będzie równa czarnej wysokości początkowego drzewa czerwono-czarnego.

13.1-5. Skorzystamy z faktu, że każda prosta ścieżka z ustalonego węzła x do jego potomnego czarnego liścia w danym drzewie czerwono-czarnym składa się z tej samej liczby czarnych węzłów. Na możliwie najkrótszej takiej ścieżce są same czarne węzły (być może z wyjątkiem x) i jej długością jest czarna wysokość węzła x. Najdłuższa możliwa ścieżka powstaje z najkrótszej poprzez wstawienie czerwonego węzła pomiędzy każdą parę sąsiednich czarnych węzłów, w wyniku czego długość ścieżki może zwiększyć się maksymalnie dwukrotnie.

13.1-6. Drzewo czerwono-czarne o czarnej wysokości k ma najmniejszą możliwą liczbę węzłów wewnętrznych, jeśli każda prosta ścieżka od korzenia do liścia jest możliwie najkrótsza. Podobnie, największa liczba węzłów wewnętrznych w tym drzewie jest osiągana, gdy każda taka ścieżka jest możliwie najdłuższa. Na podstawie poprzedniego zadania mamy, że długość takiej ścieżki, a co za tym idzie także wysokość drzewa, wynosi co najmniej k i co najwyżej $2k$. Drzewo składa się

zatem z co najmniej $2^k - 1$ i co najwyżej $2^{2k} - 1$ węzłów wewnętrznych.

13.1-7. Rozumując podobnie jak w zad. 13.1-5, możemy wywnioskować, że w drzewie czerwono-czarnym na prostej ścieżce od korzenia do ojca liścia (najniżej położonego węzła wewnętrznego na tej ścieżce) węzłów czerwonych może być co najwyżej tyle samo co czarnych. Jeśli każda taka ścieżka zawiera maksymalną liczbę czerwonych węzłów, to drzewo jest pełnym drzewem binarnym o wysokości parzystej, w którym na parzystych poziomach są węzły czarne, a na nieparzystych – czerwone. Przykładem takiego drzewa jest to zobrazowane na rys. 28(a). W takich drzewach czerwonych węzłów wewnętrznych jest 2 razy więcej niż czarnych węzłów wewnętrznych.

Z drugiej strony zauważmy, że pełne drzewo wyszukiwań binarnych, w którym wszystkie węzły są czarne, jest poprawnym drzewem czerwono-czarnym (np. to z rys. 28(c)). Minimalny stosunek czerwonych węzłów wewnętrznych do czarnych węzłów wewnętrznych wynosi więc 0.

13.2. Operacje rotacji

Linia 3 w procedurze LEFT-ROTATE *powinna być zastąpiona fragmentem:*

```
if left[y] ≠ nil[T]
   then p[left[y]] ← x
```

W rozwiązaniach przyjmujemy, że procedury LEFT-ROTATE *i* RIGHT-ROTATE *działają poprawnie także dla zwykłych drzew wyszukiwań binarnych poprzez zastąpienie w nich odwołania do* $nil[T]$ *z wiersza 5 (oraz 3 – patrz wyżej) przez* NIL.

13.2-1. Pseudokod operacji prawej rotacji wygląda analogicznie do procedury LEFT-ROTATE, w której zamieniono ze sobą pola *left* i *right*.

```
RIGHT-ROTATE(T, x)
 1  y ← left[x]
 2  left[x] ← right[y]
 3  if right[y] ≠ nil[T]
 4     then p[right[y]] ← x
 5  p[y] ← p[x]
 6  if p[x] = nil[T]
 7     then root[T] ← y
 8     else if x = right[p[x]]
 9             then right[p[x]] ← y
10             else left[p[x]] ← y
11  right[y] ← x
12  p[x] ← y
```

13.2-2. Na dowolnym węźle x w drzewie wyszukiwań binarnych T można wykonać lewą rotację, o ile prawy syn x jest różny od NIL, a prawą rotację – o ile jego lewy syn jest różny od NIL. Oznacza to, że istnieje wzajemna jednoznaczność między rotacjami a krawędziami drzewa, wokół których można przeprowadzić rotacje. W drzewie o n węzłach jest dokładnie $n - 1$ krawędzi i tyle samo różnych rotacji można w nim wykonać.

13.2-3. Wewnętrzna struktura węzłów w poddrzewach α, β, γ nie zmienia się w wyniku wykonania rotacji. A zatem głębokość dowolnego węzła a z α zwiększa się o 1, głębokość dowolnego węzła b z β nie zmienia się, a głębokość dowolnego węzła c z γ zmniejsza się o 1.

13.2-4. *Rozwiązanie zakłada, że zadanie dotyczy drzew wyszukiwań binarnych, zgodnie z treścią oryginalną. Ponadto doprecyzujemy pojęcie (prawego) łańcucha z treści zadania – jest to drzewo wyszukiwań binarnych, w którym każdy węzeł znajduje się na jego prawym kręgosłupie (patrz problem 13.4).*

Wykażemy stwierdzenie podane we wskazówce. Jeśli drzewo T nie jest prawym łańcuchem, czyli jeśli są w nim węzły nie leżące na prawym kręgosłupie T, to istnieje węzeł y na prawym kręgosłupie, którego lewy syn x nie leży na nim. Zauważmy, że wykonanie prawej rotacji na węźle y dodaje x do prawego kręgosłupa, jednocześnie nie usuwając z niego żadnego innego węzła. Prawa rotacja zwiększa więc rozmiar prawego kręgosłupa o 1, dlatego wystarczy wykonać co najwyżej $n-1$ rotacji, aby przekształcić T w prawy łańcuch.

Jeśli znamy ciąg prawych rotacji przekształcających drzewo T w prawy łańcuch T', to możemy powrócić od T' do T poprzez wykonanie tego ciągu w odwrotnej kolejności, zamieniając każdą prawą rotację w odpowiadającą jej lewą rotację.

Niech T_1, T_2 będą drzewami wyszukiwań binarnych o n węzłach, a T' – jedynym prawym łańcuchem powstałym z T_1 bądź z T_2 po przeprowadzeniu powyżej opisanej transformacji. Niech $r = \langle r_1, r_2, \ldots, r_k \rangle$ i $r' = \langle r'_1, r'_2, \ldots, r'_{k'} \rangle$ będą ciągami prawych rotacji przekształcających, odpowiednio T_1 w T' i T_2 w T'. Z poprzedniego paragrafu wiemy, że istnieją ciągi r i r', gdzie k, $k' \leqslant n-1$. Dla każdej prawej rotacji r'_i niech l'_i będzie odpowiadającą jej lewą rotacją. Wówczas ciąg $\langle r_1, r_2, \ldots, r_k, l'_{k'}, l'_{k'-1}, \ldots, l'_1 \rangle$ co najwyżej $2n-2$ rotacji pozwala przekształcić drzewo T_1 w T_2.

13.2-5. *Treść zadania nie definiuje n – przyjmiemy, że jest to rozmiar drzew T_1 i T_2.*

W drzewie T_1 stanowiącym prawy łańcuch (definicja w komentarzu do rozwiązania zad. 13.2-4) nie można wykonać żadnej prawej rotacji, gdyż nie istnieje w nim węzeł o niepustym lewym synu. A zatem T_1 nie jest prawostronnie przekształcalne do jakiegokolwiek drzewa wyszukiwań binarnych T_2 różnego od T_1. Innym przykładem drzew T_1 i T_2 są dowolne 2 drzewa, które przechowują różne zbiory kluczy, gdyż rotacje zmieniają strukturę drzewa, a nie klucze w węzłach.

Niech teraz T_1, T_2 będą drzewami wyszukiwań binarnych o n węzłach takimi, że T_1 jest prawostronnie przekształcalne do T_2. Załóżmy, że $n > 0$ – w przeciwnym przypadku bowiem drzewa są puste i nie jest potrzebna żadna rotacja, aby przekształcić T_1 w T_2.

Wykonanie prawej rotacji na węźle x skutkuje tym, że jego lewy syn zostaje przeniesiony na mniejszą głębokość w drzewie, stając się nowym synem ojca x. Wynika stąd, że jedynie węzły leżące na lewym kręgosłupie drzewa mogą być przeniesione w miejsce korzenia. Korzeń r drzewa T_2 leży zatem na lewym kręgosłupie T_1. Można więc z T_1 utworzyć drzewo T'_1 poprzez wykonywanie prawych rotacji na aktualnym korzeniu, aż r stanie się korzeniem. Oczywiście T'_1 jest prawostronnie przekształcalne do T_2, więc także lewe i prawe poddrzewo T'_1 jest prawostronnie przekształcalne do, odpowiednio, lewego i prawego poddrzewa T_2. W celu otrzymania drzewa T_2 wystarczy więc powtórzyć opisane ciągi prawych rotacji rekurencyjnie na lewym i prawym poddrzewie drzewa T'_1.

Liczbę wykonanych rotacji przekształcających drzewo T_1 w T'_1 można ograniczyć od góry przez długość lewego kręgosłupa drzewa T_1, która wynosi co najwyżej $n-1$. Wywołania rekurencyjne mogą następnie wykonywać podobne ciągi rotacji na każdym spośród $n-1$ pozostałych

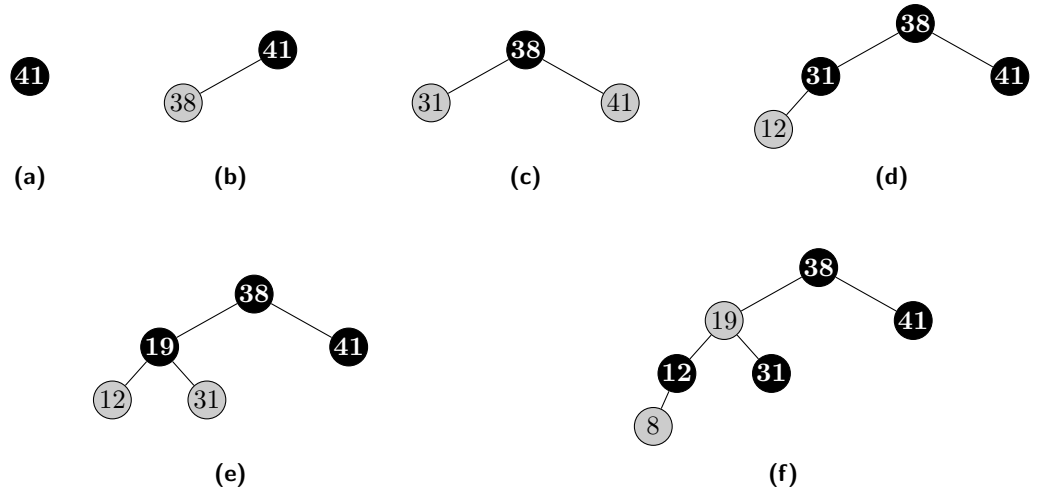
węzłów w poddrzewach drzewa T_1'. Wynika stąd, że sumaryczną liczbę rotacji wykonywanych przy przekształceniu T_1 do T_2, można ograniczyć od góry przez $O(n^2)$.

13.3. Operacja wstawiania

Linia 15 procedury RB-Insert-Fixup *powinna mówić również o zamianie lewej rotacji na prawą i vice versa.*

13.3-1. Pokolorowanie węzła z na czarno mogłoby spowodować naruszenie własności 5 drzewa czerwono-czarnego, podczas gdy procedura RB-Insert-Fixup operuje na drzewie, w którym własność ta jest zachowana i nie narusza jej w trakcie swojego działania.

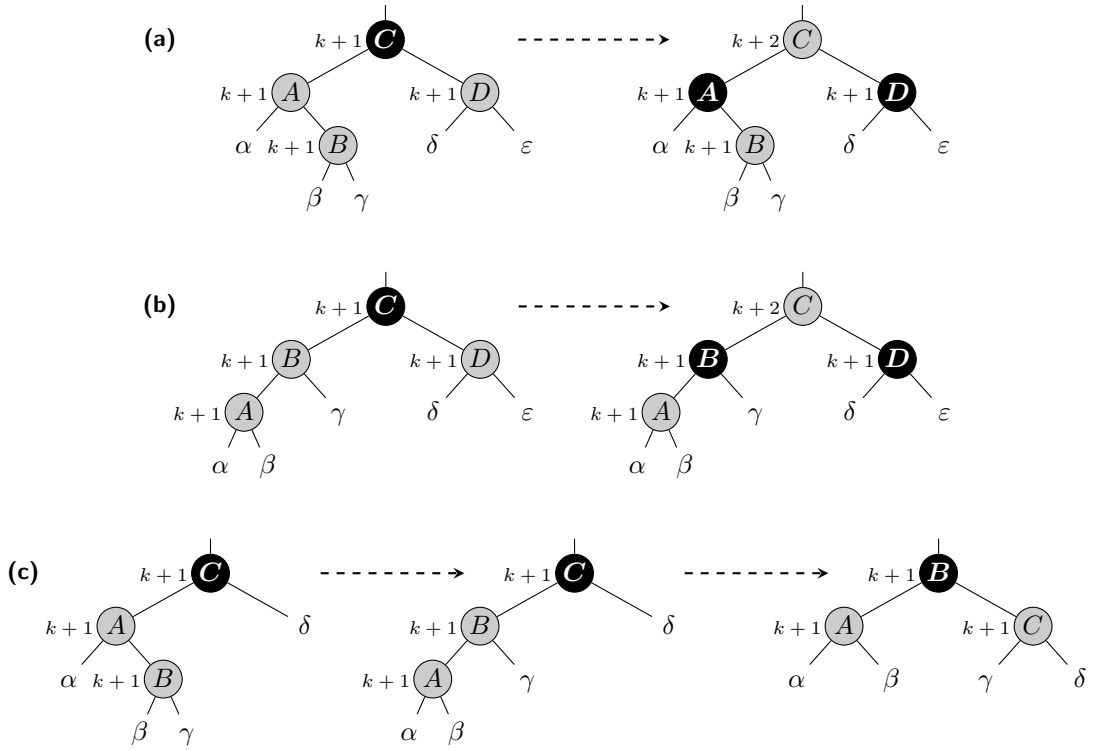
13.3-2. Na rys. 30 zilustrowano ciąg drzew czerwono-czarnych powstających po każdym wywołaniu operacji Tree-Insert dla zadanego ciągu elementów.



Rysunek 30: Drzewa czerwono-czarne powstałe po wstawieniu węzłów o kluczach 41, 38, 31, 12, 19, 8 kolejno do początkowo pustego drzewa. **(a)** Po wstawieniu pierwszego elementu drzewo składa się tylko z korzenia, który początkowo ma kolor czerwony. Przywrócenie własności 2 drzewa czerwono-czarnego następuje w ostatnim wierszu procedury RB-Insert-Fixup. **(b)** Dodanie węzła o kluczu 38 nie powoduje naruszenia żadnej własności drzewa czerwono-czarnego. **(c)–(f)** Wstawienie każdego kolejnego elementu produkuje drzewo, w którym naruszona jest własność 4 przywracana następnie w procedurze RB-Insert-Fixup.

13.3-3. Wersja rysunków 13.5 i 13.6 z Podręcznika z podanymi czarnymi wysokościami węzłów została zaprezentowana na rys. 31. Wartości te wyznaczone są jednoznacznie dla każdego węzła, co oznacza, że własność 5 drzewa czerwono-czarnego faktycznie pozostaje zachowana podczas działania procedury RB-Insert-Fixup.

13.3-4. Jedyny węzeł, który jest kolorowany na czerwono w trakcie działania procedury RB-Insert-Fixup, to $p[p[z]]$. Pętla **while** wykonuje się tylko wtedy, gdy $p[z]$ jest czerwonym węzłem.



Rysunek 31: (a)–(b) Rys. 13.5 i **(c)** rys. 13.6 z Podręcznika z wyznaczonymi czarnymi wysokościami poszczególnych węzłów.

Jeśli $p[z]$ jest korzeniem drzewa, to warunek w wierszu 2 (oraz symetryczny z wiersza 15) jest fałszywy i pętla natychmiast kończy swe działanie. Do zmiany koloru $p[p[z]]$ dochodzi więc tylko wtedy, gdy $p[z]$ nie jest korzeniem, czyli gdy $p[p[z]] \neq nil[T]$.

13.3-5. Pokażemy, że nowy węzeł wstawiony do niepustego drzewa czerwono-czarnego za pomocą procedury RB-Insert zachowuje kolor czerwony. Bezpośrednio przed wywołaniem pomocniczej procedury RB-Insert-Fixup nowo wstawiony węzeł z jest czerwonym liściem. Jeśli ojciec węzła z jest czarny, to procedura zakończy działanie, a z zachowa swój kolor. Załóżmy więc, że ojciec węzła z jest czerwony i rozważmy przypadki, jak przy omawianiu działania procedury RB-Insert. Zmiany kolorów węzłów w drzewie odbywają się tylko w przypadkach 1 i 3, ale w żadnym z nich kolor węzła z nie ulega zmianie – pole *color* może być zmodyfikowane jedynie dla dziadka (ojca ojca) węzła z i synów dziadka węzła z. Drzewo, do którego z zostało wstawione, nie jest puste, więc z nie może być korzeniem. Dlatego również ostatni wiersz procedury RB-Insert-Fixup nie zaktualizuje koloru z na czarny.

13.3-6. Podczas wstawiania węzła do drzewa czerwono-czarnego w reprezentacji bez wskaźników do ojców, do zapamiętania ścieżki od korzenia do nowego węzła użyjemy stosu. Wstawimy do niego każdy kolejno odwiedzony węzeł drzewa, zanim wywołana zostanie procedura RB-Insert-Fixup. Dokładniej, między linią 4 a 5 w procedurze RB-Insert będziemy wywoływać Push(S, y), gdzie S jest początkowo stosem zawierającym tylko $nil[T]$ (czyli ojca korzenia drzewa). Pominiemy w niej także linię 8.

Elementy zebrane na stosie S są wystarczające dla procedury RB-Insert-Fixup, ponieważ korzysta ona ze wskaźnika p jedynie dla węzłów znajdujących się w S. Musimy następnie zmodyfikować procedurę RB-Insert-Fixup tak, aby każde odwołanie do przodka danego węzła przy pomocy wskaźnika p było zamienione na odwołanie do odpowiedniego elementu stosu. Implementacja tej procedury została przedstawiona na poniższym pseudokodzie.

```
RB-Parentless-Insert-Fixup(T, S, z)
 1  p ← Pop(S)
 2  while color[p] = red
 3        do r ← Pop(S)
 4           if p = left[r]
 5              then y ← right[r]
 6                   if color[y] = red
 7                      then color[y] ← color[p] ← black
 8                           color[r] ← red
 9                           z ← r
10                           p ← Pop(S)
11                      else if z = right[p]
12                              then zamień z ↔ p
13                                   RB-Parentless-Left-Rotate(T, z, r)
14                           color[p] ← black
15                           color[r] ← red
16                           RB-Parentless-Right-Rotate(T, r, Pop(S))
17              else (to samo co po then z zamienionymi „right" i „left")
18  color[root[T]] ← black
```

W każdej iteracji pętli **while**, oprócz węzła z, potrzebujemy znać także jego ojca i dziadka (ojca ojca). Węzły te są pobierane ze stosu S przekazywanego do procedury i umieszczane w zmiennych, odpowiednio, p i r. Ponieważ rotacje także wykorzystują wskaźniki do ojca, to wyeliminujemy te użycia poprzez przekazanie im dodatkowego parametru – ojca węzła, dla którego rotację wykonujemy. W wierszu 16 powyższej procedury ojciec węzła r jest ściągany ze stosu i przekazywany do prawej rotacji. Pobranie dodatkowego elementu ze stosu S nie powoduje jednak problemów, bo pętla **while** nie wykona więcej iteracji i żaden element nie będzie już ściągany z S.

Poniżej znajduje się operacja lewej rotacji na węźle x w drzewie T, która nie korzysta ze wskaźników na ojca. Zamiast tego otrzymuje ona trzeci parametr, będący ojcem węzła x.

```
RB-Parentless-Left-Rotate(T, x, p)
1  y ← right[x]
2  right[x] ← left[y]
3  if p = nil[T]
4     then root[T] ← y
5     else if x = left[p]
6             then left[p] ← y
7             else right[p] ← y
8  left[y] ← x
```

Implementacja prawej rotacji nie wykorzystującej wskaźników na ojca jest analogiczna.

13.4. Operacja usuwania

W procedurze RB-DELETE *w linii 3 powinna być wywoływana wersja operacji następnika dla drzew czerwono-czarnych, która różni się od* TREE-SUCCESSOR *tym, że przyjmuje drzewo T jako dodatkowy parametr oraz używa $nil[T]$ zamiast* NIL. *Z kolei w linii 15 powinny być kopiowane tylko dodatkowe dane z węzła y do z. W szczególności kolor węzła z powinien pozostać bez zmian. Ponadto w linii 22 procedury* RB-DELETE-FIXUP *nie chodzi tylko o zamianę wskaźników, ale także o zamianę każdej lewej rotacji na prawą i vice versa.*

13.4-1. Zanim wywołana zostanie procedura RB-DELETE-FIXUP, korzeniem może zostać czerwony węzeł tylko w przypadku, gdy usuwany jest korzeń drzewa, którego dokładnie jeden z synów jest czerwony. Wtedy jednak w wywołaniu RB-DELETE-FIXUP nie jest wykonywana ani jedna iteracja pętli **while** i korzeń jest kolorowany na czarno w wierszu 23.

Procedura RB-DELETE-FIXUP aktualizuje jednak kolor niektórych węzłów na czerwono i dokonuje pewnych rotacji. Przyjrzymy się tym sytuacjom i sprawdzimy, że w żadnej z nich czerwony węzeł nie staje się korzeniem drzewa. Zastosujemy identyczny podział na przypadki, jak w omawianiu działania procedury w Podręczniku. W przypadkach 1 i 3 zamiana kolorów między czarnym węzłem i jego czerwonym synem, a następnie wykonanie rotacji na tym węźle powoduje, że czerwony z nich zostaje umieszczony w drzewie głębiej niż czarny, przez co nie może on być korzeniem. W przypadku 2 na czerwono zostaje pokolorowany brat węzła x, ale oczywiście nie może być on korzeniem drzewa z racji istnienia jego ojca. W końcu, po wykonaniu przypadku 4, x jest ustawione na $root[T]$ i pętla kończy działanie, po czym koloruje x na czarno.

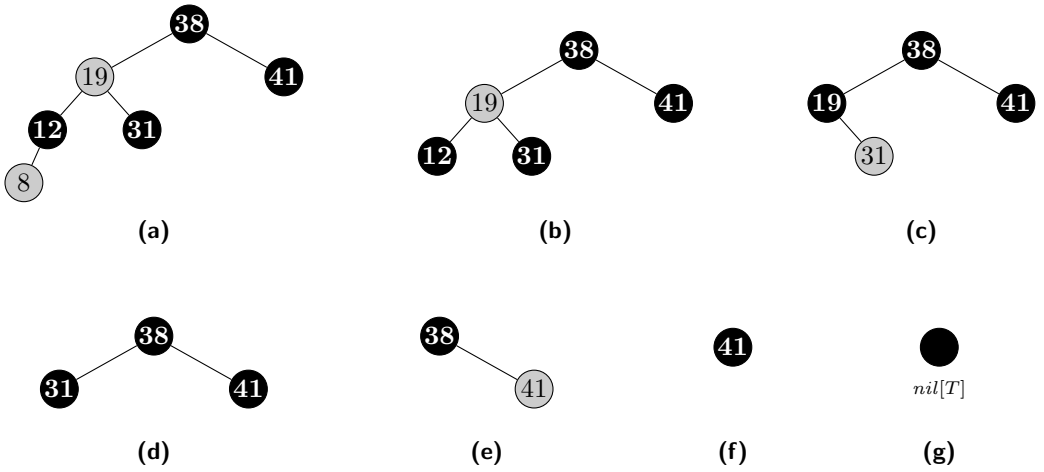
13.4-2. Jeśli węzeł x jest czerwony, to pętla **while** w procedurze RB-DELETE-FIXUP nie wykonuje się i węzeł x jest kolorowany na czarno w wierszu 23, co przywraca własność 4.

13.4-3. Ciąg drzew po każdej operacji usunięcia węzła został zilustrowany na rys. 32.

13.4-4. Gdy z drzewa czerwono-czarnego T efektywnie usuwany jest czarny węzeł y, którego obaj synowie są $nil[T]$, to do procedury RB-DELETE-FIXUP przekazywany jest jako parametr wartownik $nil[T]$, którego pole p pokazuje na $p[y]$. A zatem wszystkie odniesienia do pól parametru x w liniach 1, 2, 3, 6, 7, 8, 11, 16, 17, 18, 20, 22 i 23 tej procedury są odniesieniami do pól wartownika $nil[T]$. Ponadto w liniach 9, 12 i 13 odczytywany bądź modyfikowany jest kolor synów brata w węzła x, przy czym w także może mieć obu synów równych $nil[T]$ i operacje te mogą odbywać się na atrybucie *color* wartownika.

Zastanówmy się teraz, czy w jakimkolwiek innym miejscu w RB-DELETE-FIXUP pola $nil[T]$ mogą być odczytywane lub aktualizowane. Jeśli efektywnie usuniętym węzłem y jest korzeń drzewa T, to procedura zostaje wywołana dla $x = root[T]$ i pętla **while** nie wykonuje się. Gdy z kolei y nie jest korzeniem, to posiada brata $w \neq nil[T]$, gdyż w przeciwnym przypadku przed wywołaniem operacji usuwania własność 5 byłaby zaburzona dla $p[y]$. Jak łatwo też zauważyć, w nie zostaje nigdy zaktualizowane na $nil[T]$ w trakcie działania procedury – zarówno w linii 8, jak i 16 $right[p[x]] \neq nil[T]$. Dzięki temu w wierszach 4, 5, 10, 14 i 15 nigdy nie pojawi się wartownik $nil[T]$. Możemy wyeliminować też linijki 19 – węzeł $right[w]$ początkowo jest czerwony, dlatego nie może być $nil[T]$ – oraz 21.

Można także sprawdzić, że w żadnym z wywołań LEFT-ROTATE i RIGHT-ROTATE w procedurze RB-DELETE-FIXUP nie ma odwołań do pól wartownika, ponieważ wszystkie węzły, na których operacje te pracują, są wewnętrznymi węzłami drzewa.



Rysunek 32: Drzewa czerwono-czarne powstałe po usunięciu elementów 8, 12, 19, 31, 38, 41 kolejno z drzewa czerwono-czarnego T skonstruowanego w zad. 13.3-2. **(a)** Drzewo T przed rozpoczęciem usuwania. **(b)** Wycięcie węzła o kluczu 8 nie powoduje naruszenia żadnej własności drzewa czerwono-czarnego. **(c)–(e)** Po wycięciu kolejnych elementów, w drzewie naruszona zostaje wyłącznie własność 5. Przywracana jest ona następnie w procedurze RB-Delete-Fixup. **(f)** Po pozbawieniu drzewa przedostatniego węzła pozostaje w nim jedynie czerwony korzeń, co stanowi naruszenie własności 2. W linii 23 procedury RB-Delete-Fixup jest ona jednak natychmiast przywracana. **(g)** Usunięcie ostatniego węzła pozostawia puste drzewo czerwono-czarne, czyli składające się tylko z wartownika $nil[T]$ (którego pominęliśmy na pozostałych częściach rysunku dla zwiększenia czytelności). Wywołanie procedury RB-Delete-Fixup na wartowniku nie powoduje żadych zmian.

13.4-5. W tabeli 7 zebrano liczby czarnych węzłów znajdujących się na ścieżkach od korzenia każdego poddrzewa sprzed transformacji do poddrzew $\alpha, \beta, \ldots, \zeta$. Liczby te w każdym przypadku zgadzają się z ilościami czarnych węzłów na odpowiednich ścieżkach w drzewie po transformacji. Oznacza to, że każde przekształcenie drzewa w procedurze RB-Delete-Fixup zachowuje własność 5 drzewa czerwono-czarnego. Pamiętajmy, że węzeł x wnosi dodatkową „czarną jednostkę".

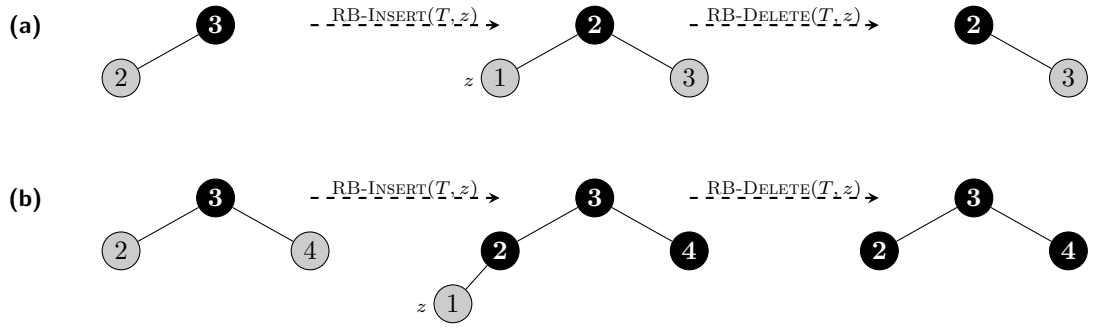
	α, β	γ, δ	ε, ζ
(a)	3	2	2
(b)	$\text{count}(c) + 2$	$\text{count}(c) + 2$	$\text{count}(c) + 2$
(c)	$\text{count}(c) + 2$	$\text{count}(c) + 1$	$\text{count}(c) + 2$
(d)	$\text{count}(c) + 2$	$\text{count}(c) + \text{count}(c') + 1$	$\text{count}(c) + 1$

Tabela 7: Liczby czarnych węzłów od korzeni poddrzew z rys. 13.7 z Podręcznika do każdego z poddrzew $\alpha, \beta, \ldots, \zeta$, zarówno sprzed, jak i po przekształceniu poddrzew.

13.4-6. Przypadek 1 ma miejsce wtedy, gdy brat w węzła x ma kolor czerwony, a zatem węzeł $p[x] = p[w]$ ma kolor czarny na mocy własności 4. Własność ta może być naruszona podczas usuwania węzła jedynie w przypadku, gdy czerwone są węzły x oraz $p[x]$, ale wtedy zostaje ona natychmiast przywrócona (zad. 13.4-2) i nie dochodzi do przypadku 1.

13.4-7. Wstawienie nowego węzła do drzewa czerwono-czarnego za pomocą procedury RB-Insert, a następnie natychmiastowe jego usunięcie przez RB-Delete nie zawsze pozostawia

takie samo drzewo. Rys. 33 przedstawia przykład, w którym zmienia się struktura drzewa po takim działaniu oraz taki, w którym struktura jest zachowana, ale zmianie ulegają kolory węzłów.



Rysunek 33: Drzewa czerwono-czarne po wstawieniu nowego węzła z o kluczu 1, a następnie natychmiastowym jego usunięciu. **(a)** Drzewo T, w którym opisane operacje powodują zmianę struktury drzewa. **(b)** Drzewo T, w którym działania te zachowują strukturę, ale zmieniają kolory węzłów.

Problemy

13-1. Zbiory dynamiczne z historią

(a) Gdy do drzewa wstawiany jest klucz k, zmianie ulegają wszystkie węzły na prostej ścieżce od korzenia drzewa do nowego węzła z kluczem k, który staje się nowym liściem drzewa.

Podczas usuwania węzła z, z drzewa rzeczywiście wycinany jest węzeł y, który jest równy z, gdy ten posiada co najwyżej jednego syna, albo jest następnikiem z, w przypadku gdy ten ma dwóch synów. Aktualizowani są wtedy wszyscy właściwi przodkowie y.

(b) Zdefiniujmy dwie pomocnicze operacje, z których będziemy korzystać:

- New-Node(k) – tworzy nowy węzeł, którego pole *key* ma wartość k, a pola *left* i *right* są ustawione na NIL; zwraca wskaźnik do nowo utworzonego węzła;
- Copy-Node(x) – tworzy nowy węzeł, którego pola *key*, *left* oraz *right* mają identyczne wartości jak odpowiadające pola węzła x; zwraca wskaźnik do nowo utworzonego węzła.

Poniższa rekurencyjna procedura Persistent-Subtree-Insert wywoływana jest z dwoma parametrami – korzeniem x drzewa, do którego wstawiany jest nowy węzeł, oraz kluczem k nowego węzła. Ścieżka od x do jednego z jego potomnych liści jest kopiowana, a na końcu kopii tej ścieżki umieszczany jest nowy węzeł o kluczu k.

```
Persistent-Subtree-Insert(x, k)
1  if x = NIL
2     then z ← New-Node(k)
3     else  z ← Copy-Node(x)
4           if k < key[x]
5              then left[z] ← Persistent-Subtree-Insert(left[x], k)
6              else  right[z] ← Persistent-Subtree-Insert(right[x], k)
7  return z
```

Wynikiem zwracanym przez tę procedurę jest nowo wstawiony węzeł albo kopia węzła x będąca jego przodkiem.

Następujący pseudokod przyjmuje na wejściu drzewo z historią T oraz klucz k i zwraca nowe drzewo z historią T' powstałe przez dodanie do T klucza k.

PERSISTENT-TREE-INSERT(T, k)
1 utwórz puste drzewo z historią T'
2 $root[T'] \leftarrow$ PERSISTENT-SUBTREE-INSERT$(root[T], k)$
3 **return** T'

(c) Każde kolejne wywołanie rekurencyjne procedury PERSISTENT-SUBTREE-INSERT schodzi o 1 poziom w dół drzewa T. Ścieżka pokonywana przez tę procedurę, wywołaną z PERSISTENT-TREE-INSERT, ma długość co najwyżej h. A zatem, przy założeniu, że operacje NEW-NODE oraz COPY-NODE działają w czasie stałym, czas potrzebny do wstawienia nowego klucza do T wynosi $O(h)$.

Z punktu (a) mamy, że podczas wstawiania nowego klucza do T w procedurze PERSISTENT-SUBTREE-INSERT skopiowany zostanie każdy przodek nowego węzła. Złożoność pamięciowa operacji wstawiania wynosi więc także $O(h)$.

(d) Jeśli w każdym węźle byłby przechowywany wskaźnik na ojca, to skopiowanie korzenia drzewa podczas wstawiania nowego węzła pociągałoby za sobą konieczność skopiowania obydwu synów korzenia, a to z kolei konieczność skopiowania także ich synów – i tak dalej, aż do liści drzewa. A zatem obecność wskaźników na ojca wymaga wykonania kopii każdego węzła w drzewie w procedurze PERSISTENT-SUBTREE-INSERT. Dla drzewa o n węzłach operacja PERSISTENT-TREE-INSERT potrzebowałaby więc czasu $\Omega(n)$ oraz $\Omega(n)$ dodatkowej pamięci.

(e) Jak zobaczyliśmy w punkcie (c), operacja wstawiania do drzewa z historią działa w czasie proporcjonalnym do wysokości tego drzewa. Jeśli do implementacji drzew z historią użylibyśmy drzew czerwono-czarnych, to moglibyśmy zagwarantować niski czas działania tej operacji. W implementacji tej w żadnym węźle nie możemy jednak przechowywać pola wskazującego na ojca, gdyż wtedy bylibyśmy zmuszeni do kopiowania całego drzewa przy wstawianiu jednego węzła (patrz punkt (d)). Możliwe jest jednak zaimplementowanie operacji wstawiania do drzewa czerwono-czarnego nie wykorzystującego wskaźników na ojca, poprzez użycie stosu – szczegóły zostały opisane w zad. 13.3-6. Możemy zmodyfikować nieco implementację podaną w tamtym rozwiązaniu tak, aby zamiast dokonywać zmian na drzewie wejściowym, odpowiednie węzły były kopiowane, a na końcu działania, by zwracane było nowe drzewo uzupełnione o nowy węzeł, podobnie jak w PERSISTENT-TREE-INSERT z punktu (b).

Wystarczy jeszcze pokazać, że w trakcie wstawiania do drzewa z historią T o n węzłach opisana wersja procedury RB-INSERT kopiuje nie więcej niż $O(\lg n)$ węzłów w wyniku wykonywania rotacji i zmian kolorów. Niech s będzie ścieżką złożoną ze skopiowanych węzłów, zanim została wywołana procedura RB-INSERT-FIXUP – czyli od kopii korzenia do nowo wstawionego węzła. Procedura ta wykonuje co najwyżej 2 rotacje, a każda z nich modyfikuje wskaźniki do synów 3 węzłów – tego, wokół którego rotacja jest wykonywana, jego ojca oraz jednego z jego synów. Wszystkie one stanowią jednak fragment ścieżki s, dlatego zostały skopiowane jeszcze wewnątrz RB-INSERT.

W przypadku 1 w procedurze RB-INSERT-FIXUP zmieniany jest kolor dziadka (ojca ojca) aktualnego węzła oraz synowie dziadka. Zarówno dziadek, jak i jeden z jego synów znajdują się na ścieżce s – należy więc wykonać kopię drugiego syna (stryja aktualnego węzła). Jest to jednak jedyny dodatkowo kopiowany węzeł w aktualnej iteracji pętli w procedurze RB-INSERT-FIXUP,

ponieważ jego ojciec leży już na s. Pętla może sumarycznie wykonać $O(\lg n)$ iteracji i tyle samo węzłów może wymagać wykonania ich kopii z powodu aktualizacji kolorów.

Widzimy zatem, że wstawianie do drzewa z historią reprezentowanego przez drzewo czerwono-czarne o n węzłach bez wskaźników do ojca wymaga czasu $O(\lg n)$.

Do usuwania węzła z drzewa z historią reprezentowanego przez drzewo czerwono-czarne możemy podać wersję procedury RB-DELETE z modyfikacjami podobnymi do tych opisanych dla wstawiania. Skopiowane węzły sprzed wywołania RB-DELETE-FIXUP, czyli z punktu (a) właściwych przodków efektywnie usuniętego węzła (i jego syna x w przypadku, gdy $color[x] = \text{RED}$), można przechować na stosie, co pozwoli dostać się do nich bez użycia w tym celu wskaźników na ojca.

Analogicznie do analizy operacji wstawiania można sprawdzić, że w wywołaniu RB-DELETE-FIXUP w każdej iteracji pętli aktualizowanych jest $O(1)$ węzłów przez wykonane rotacje i modyfikacje kolorów. Sumarycznie procedura ta skopiuje zatem dodatkowo co najwyżej $O(\lg n)$ węzłów, dlatego czas działania operacji usuwania także wynosi $O(\lg n)$.

13-2. Złączanie drzew czerwono-czarnych

(a) Podczas wstawiania nowego węzła do drzewa T czarna wysokość T może zmienić się tylko wtedy, gdy czerwony korzeń T zostaje pokolorowany na czarno. Można więc w procedurze RB-INSERT-FIXUP bezpośrednio przed ostatnim wierszem inkrementować wartość $bh[T]$ w przypadku, gdy $color[root[T]] = \text{RED}$.

W usuwaniu węzła wszystkie przypadki w procedurze RB-DELETE-FIXUP zachowują własność 5, o ile przyjmiemy, że węzeł aktualnie pokazywany przez x wnosi dodatkową „czarną jednostkę". Czarna wysokość drzewa T może być więc zmodyfikowana tylko wtedy, kiedy nadmiarowa „czarna jednostka" jest przenoszona na korzeń drzewa i „zapominana" w przypadku 2. Wystarczy zatem bezpośrednio po linii 11 zmniejszyć $bh[T]$ o 1, gdy $x = root[T]$. W przypadku 4 wskaźnik x także jest ustawiany na korzeń, ale zanim zostanie wykonane przypisanie z wiersza 21, w drzewie zachowana jest już własność 5 bez potraktowania węzła wskazywanego przez x jako „nadmiarowo" czarnego. Przypisanie to służy tylko do przerwania pętli.

Czarna wysokość korzenia drzewa T to oczywiście $bh[T]$. Jeśli dany węzeł jest czerwony, to ma tę samą czarną wysokość, co jego ojciec. Gdy natomiast jest czarny (ale nie jest korzeniem), to jego czarna wysokość jest o 1 mniejsza od czarnej wysokości ojca. Możemy więc wyznaczyć czarne wysokości wszystkich węzłów na prostej ścieżce od korzenia drzewa T do jednego z jego liści w czasie proporcjonalnym do długości tej ścieżki.

(b) Jak wyjaśnimy w punkcie (f), w celu zaprojektowania efektywnej operacji złączania drzew czerwono-czarnych, musimy zrezygnować w ich implementacji z wartownika, a zamiast niego korzystać z wartości NIL, podobnie jak w zwykłych drzewach wyszukiwań binarnych.

Poniższy pseudokod realizuje opisany w treści algorytm, przy założeniu, że drzewa czerwono-czarne zaimplementowane są bez użycia wartownika.

```
RB-JOIN-POINT(T_1, T_2)
1  y ← root[T_1]
2  b ← bh[T_1]
3  while b > bh[T_2]
4        do if right[y] ≠ NIL
5              then y ← right[y]
6              else  y ← left[y]
7           if y = NIL lub color[y] = BLACK
8              then b ← b − 1
9  return y
```

Algorytm przechodzi drzewo T_1 od korzenia w dół, wybierając w pierwszej kolejności prawego syna aktualnego węzła y, a jeżeli ten jest NIL, to wtedy lewego syna y. Dzięki temu odwiedzane są węzły o największym możliwym kluczu na danej czarnej wysokości w drzewie T_1. Algorytm kończy działanie, gdy czarna wysokość b węzła y, wyznaczona na podstawie obserwacji z punktu (a), zrówna się z wartością $bh[T_2]$. Gdy to się wydarzy, węzeł y jest czarny i następuje jego zwrócenie. Zauważmy, że test z linii 7 traktuje NIL jak wirtualny czarny węzeł, którego napotkanie także prowadzi do dekrementacji b.

Algorytm może odwiedzić wszystkie węzły na prostej ścieżce od korzenia do liścia NIL. Ponieważ T_1 jest drzewem czerwono-czarnym o co najwyżej n węzłach, to czas działania algorytmu można ograniczyć od góry przez $O(\lg n)$.

(c) Na podstawie założenia klucz korzenia y drzewa T_y, będącego poddrzewem T_1, jest mniejszy lub równy od $key[x]$. Można zatem, bez naruszenia własności drzewa wyszukiwań binarnych, umieścić węzeł x między y a $p[y]$, czyniąc T_y lewym poddrzewem x, a T_2 – jego prawym poddrzewem.

(d) Jeżeli pokolorujemy x na czerwono, to czarna wysokość węzła $p[x]$ (o ile $x \neq root[T]$) nie zmieni się i czerwono-czarne własności 1, 3 i 5 będą spełnione. W przypadku gdy $x = root[T]$, to naruszona może być czerwono-czarna własność 2, a jeśli $x \neq root[T]$, to węzeł $p[x]$ także może być czerwony, co powoduje naruszenie własności 4. Z identyczną sytuacją mieliśmy do czynienia podczas wstawiania węzła do drzewa czerwono-czarnego przy pomocy procedury RB-INSERT. Wystarczy więc wywołać RB-INSERT-FIXUP(T, x), aby przywrócić naruszone własności 2 i 4 w drzewie T w czasie $O(\lg n)$.

(e) W sytuacji symetrycznej algorytm z punktu (b) będzie wyszukiwał w drzewie T_2 czarny węzeł y o możliwie najmniejszym kluczu spośród węzłów o czarnej wysokości $bh[T_1]$, preferując lewych synów w poruszaniu się po ścieżce w dół drzewa. Można to zrealizować, modyfikując pseudokod RB-JOIN-POINT poprzez zamianę T_1 z T_2 w liniach 1–3 oraz zamianę pól *left* i *right* w liniach 4–6. Węzeł x należy następnie umieścić w drzewie T_2 jako nowego ojca węzła y, który teraz staje się prawym synem x, a drzewo T_1 uczynić lewym poddrzewem x. Nadanie koloru węzłowi x i przywrócenie własności drzewa czerwono-czarnego w wynikowym drzewie T odbywa się identycznie jak w punkcie (d).

(f) Prześledźmy działanie algorytmu RB-JOIN, którego pseudokod podaliśmy poniżej.

```
RB-JOIN(T_1, x, T_2)
 1  utwórz puste drzewo czerwono-czarne T
 2  if bh[T_1] ⩾ bh[T_2]
 3     then if root[T_2] = NIL
 4             then RB-INSERT(T_1, x)
 5                  return T_1
 6          root[T] ← x
 7          bh[T] ← bh[T_1]
 8          y ← RB-JOIN-POINT(T_1, T_2)
 9          left[x] ← y
10          right[x] ← root[T_2]
11          if y ≠ root[T_1]
12             then if y = left[p[y]]
13                     then left[p[y]] ← x
14                     else  right[p[y]] ← x
15                  root[T] ← root[T_1]
16                  p[x] ← p[y]
17          p[root[T_2]] ← p[y] ← x
18     else (to samo co po then odwrócone symetrycznie)
19  color[x] ← RED
20  RB-INSERT-FIXUP(T, x)
21  return T
```

W wierszach 2–17 wykonywane są działania w przypadku, gdy czarna wysokość drzewa T_2 nie przekracza czarnej wysokości drzewa T_1. Jeśli drzewo T_2 jest puste, to w celu złączenia T_1 oraz węzła x, wystarczy ten ostatni wstawić do T_1 za pomocą zwykłej operacji wstawiania, po czym zwrócić wynikowe drzewo T_1. W przeciwnym przypadku wykonywane są operacje opisane w punkcie (c). W wierszu 18 obsługiwany jest przypadek, gdy $bh[T_1] < bh[T_2]$, poprzez przeprowadzenie analogicznych działań do tych z linii 2–17, ale z zamienionymi T_1 i T_2 oraz *left* i *right*. Wywołanie odpowiadające temu z linii 8 jest z kolei zastąpione wersją symetryczną opisaną w części (e). Pod koniec działania procedura koloruje węzeł x na czerwono i przywraca naruszone własności czerwono-czarne w drzewie T.

Na podstawie analizy z poprzednich punktów nietrudno wywnioskować, że czas działania algorytmu RB-JOIN, działającego na drzewach o sumarycznie n węzłach, wynosi $O(\lg n)$.

Zastanówmy się teraz, dlaczego wartownik w reprezentacji drzew czerwono-czarnych powoduje problemy w efektywnej implementacji operacji złączania. W takiej reprezentacji po zakończeniu działania algorytmu wskaźniki na synów niektórych węzłów w T_1 pokazują na $nil[T_1]$, a niektórych węzłów w T_2 – na $nil[T_2]$. Tuż przed zakończeniem działania algorytmu moglibyśmy wykonać przypisanie $nil[T] \leftarrow nil[T_1]$, dzięki któremu wartownik w drzewie T byłby poprawnie ustawiony dla węzłów oryginalnie znajdujących się w drzewie T_1. Jednakże wszystkie węzły początkowo znajdujące się w T_2, których co najmniej jeden z synów jest liściem, nadal pokazywałyby na wartownika $nil[T_2] \neq nil[T]$. Liczba takich węzłów może być rzędu $O(n)$, dlatego modyfikacja ich pól wskazujących na $nil[T_2]$ zwiększyłaby czas działania procedury RB-JOIN do $O(n)$.

Wartownik służy jedynie do uproszczenia operacji na drzewach czerwono-czarnych. Drzewa w implementacji bez wartownika zachowują własność zrównoważenia, dzięki czemu podstawowe operacje słownikowe wciąż działają na nich w czasie logarytmicznym względem ich rozmiaru. Zmodyfikowane wersje tych operacji są tylko nieznacznie bardziej skomplikowane z powodu konieczności wykrywania przypadków, gdy dany wskaźnik jest NIL i traktowania tej wartości jako wirtualnego czarnego liścia.

13-3. Drzewa AVL

(a) Udowodnimy stwierdzenie ze wskazówki, wykorzystując indukcję po wysokościach drzew AVL. Jako podstawę indukcji przyjmiemy drzewa AVL o wysokościach 0 i 1. W drzewie o wysokości 0 jest tylko jeden węzeł, czyli więcej niż $F_0 = 0$, zaś drzewo o wysokości 1 może składać się z minimalnie 2 węzłów, co jest większe niż $F_1 = 1$.

Niech teraz dane będzie drzewo AVL o wysokości $h \geqslant 2$ i niech h_L będzie wysokością jego lewego poddrzewa, a h_R – wysokością jego prawego poddrzewa. Bez utraty ogólności możemy przyjąć, że $h_L \leqslant h_R$, skąd $h = h_R+1$. Na podstawie definicji drzewa AVL mamy, że $|h_R - h_L| \leqslant 1$, a więc $h_L \geqslant h_R - 1$. Z kolei z założenia indukcyjnego mamy, że liczby węzłów n_L i n_R w lewym i prawym poddrzewie wynoszą, odpowiednio, co najmniej F_{h_L} i co najmniej F_{h_R}. Stąd liczba węzłów drzewa wynosi

$$n = n_L + n_R + 1 \geqslant F_{h_L} + F_{h_R} + 1 \geqslant F_{h_R-1} + F_{h_R} + 1 = F_{h_R+1} + 1 = F_h + 1 > F_h.$$

Z zad. 3.2-7 mamy $F_h \geqslant \phi^{h-2}$ dla każdego $h \geqslant 2$, a zatem $n > F_h \geqslant \phi^{h-2}$, skąd otrzymujemy ostatecznie $h < \log_\phi n + 2 = O(\lg n)$.

(b) Zanim podamy pseudokod algorytmu BALANCE, zdefiniujemy pomocnicze procedury, które będą w nim wykorzystywane.

Dla każdego węzła x w drzewie wyszukiwań binarnych definiujemy **współczynnik zrównoważenia** x jako wartość zwracaną przez następującą procedurę:

```
BALANCE-FACTOR(x)
1  hl ← hr ← −1
2  if left[x] ≠ NIL
3      then hl ← h[left[x]]
4  if right[x] ≠ NIL
5      then hr ← h[right[x]]
6  return −hl + hr
```

Współczynnik zrównoważenia dowolnego węzła w drzewie AVL przyjmuje wartość -1, 0 lub 1.

Dla danego węzła x procedura HEIGHT(x) zwraca aktualną wysokość x w drzewie na podstawie wartości pola h jego synów. Jej działanie jest identyczne z BALANCE-FACTOR(x) z wyjątkiem ostatniego wiersza, w którym zwracana jest wartość $\max(hl, hr) + 1$.

Będziemy też korzystać z procedur AVL-LEFT-ROTATE i AVL-RIGHT-ROTATE, których zadaniem jest wykonanie rotacji na węźle x w drzewie niekoniecznie zrównoważonym po wysokościach. Różnią się one od LEFT-ROTATE i RIGHT-ROTATE tym, że nie przyjmują T jako parametru i nie modyfikują wskaźnika $root[T]$ w wierszu 6, a tuż przed zakończeniem działania aktualizują jeszcze $h[x]$ i $h[y]$ na wartości, odpowiednio, HEIGHT(x) i HEIGHT(y).

```
BALANCE(x)
 1  if BALANCE-FACTOR(x) = −2
 2     then if BALANCE-FACTOR(left[x]) = 1
 3             then AVL-LEFT-ROTATE(left[x])          ▷ Przypadek „lewo-prawo”
 4          AVL-RIGHT-ROTATE(x)                       ▷ Przypadek „lewo-lewo”
 5     else if BALANCE-FACTOR(x) = 2
 6             then if BALANCE-FACTOR(right[x]) = −1
 7                     then AVL-RIGHT-ROTATE(right[x]) ▷ Przypadek „prawo-lewo”
 8                  AVL-LEFT-ROTATE(x)                ▷ Przypadek „prawo-prawo”
 9          return p[x]
10  return x
```

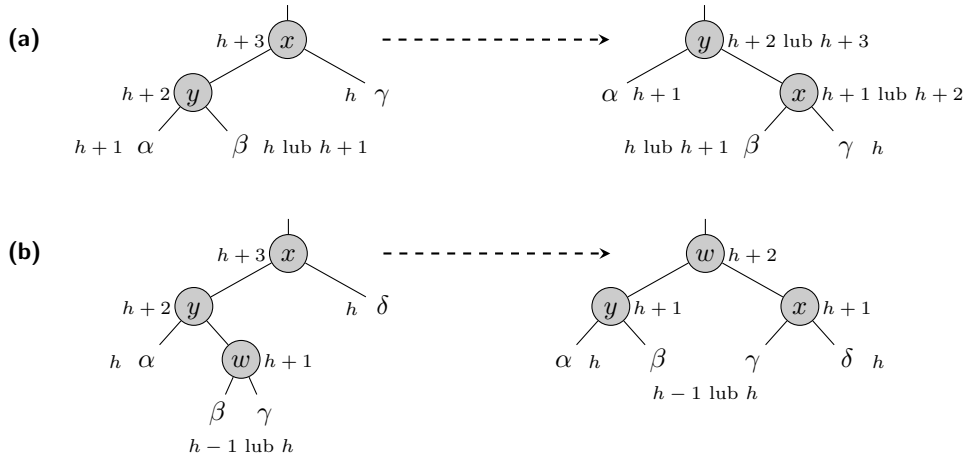
Wywołanie BALANCE(x) modyfikuje drzewo, gdy $|\text{BALANCE-FACTOR}(x)| = 2$ i przy założeniu, że dla każdego właściwego potomka y węzła x zachodzi $|\text{BALANCE-FACTOR}(y)| \leqslant 1$. Rozważmy 4 przypadki w zależności od współczynników zrównoważenia x oraz jego synów. W przypadku, który określimy jako „lewo-lewo”, prawdziwe są warunki $\text{BALANCE-FACTOR}(x) = -2$ oraz $\text{BALANCE-FACTOR}(left[x]) \in \{-1, 0\}$. Aby zrównoważyć poddrzewo o korzeniu w x, wystarczy wykonać na x prawą rotację. Przypadek „lewo-prawo” zachodzi, gdy $\text{BALANCE-FACTOR}(x) = -2$ i $\text{BALANCE-FACTOR}(left[x]) = 1$. Można go jednak sprowadzić do przypadku „lewo-lewo”, wykonując lewą rotację na $left[x]$. Wynikiem zwracanym przez procedurę jest węzeł o najmniejszej głębokości, którego głębokość została zmodyfikowana przez rotacje w danym wywołaniu, czyli taki, który zastąpił x w roli korzenia modyfikowanego poddrzewa, albo x, jeżeli drzewo nie było zmieniane. Działanie procedury dla opisanych przypadków przedstawione zostało na rys. 34.

Pozostałe przypadki, "prawo-prawo" oraz „prawo-lewo”, są symetryczne do poprzednich, gdzie tym razem $\text{BALANCE-FACTOR}(x) = 2$.

Można zauważyć, że w trakcie działania procedury BALANCE wskaźnik $root[T]$ nie jest aktualizowany. Jest ona jednak procedurą pomocniczą wykorzystywaną w algorytmie wstawiania węzła do drzewa AVL, który opiszemy w następnym punkcie i to na nim będzie spoczywać odpowiedzialność za utrzymanie aktualnego wskaźnika na korzeń drzewa T.

(c) Operacja AVL-INSERT opiera się na zmodyfikowanej procedurze RECURSIVE-TREE-INSERT z zad. 12.3-1. Wstawienie nowego węzła z do zrównoważonego po wysokościach drzewa wyszukiwań binarnych może jednak zaburzyć to zrównoważenie. Dokładniej, dla każdego przodka x nowo wstawionego węzła z, wartość $h[x]$ może być nieaktualna, a poddrzewo o korzeniu w x może nie być drzewem AVL. Problemy te da się naprawić poprzez aktualizację $h[x]$ wynikiem zwracanym przez HEIGHT(x) oraz wywołanie dla x procedury BALANCE opisanej w części (b). Wartością zwracaną przez AVL-INSERT jest aktualny korzeń drzewa, do którego wstawiane było z.

```
AVL-INSERT(x, z)
1  if x = NIL
2     then return z
3  if key[z] < key[x]
4     then left[x] ← AVL-INSERT(left[x], z)
5          p[left[x]] ← x
6     else right[x] ← AVL-INSERT(right[x], z)
7          p[right[x]] ← x
8  h[x] ← HEIGHT(x)
9  return BALANCE(x)
```



Rysunek 34: Działanie procedury BALANCE wywołanej dla węzła x, który nie jest zrównoważony po wysokościach. Obok węzłów x, y, w i poddrzew α, β, γ, δ zaznaczone zostały ich wysokości. **(a)** Przypadek „lewo-lewo". Jeśli przyjmiemy, że wysokością poddrzewa γ jest h, to wysokość poddrzewa o korzeniu w y wynosi $h + 2$. Zachodzi BALANCE-FACTOR$(y) \in \{-1, 0\}$, więc poddrzewo α ma wysokość $h + 1$, a poddrzewo β może mieć wysokość równą h lub $h + 1$. Poddrzewo o korzeniu w x po lewej stronie rysunku zostaje przekształcone w zrównoważone poddrzewo o korzeniu w y po prawej stronie rysunku po wykonaniu AVL-RIGHT-ROTATE(x). **(b)** W przypadku „lewo-prawo" po przyjęciu, że wysokością δ jest h, mamy, że wysokość poddrzewa o korzeniu w węźle y wynosi $h+2$. Tutaj jednak BALANCE-FACTOR$(y) = 1$, więc wyróżniamy w – prawego syna y o wysokości $h + 1$, którego obydwa poddrzewa, β i γ, mają wysokość $h-1$ lub h. Zrównoważenie poddrzewa o korzeniu w x odbywa się poprzez wywołanie najpierw AVL-LEFT-ROTATE(y), a następnie AVL-RIGHT-ROTATE(x).

Podobnie jak w przypadku RECURSIVE-TREE-INSERT, z powyższej procedury nie należy korzystać bezpośrednio, ale poprzez wywołanie poniższego pseudokodu opakowującego, który aktualizuje pole *root* drzewa T, do którego wstawiany jest węzeł z.

```
AVL-INSERT-WRAPPER(T, z)
1  root[T] ← AVL-INSERT(root[T], z)
```

(d) *Przykład, którego podania wymaga polecenie, nie istnieje. Przedstawimy więc rozwiązanie dla treści z erraty do oryginału, czego tłumaczenie brzmi:*

Pokaż, że operacja AVL-INSERT dla drzewa AVL o n węzłach działa w czasie $O(\lg n)$ i wykonuje $O(1)$ rotacji.

Rozwiązanie:

Wszystkie procedury pomocnicze, które opisaliśmy w punkcie (b), czyli BALANCE-FACTOR, HEIGHT, operacje rotacji oraz BALANCE, działają w czasie stałym. Na każdym poziomie rekurencji algorytm AVL-INSERT wykonuje więc $O(1)$ operacji. Liczbę poziomów rekurencji można z kolei ograniczyć od góry przez wysokość h drzewa, na którym on działa. W drzewie o n węzłach $h = O(\lg n)$, zatem czas działania tego algorytmu dla drzewa o n węzłach wynosi $O(\lg n)$.

Na każdym poziomie rekursji w AVL-INSERT wywoływana jest operacja BALANCE. Wykażemy jednak, że sumaryczna liczba rotacji w niej przeprowadzanych jest nie większa niż 2. Po umieszczeniu węzła z w drzewie rekurencja wraca, wywołując BALANCE dla przodków z na coraz

mniejszych głębokościach drzewa. Rotacje zostaną wykonane w BALANCE tylko wówczas, gdy współczynnik zrównoważenia aktualnego przodka z wynosi -2 lub 2. Niech x będzie pierwszym napotkanym węzłem o tej własności, czyli najgłębiej położonym przodkiem węzła z, który nie jest zrównoważony po wysokościach. Sprawdźmy, jak zmieni się wysokość poddrzewa o korzeniu w x w każdym z przypadków rozważanych w procedurze BALANCE.

Posługując się oznaczeniami z rys. 34, w przypadku „lewo-lewo" zanim nowy węzeł z został umieszczony w drzewie, węzeł x znajdujący się na wysokości $h+2$ był zrównoważony po wysokościach, więc węzeł y miał wysokość $h+1$, a wysokość poddrzew α i β wynosiła h. Nowy węzeł musiał więc zostać dodany do poddrzewa α. Po wykonaniu rotacji w wierszu 4 procedury BALANCE wysokość węzła x wynosi $h+1$, a wysokością y jest $h+2$. Wywołanie procedury BALANCE przywraca zatem nie tylko zrównoważenie poddrzewa, na którym ona działa, ale także przywraca wysokość tego poddrzewa do wartości sprzed wstawienia węzła z. Dzięki temu zrównoważenie po wysokościach każdego przodka węzła y pozostaje nienaruszone i w dalszym działaniu algorytmu AVL-INSERT nie zostanie wykonana już żadna rotacja.

Jeśli z kolei wstawienie z doprowadziło do przypadku „lewo-prawo", to początkowo wysokość węzła y wynosiła $h+1$, a wysokość węzła x wynosiła $h+2$. Po wstawieniu z do drzewa oraz wykonaniu rotacji w liniach 3 i 4 poddrzewo o korzeniu w x zostaje przekształcone w zrównoważone po wysokościach poddrzewo o korzeniu w w o wysokości $h+2$ równej tej sprzed przekształcenia. Każdy przodek węzła w pozostaje więc zrównoważony, dlatego rotacje nie będą już wykonywane.

Analiza w przypadkach „prawo-prawo" i „prawo-lewo" jest analogiczna.

13-4. Drzepce, czyli drzewa „treaps"

(a) Dla danego zbioru węzłów $x_1, x_2, \ldots, x_n$ istnieje wiele możliwych drzew wyszukiwań binarnych zbudowanych z tych węzłów. Jeśli $key[x_i] < key[x_j]$, to węzeł x_i może znajdować się w lewym poddrzewie węzła x_j albo x_j może znajdować się w prawym poddrzewie x_i. Powiązanie z każdym węzłem priorytetów i utrzymywanie na ich podstawie własności kopca typu min w drzepcu sprawia, że niejednoznaczność ta znika – węzeł o wyższym priorytecie znajduje się w drzepcu głębiej od węzła o niższym priorytecie. A zatem, jeśli dodatkowo $priority[x_i] < priority[x_j]$, to węzeł x_i jest przodkiem x_j, a w przeciwnym przypadku x_i jest potomkiem x_j. Dla każdej pary węzłów z danego zbioru istnieje jednoznacznie wyznaczone ich wzajemne rozmieszczenie w drzepcu, dlatego istnieje dokładnie jeden drzepiec dla tego zbioru węzłów i powiązanych z nimi priorytetów.

(b) Niech T będzie drzepcem zbudowanym z węzłów $x_1, x_2, \ldots, x_n$. Załóżmy bez utraty ogólności, że priorytety węzłów w T zostały wybrane ze zbioru $\{1, 2, \ldots, n\}$ losowo i niezależnie od każdego węzła. Oznaczmy przez π permutację tego zbioru przyporządkowującą priorytety dla ciągu węzłów drzepca T taką, że dla $i = 1, 2, \ldots, n$ priorytetem x_i jest $\pi(i)$. Oczywiście każda z $n!$ takich permutacji jest jednakowo prawdopodobna, co oznacza, że każda permutacja kluczy węzłów, wyznaczona przez ich rosnące priorytety, czyli π^{-1}, jest również jednakowo prawdopodobna. Możemy zatem potraktować drzepiec T jak drzewo wyszukiwań binarnych utworzone przez wstawienie do niego węzłów o coraz większym priorytecie, czyli, kolejno $x_{\pi^{-1}(1)}, x_{\pi^{-1}(2)}, \ldots, x_{\pi^{-1}(n)}$. Drzewo T jest zatem losowo skonstruowanym drzewem wyszukiwań binarnych, które, na podstawie analizy z podrozdziału 12.4, ma wysokość $h = O(\lg n)$. Wynik ten, wraz z dolnym oszacowaniem na h otrzymanym w zad. B.5-4, daje nam oczekiwaną wysokość drzepca równą $\Theta(\lg n)$.

(c) Procedura TREAP-INSERT wstawia nowy węzeł x do drzepca T tak, jakby T było zwykłym drzewem wyszukiwań binarnych. Jeśli priorytet nowego węzła jest mniejszy od priorytetu jego ojca, to mamy naruszoną własność kopca typu min i procedura przystępuje do jej przywrócenia.

Wykonuje w tym celu odpowiednie rotacje, przenosząc węzeł x o jeden poziom w górę drzepca aż do momentu, gdy x jest korzeniem albo gdy własność kopca między x i $p[x]$ jest już spełniona. Gdy x jest lewym synem $p[x]$, to na $p[x]$ wykonywana jest prawa rotacja, natomiast gdy x jest prawym synem $p[x]$ – lewa rotacja.

Operacja wstawiania węzła do drzepca została przedstawiona na poniższym pseudokodzie:

```
TREAP-INSERT(T, x)
1  TREE-INSERT(T, x)
2  while x ≠ root[T] i priority[x] < priority[p[x]]
3        do if x = left[p[x]]
4              then RIGHT-ROTATE(T, p[x])
5              else  LEFT-ROTATE(T, p[x])
```

(d) Jeśli h jest wysokością drzepca T przed wstawieniem nowego węzła, to wywołanie z linii 1 potrzebuje czasu proporcjonalnego do h, a liczba rotacji wykonywanych w trakcie działania pętli **while** nigdy nie przekracza $h+1$. A zatem oczekiwany czas działania procedury TREAP-INSERT wynosi $\Theta(h)$, co na mocy faktu wykazanego w punkcie (b) jest równe $\Theta(\lg n)$.

(e) Bezpośrednio po umieszczeniu nowego węzła x w drzepcu w wierszu 1 procedury TREAP-INSERT, zarówno lewe, jak i prawe poddrzewo x są puste – ich lewy i prawy kręgosłup mają długość 0. Każde wykonanie lewej rotacji w tej procedurze na węźle $y = p[x]$ przenosi węzeł x o jeden poziom w górę drzepca oraz dodaje y do prawego kręgosłupa lewego poddrzewa x. Analogicznie prawa rotacja zwiększa o 1 lewy kręgosłup prawego poddrzewa węzła x. Wartość C jest więc równa ilości wywołań z linii 5, a D jest równe ilości wywołań z linii 4. Łączna liczba wykonanych rotacji w procedurze TREAP-INSERT wynosi zatem $C + D$.

(f) *Definicja zmiennej losowej $X_{i,k}$ powinna brzmieć następująco:*

$$X_{i,k} = \mathrm{I}(y \text{ jest na prawym kręgosłupie lewego poddrzewa węzła } x \text{ (w } T)).$$

Wprowadźmy oznaczenia T_x^L i T_x^R na, odpowiednio, lewe i prawe poddrzewo węzła x. Będziemy też pisać $x \in T$ dla oznaczenia, że x jest węzłem w drzewie T.

Koniunkcja warunków $priority[y] > priority[x]$ i $key[y] < key[x]$ jest równoważna temu, że $y \in T_x^L$. Zastanówmy się teraz, które węzły $z \in T$ spełniają nierówności $key[y] < key[z] < key[x]$. Pierwsza z nich jest prawdziwa wtedy, gdy $z \in T_y^R$ albo $y \in T_z^L$, a druga, gdy $z \in T_x^L$ albo $x \in T_z^R$. Przy założeniu, że $y \in T_x^L$, obydwie nierówności zachodzą w następujących przypadkach:

(i) $z \in T_x^L$ i $y \in T_z^L$;

(ii) $z \in T_y^R$.

Przypadek (i) obejmuje węzły z z lewego poddrzewa x, które z kolei w swoim lewym poddrzewie mają y. Zgodnie z własnościami drzepca oznacza to, że $priority[x] < priority[z] < priority[y]$. W przypadku (ii) zaś zachodzi $priority[y] < priority[z]$. Jeśli nie istnieje węzeł $z \in T$, który spełnia warunek (i), to y znajduje się na prawym kręgosłupie drzewa T_x^L. Podobnie w drugą stronę, jeśli y jest na prawym kręgosłupie T_x^L, to nie istnieje węzeł $z \in T_x^L$, dla którego $y \in T_z^L$.

(g) Załóżmy, że $i < k$. Niech p_i, p_{i+1}, …, p_k będą priorytetami węzłów o kluczach od i do k włącznie. Z punktu (f) mamy, że $X_{i,k} = 1$ wtedy i tylko wtedy, gdy $p_i > p_k$ oraz dla każdego

$j = i+1, i+2, \ldots, k-1$ zachodzi $p_j > p_i$. Istnieje $(k-i+1)!$ permutacji priorytetów p_i, $p_{i+1}, \ldots, p_k$. Spośród nich, w tych, dla których zmienna $X_{i,k}$ przyjmuje 1, p_k i p_i są, odpowiednio, najmniejszą i drugą najmniejszą wartością. Pozostałe priorytety mogą dowolnie przyjmować pozostałe wartości, dlatego permutacji sprzyjających zdarzeniu $X_{i,k} = 1$ jest $(k-i-1)!$. Stąd

$$\Pr(X_{i,k} = 1) = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}.$$

(h) W lewym poddrzewie węzła x o kluczu $k = key[x]$ znajdują się węzły o kluczach $j < k$, dlatego możemy napisać $C = \sum_{j=1}^{k-1} X_{j,k}$. Stąd

$$\begin{aligned}\mathrm{E}(C) &= \mathrm{E}\left(\sum_{j=1}^{k-1} X_{j,k}\right) = \sum_{j=1}^{k-1} \mathrm{E}(X_{j,k}) = \sum_{j=1}^{k-1} \Pr(X_{j,k} = 1) \\ &= \sum_{j=1}^{k-1} \frac{1}{(k-j+1)(k-j)} = \sum_{j=1}^{k-1} \frac{1}{j(j+1)} = \sum_{j=1}^{k-1} \left(\frac{1}{j} - \frac{1}{j+1}\right) = 1 - \frac{1}{k}.\end{aligned}$$

(i) Dla tych samych oznaczeń jak w definicji zmiennej losowej $X_{i,k}$, niech

$$Y_{i,k} = \mathrm{I}(y \text{ jest na lewym kręgosłupie prawego poddrzewa węzła } x \text{ (w } T)).$$

Można udowodnić symetryczny fakt do tego z punktu (f), czyli że $Y_{i,k} = 1$ wtedy i tylko wtedy, gdy $priority[y] > priority[x]$, $key[x] < key[y]$ oraz dla każdego z takiego, że $key[x] < key[z] < key[y]$, zachodzi $priority[y] < priority[z]$. Rozumując analogicznie jak w części (g), otrzymujemy

$$\Pr(Y_{i,k} = 1) = \frac{(i-k-1)!}{(i-k+1)!} = \frac{1}{(i-k+1)(i-k)}.$$

Ponieważ prawe poddrzewo węzła x o kluczu k zawiera węzły o kluczach $j > k$, to stąd $D = \sum_{j=k+1}^{n} Y_{j,k}$ i podobnie jak w (h):

$$\mathrm{E}(D) = \sum_{j=k+1}^{n} \Pr(Y_{j,k} = 1) = \sum_{j=k+1}^{n} \frac{1}{(j-k+1)(j-k)} = \sum_{j=1}^{n-k} \frac{1}{j(j+1)} = \frac{1}{n-k+1}.$$

(j) Na podstawie wyników uzyskanych w poprzednich punktach mamy, że oczekiwana liczba rotacji wykonywanych przez TREAP-INSERT wynosi

$$\mathrm{E}(C+D) = \mathrm{E}(C) + \mathrm{E}(D) = 1 - \frac{1}{k} + 1 - \frac{1}{n-k+1} < 2.$$

Rozdział **14**

Wzbogacanie struktur danych

14.1. Dynamiczne statystyki pozycyjne

14.1-1. Algorytm rozpoczyna działanie od $i = 10$ oraz ze zmienną x pokazującą na korzeń drzewa T. Ranga klucza 26 wyznaczona w linii 1 wynosi 13, więc algorytm zostaje wywołany rekurencyjnie dla lewego poddrzewa, czyli dla węzła o kluczu 17. W wywołaniu tym obliczona ranga klucza 17 wynosi 8. Szukany element jest więc $10-8 = 2$ (drugim) największym elementem w prawym poddrzewie aktualnego węzła. Po wywołaniu rekurencyjnym zmienna x wskazuje na jeden z węzłów o kluczu 21, ten na głębokości 2, a $i = 2$. Ranga tego klucza zostaje wyznaczona na 3, dlatego szukany klucz należy do lewego poddrzewa. W kolejnym wywołaniu mamy x wskazujące na węzeł o kluczu 19 oraz $i = 2$. Tym razem jednak lewe poddrzewo jest puste, ale zdefiniowanie $size[nil[T]]$ jako 0 pozwala obliczyć rangę obecnego elementu jako 1. Algorytm wywoływany jest więc rekurencyjnie jeszcze raz w celu znalezienia $2 - 1 = 1$ (pierwszego) elementu w prawym poddrzewie. Po wyznaczeniu rangi i porównaniu z wartością zmiennej $i = 1$ zwracany jest wskaźnik do węzła o kluczu 20.

14.1-2. Na początku działania procedury zmienna r zostaje zainicjalizowana na 1, a y początkowo wskazuje węzeł x. W tabeli 8 zamieszczono wartości $key[y]$ oraz r na początku każdej iteracji pętli **while**. Wynikiem zwracanym w wywołaniu procedury jest 16.

iteracja	$key[y]$	r
1	35	1
2	38	1
3	30	3
4	41	3
5	26	16

Tabela 8: Przebieg działania pętli **while** w procedurze OS-Rank wywołanej dla drzewa T z rys. 14.1 z Podręcznika oraz x takiego, że $key[x] = 35$.

14.1-3. Iteracyjna wersja OS-Select wykorzystuje pętlę **while**, w której symulowana jest rekurencja z wersji rekurencyjnej przez odpowiednie zaktualizowanie zmiennych x oraz i. Gdy x wskazuje na węzeł o szukanym kluczu, pętla jest przerywana poprzez zwrócenie x jako wyniku procedury.

```
ITERATIVE-OS-SELECT(x, i)
1  while TRUE
2        do r ← size[left[x]] + 1
3           if i = r
4              then return x
5           if i < r
6              then x ← left[x]
7              else  x ← right[x]
8                    i ← i − r
```

14.1-4. W naszej implementacji procedura przyjmować będzie jako parametr węzeł x drzewa T zamiast samego drzewa. Pozwoli to wywoływać procedurę rekurencyjnie w celu znalezienia rangi klucza w poddrzewach T.

```
OS-KEY-RANK(x, k)
1  r = size[left[x]] + 1
2  if k = key[x]
3     then return r
4  if k < key[x]
5     then return OS-KEY-RANK(left[x], k)
6     else  return OS-KEY-RANK(right[x], k) + r
```

Po wyznaczeniu rangi r klucza $key[x]$, jeśli jest on elementem, którego rangi szukamy, to w wierszu 3 nastąpi zwrócenie r. W przeciwnym przypadku procedura wywołuje się rekurencyjne dla lewego poddrzewa x albo prawego poddrzewa x, w zależności od tego, w którym z nich szukany klucz się znajduje na podstawie własności drzewa wyszukiwań binarnych. W przeciwnym przypadku, jeśli $k < key[x]$, to ranga klucza k w poddrzewie o korzeniu x jest równa jego randze w poddrzewie o korzeniu $left[x]$. Gdy natomiast $k > key[x]$, to ranga k w poddrzewie o korzeniu x jest o r większa od jego rangi w poddrzewie o korzeniu $right[x]$.

Aby wyznaczyć rangę klucza k w drzewie statystyk pozycyjnych T zawierającego parami różne klucze, należy wywołać OS-KEY-RANK$(root[T], k)$.

14.1-5. Załóżmy, że ranga klucza $key[x]$ wynosi r. Wystarczy zauważyć, że i-ty następnik węzła x jest węzłem z kluczem o randze $r+i$. Najpierw wyznaczamy więc r w czasie $O(\lg n)$, wywołując OS-RANK(T, x), a następnie, również w czasie $O(\lg n)$, szukamy w T elementu o randze $r + i$, czyli wykonujemy OS-SELECT$(root[T], r + i)$.

14.1-6. Pamiętajmy, że rotacje nie zmieniają uszeregowania inorder kluczy w drzewie, dlatego wykonanie rotacji nie zmienia ich rang.

Podczas wstawiania nowego węzła z do drzewa statystyk pozycyjnych szukamy dla niego odpowiedniego miejsca na ścieżce w dół od korzenia drzewa. Dla każdego węzła x na tej ścieżce, jeśli $key[z] < key[x]$, to węzeł z jest umieszczany w lewym poddrzewie x i ranga każdego węzła y z prawego poddrzewa węzła x rośnie o 1, bo z będzie poprzedzał każde takie y w porządku inorder po zakończeniu wstawiania. Jeśli z kolei z ląduje w prawym poddrzewie x, to rangi węzłów lewego poddrzewa x nie zmieniają się.

W przypadku usuwania jeśli x jest dowolnym węzłem na ścieżce od efektywnie usuniętego węzła w górę drzewa aż do korzenia, to zmianie, a dokładniej zmniejszeniu o 1, ulegają rangi każdego węzła y należącego do prawego poddrzewa węzła x.

Widać więc, że utrzymywanie w każdym węźle drzewa jego rangi powoduje, że operacje wstawiania i usuwania wymagają czasu $O(n)$, gdzie n jest liczbą węzłów w drzewie.

14.1-7. Niech $A = [1\,..\,n]$ będzie tablicą zawierającą liczby parami różne. Oznaczmy przez r_i rangę elementu $A[i]$ w podtablicy $A[1\,..\,i]$. Dla każdego $i = 1, 2, \ldots, n$ element $A[i]$ jest zatem r_i-tym najmniejszym elementem w podtablicy $A[1\,..\,i]$, dlatego z innymi elementami w tej podtablicy tworzy $i-r_i$ inwersji. Sumaryczna liczba inwersji w tablicy A jest więc równa $\sum_{i=1}^{n}(i-r_i)$.

Rangi elementów tablicy A możemy wyznaczyć, wstawiając do drzewa statystyk pozycyjnych kolejno, $A[1]$, $A[2]$, $\ldots$, $A[n]$. Tuż po wstawieniu elementu $A[i]$ jego ranga w aktualnym drzewie wynosi r_i, którą to wartość otrzymujemy za pomocą algorytmu OS-Rank. Wstawienie dowolnego elementu tablicy A do drzewa oraz obliczenie jego rangi wymaga czasu $O(\lg n)$, dlatego inwersje w A możemy policzyć w czasie $O(n \lg n)$.

14.1-8. Ponumerujmy końce cięciw liczbami od 1 do n w taki sposób, aby dwa różne punkty miały ten sam numer wtedy i tylko wtedy, gdy są końcami tej samej cięciwy. Niech σ będzie ciągiem wszystkich tych etykiet czytanych wzdłuż okręgu w ustalonym kierunku. Zauważmy, że σ składa się z $2n$ wyrazów, a każdy z nich pojawia się dokładnie 2 razy. Cięciwa oznaczona jako i przecina się z cięciwą oznaczoną jako j wtedy i tylko wtedy, gdy dokładnie jeden egzemplarz i znajduje się w σ pomiędzy dwoma egzemplarzami j.

Algorytm będzie przeglądał ciąg σ od lewej do prawej. Będzie on zapamiętywał, które z etykiet pojawiły się dotychczas dokładnie raz w trakcie tego przeglądania. W momencie gdy pewna liczba i pojawia się drugi raz, policzy, ile liczb różnych od i pojawiło się dokładnie raz po pierwszym wystąpieniu i. Wartość ta oznacza liczbę cięciw przecinających się z cięciwą o numerze i.

W algorytmie użyjemy dwóch tablic do zapamiętania cięciw. W tablicy $E[1\,..\,n]$ w komórce $E[i]$ będziemy zapisywać pozycję pierwszego wystąpienia i w ciągu σ, a w wektorze bitowym $seen[1\,..\,n]$ wartość TRUE na i-tej pozycji będzie oznaczać, że i zostało już napotkane w trakcie przeglądania ciągu σ. Potrzebny będzie nam też zbiór dynamiczny S przechowujący wartości z tablicy E dla liczb, które pojawiły się dokładnie raz. Zbiór S będziemy reprezentować, używając w tym celu drzewa statystyk pozycyjnych, które pozwoli na generowanie odpowiedzi na pytania „ile elementów zbioru S jest większych od x" w czasie $O(\lg n)$.

```
INTERSECTING-CHORDS(σ)
 1  for k = 1 to 2n
 2       do seen[k] = FALSE
 3  intersections ← 0
 4  S ← ∅
 5  for k = 1 to 2n
 6       do j ← σ_k
 7          if seen[j] = FALSE
 8             then seen[j] ← TRUE
 9                  E[j] = k
10                  S ← S ∪ {k}
11             else intersections ← intersections + |{s ∈ S : s > E[j]}|
12                  S ← S \ {E[j]}
13  return intersections
```

Zastanówmy się nad poprawnością przedstawionego algorytmu. Dla każdej etykiety i niech $\sigma_{s_i} = \sigma_{t_i} = i$, gdzie $s_i < t_i$ są pozycjami końców cięciwy i w ciągu σ. Rozważmy cięciwy i oraz

j, dla których $s_i < s_j$. Jeśli cięciwa i i j przecinają się, to $s_j < t_i < t_j$ i przecięcie to zostanie policzone dokładnie raz – wtedy, gdy przetwarzany będzie t_i-ty wyraz ciągu σ. W przeciwnym przypadku $t_j < t_i$, więc tego (nieistniejącego) przecięcia nie policzymy.

Aby otrzymać czas działania algorytmu, zauważmy, że każda etykieta jest wstawiana do S dokładnie raz i co najwyżej jedno zapytanie z wiersza 11 zostaje wykonane w celu policzenia elementów ze zbioru S większych od tej etykiety. Ponieważ obydwie te operacje działają w czasie $O(\lg n)$, to stąd czasem działania algorytmu jest $O(n \lg n)$.

14.2. Jak wzbogacać strukturę danych

14.2-1. Aby efektywnie wykonywać podane operacje, wzbogacimy każdy węzeł x drzewa statystyk pozycyjnych T o 4 nowe pola:

- $min[x]$ – wskaźnik na węzeł o najmniejszym kluczu w poddrzewie o korzeniu w x,
- $max[x]$ – wskaźnik na węzeł o największym kluczu w poddrzewie o korzeniu w x,
- $pred[x]$ – wskaźnik na poprzednika węzła x (lub $nil[T]$ jeśli poprzednik nie istnieje),
- $succ[x]$ – wskaźnik na następnika węzła x (lub $nil[T]$ jeśli następnik nie istnieje).

Przyjmujemy, że wartością każdego z tych pól dla $nil[T]$ jest $nil[T]$.

Wartości pól min oraz max każdego wewnętrznego węzła x można wyznaczyć na podstawie jedynie wartości tych pól w synach węzła x. Niech

$$\mu_x = \min(key[min[left[x]]], key[x], key[min[right[x]]]).$$

Wówczas:

$$min[x] = \begin{cases} min[left[x]], & \text{jeśli } \mu_x = key[min[left[x]]], \\ x, & \text{jeśli } \mu_x = key[x], \\ min[right[x]], & \text{jeśli } \mu_x = key[min[right[x]]]. \end{cases}$$

Analogicznie można wyznaczyć wartość pola $max[x]$. Zatem zgodnie z tw. 14.1 można zachować poprawne wartości tych pól podczas wstawiania i usuwania, nie zwiększając asymptotycznej złożoności tych operacji.

Jednakże w przypadku pól *pred* i *succ* nie można zastosować tw. 14.1, bo wartości tych pól mogą zależeć nie tylko od pól synów danego węzła – zarówno następnik, jak i poprzednik węzła może znajdować się na wyższym poziomie drzewa. Pokażemy jednak, że nadal możliwe jest efektywne zaimplementowanie operacji wstawiania i usuwania, które aktualizują wartości tych pól.

Zauważmy najpierw, że rotacje nie pociągają za sobą konieczności modyfikacji pól *pred* i *succ*. Jest tak dlatego, że porządek inorder węzłów w drzewie nie zostaje zmieniony przez rotacje, więc nie ma zmian w poprzednikach i następnikach. Po wstawieniu nowego węzła do drzewa wystarczy więc nadać jego atrybutowi *pred* wartość zwróconą przez wywołanie oryginalnej wersji operacji Predecessor, a atrybutowi *succ* – oryginalnej wersji operacji Successor. Nowe pola można zaktualizować także w operacji Delete. Wystarczy tuż przed usunięciem węzła odnaleźć jego poprzednik p i następnik s poprzez wywołanie zwykłych wersji operacji Predecessor i Successor. Następnie tuż po usunięciu węzła wystarczy ustawić $succ[p]$ na s (o ile $p \neq nil[T]$), a $pred[s]$ na p (o ile $s \neq nil[T]$). Widać więc, że dodatkowe wywołania nie powiększają czasu działania operacji Insert i Delete.

Dzięki wykorzystaniu nowo dodanych atrybutów wywołania MINIMUM(T), MAXIMUM(T), PREDECESSOR(T, x) i SUCCESSOR(T, x) można zaimplementować w czasie $O(1)$ tak, aby zwracały, odpowiednio, $min[root[T]]$, $max[root[T]]$, $pred[x]$ i $succ[x]$.

14.2-2. Jeśli czarną wysokość wewnętrznego węzła x w drzewie czerwono-czarnym T przechowamy w polu $bh[x]$, a ponadto zdefiniujemy $bh[nil[T]] = 0$, to zachodzi następująca zależność:

$$bh[x] = \begin{cases} bh[left[x]], & \text{jeśli } color[left[x]] = \text{RED}, \\ bh[left[x]] + 1, & \text{jeśli } color[left[x]] = \text{BLACK}. \end{cases}$$

A zatem na mocy tw. 14.1 mamy, że złożoność asymptotyczna operacji na tak wzbogaconym drzewie czerwono-czarnym nie ulegnie zmianie.

14.2-3. Nie można tego zrobić przy zachowaniu efektywnych czasów działania operacji na drzewie, ponieważ głębokość węzła zależy od głębokości jego ojca. Gdy zmienia się głębokość węzła x, to zmieniają się także głębokości wszystkich potomków x. A zatem aktualizacja głębokości korzenia drzewa niesie za sobą konieczność aktualizacji pozostałych $n-1$ węzłów drzewa i operacje wstawiania i usuwania działają wtedy w czasie $O(n \lg n)$.

14.2-4. Posługując się rys. 13.2 z Podręcznika, oznaczmy przez r_α, r_β, r_γ korzenie poddrzew, odpowiednio, α, β, γ. Ponieważ operacja $\otimes$ jest łączna, to mamy

$$\begin{aligned} f[x] &= f[r_\alpha] \otimes a[x] \otimes f[r_\beta] \otimes a[y] \otimes f[r_\gamma], \\ f[y] &= \qquad\qquad f[r_\beta] \otimes a[y] \otimes f[r_\gamma]. \end{aligned}$$

Rotacje nie zmieniają porządku inorder węzłów w żadnym z poddrzew α, β i γ, dlatego po przeprowadzeniu lewej rotacji wartości pola f wynoszą

$$\begin{aligned} f[x] &= f[r_\alpha] \otimes a[x] \otimes f[r_\beta], \\ f[y] &= f[r_\alpha] \otimes a[x] \otimes f[r_\beta] \otimes a[y] \otimes f[r_\gamma] \end{aligned}$$

i mogą zostać obliczone w czasie $O(1)$. Rozumowanie w przypadku prawej rotacji przeprowadza się analogicznie.

W drzewie czerwono-czarnym T zdefiniujmy teraz dla każdego węzła pole a przyjmujące wartość 0 dla liści drzewa (reprezentowanych przez $nil[T]$) oraz 1 dla jego węzłów wewnętrznych. Jeśli operacją $\otimes$ będzie zwykłe dodawanie, to wartość $f[x]$ będzie rozmiarem poddrzewa o korzeniu w x, czyli pole f będzie identyczne z polem *size* z drzew statystyk pozycyjnych. Dzięki powyżej przedstawionej argumentacji pole *size* może być aktualizowane w czasie $O(1)$ po każdym wykonaniu rotacji w drzewie T.

14.2-5. Do zaimplementowania tej operacji potrzebna nam będzie pomocnicza procedura wyszukująca w drzewie czerwono-czarnym węzeł o najmniejszym kluczu większym lub równym podanej wartości.

```
RB-Search-Upper-Bound(T, x, k)
 1  if z ← nil[T]
 2  while TRUE
 3        do if x = nil[T]
 4              then return y
 5           if key[x] = k
 6              then return x
 7           if k < key[x]
 8              then z ← x
 9                   x ← left[x]
10              else  x ← right[x]
```

Jest to modyfikacja procedury Iterative-RB-Search, która z kolei jest wersją procedury Iterative-Tree-Search przystosowaną do drzew czerwono-czarnych. W drzewie T wyszukiwany jest węzeł o kluczu k i jeśli istnieje, to zostaje zwrócony w linii 6. W przeciwnym przypadku w wierszu 4 następuje zwrócenie następnika węzła o kluczu k, gdyby znajdował się on w drzewie T. Zmienna z w kolejnych iteracjach pętli **while** wskazuje na węzeł będący przodkiem aktualnie testowanego węzła x znajdującego się w lewym poddrzewie z, albo na $nil[T]$, jeżeli węzeł taki nie istnieje w T. Poza utrzymywaniem odpowiedniej wartości zmiennej z, procedura wykonuje te same działania, co Iterative-RB-Search, a zatem działa w czasie $O(\lg n)$ dla drzewa T o n węzłach.

Rozważymy także modyfikację operacji następnika, RB-Subtree-Successor, która przyjmuje dodatkowy, trzeci parametr z i znajduje następnik węzła x w poddrzewie o korzeniu z. Jeśli taki węzeł nie istnieje w tym poddrzewie, to zwracane jest $nil[T]$. Modyfikacja polega na dodaniu następującego fragmentu między wiersz 2 i 3 w RB-Successor, co nie zwiększa czasu działania tej procedury:

```
if x = RB-Maximum(T, z)
   then return nil[T]
```

Następujący algorytm stanowi implementację szukanej operacji. Przyjmuje on dodatkowo jako parametr drzewo T potrzebne do odwołania się do $nil[T]$ i wypisuje szukane klucze w porządku niemalejącym.

```
RB-Enumerate(T, x, a, b)
1  z ← RB-Search-Upper-Bound(T, x, a)
2  while z ≠ nil[T] i a ⩽ key[z] ⩽ b
3        do wypisz key[z]
4           z ← RB-Subtree-Successor(T, x, z)
```

Zmienna z jest inicjalizowana na węzeł o kluczu $k \geqslant a$ z poddrzewa o korzeniu w x. Jeśli węzeł taki nie istnieje (bo poddrzewo x zawiera klucze mniejsze od a), to $z = nil[T]$ i procedura kończy działanie. W przeciwnym przypadku następuje seria wywołań operacji następnika y w wierszu 4, aż do wyczerpania węzłów w poddrzewie x albo odnalezienia węzła o kluczu przekraczającym b.

Wywołanie z linii 1 zajmuje czas $O(\lg n)$. Na mocy zad. 12.2-8 niezależnie od tego, od którego węzła rozpoczniemy, seria m wywołań operacji następnika w drzewie o wysokości $h = O(\lg n)$ potrzebuje czasu $O(m + \lg n)$, co stanowi czas działania przedstawionego algorytmu.

14.3. Drzewa przedziałowe

14.3-1. Procedura INTERVAL-LEFT-ROTATE dla drzew przedziałowych działa identycznie jak procedura LEFT-ROTATE dla drzew czerwono-czarnych, z tą różnicą, że tuż przed zakończeniem działania modyfikuje $max[x]$ i $max[y]$, gdzie y jest nowym ojcem x po przeprowadzeniu rotacji. Modyfikacje te polegają najpierw na aktualizacji $max[x]$ wprost z definicji pola max, a następnie analogicznym zaktualizowaniu $max[y]$, które po wykonaniu rotacji zależy od $max[x]$.

INTERVAL-LEFT-ROTATE(T, x)
1 LEFT-ROTATE(T, x)
2 $y \leftarrow p[x]$
3 $max[x] \leftarrow \max(high[int[x]], max[left[x]], max[right[x]])$
4 $max[y] \leftarrow \max(high[int[y]], max[x], max[right[y]])$

14.3-2. Pseudokod procedury INTERVAL-SEARCH nie zmieni się. Jedyna różnica będzie ukryta w linii 2 tej procedury w definicji zachodzenia przedziałów. Powiemy, że dwa przedziały otwarte i oraz i' zachodzą na siebie, jeśli $i \cap i' \neq \emptyset$, tj. jeśli $low[i] < high[i']$ i $low[i'] < high[i]$.

14.3-3. Algorytm wyszukuje w drzewie przedziałowym T dowolny węzeł x, taki że przedział $int[x]$ zachodzi na przedział i, wywołując w tym celu procedurę INTERVAL-SEARCH. Jeśli przedział taki istnieje, czyli $x \neq nil[T]$, to począwszy od x, algorytm przechodzi w dół drzewa przedziałowego w poszukiwaniu przedziału zachodzącego na i o najmniejszym lewym końcu.

MIN-INTERVAL-SEARCH(T, i)
```
 1  x ← INTERVAL-SEARCH(T, i)
 2  if x ≠ nil[T]
 3     then y ← left[x]
 4          while y ≠ nil[T]
 5                do if i zachodzi na int[y]
 6                      then x ← y
 7                           y ← left[x]
 8                      else  if left[y] ≠ nil[T] i max[left[y]] ⩾ low[i]
 9                               then y ← left[y]
10                               else  y ← right[y]
11  return x
```

Udowodnimy następujący niezmiennik pętli **while**:

> Przed każdą iteracją pętli **while** w wierszach 4–10, jeśli w drzewie T znajduje się przedział zachodzący na i o lewym końcu mniejszym niż $low[int[x]]$, to należy on do poddrzewa o korzeniu w y.

Inicjowanie: Przed pierwszą iteracją pętli **while** $y = left[x]$. Przedziały o lewych końcach mniejszych niż $low[int[x]]$ znajdują się w poddrzewie o korzeniu w $left[x]$, dlatego niezmiennik jest spełniony.

Utrzymanie: Załóżmy, że niezmiennik jest prawdziwy przed każdą następną iteracją pętli **while** i zobaczmy, co zmienia wykonanie iteracji. Zauważmy wpierw, że każda aktualizacja zmiennej x w trakcie działania procedury pociąga za sobą przestawienie y na $left[x]$. Jeśli przedział i zachodzi na $int[y]$, to można zaktualizować wskaźnik x – przechowujący przedział zachodzący na i o najmniejszym dotychczas napotkanym lewym końcu – na y, ponieważ y jest węzłem z lewego poddrzewa x, więc $low[int[y]] < low[int[x]]$. Niezmiennik

pozostaje spełniony, bo jeśli w drzewie T istnieje przedział zachodzący na i o mniejszym lewym końcu od nowej wartości x, to należy on do lewego poddrzewa x, a w wierszu 7 y jest zaktualizowane na $left[x]$.

Załóżmy teraz, że przedział i nie zachodzi na $int[y]$. Zauważmy, że wiersze 8–10 stanowią ciało pętli **while** z procedury INTERVAL-SEARCH z y w miejscu x. Wykorzystując niezmiennik pętli **while** tamtej procedury do obecnej iteracji pętli w procedurze MIN-INTERVAL-SEARCH otrzymujemy, że jeśli przed aktualizacją y poddrzewo o korzeniu y zawierało przedział zachodzący na i, to zawiera go też poddrzewo o korzeniu w węźle wskazywanym przez nową wartość wskaźnika y. Niezmiennik jest więc zachowany, gdyż y znajduje się zawsze w lewym poddrzewie x.

Zakończenie: Pętla kończy działanie, gdy $y = nil[T]$. Poddrzewo o korzeniu w y jest puste, skąd wynika, że nie istnieje w T przedział zachodzący na i o lewym końcu mniejszym niż $low[int[x]]$, czyli x jest szukanym przedziałem.

Algorytm działa w czasie $O(\lg n)$ dla drzewa o n węzłach, ponieważ oprócz wywołania operacji INTERVAL-SEARCH schodzi jeszcze od znalezionego węzła do liścia drzewa T po ścieżce prostej.

14.3-4. Poniższy pseudokod implementuje szukany algorytm, wypisując szukane przedziały niemalejąco według ich lewych końców. Wejściowe drzewo przedziałowe nie jest modyfikowane.

```
INTERVAL-SEARCH-ALL(T, x, i)
1  if left[x] ≠ nil[T] i max[left[x]] ⩾ low[i]
2      then INTERVAL-SEARCH-ALL(T, left[x], i)
3  if x ≠ nil[T] oraz i zachodzi na int[x]
4      then wypisz int[x]
5  if right[x] ≠ nil[T] i max[right[x]] ⩾ low[i] i low[int[x]] ⩽ high[i]
6      then INTERVAL-SEARCH-ALL(T, right[x], i)
```

Aby wyznaczyć wszystkie przedziały zachodzące na i w drzewie T, należy skorzystać z wywołania INTERVAL-SEARCH-ALL$(T, root[T], i)$.

Zastanówmy się nad testami przeprowadzanymi, zanim wykonane zostaną wywołania rekurencyjne. W linii 1, gdyby było $max[left[x]] < low[i]$, to każdy przedział z lewego poddrzewa x leżałby na lewo od i, więc wywołanie byłoby zbędne. Podobnie w wierszu 5 – gdyby $max[right[x]] < low[i]$, to wszystkie przedziały z prawego poddrzewa x leżałyby na lewo od i. Gdyby ponadto zachodziło $low[int[x]] > high[i]$, to zarówno $int[x]$, jak i każdy przedział z prawego poddrzewa x znajdowałby się na prawo od i, dlatego także wtedy blokowane jest wywołanie rekurencyjne.

Załóżmy teraz, że poddrzewo o korzeniu w x nie zawiera przedziałów zachodzących na i. Jeśli i leży na lewo od $int[x]$, to procedura może zostać wywołana rekurencyjnie tylko dla lewego poddrzewa x. Jeśli z kolei i leży na prawo od $int[x]$, to procedura nie zostanie wywołana rekurencyjne dla lewego poddrzewa. Gdyby tak było, to musiałoby zachodzić $max[left[x]] \geqslant low[i]$, czyli w lewym poddrzewie x znajdowałby się przedział i', dla którego $high[i'] = max[left[x]] \geqslant low[i]$, a zatem przedział i' zachodziłby na i, co przeczyłoby założeniu. Wynika stąd, że jeśli w poddrzewie o korzeniu w x nie ma przedziału zachodzącego na i, to procedura na każdym poziomie wykonuje co najwyżej jedno zejście rekurencyjne, schodząc najdalej do potomnego liścia x.

Każdy węzeł drzewa T testowany jest w algorytmie co najwyżej dwa razy – raz w linii 1 albo 5 i raz w linii 3. Każdy taki test zajmuje czas stały, dlatego czas działania algorytmu nie przekracza $O(n)$. Z drugiej strony procedura dociera do każdego węzła z przedziałem zachodzącym na i, po

czym może zejść rekurencyjnie od niego aż do jednego z jego potomnych liści w drzewie (na podstawie poprzedniego paragrafu). Dlatego czas działania nie może też przekroczyć $O(k \lg n)$, gdyż wysokość drzewa T wynosi $O(\lg n)$, a liczba przedziałów zachodzących na i w T wynosi k. Łącząc te ograniczenia, dostajemy, że algorytm działa w czasie $O(\min(n, k \lg n))$.

14.3-5. Zmodyfikujemy operację wstawiania do drzewa przedziałowego w taki sposób, aby w poddrzewach składających się z przedziałów o takich samych lewych końcach, przedziały uporządkowane były względem prawych końców. Załóżmy, że do drzewa przedziałowego T o takiej własności wstawiany jest element x. Niech y będzie pierwszym napotkanym węzłem w T podczas tego wstawiania, gdzie $low[int[y]] = low[int[x]]$, albo $nil[T]$, jeżeli węzeł taki nie istnieje. Wszystkie przedziały w drzewie T o takich samych lewych końcach jak lewy koniec $int[x]$ znajdują się w poddrzewie o korzeniu w y. Korzystając z uporządkowania przedziałów w takim poddrzewie i traktując jako klucze prawe końce przedziałów, wyszukujemy dla x odpowiednie miejsce w tym poddrzewie. Dzięki temu po wstawieniu elementu x uporządkowanie w drzewie T pozostaje zachowane. Łatwo zauważyć, że następujące potem rotacje nie naruszają wprowadzonego uporządkowania, bo zachowują porządek inorder w drzewie, jak również to, że wykonana modyfikacja nie zwiększa asymptotycznego czasu działania operacji wstawiania.

Operacja Interval-Search-Exactly wyznacza szukany węzeł w identyczny sposób jak opisane powyżej wyszukiwanie miejsca w drzewie podczas wstawiania do niego nowego węzła. Stąd też czas działania nowej operacji można ograniczyć od góry przez $O(\lg n)$, gdzie n jest liczbą węzłów w drzewie wejściowym.

```
Interval-Search-Exactly(T, i)
 1  x ← root[T]
 2  while x ≠ nil[T] i low[int[x]] ≠ low[i]
 3        do if low[int[x]] < low[i]
 4              then x ← left[x]
 5              else  x ← right[x]
 6  while x ≠ nil[T] i high[int[x]] ≠ high[i]
 7        do if high[int[x]] < high[i]
 8              then x ← left[x]
 9              else  x ← right[x]
10  return x
```

14.3-6. Strukturę danych Q zaimplementujemy w postaci odpowiednio wzbogaconego drzewa czerwono-czarnego. W każdym węźle x drzewa Q przechowamy dodatkowe pola:

- $min\text{-}key[x]$ – równe najmniejszemu kluczowi w poddrzewie o korzeniu w x;
- $max\text{-}key[x]$ – równe największemu kluczowi w poddrzewie o korzeniu w x;
- $min\text{-}gap[x]$ – równe najmniejszej odległości między kluczami w poddrzewie o korzeniu w x.

Wszystkie nowe pola każdego wewnętrznego węzła x są zależne od innych pól węzła x oraz od pól węzłów $left[x]$ i $right[x]$:

$$\begin{aligned} min\text{-}key[x] &= \min(min\text{-}key[left[x]], key[x], min\text{-}key[right[x]]), \\ max\text{-}key[x] &= \max(max\text{-}key[left[x]], key[x], max\text{-}key[right[x]]), \\ min\text{-}gap[x] &= \min(min\text{-}gap[left[x]], min\text{-}gap[right[x]], \\ &\qquad key[x] - max\text{-}key[left[x]], min\text{-}key[right[x]] - key[x]). \end{aligned}$$

Definiujemy ponadto $min\text{-}key[nil[Q]] = min\text{-}gap[nil[Q]] = \infty$ oraz $max\text{-}key[nil[Q]] = -\infty$.

Wywołanie MIN-GAP(Q) polega na zwróceniu wartości $min\text{-}gap[root[Q]]$ i działa w czasie $O(1)$. Oczywiście wzbogacenie drzewa o nowe pola nie zmienia działania operacji SEARCH. Dzięki zastosowaniu tw. 14.1 mamy z kolei, że wprowadzenie nowych pól nie zwiększa asymptotycznej złożoności czasowej operacji INSERT i DELETE.

14.3-7. Ogólna idea algorytmu polega na pomyśle ze wskazówki podanej w treści zadania: wykorzystywana jest tzw. „miotła", czyli pionowa prosta zamiatająca zbiór prostokątów od lewej do prawej. W danym momencie pamiętane będą prostokąty, których bok poziomy wyznacza przedział zawierający aktualną pozycję „miotły". Tak naprawdę zamiast przechowywania wszystkich informacji o prostokątach wystarczy pamiętać przedziały wyznaczone przez boki pionowe tych prostokątów. Do tego celu zastosujemy drzewo przedziałowe.

Omówmy teraz szczegóły techniczne algorytmu. Najpierw utworzone zostanie puste drzewo przedziałowe T oraz lista współrzędnych x wszystkich prostokątów wraz z informacją o tym, czy jest to lewa współrzędna swojego prostokąta, czy prawa. Lista ta zostanie następnie posortowana i będzie przeglądana od lewej do prawej. Jeśli kolejna napotkana współrzędna na tej liście stanowi lewą współrzędną x pewnego prostokąta R, to na drzewie T wywołana zostanie operacja INTERVAL-SEARCH dla przedziału wyznaczonego przez współrzędne y prostokąta R (czyli jego bok pionowy). Wynik działania tego wywołania stanowi informację o tym, czy w zbiorze prostokątów istnieje prostokąt różny od R i zachodzący na R. Następnie pionowy bok prostokąta R zostanie wstawiony do drzewa T. Z kolei napotkawszy prawą współrzędną x prostokąta R, algorytm usunie pionowy bok R z drzewa T. W dowolnym momencie działania algorytmu drzewo T reprezentuje zatem zbiór prostokątów, które są aktualnie przecinane przez prostą zamiatającą.

Czas działania algorytmu dla n prostokątów wynosi $O(n \lg n)$. Tyle czasu zajmuje sortowanie listy współrzędnych x oraz $O(n)$ wywołań operacji na drzewie przedziałowym (wstawianie, usuwanie i wyszukiwanie przedziałów zachodzących), z których każda wymaga czasu $O(\lg n)$.

Problemy

14-1. Punkt o największej liczbie przecięć

(a) Niech I będzie rozważanym zbiorem przedziałów, a p – jednym z punktów o największej liczbie s przecięć z przedziałami z I. Oznaczmy przez $i_1, i_2, \ldots, i_s$ parami różne przedziały z I zawierające p. Oczywiście przecięcie $i = \bigcap_{k=1}^{s} i_k$ również jest przedziałem, który zawiera punkt p. Przedział i składa się z punktów, na które zachodzi dokładnie s przedziałów z I, w szczególności końce i stanowią punkty o największej liczbie przecięć z przedziałami z I.

(b) Skorzystamy ze wskazówki i do zapamiętania wszystkich n przedziałów wykorzystamy wzbogacone drzewo czerwono-czarne T przechowujące nie przedziały, ale ich końce. Do każdego wewnętrznego węzła x w tym drzewie dodamy atrybut *int* będący wskaźnikiem na przedział, którego końcem jest klucz węzła x oraz atrybut *side* przyjmujący wartość $+1$, jeśli klucz węzła x stanowi lewy koniec przedziału $int[x]$ oraz -1 – jeśli klucz x stanowi prawy koniec przedziału $int[x]$. Do początkowo pustego drzewa T wstawione zostaną najpierw wszystkie lewe końce przedziałów, a następnie wszystkie prawe końce. Dzięki temu, jeśli pewien lewy koniec ma tę samą wartość co pewien prawy koniec, to lewy będzie poprzedzał prawy w uporządkowaniu inorder w drzewie T.

Niech $\langle e_1, e_2, \ldots, e_{2n} \rangle$ będzie posortowanym ciągiem końców przedziałów ze zbioru wejścio-

wego. Zdefiniujmy

$$s(j,k) = \sum_{i=j}^{k} side[e_i]$$

dla $1 \leqslant j \leqslant k \leqslant 2n$. Na podstawie poprzedniej części, punkt o największej liczbie przecięć znajduje się w lewym końcu e_k przedziału, dla którego wartość $s(1,k)$ jest maksymalna.

Z każdym węzłem x związżemy kolejne atrybuty. Niech e_{j_x} i e_{k_x} będą końcami przedziałów znajdującymi się w poddrzewie o korzeniu w x, odpowiednio, najbardziej na lewo i najbardziej na prawo w porządku inorder. Atrybut $sum[x]$ definiujemy jako $s(j_x, k_x)$, czyli sumę wartości *side* po wszystkich przedziałach w poddrzewie w x. W polu $max[x]$ przechowamy maksymalną wartość wyrażenia $s(j_x, i)$ dla $i = j_x, j_x + 1, \ldots, k_x$. W końcu pole $pom[x]$ przechowywać będzie przedział o końcu e_i, dla którego $max[x]$ osiąga maksimum. Przyjmijmy ponadto $sum[nil[T]] = max[nil[T]] = 0$.

Wartości nowych pól można wyznaczyć na podstawie ich wartości w synach, co spełnia założenie tw. 14.1:

$$sum[x] = sum[left[x]] + side[x] + sum[right[x]],$$
$$max[x] = \max(max[left[x]], sum[left[x]] + side[x], sum[left[x]] + side[x] + max[right[x]]).$$

O ile sposób obliczania $sum[x]$ jest oczywisty, to opiszemy na czym polega wzór na $max[x]$. Koniec e_i, gdzie i maksymalizuje sumę $s(j_x, i)$, znajduje się albo w węźle x albo w którymś z jego poddrzew. Jeśli e_i jest w lewym poddrzewie x, to maksymalna wartość $s(j_x, i)$ jest równa maksymalnej wartości $s(j_{left[x]}, i)$, skąd $max[x] = max[left[x]]$. Gdy e_i jest w węźle x, to $max[x]$ stanowi sumę wartości *side* po każdym węźle w lewym poddrzewie x łącznie z $side[x]$, więc $max[x] = sum[left[x]] + side[x]$. W końcu, gdy e_i jest w jednym z węzłów w prawym poddrzewie x, to $max[x]$ stanowi sumę wartości *side* po każdym węźle z lewego poddrzewa x, łącznie z $side[x]$ i wartościami *side* w pewnym zbiorze węzłów z prawego poddrzewa x. Zbiór ten przebiega węzły, począwszy od najbardziej na lewo położonego w tym poddrzewie, aż do węzła zawierającego e_i, który maksymalizuje $s(j_{right[x]}, i)$. Maksymalna wartość tego wyrażenia jest równa dokładnie $max[right[x]]$, skąd w tym przypadku dostajemy $max[x] = sum[left[x]] + side[x] + max[right[x]]$. Na podstawie wartości, która przypisywana jest do $max[x]$, ustalana jest odpowiednia wartość dla $pom[x]$, czyli $pom[left[x]]$, $int[x]$ albo $pom[right[x]]$, kolejno dla przypadków opisywanych powyżej.

Stosując teraz tw. 14.1 mamy, że operacje wstawiająca i usuwająca koniec przedziału do opisanej struktury danych, działają w czasie $O(\lg n)$. Operacja FIND-POM(T) działa w czasie $O(1)$, zwracając po prostu przedział $pom[root[T]]$.

14-2. Permutacja Józefa

(a) Algorytm można zaimplementować jako symulację opisanego procesu eliminowania osób. W tym celu można wykorzystać dwukierunkową listę cykliczną zawierającą początkowo liczby całkowite, kolejno 1, 2, ..., n. Symulacja polega na usuwaniu co m-tego elementu z tej listy poprzez przejście m razy wskaźnikami *next* po tej liście, a następnie usunięcie aktualnego elementu. Ostatni element, który pozostanie na liście, zostaje wypisany.

Operacja usuwania z dwukierunkowej listy cyklicznej działa jak LIST-DELETE, ale polega na założeniu, że $prev[x] \neq$ NIL oraz $next[x] \neq$ NIL. Operacja składa się zatem jedynie z wierszy 2 i 5 procedury LIST-DELETE, dlatego jej czasem działania jest oczywiście $O(1)$. Stąd całkowity czas

działania symulacji wynosi

$$\sum_{k=2}^{n}(m + O(1)) = m(n-1) + O(n) = O(n),$$

ponieważ m jest stałą.

(b) Załóżmy, że w pewnym momencie odliczanie zatrzymało się na osobie o j-tym największym numerze spośród pozostałych $k \leqslant n$ osób. Osoba ta zostaje usunięta, co powoduje dekrementację zmiennej k, a numer kolejnej osoby do usunięcia można wyznaczyć następująco. Do j dodajemy m, bo przesuwamy się o m numerów i odejmujemy 1, bo osoba o j-tym największym numerze została właśnie usunięta. Ponieważ liczba $j + m - 1$ może przekraczać liczbę pozostałych k osób w okręgu, to należy zastosować arytmetykę modularną. W rezultacie otrzymujemy, że następna osoba ma numer na pozycji $(j + m - 2) \bmod k + 1$ na uszeregowanej rosnąco liście numerów pozostałych osób.

Na podstawie powyższego opisu w algorytmie wykorzystamy drzewo statystyk pozycyjnych. Będziemy używać operacji OS-Insert oraz OS-Delete, które stanowią implementacje operacji słownikowych Insert i Delete na drzewie statystyk pozycyjnych.

```
JOSEPHUS(n, m)
 1  utwórz puste drzewo statystyk pozycyjnych T
 2  for j ← 1 to n
 3       do utwórz węzeł x, w którym key[x] = j
 4          OS-INSERT(T, x)
 5  j ← 1
 6  for k ← n downto 1
 7       do j ← (j + m − 2) mod k + 1
 8          x ← OS-SELECT(root[T], j)
 9          wypisz key[x]
10          OS-DELETE(T, x)
```

Zbudowanie drzewa statystyk pozycyjnych T zajmuje czas $O(n \lg n)$. Następnie wykonywanych jest n wywołań procedur działających na tym drzewie, z których każde potrzebuje czasu $O(\lg n)$. A zatem czas działania algorytmu wynosi $O(n \lg n)$.

Część **VIII**

Dodatek: Podstawy matematyczne

Dodatek **A**

Sumy

A.1. Wzory i własności dotyczące sum

A.1-1.

$$\sum_{k=1}^{n}(2k-1) = 2\sum_{k=1}^{n} k - \sum_{k=1}^{n} 1 = \frac{2n(n+1)}{2} - n = n^2$$

A.1-2. Korzystając z oszacowania na H_n (wzór (A.7)), mamy

$$\begin{aligned}
\sum_{k=1}^{n} \frac{1}{2k-1} &= H_{2n-1} - \sum_{k=1}^{n-1} \frac{1}{2k} \\
&= H_{2n-1} - \frac{H_{n-1}}{2} \\
&= \ln(2n-1) + O(1) - \frac{\ln(n-1) + O(1)}{2} \\
&= \ln(2n-2) + \ln\frac{2n-1}{2n-2} - \ln\sqrt{n-1} + O(1) \\
&= \ln\frac{2n-2}{\sqrt{n-1}} + O(1) \\
&= \ln(2\sqrt{n-1}) + O(1) \\
&= \ln 2 + \ln\sqrt{n} + \ln\frac{\sqrt{n-1}}{\sqrt{n}} + O(1) \\
&= \ln\sqrt{n} + O(1).
\end{aligned}$$

Wyrażenia $\ln\frac{2n-1}{2n-2}$ i $\ln\frac{\sqrt{n-1}}{\sqrt{n}}$ zostały potraktowane jak funkcje klasy $O(1)$.

A.1-3. Wykorzystując wzór (A.8), dostajemy

$$\sum_{k=0}^{\infty} k^2 x^k = x \cdot \frac{d}{dx} \sum_{k=0}^{\infty} k x^k = \frac{x(x+1)}{(1-x)^3}.$$

A.1-4. Korzystając ze wzorów (A.6) oraz (A.8), mamy

$$\sum_{k=0}^{\infty}\frac{k-1}{2^k}=\sum_{k=0}^{\infty}\frac{k}{2^k}-\sum_{k=0}^{\infty}\frac{1}{2^k}=\sum_{k=0}^{\infty}k\left(\frac{1}{2}\right)^k-\sum_{k=0}^{\infty}\left(\frac{1}{2}\right)^k=\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2}-\frac{1}{1-\frac{1}{2}}=0.$$

A.1-5. Załóżmy, że $|x^2|<1$, czyli $-1<x<1$. Wówczas na podstawie wzorów (A.6) i (A.8) obliczamy:

$$\sum_{k=1}^{\infty}(2k+1)x^{2k}=2\sum_{k=0}^{\infty}k(x^2)^k+\sum_{k=0}^{\infty}(x^2)^k-1=\frac{2x^2}{(1-x^2)^2}+\frac{1}{1-x^2}-1=\frac{x^2(3-x^2)}{(1-x^2)^2}.$$

W przypadku, gdy $|x^2|\geqslant 1$, badana suma jest rozbieżna do ∞.

A.1-6. Pokażemy mocniejszy wynik, zamieniając w dowodzonej tożsamości O na Θ. Dopuścimy też inną zmienną jako parametr funkcji f_k po obu stronach równości.

Zgodnie z definicją notacji Θ, dla dowolnej funkcji $F(i)$ zachodzi $F(i)=\Theta\bigl(\sum_{k=1}^{n}f_k(i)\bigr)$ wtedy i tylko wtedy, gdy istnieją dodatnie stałe i_0, c_1, c_2 takie, że dla każdego całkowitego $i\geqslant i_0$ prawdą jest

$$0\leqslant c_1\sum_{k=1}^{n}f_k(i)\leqslant F(i)\leqslant c_2\sum_{k=1}^{n}f_k(i).$$

Ustalmy stałe i_0, c_1 i c_2. Niech $F_1(i)$, $F_2(i)$, …, $F_n(i)$ będą funkcjami, które dla każdego całkowitego $i\geqslant i_0$ spełniają układ nierówności

$$\begin{cases}0\leqslant c_1f_1(i) & \leqslant F_1(i)\leqslant c_2f_1(i)\\ 0\leqslant c_1f_1(i)+c_1f_2(i) & \leqslant F_2(i)\leqslant c_2f_1(i)+c_2f_2(i)\\ & \vdots\\ 0\leqslant\sum_{k=1}^{n}c_1f_k(i) & \leqslant F_n(i)\leqslant\sum_{k=1}^{n}c_2f_k(i)\end{cases}.$$

Wprost z definicji notacji Θ mamy:

$$\begin{aligned}F_1(i)&=\Theta(f_1(i)),\\ F_2(i)-F_1(i)&=\Theta(f_2(i)),\\ &\vdots\\ F_n(i)-F_{n-1}(i)&=\Theta(f_n(i)).\end{aligned}$$

Dodając powyższe równania stronami, otrzymujemy

$$F_1(i)+(F_2(i)-F_1(i))+\cdots+(F_n(i)-F_{n-1}(i))=\sum_{k=1}^{n}\Theta(f_k(i)).$$

Wyrażenie po lewej stronie skraca się do $F_n(i)$. Ale z ostatniej nierówności układu wynika, że $F_n(i)=\Theta\bigl(\sum_{k=1}^{n}f_k(i)\bigr)$, skąd dostajemy tożsamość

$$\sum_{k=1}^{n}\Theta(f_k(i))=\Theta\left(\sum_{k=1}^{n}f_k(i)\right).$$

A.1-7.

$$\prod_{k=1}^{n} 2 \cdot 4^k = 2^n \cdot 4^{1+2+\cdots+n} = 2^n \cdot 4^{\frac{n(n+1)}{2}} = 2^{n(n+2)}$$

A.1-8.

$$\prod_{k=2}^{n}\left(1 - \frac{1}{k^2}\right) = \prod_{k=2}^{n} \frac{k^2-1}{k^2} = \frac{\prod_{k=2}^{n}(k^2-1)}{\prod_{k=2}^{n} k^2} = \frac{\prod_{k=2}^{n}(k-1) \cdot \prod_{k=2}^{n}(k+1)}{\left(\prod_{k=1}^{n} k\right)^2}$$

$$= \frac{\prod_{k=1}^{n-1} k \cdot \prod_{k=3}^{n+1} k}{(n!)^2} = \frac{(n-1)! \cdot \frac{(n+1)!}{2}}{(n!)^2} = \frac{n+1}{2n}$$

A.2. Szacowanie sum

A.2-1. Wykorzystując własności szeregu teleskopowego, dostajemy

$$\sum_{k=1}^{n} \frac{1}{k^2} \leqslant 1 + \sum_{k=2}^{n} \frac{1}{k(k-1)} = 1 + \sum_{k=2}^{n}\left(\frac{1}{k-1} - \frac{1}{k}\right) = 1 + 1 - \frac{1}{n} < 2,$$

a zatem badana suma jest ograniczona z góry przez stałą. Można pokazać, że wraz ze wzrostem n wartość tej sumy zbliża się do $\pi^2/6$.

A.2-2. Zauważmy, że gdy n osiąga wartość będącą potęgą 2, to zwiększa się o 1 liczba sumowanych składników. Funkcja $F(n) = \sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil$, zdefiniowana dla dodatnich liczb całkowitych, jest niemalejąca, więc jeśli m jest liczbą całkowitą dodatnią, to $F(n) < F(2^m)$ dla wszystkich $n < 2^m$. Oszacowaniem górnym tej funkcji jest zatem oszacowanie górne jej wartości przyjmowanych dla potęg 2. Dla $n = 2^m$ zachodzi

$$F(n) = F(2^m) = \sum_{k=0}^{m} \left\lceil \frac{2^m}{2^k} \right\rceil = 2^m + 2^{m-1} + \cdots + 2^0 = 2^{m+1} - 1 = 2n - 1,$$

z czego wynika, że $F(n) = O(n)$ i asymptotycznym górnym ograniczeniem sumy jest $O(n)$.

A.2-3. Postępując podobnie jak podczas badania oszacowania górnego n-tej liczby harmonicznej, dzielimy zakres indeksów od 1 do n na $\lfloor \lg n \rfloor$ części i ograniczamy sumę każdej części przez $1/2$:

$$\sum_{k=1}^{n} \frac{1}{k} \geqslant \sum_{i=0}^{\lfloor \lg n \rfloor - 1} \sum_{j=0}^{2^i - 1} \frac{1}{2^i + j} \geqslant \sum_{i=0}^{\lfloor \lg n \rfloor - 1} \sum_{j=0}^{2^i - 1} \frac{1}{2^{i+1}} = \sum_{i=0}^{\lfloor \lg n \rfloor - 1} \frac{1}{2} = \frac{\lfloor \lg n \rfloor}{2} = \Omega(\lg n).$$

A.2-4. Funkcja $f(k) = k^3$, gdzie k to dodatnia liczba całkowita, jest monotonicznie rosnąca, zatem szacujemy sumę, korzystając z nierówności (A.11):

$$\int_0^n x^3\, dx \leqslant \sum_{k=1}^{n} k^3 \leqslant \int_1^{n+1} x^3\, dx.$$

Znajdujemy oszacowania obu całek:

$$\int_0^n x^3\,dx = \frac{n^4}{4} = \Theta(n^4),$$

$$\int_1^{n+1} x^3\,dx = \frac{(n+1)^4 - 1}{4} = \Theta(n^4)$$

i otrzymujemy

$$\sum_{k=1}^{n} k^3 = \Theta(n^4).$$

A.2-5. Zastosowanie nierówności (A.12) do sumy $\sum_{k=1}^{n} 1/k$ doprowadza do całki niewłaściwej:

$$\sum_{k=1}^{n} \frac{1}{k} \leqslant \int_0^n \frac{dx}{x} = \lim_{a\to 0^+} \int_a^n \frac{dx}{x} = \infty.$$

W rezultacie nie uzyskujemy żadnej informacji o oszacowaniu górnym badanej sumy. Dzięki zapisaniu jej w postaci $1 + \sum_{k=2}^{n} 1/k$, można zastosować wzór (A.12) do drugiego składnika i otrzymać oszacowanie górne $\ln n + 1$ na sumę wyjściową.

Problemy

A-1. Szacowanie sum

W celu wyznaczenia asymptotycznych oszacowań dokładnych dla każdej z sum, znajdziemy ich oszacowania górne i dolne poprzez zastąpienie odpowiednimi wartościami każdego składnika danej sumy. Ponieważ parzystość n nie ma znaczenia dla postaci oszacowań tych sum, to dla uproszczenia rachunków będziemy przyjmować, że n jest liczbą parzystą.

(a) Oszacowanie górne:

$$\sum_{k=1}^{n} k^r \leqslant \sum_{k=1}^{n} n^r = n \cdot n^r = O(n^{r+1}).$$

Oszacowanie dolne:

$$\sum_{k=1}^{n} k^r \geqslant \sum_{k=n/2+1}^{n} k^r \geqslant \sum_{k=n/2+1}^{n} (n/2)^r = (n/2) \cdot (n/2)^r = \Omega(n^{r+1}).$$

Na podstawie otrzymanych wyników stwierdzamy, że oszacowaniem dokładnym sumy jest

$$\sum_{k=1}^{n} k^r = \Theta(n^{r+1}).$$

(b) Oszacowanie górne:

$$\sum_{k=1}^{n} \lg^s k \leqslant \sum_{k=1}^{n} \lg^s n = n \cdot \lg^s n = O(n \lg^s n).$$

Oszacowanie dolne:

$$\sum_{k=1}^{n} \lg^s k \geqslant \sum_{k=n/2+1}^{n} \lg^s k \geqslant \sum_{k=n/2+1}^{n} \lg^s(n/2) = (n/2) \cdot \lg^s(n/2) = \Omega(n \lg^s n).$$

Oszacowanie dokładne:

$$\sum_{k=1}^{n} \lg^s k = \Theta(n \lg^s n).$$

(c) Oszacowanie górne:

$$\sum_{k=1}^{n} k^r \lg^s k \leqslant \sum_{k=1}^{n} n^r \lg^s n = n \cdot n^r \lg^s n = O(n^{r+1} \lg^s n).$$

Oszacowanie dolne:

$$\sum_{k=1}^{n} k^r \lg^s k \geqslant \sum_{k=n/2+1}^{n} k^r \lg^s k \geqslant \sum_{k=n/2+1}^{n} (n/2)^r \lg^s(n/2) = (n/2) \cdot (n/2)^r \lg^s(n/2) = \Omega(n^{r+1} \lg^s n).$$

Oszacowanie dokładne:

$$\sum_{k=1}^{n} k^r \lg^s k = \Theta(n^{r+1} \lg^s n).$$

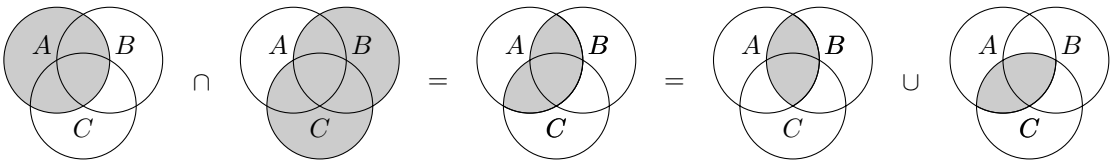
Przyjmując w powyższym oszacowaniu, odpowiednio, $s = 0$ i $r = 0$, otrzymujemy sumy i ich oszacowania z punktów (a) i (b).

Dodatek **B**

Zbiory i nie tylko

B.1. Zbiory

B.1-1. Pierwsze prawo rozdzielności zostało zilustrowane na rys. 35.



Rysunek 35: Diagramy Venna ilustrujące pierwsze prawo rozdzielności.

B.1-2. Przeprowadzimy dowód pierwszego wzoru przez indukcję względem n. Dla $n = 1$ dowód jest trywialny, a dla $n = 2$ wzór stanowi pierwsze prawo de Morgana. Załóżmy więc, że $n > 2$ i że wzór zachodzi dla rodziny $n - 1$ zbiorów. Mamy

$$\begin{aligned}\overline{A_1 \cap A_2 \cap \cdots \cap A_{n-1} \cap A_n} &= \overline{(A_1 \cap A_2 \cap \cdots \cap A_{n-1}) \cap A_n} \\ &= \overline{A_1 \cap A_2 \cap \cdots \cap A_{n-1}} \cup \overline{A_n} \\ &= \overline{A_1} \cup \overline{A_2} \cup \cdots \cup \overline{A_{n-1}} \cup \overline{A_n}.\end{aligned}$$

W drugiej równości skorzystano z pierwszego prawa de Morgana, a w trzeciej – z założenia indukcyjnego.

Dowód drugiego wzoru przebiega analogicznie. Jeśli $n = 2$, to wzór jest drugim prawem de Morgana, a jeśli $n > 2$, to wystarczy zastosować powyższe rozumowanie z zamienionymi symbolami sumy i przecięcia zbiorów oraz skorzystać z drugiego prawa de Morgana.

B.1-3. Udowodnimy zasadę włączania i wyłączania przez indukcję względem n. Dla $n = 1$ dowód jest trywialny, a dla $n = 2$ zasada stanowi wzór (B.3). Jeśli $n > 2$, to na mocy tegoż wzoru, jak i uogólnionego pierwszego prawa rozdzielności, mamy

$$\begin{aligned}|A_1 \cup A_2 \cup \cdots \cup A_n| &= \big|(A_1 \cup A_2 \cup \cdots \cup A_{n-1}) \cup A_n\big| \\ &= |A_1 \cup A_2 \cup \cdots \cup A_{n-1}| + |A_n| - \big|(A_1 \cup A_2 \cup \cdots \cup A_{n-1}) \cap A_n\big|\end{aligned}$$

$$= |A_1 \cup A_2 \cup \cdots \cup A_{n-1}| + |A_n| - \big|(A_1 \cap A_n) \cup \cdots \cup (A_{n-1} \cap A_n)\big|.$$

Stosujemy teraz założenie indukcyjne do pierwszego i ostatniego składnika, w wyniku czego otrzymujemy

$$|A_1 \cup A_2 \cup \cdots \cup A_{n-1}| = \sum_{1 \leqslant i_1 < n} |A_{i_1}| - \sum_{1 \leqslant i_1 < i_2 < n} |A_{i_1} \cap A_{i_2}| + \sum_{1 \leqslant i_1 < i_2 < i_3 < n} |A_{i_1} \cap A_{i_2} \cap A_{i_3}| - \cdots + (-1)^{n-2} |A_1 \cap A_2 \cap \cdots \cap A_{n-1}|$$

oraz

$$\big|(A_1 \cap A_n) \cup \cdots \cup (A_{n-1} \cap A_n)\big| = \sum_{1 \leqslant i_1 < n} |A_{i_1} \cap A_n| - \sum_{1 \leqslant i_1 < i_2 < n} |A_{i_1} \cap A_{i_2} \cap A_n| + \cdots + (-1)^{n-2} |A_1 \cap A_2 \cap \cdots \cap A_{n-1} \cap A_n|.$$

Wystarczy wstawić otrzymane wyrażenia do początkowego wzoru:

$$|A_1 \cup A_2 \cup \cdots \cup A_n| = \sum_{1 \leqslant i_1 \leqslant n} |A_{i_1}| - \sum_{1 \leqslant i_1 < i_2 \leqslant n} |A_{i_1} \cap A_{i_2}| + \sum_{1 \leqslant i_1 < i_2 < i_3 \leqslant n} |A_{i_1} \cap A_{i_2} \cap A_{i_3}| - \cdots + (-1)^{n-1} |A_1 \cap A_2 \cap \cdots \cap A_n|.$$

A zatem zasada zachodzi dla dowolnej skończonej rodziny zbiorów.

B.1-4. *Poniższe rozwiązanie dowodzi przeliczalności zbioru nieparzystych liczb naturalnych, czego dotyczy oryginalna treść zadania. Tłumaczenie pyta natomiast o przeliczalność zbioru wszystkich liczb nieparzystych.*

Aby wykazać ten fakt, należy znaleźć wzajemnie jednoznaczne odwzorowanie zbioru $\mathbb{N}$ w zbiór $\{\, 2n+1 : n \in \mathbb{N} \,\}$. Każdej liczbie naturalnej n przyporządkujmy liczbę $2n+1$. Bijektywność tego odwzorowania jest oczywista, wnioskujemy zatem, że zbiór nieparzystych liczb naturalnych jest przeliczalny.

B.1-5. Dowodzimy przez indukcję względem liczby elementów zbioru S. Jeśli $|S| = 0$, czyli $S = \emptyset$, to $|2^S| = 2^{|S|} = 1$, bo S ma tylko jeden podzbiór – zbiór pusty. Niech teraz $|S| > 0$ i załóżmy, że $|2^S| = 2^{|S|}$. Ustalmy $p \notin S$ i rozważmy zbiór $S' = S \cup \{p\}$. Podzbiory zbioru S' można podzielić na takie, które zawierają p i na takie, które nie zawierają p. Tych ostatnich jest $|2^S| = \big|2^{S' \setminus \{p\}}\big| = 2^{|S' \setminus \{p\}|}$ na mocy założenia indukcyjnego. Okazuje się, że podzbiorów zawierających p jest tyle samo, ponieważ każdy powstaje przez zsumowanie singletonu $\{p\}$ z pewnym podzbiorem niezawierającym p. Mamy zatem

$$\big|2^{S'}\big| = 2 \cdot 2^{|S' \setminus \{p\}|} = 2^{|S' \setminus \{p\}|+1} = 2^{|S'|}.$$

Na mocy indukcji twierdzenie jest spełnione dla dowolnego zbioru skończonego.

B.1-6.

$$\langle a_1, a_2, \ldots, a_n \rangle = \begin{cases} \emptyset, & \text{jeśli } n = 0, \\ \{a_1\}, & \text{jeśli } n = 1, \\ \{a_1, \{a_1, a_2\}\}, & \text{jeśli } n = 2, \\ \langle a_1, \langle a_2, \ldots, a_n \rangle \rangle, & \text{jeśli } n \geqslant 3. \end{cases}$$

B.2. Relacje

B.2-1. Porządek częściowy to relacja zwrotna, antysymetryczna i przechodnia. Zauważmy, że relacja $\subseteq$ w $2^{\mathbb{Z}}$ posiada każdą z tych własności. Dla zbioru $A \in 2^{\mathbb{Z}}$ zachodzi $A \subseteq A$ (zwrotność). Dla zbiorów A, $B \in 2^{\mathbb{Z}}$, jeśli zachodzi $A \subseteq B$ i $B \subseteq A$, to $A = B$ (antysymetria). Wreszcie dla zbiorów A, B, $C \in 2^{\mathbb{Z}}$, jeżeli $A \subseteq B$ i $B \subseteq C$, to $A \subseteq C$ (przechodniość). Jednak w $2^{\mathbb{Z}}$ porządek $\subseteq$ nie jest liniowy, bo np. $\{0,1\} \not\subseteq \{1,2\}$ i $\{1,2\} \not\subseteq \{0,1\}$.

B.2-2. Oznaczmy przez R_n, dla $n \in \mathbb{N} \setminus \{0\}$, relację „przystaje modulo n”:

$$R_n = \big\{ \langle a,b\rangle \in \mathbb{Z} \times \mathbb{Z} \;:\; a \equiv b \pmod{n} \big\}.$$

Dla dowolnego $a \in \mathbb{Z}$ mamy $a \equiv a \pmod{n}$, bo $a - a = 0$, więc relacja R_n jest zwrotna. Dla dowolnych a, $b \in \mathbb{Z}$, jeśli istnieje $q \in \mathbb{Z}$, że $a - b = qn$, to $b - a = -qn$, a zatem z faktu, że $a \equiv b \pmod{n}$ wynika, że $b \equiv a \pmod{n}$, co dowodzi symetrii R_n. Dla dowodu przechodniości wybierzmy dowolne $a, b, c \in \mathbb{Z}$ i załóżmy, że zachodzi $a \equiv b \pmod{n}$ oraz $b \equiv c \pmod{n}$. Oznacza to, że istnieją q, $r \in \mathbb{Z}$, że $a-b = qn$ oraz $b-c = rn$. Stąd $a-c = a-b+b-c = qn+rn = (q+r)n$, a zatem $a \equiv c \pmod{n}$.

Na mocy powyższych faktów R_n jest relacją równoważności i dzieli zbiór $\mathbb{Z}$ na n klas abstrakcji; i-ta klasa, gdzie $i = 1, 2, \ldots, n$, jest zbiorem takich liczb całkowitych, które przy dzieleniu przez n dają resztę $i - 1$.

B.2-3.

(a) Relacja $R = \big\{\langle a,a\rangle, \langle b,b\rangle, \langle c,c\rangle, \langle a,b\rangle, \langle b,a\rangle, \langle a,c\rangle, \langle c,a\rangle\big\}$ określona w zbiorze $\{a,b,c\}$.

(b) Relacja porządku $\leqslant$ określona w zbiorze liczb rzeczywistych.

(c) Relacja $R = \big\{\langle a,a\rangle, \langle b,b\rangle, \langle a,b\rangle, \langle b,a\rangle\big\}$ określona w zbiorze $\{a,b,c\}$.

B.2-4. Jeśli R jest relacją równoważności, to dla każdego $s \in S$ zachodzi $s \in [s]$. Na mocy antysymetrii R, jeśli zachodzi $s' \,R\, s$ oraz $s \,R\, s'$, to $s = s'$, a więc nie istnieją takie elementy s', że $s' \in [s] \setminus \{s\}$. To oznacza, że klasy abstrakcji S względem relacji R są singletonami.

B.2-5. Symetria i przechodniość relacji zdefiniowane są za pomocą implikacji, do spełnienia których nie jest konieczne spełnienie ich poprzedników. Relacja pozostanie symetryczna i przechodnia, jeśli w zbiorze, w którym jest określona, istnieje element niebędący w relacji z żadnym elementem z tego zbioru. Zwrotność wymaga natomiast, aby każdy element był w relacji z samym sobą. Istnieją zatem relacje symetryczne i przechodnie, ale nie zwrotne (patrz punkt (c) zad. B.2-3).

B.3. Funkcje

B.3-1.

(a) Zbiór wartości funkcji $f\colon A \to B$, czyli obraz jej dziedziny, jest zdefiniowany następująco:

$$f(A) = \big\{\, b \in B : b = f(a) \text{ dla pewnego } a \in A \,\big\}.$$

Z tego, że f jest injekcją, mamy, że $|A| = |f(A)|$. Z kolei $|f(A)| \leqslant |B|$, bo w B mogą być takie elementy b, dla których nie istnieje $a \in A$ takie, że $b = f(a)$. Stąd $|A| \leqslant |B|$.

(b) Dla surjekcji $f\colon A \to B$ zachodzi $f(A) = B$, więc $|f(A)| = |B|$. Dla pewnych elementów a_1, $a_2 \in A$ może zachodzić $f(a_1) = f(a_2)$, mamy zatem $|A| \geqslant |f(A)|$, a stąd $|A| \geqslant |B|$.

Z powyższych faktów wynika, że jeśli $f\colon A \to B$ jest bijekcją, to $|A| = |B|$.

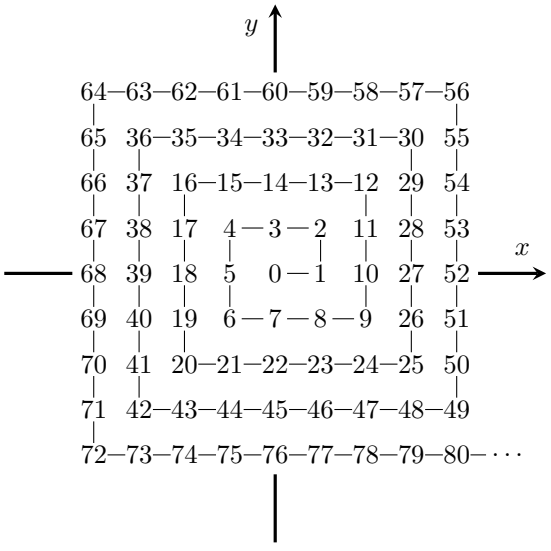
B.3-2. Funkcja $f(x) = x + 1$ o dziedzinie i przeciwdziedzinie $\mathbb{N}$ nie jest bijekcją, gdyż dla żadnego $x \in \mathbb{N}$ nie zachodzi $f(x) = 0$. Jeśli zamiast $\mathbb{N}$ rozważymy $\mathbb{Z}$, to f będzie bijekcją – każda liczba całkowita jest wartością funkcji f dla pewnej jednoznacznie wyznaczonej liczby całkowitej.

B.3-3. Niech R będzie relacją binarną w zbiorze A. Relację R^{-1} w zbiorze A nazywamy relacją odwrotną do R, jeżeli dla dowolnych a, $b \in A$, $a\,R^{-1}\,b$ wtedy i tylko wtedy, gdy $b\,R\,a$. Łatwo sprawdzić, że jeśli R jest bijekcją, to R^{-1} jest jej funkcją odwrotną.

B.3-4. Ponieważ każda bijekcja posiada funkcję odwrotną, która także jest bijekcją, to znajdziemy funkcję $F\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ będącą odwrotnością szukanego odwzorowania. Wyznaczenie F jest równoważne znalezieniu sposobu ponumerowania kolejnymi liczbami całkowitymi każdej pary o elementach całkowitych tak, aby każda liczba całkowita była wykorzystana jako numer pewnej pary. Opiszemy teraz konstrukcję jednej z takich funkcji.

Dokonajmy pewnego uproszczenia – zamiast numerować pary liczbami całkowitymi, ograniczymy się do liczb naturalnych. Niech $g\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$ oraz $h\colon \mathbb{N} \to \mathbb{Z}$ będą takimi bijekcjami, że $F = h \circ g$. Łatwo wykazać, że $h(n) = (-1)^n \lceil n/2 \rceil$ jest bijekcją, pozostaje więc znaleźć funkcję g.

Rozważmy numerację par o elementach całkowitych przedstawioną na rys. 36 w formie spirali. Ponieważ każdej takiej parze $\langle x, y \rangle$ przypisywana jest unikalna liczba naturalna, to możemy tę spiralę potraktować jak opis funkcji g. Przyjmijmy wpierw oznaczenia: $d = \max(|x|, |y|)$ oraz



Rysunek 36: Bijekcja ze zbioru $\mathbb{Z} \times \mathbb{Z}$ w zbiór $\mathbb{N}$. Poszczególne liczby naturalne oznaczają wartości tej bijekcji dla punktów o współrzędnych całkowitych w kartezjańskim układzie współrzędnych.

$D = (2d-1)^2 - 1$. Nieformalnie liczby te oznaczają, odpowiednio, numer „okrążenia" punktu $\langle 0,0\rangle$ pokonywanego przez spiralę w momencie przechodzenia przez punkt $\langle x,y\rangle$ oraz największą wartość przyjmowaną przez spiralę podczas pokonywania poprzedniego „okrążenia". Można przyjąć następującą definicję funkcji g:

$$g(x,y) = \begin{cases} 0, & \text{jeśli } d = |x| = |y| = 0, \\ D + d + y, & \text{jeśli } d = x \neq |y|, \\ D + 3d - x, & \text{jeśli } d = y \neq 0, \\ D + 5d - y, & \text{jeśli } d = -x \neq |y|, \\ D + 7d + x, & \text{jeśli } d = -y \neq 0. \end{cases}$$

Zaprezentowana tutaj spirala przypomina znaną w literaturze **spiralę Ulama**, opisaną w [11] i wykorzystywaną do znajdowania własności liczb pierwszych.

B.4. Grafy

B.4-1. Jeśli będziemy reprezentować zbiór pracowników przez zbiór wierzchołków V, a dla każdych u, $v \in V$ relację „pracownik u podał rękę pracownikowi v" przez zbiór krawędzi E, to otrzymamy graf nieskierowany $G = \langle V, E\rangle$. Sumując stopnie wszystkich wierzchołków tego grafu, otrzymamy podwojoną liczbę krawędzi, gdyż każdą krawędź policzymy dwa razy (każda krawędź jest incydentna z dokładnie dwoma wierzchołkami). Mamy więc

$$\sum_{v \in V} \deg(v) = 2|E|.$$

B.4-2. Ścieżka z wierzchołka u do wierzchołka v w dowolnym grafie jest skończonym ciągiem wierzchołków kolejno odwiedzanych na tej ścieżce, $\langle v_0, v_1, \ldots, v_n\rangle$, przy czym $v_0 = u$ i $v_n = v$. Jeśli ścieżka jest prosta, to wyrazy tego ciągu nie powtarzają się. W przeciwnym przypadku, jeśli podciągiem spójnym ścieżki z u do v jest $\langle v_i, v_{i+1}, \ldots, v_{i+k}, v_i\rangle$, to eliminując jego podciąg $\langle v_i, v_{i+1}, \ldots, v_{i+k}\rangle$, odrzucamy jedno powtórzenie v_i, a tym samym podcykl ścieżki, który sprawia, że nie jest ona prosta. Po eliminacji wszystkich takich podcykli otrzymujemy ścieżkę prostą. Oznacza to, że każda ścieżka zawiera ścieżkę prostą reprezentowaną przez ciąg wierzchołków pozostawiony z początkowego ciągu po zastosowaniu opisanej procedury.

Dowód dla cykli przeprowadzamy analogicznie z $v_n = u$, pamiętając jednak, by nie eliminować ostatniego powtórzenia u, które jest wymagane do tego, by ścieżka stanowiła cykl.

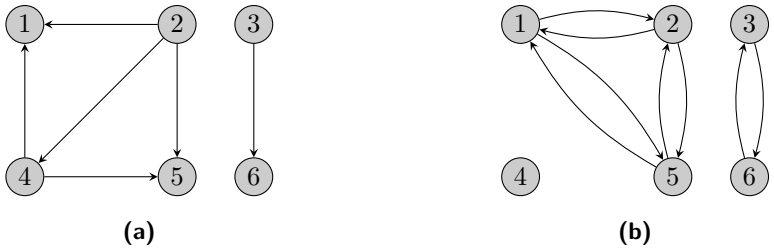
B.4-3. Z twierdzenia B.2 mamy, że graf $G = \langle V, E\rangle$ będący drzewem, jest spójny i acykliczny oraz że ma $|E| = |V| - 1$ krawędzi. Gdy dodamy do E nową krawędź, to G nie będzie już drzewem, ale nadal będzie spójny – może być zatem $|E| > |V| - 1$. Z kolei gdy usuniemy z E jakąkolwiek krawędź, to rozspójnimy G, przez co nie może zachodzić $|E| < |V| - 1$.

B.4-4. Każdy wierzchołek grafu skierowanego lub nieskierowanego jest osiągalny z samego siebie, ponieważ istnieje ścieżka o długości równej 1 zawierająca tylko ten wierzchołek, zatem relacja osiągalności jest zwrotna.

Dla dowolnych wierzchołków u, v i w grafu skierowanego lub nieskierowanego z faktu, że $u \rightsquigarrow v$ i $v \rightsquigarrow w$ wynika, że $u \rightsquigarrow w$. Istnieje bowiem ścieżka z u do w będąca konkatenacją ciągów reprezentujących ścieżki z u do v i z v do w (z pominięciem powtórzenia v między nimi).

Relacja osiągalności jest symetryczna jedynie w grafach nieskierowanych, gdyż dla dowolnych wierzchołków u i v, jeśli $u \rightsquigarrow v$, to $v \rightsquigarrow u$. Ścieżka z v do u powstaje przez lustrzane odbicie ścieżki z u do v; powstały ciąg reprezentuje poprawną ścieżkę, bo każdą krawędzią można poruszać się w obie strony. W grafie skierowanym krawędzie są jednokierunkowe, więc symetria nie zachodzi.

B.4-5. Szukane grafy przedstawiono na rys. 37.



Rysunek 37: (a) Wersja nieskierowana grafu skierowanego z rysunku B.2(a). **(b)** Wersja skierowana grafu nieskierowanego z rysunku B.2(b).

B.4-6. Hipergraf $H = \langle V_H, E_H \rangle$ można reprezentować jako graf dwudzielny $G = \langle V_1 \cup V_2, E \rangle$, w którym $V_1 = V_H$ oraz $V_2 = E_H$. Krawędź $\langle u, v \rangle \in V_1 \times V_2$ w grafie G istnieje wtedy i tylko wtedy, gdy hiperkrawędź v jest incydentna z u (hiperkrawędzie mogą być incydentne z więcej niż dwoma wierzchołkami). W grafie G nie istnieją krawędzie pomiędzy elementami z V_1 ani pomiędzy elementami z V_2, zatem G istotnie jest dwudzielny.

B.5. Drzewa

B.5-1. Rys. 38 ilustruje wszystkie szukane drzewa.

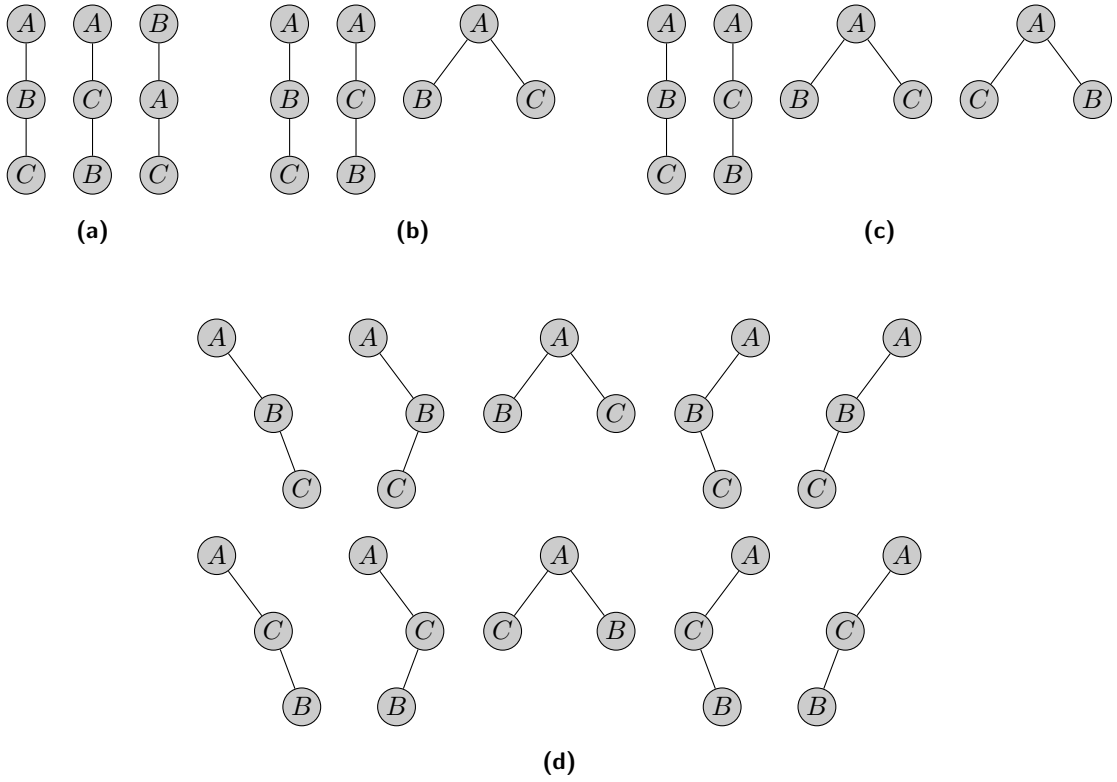
B.5-2. Przypuśćmy, że twierdzenie jest fałszywe, czyli że wersja nieskierowana grafu G nie tworzy drzewa, a więc posiada cykl, w szczególności cykl prosty (zad. B.4-2). Niech $\langle v_1, v_2, \ldots, v_k, v_1 \rangle$ będzie takim cyklem. Graf G jest acykliczny, zatem dla pewnego całkowitego l, gdzie $1 \leqslant l \leqslant k$, istnieją krawędzie $\langle v_l, v_{l+1} \rangle$, $\langle v_{l+2}, v_{l+1} \rangle \in E$, przy czym v_{k+1} utożsamiamy z v_1, a v_{k+2} z v_2. Wiemy z założenia, że $v_0 \rightsquigarrow v_l$ oraz $v_0 \rightsquigarrow v_{l+2}$, zatem istnieją dwie różne ścieżki z v_0 do v_{l+1}:

$$\langle v_0, \ldots, v_l, v_{l+1} \rangle \quad \text{oraz} \quad \langle v_0, \ldots, v_{l+2}, v_{l+1} \rangle.$$

Otrzymana sprzeczność prowadzi do wniosku, że wersja nieskierowana grafu G jest acykliczna, zatem istotnie stanowi drzewo.

B.5-3. W drzewie o jednym węźle jest jeden liść i brak węzłów wewnętrznych, więc krok bazowy indukcji zachodzi. Zauważmy, że drzewo można ściągnąć wzdłuż wszystkich krawędzi pomiędzy węzłami stopnia 1, a ich synami, nie powodując zmian w liczbie węzłów stopnia 2. Wykonanie tej operacji pozbawia drzewo wszystkich węzłów stopnia 1. W dalszej części dowodu będziemy zatem rozważać tylko drzewa regularne.

Lemat. *Niepuste regularne drzewo binarne ma nieparzystą liczbę węzłów.*



Rysunek 38: (a) Drzewa wolne o 3 wierzchołkach A, B i C. **(b)** Drzewa ukorzenione o węzłach A, B i C, w których A jest korzeniem. **(c)** Drzewa uporządkowane o węzłach A, B i C, w których A jest korzeniem. **(d)** Drzewa binarne o węzłach A, B i C, w których A jest korzeniem.

Dowód. Niech $T = \langle V, E\rangle$ będzie niepustym regularnym drzewem binarnym, a $L \subseteq V$ – zbiorem liści tego drzewa. Obliczmy sumę stopni wszystkich węzłów T (w sensie grafowym, czyli uwzględniając ojca węzła):

$$\sum_{v\in V} \deg(v) = \sum_{v\in L} 1 + \sum_{v\in V\setminus L} 3 \ - 1 = 3|V| - 2|L| - 1.$$

Z lematu o podawaniu rąk (zad. B.4-1) mamy, że $\sum_{v\in V} \deg(v) = 2|E|$, a stąd

$$|V| = \frac{2|E| + 2|L| + 1}{3}.$$

Licznik ułamka jest nieparzysty, zatem liczba węzłów T także jest nieparzysta. □

Korzystając z powyższego lematu, założymy, że twierdzenie jest prawdziwe dla drzewa o $2k-1$ węzłach ($k \geqslant 1$) i wykażemy jego prawdziwość dla drzewa o $2k+1$ węzłach. Mamy, że liczba węzłów w stopnia 2 w regularnym drzewie binarnym o $2k-1$ węzłach jest o 1 mniejsza od liczby jego liści l. Wybierając dowolny liść i czyniąc z niego węzeł wewnętrzny, poprzez dołączenie do niego dwóch synów, tworzymy regularne drzewo binarne o $2k+1$ węzłach. W nowym drzewie mamy $w' = w+1$ węzłów stopnia 2 oraz $l' = (l-1)+2 = l+1$ liści, więc z założenia indukcyjnego dostajemy $w' = w + 1 = (l-1) + 1 = l' - 1$, a zatem twierdzenie jest prawdziwe.

B.5-4. Udowodnimy nierówność $h \geqslant \lfloor \lg n \rfloor$ przez indukcję względem n. Jeśli $n = 1$, to drzewo posiada tylko jeden węzeł, więc $h = 0$ i nierówność oczywiście zachodzi. Załóżmy teraz, że $n \geqslant 2$ oraz że nierówność jest spełniona dla wszystkich drzew binarnych o $n - 1$ węzłach i wysokości h. Niech T będzie jednym z nich. Dodając do niego nowy węzeł, tworzymy nowe drzewo T' o wysokości h'. Rozważmy dwa przypadki w zależności od położenia tego węzła w drzewie T'.

Przyjmijmy najpierw, że nowy węzeł został umieszczony na co najwyżej h-tym poziomie. Wówczas $h' = h$. Jedyny przypadek, gdy nierówność $h' \geqslant \lfloor \lg n \rfloor$ nie jest spełniona, występuje wówczas, gdy $n = 2^{h+1}$, czyli gdy T jest pełnym drzewem binarnym. Ale każdy poziom takiego drzewa ma komplet węzłów, dlatego nowy węzeł może zostać umieszczony jedynie na $(h+1)$-szym poziomie, co przeczy założeniu. A zatem nierówność jest spełniona.

W przypadku, gdy nowy węzeł zajął w T' poziom $h + 1$, jest $h' = h + 1$. Z założenia indukcyjnego mamy $h \geqslant \lfloor \lg(n-1) \rfloor$, zatem wystarczy pokazać, że $\lfloor \lg(n-1) \rfloor + 1 \geqslant \lfloor \lg n \rfloor$. Zauważmy, że dla dowolnej liczby rzeczywistej x i dowolnej liczby całkowitej k zachodzi $\lfloor x \rfloor + k = \lfloor x + k \rfloor$. Mamy więc

$$\lfloor \lg(n-1) \rfloor + 1 = \lfloor \lg(n-1) + 1 \rfloor = \lfloor \lg(n-1) + \lg 2 \rfloor = \lfloor \lg(2n-2) \rfloor.$$

Podłoga oraz logarytm przy podstawie 2 są funkcjami niemalejącymi, a więc $\lfloor \lg(2n-2) \rfloor \geqslant \lfloor \lg n \rfloor$, o ile $2n - 2 \geqslant n$, czyli gdy $n \geqslant 2$. Nierówność zachodzi zatem dla dowolnego drzewa binarnego.

B.5-5. Dowodzimy przez indukcję względem n. Gdy $n = 0$, to drzewo jest puste, więc $i = e = 0$ i baza zachodzi. Załóżmy więc, że $n > 0$ i że równanie $e = i + 2(n - 1)$ jest spełnione przez drzewo regularne o $n - 1$ węzłach wewnętrznych. W wyniku dołączenia dwóch nowych węzłów do pewnego liścia, ten staje się węzłem wewnętrznym. Zwiększamy przez to zarówno liczbę liści, jak i liczbę węzłów wewnętrznych drzewa o 1.

Zbadajmy, co się dzieje z długościami ścieżek wewnętrznej i zewnętrznej po takiej modyfikacji. Oznaczmy przez e' oraz i', odpowiednio, długość nowej ścieżki zewnętrznej i długość nowej ścieżki wewnętrznej, a przez d – głębokość nowego węzła wewnętrznego. Zachodzi $e' = e - d + 2(d+1) = i + 2(n-1) + d + 2$ oraz $i' = i + d$, a zatem $e' = i' + 2n$ i twierdzenie jest spełnione, gdyż teraz w drzewie jest n węzłów wewnętrznych.

B.5-6. Niech h będzie wysokością drzewa binarnego T. Zauważmy, że badana suma wag liści drzewa T nie zmniejszy się, jeśli do każdego węzła o stopniu 1 w tym drzewie dołączymy jego brakującego syna. Nowe węzły są nowymi liśćmi drzewa, zatem powiększają one sumę wag liści. Wagą liścia x na głębokości d jest $2^{-d} = 2 \cdot 2^{-(d+1)}$, więc jeśli uczynimy z x ojca dwóch nowych węzłów, to suma wag liści pozostanie niezmieniona. Powtarzając tę czynność dla każdego liścia x (również dla tych, które powstają w wyniku tej procedury) o głębokości mniejszej niż h, otrzymamy w końcu pełne drzewo binarne T' o wysokości h. Suma wag liści drzewa T nie przekracza sumy wag liści drzewa T', która wynosi

$$\sum_x w(x) = 2^h \cdot 2^{-h} = 1,$$

gdzie sumujemy względem wszystkich liści x z T'.

B.5-7. *W tekście zadania występuje błąd. Twierdzenie nie zachodzi bowiem dla drzew o jednym liściu, dlatego w rozwiązaniu zakładamy, że $L \geqslant 2$.*

Niech T będzie drzewem binarnym o $L \geqslant 2$ liściach oraz niech LT i RT będą, odpowiednio, jego lewym i prawym poddrzewem. Niech ponadto L_1 i L_2 stanowią, odpowiednio, liczbę liści LT

i liczbę liści RT. Bez straty ogólności załóżmy, że $L_1 \leqslant L_2$. Jeśli $L_2 \leqslant 2L/3$, to zarówno LT, jak i RT stanowi szukane poddrzewo. W przeciwnym przypadku zachodzi $L_2 > 2L/3$, więc szukane poddrzewo będzie częścią drzewa RT. Po skończonej liczbie kroków dojdziemy do drzewa T', którego większe poddrzewo RT' będzie mieć nie więcej niż $2L/3$ liści. Ale ponieważ T' ma więcej niż $2L/3$ liści, to liczba liści RT' jest większa niż $L/3$, zatem RT' jest szukanym poddrzewem.

Problemy

B-1. Kolorowanie grafów

(a) Zamiast dowolnych drzew rozważmy bez straty ogólności drzewa ukorzenione. Krawędzie w takim drzewie są incydentne z węzłami z sąsiednich poziomów, a więc można węzłom nadać kolory na podstawie parzystości ich głębokości w drzewie.

(b) Rozważmy bez straty ogólności tylko grafy spójne, gdyż stwierdzenia te pozostaną równoważne, jeżeli zostaną zastosowane osobno dla każdej składowej.

1. $\Rightarrow$ 2.: W grafie dwudzielnym krawędzie łączą wierzchołki między dwoma rozłącznymi zbiorami, zatem można pokolorować wierzchołki dwiema barwami w zależności od ich przynależności do danego zbioru, uzyskując prawidłowe 2-kolorowanie.

2. $\Rightarrow$ 3.: Załóżmy, że graf G jest 2-kolorowalny i że ma cykl nieparzysty $\langle v_1, v_2, \ldots, v_{2k+1}, v_1\rangle$ dla pewnego $k \geqslant 1$. Bez utraty ogólności niech $c(v_1) = 0$. Wtedy musi być $c(v_{2i}) = 1$ oraz $c(v_{2i+1}) = 0$, gdzie $i = 1, 2, \ldots, k$. Jednak wówczas dwa sąsiednie wierzchołki mają ten sam kolor, $c(v_1) = c(v_{2k+1}) = 0$, co przeczy założeniu, że G jest 2-kolorowalny. Wnioskujemy zatem, że G nie zawiera cyklu o długości nieparzystej.

3. $\Rightarrow$ 1.: Ustalmy pewne $v \in V$. Niech V_1 będzie zbiorem wszystkich wierzchołków grafu G, które znajdują się w odległości parzystej od v oraz niech $V_2 = V \setminus V_1$. Ponieważ G nie zawiera cyklu nieparzystego, to żaden jego wierzchołek nie sąsiaduje z innym wierzchołkiem ze swojego zbioru, a to oznacza, że graf $G = \langle V_1 \cup V_2, E\rangle$ jest dwudzielny.

(c) Dowód przeprowadzimy indukcyjnie ze względu na liczbę wierzchołków w grafie G. Jeśli graf ma jeden wierzchołek, to oczywiście wystarcza jeden kolor. Załóżmy więc, że $|V| \geqslant 2$. Wybierzmy dowolny wierzchołek $v \in V$ i oznaczmy przez G' podgraf G indukowany przez zbiór $V \setminus \{v\}$. Na mocy założenia indukcyjnego G' da się pokolorować $d' + 1$ barwami, gdzie d' to maksymalny stopień wierzchołka w G'. Ale $d' \leqslant d$, więc tym bardziej wystarczy $d + 1$ kolorów. Ponieważ wierzchołek v ma co najwyżej d sąsiadów, to wśród $d+1$ kolorów użytych w kolorowaniu podgrafu G' jest jeden nieprzypisany żadnemu sąsiadowi wierzchołka v. Wybierając ten kolor dla v, uzyskujemy poprawne $(d + 1)$-kolorowanie grafu G.

(d) Jeżeli k barw wystarcza do optymalnego pokolorowania (czyli takiego, które wykorzystuje możliwie najmniej kolorów) grafu G, to dla każdych dwóch różnych barw istnieje krawędź w E incydentna z wierzchołkami o takich barwach. W przeciwnym przypadku istniałyby dwie różne barwy, których moglibyśmy nie rozróżniać. Do pokolorowania grafu wystarczyłoby ich zatem $k - 1$, co przeczyłoby optymalności kolorowania.

Na podstawie powyższego rozumowania mamy $\binom{k}{2} \leqslant |E|$. Korzystamy z tego, że dla $k \geqslant 2$ zachodzi $k/2 \leqslant k - 1$ i stąd

$$k^2 = 4 \cdot \frac{k}{2} \cdot \frac{k}{2} \leqslant 4 \cdot \frac{k(k-1)}{2} = 4\binom{k}{2} \leqslant 4|E| = O(|V|),$$

czyli $k = O\big(\sqrt{|V|}\big)$.

B-2. Grafy znajomości

(a) Twierdzenie. *W grafie nieskierowanym $G = \langle V, E\rangle$, w którym $|V| \geqslant 2$, istnieją dwa wierzchołki o tym samym stopniu.*

Dowód. W grafie G o $n \geqslant 2$ wierzchołkach możliwymi stopniami wierzchołków są liczby 0, 1, …, $n-1$. Jeśli jednak pewien wierzchołek ma stopień równy 0, to żaden z pozostałych nie ma stopnia $n-1$. Oznacza to, że jest n wierzchołków, ale tylko co najwyżej $n-1$ liczb mogących jednocześnie być ich stopniami w G, zatem pewne dwa wierzchołki mają równe stopnie. □

(b) Twierdzenie. *Graf pełny posiadający 3 wierzchołki jest podgrafem dowolnego grafu nieskierowanego $G = \langle V, E\rangle$, w którym $|V| = 6$, lub jego dopełnienia $\overline{G}$.*

Dowód. Wybierzmy pewne $v \in V$. Istnieją wtedy w V trzy inne wierzchołki v_1, v_2, v_3 wszystkie sąsiednie z v albo wszystkie niesąsiednie z v. Ponieważ przypadki te są symetryczne, rozważmy pierwszy z nich. Jeśli nie istnieje wśród v_1, v_2, v_3 para wierzchołków sąsiednich, to twierdzenie zachodzi. Załóżmy więc, że istnieje krawędź między pewnymi dwoma. Wtedy jednak tworzą one wraz z v graf pełny, a więc również w tym przypadku twierdzenie jest prawdziwe. □

Problem rozważany w tym punkcie jest związany z **liczbami Ramseya** $R(m, n)$ [13]; powyższe twierdzenie pokazuje, że $R(3, 3) \leqslant 6$.

(c) Twierdzenie. *Zbiór wierzchołków dowolnego grafu nieskierowanego $G = \langle V, E\rangle$ można podzielić na dwa rozłączne zbiory tak, aby dla dowolnego wierzchołka $v \in V$ co najmniej połowa jego sąsiadów nie należała do zbioru, do którego należy v.*

Dowód. Przypiszmy każdemu wierzchołkowi $v \in V$ wagę $d(v)$ równą liczbie jego sąsiadów spoza jego zbioru pomniejszoną o liczbę sąsiadów ze zbioru, do którego v należy. Dowód sprowadza się do pokazania, że istnieje taki podział zbioru wierzchołków, że $d(v) \geqslant 0$ dla każdego $v \in V$.

Zdefiniujmy $\sigma = \sum_{v \in V} d(v)$ i zastanówmy się, jak tę wartość można zmaksymalizować, wyznaczając podziały V na podzbiory V_1 i V_2. Załóżmy, że dokonaliśmy już pewnego takiego podziału i wybierzmy pewien wierzchołek v należący do zbioru V_1. Dowód w przypadku gdy $v \in V_2$ przebiega symetrycznie. Zauważmy, że jeśli przenieślibyśmy wierzchołek v do V_2, to jego waga $d(v)$ zmieniłaby znak na przeciwny. Ponadto waga każdego sąsiada v należącego do zbioru V_1 wzrosłaby o 2, a waga każdego sąsiada v z V_2 zmalałaby o 2. W wyniku przeniesienia v wartość σ wzrosłaby o $-4d(v)$. Widać więc, że aby zwiększyć σ, należy przenosić wierzchołki o ujemnych wagach. Ponieważ σ nie może rosnąć w nieskończoność (jest ograniczone od góry przez $2|E|$ w grafach dwudzielnych), to po skończonej liczbie takich operacji wszystkie wierzchołki grafu G będą mieć wagi nieujemne, czego należało dowieść. □

(d) Twierdzenie (Dirac)**.** *Jeśli dla każdego wierzchołka v grafu nieskierowanego $G = \langle V, E\rangle$, w którym $|V| \geqslant 3$, zachodzi $\deg(v) \geqslant |V|/2$, to G jest hamiltonowski.*

Zanim zajmiemy się dowodem twierdzenia, udowodnimy następujący lemat.

Lemat (Ore)**.** *Jeśli w grafie nieskierowanym $G = \langle V, E\rangle$, gdzie $|V| \geqslant 3$, dla każdej pary niesąsiednich wierzchołków u i v zachodzi nierówność $\deg(u) + \deg(v) \geqslant |V|$, to G jest hamiltonowski.*

Dowód. Oznaczmy przez n liczbę wierzchołków grafu G. Przypuśćmy, że lemat jest fałszywy, czyli że dla pewnego n istnieje kontrprzykład – graf $G = \langle V, E\rangle$, w którym $|V| = n$ i który spełnia założenie lematu, ale nie jest hamiltonowski. Spośród wszystkich takich grafów rozpatrzmy ten, dla którego $|E|$ jest maksymalne. Wówczas G musi mieć ścieżkę Hamiltona $\langle v_1, v_2, \ldots, v_n\rangle$ – w przeciwnym przypadku moglibyśmy uzupełnić go o pewne brakujące krawędzie, nie naruszając warunku z założenia lematu i otrzymując w wyniku graf o więcej niż $|E|$ krawędziach. Ponieważ G nie ma cyklu Hamiltona, to nie istnieje krawędź łącząca v_1 z v_n. Z kolei z założenia wiemy, że $\deg(v_1) + \deg(v_n) \geqslant n$.

Można teraz zdefiniować podzbiory S_1 i S_n zbioru $\{2, 3, \ldots, n\}$ takie, że

$$S_1 = \big\{\, i : \{v_1, v_i\} \in E \,\big\} \quad \text{oraz} \quad S_n = \big\{\, i : \{v_{i-1}, v_n\} \in E \,\big\}.$$

Wtedy $|S_1| = \deg(v_1)$ i $|S_n| = \deg(v_n)$. Ponieważ $|S_1| + |S_n| \geqslant n$ i zbiór $S_1 \cup S_n$ ma co najwyżej $n-1$ elementów, to zbiór $S_1 \cap S_n$ musi być niepusty. Istnieje więc i, dla którego istnieją krawędzie $\{v_1, v_i\}$ oraz $\{v_{i-1}, v_n\}$. Stąd ścieżka $\langle v_1, \ldots, v_{i-1}, v_n, v_{n-1}, \ldots, v_i, v_1\rangle$ jest cyklem Hamiltona w grafie G. Sprzeczność – lemat jest prawdziwy. □

Można teraz udowodnić główne twierdzenie.

Dowód. Jeśli dla każdego $v \in V$ zachodzi $\deg(v) \geqslant |V|/2$, to $\deg(u) + \deg(v) \geqslant |V|$ dla każdych u, $v \in V$ niezależnie od tego, czy są sąsiednie, czy nie, a więc G spełnia założenia powyższego lematu, czyli jest hamiltonowski. □

B-3. Podziały drzew

(a) Niech $T = \langle V, E\rangle$ będzie drzewem binarnym, w którym $|V| = n \geqslant 2$. Przez **krawędź dzielącą** będziemy rozumieć krawędź, po usunięciu której zbiór wierzchołków drzewa T dzieli się na zbiory A i B takie, że $|A| \leqslant 3n/4$ oraz $|B| \leqslant 3n/4$. Udowodnimy przez indukcję względem n, że w każdym takim drzewie istnieje krawędź dzieląca.

Dla $n = 2$ twierdzenie zachodzi, ponieważ w drzewie T istnieje tylko jedna krawędź, po usunięciu której dostajemy zbiory jednoelementowe. Niech zatem $n > 2$ i załóżmy, że w drzewie o $n-1$ wierzchołkach istnieje krawędź dzieląca $e \in E$ taka, że po podziale każdy ze zbiorów A i B ma co najwyżej $3(n-1)/4$ elementów. Przyjmijmy bez utraty ogólności, że $|A| \leqslant |B|$, co oznacza, że $|A| \leqslant (n-1)/2$. Utwórzmy teraz nowe drzewo T', dodając do V nowy wierzchołek v' oraz nową krawędź $\{v', v\}$ do E dla pewnego $v \in V$. Niech teraz A' oraz B' będą zbiorami wierzchołków w nowym drzewie utworzonymi w wyniku podziału krawędzią e. Jeśli $v \in A$, to $A' = A \cup \{v'\}$ oraz $B' = B$. Oczywiście $|B'| < 3n/4$, zbadajmy zatem A':

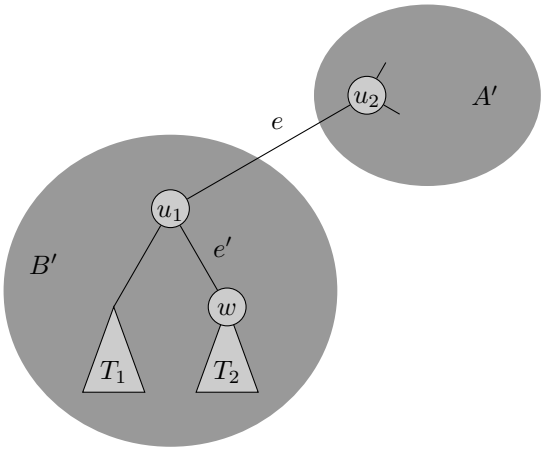
$$|A'| = |A| + 1 \leqslant \frac{n-1}{2} + 1 = \frac{n+1}{2} \leqslant \frac{3n}{4},$$

co jest prawdą, o ile $n \geqslant 2$, zatem w tym przypadku twierdzenie zachodzi.

Niech teraz $v \in B$. Stąd $A' = A$ i $B' = B \cup \{v'\}$, ale z założenia $|B'| = |B| + 1 \leqslant (3n+1)/4$, a zatem $|B'|$ może przekroczyć $3n/4$, co oznacza, że musimy znaleźć inną krawędź dzielącą dla drzewa T' w przypadku, gdy $|B| = (3n-3)/4$, przy czym $n \geqslant 5$.

Rozważmy drzewo T' przedstawione na rys. 39. Niech $u_1 \in B$ oraz $e = \{u_1, u_2\}$. Oprócz u_1 do zbioru B należą wierzchołki ze zbiorów V_1 i V_2, a do zbioru A – wierzchołek u_2 oraz wierzchołki ze zbiorów V_3 i V_4. Załóżmy bez straty ogólności, że $|V_1| \leqslant |V_2|$. Zbiór V_2 jest niepusty, gdyż $|B'| \geqslant 4$, istnieje zatem krawędź $e' = \{u_1, w\}$, gdzie $w \in V_2$. Pokażemy, że jest to krawędź dzieląca drzewa T'. Rozważmy w tym celu zbiory A'' i B'', na które krawędź e' dzieli zbiór $V \cup \{v'\}$. Mamy

$$|B''| = |V_2| \leqslant |B| = \frac{3n-3}{4} < \frac{3n}{4}$$



Rysunek 39: Drzewo T' z drugiego przypadku dowodu.

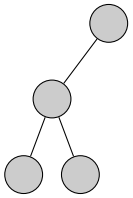
oraz

$$|A''| = \left|A \cup \{u_1\} \cup V_1\right| \leqslant (n-1-|B|)+1+\frac{|B|}{2} = n-\frac{|B|}{2} = \frac{5n+4}{8}.$$

Skorzystaliśmy z tego, że $|A|+|B| = n-1$ oraz $|V_1| \leqslant |B|/2$. Nierówność $|A''| \leqslant 3n/4$ zachodzi, o ile $n \geqslant 4$, więc istotnie e' jest krawędzią dzielącą drzewa T'.

Rozpatrzyliśmy wszystkie przypadki, zatem na mocy indukcji twierdzenie zachodzi dla każdego drzewa binarnego T.

(b) Stała 3/4 jest wystarczająca do dokonywania zrównoważonych podziałów, jak to wykazaliśmy w punkcie (a). Przykład drzewa binarnego z rys. 40 pokazuje, że nie można przyjąć na jej miejsce mniejszej wartości. Usuwając dowolną krawędź tego drzewa, dzielimy zbiór jego wierzchołków na podzbiory, z których jeden ma trzy elementy.



Rysunek 40: Drzewo binarne, w którym najbardziej zrównoważony podział tworzy podzbiór zawierający 3 wierzchołki.

(c) Rozważmy następującą procedurę podziału zbioru wierzchołków. Na początku przyjmujemy, że wynikowe zbiory A i B są puste. Usuwając jedną krawędź, możemy podzielić n-elementowy zbiór wierzchołków drzewa na dwa podzbiory, z których większy będzie składać się z co najwyżej $3n/4$ wierzchołków, co wynika na podstawie punktu (a). Mniejszy podzbiór sumujemy z jednym ze zbiorów wynikowych, natomiast większy z nich będzie podlegał dalszemu podziałowi. Podczas działania procedury pilnujemy, aby rozmiary zbiorów A i B nie przekroczyły $\lceil n/2 \rceil$. Procedurę podziału zakończymy w momencie, gdy jeden z tych zbiorów będzie zawierał $\lceil n/2 \rceil$ elementów, gdyż drugi zbiór zawiera wtedy $\lfloor n/2 \rfloor$ elementów.

Zauważmy, że maksymalną liczbę podziałów dla zadanego drzewa wykonamy w przypadku, gdy po każdym kroku zostanie do podziału zbiór o rozmiarze $3/4$ rozmiaru zbioru z poprzedniego kroku. Niech k będzie taką maksymalną liczbą podziałów drzewa o n wierzchołkach. Zachodzi wtedy $(3/4)^k n = 1$, ponieważ zbioru jednoelementowego nie trzeba już dalej dzielić. Stąd mamy $k = \log_{4/3} n$, a zatem należy usunąć co najwyżej $k = O(\lg n)$ krawędzi.

Dodatek **C**

Zliczanie i prawdopodobieństwo

C.1. Zliczanie

C.1-1. Załóżmy, że $k = 1, 2, \ldots, n$, to znaczy nie rozważamy podsłów pustych. Pierwsze k-podsłowo zajmuje w n-słowie pozycje $1, 2, \ldots, k$, drugie – $2, 3, \ldots, k+1$ itd. Ostatnie k-podsłowo leży na pozycjach $n-k+1, n-k+2, \ldots, n$. Istnieje zatem $n-k+1$ wszystkich k-podsłów n-słowa.

By obliczyć łączną ilość podsłów n-słowa, należy zsumować liczby k-podsłów po wszystkich $k = 1, 2, \ldots, n$, co daje

$$\sum_{k=1}^{n}(n-k+1) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

C.1-2. Niech $X = \{0, 1, \ldots, 2^n - 1\}$ i $Y = \{0, 1, \ldots, 2^m - 1\}$ będą zbiorami liczb całkowitych, odpowiednio, n-bitowych i m-bitowych. Funkcji logicznych o n wejściach i m wyjściach jest tyle samo, co funkcji $f\colon X \to Y$. Zagadnienie sprowadza się zatem do pytania o liczbę wszystkich ciągów $\langle y_1, \ldots, y_{2^n}\rangle$ o wyrazach ze zbioru Y. Każdy wyraz możemy wybrać na 2^m sposobów, co daje $(2^m)^{2^n} = 2^{m2^n}$ możliwości wyboru ciągu $\langle y_1, \ldots, y_{2^n}\rangle$. Jest zatem 2^{m2^n} funkcji logicznych o n wejściach i m wyjściach, a stąd 2^{2^n} funkcji logicznych o n wejściach i 1 wyjściu.

C.1-3. Niech S_n oznacza szukaną liczbę sposobów ustawienia n osób przy okrągłym stole. Każdy sposób jest nierozróżnialny z $n-1$ innymi, które powstają przez przesunięcie wszystkich osób o pewną ilość miejsc bez zmiany ich kolejności. Istnieje $n!$ możliwych permutacji n osób, zatem nS_n jest równe $n!$. Mamy zatem

$$S_n = \frac{n!}{n} = (n-1)!.$$

C.1-4. Aby wybrać trzy liczby ze zbioru $\{1, 2, \ldots, 100\}$ sumujące się do liczby parzystej, można postąpić według jednej z dwóch strategii – wybrać trzy liczby parzyste albo wybrać dwie liczby nieparzyste i jedną liczbę parzystą. W pierwszym przypadku możemy dokonać wyboru na $\binom{50}{3}$ sposobów, a w drugim – na $\binom{50}{2}\binom{50}{1}$ sposobów. Łączna liczba możliwości wyboru liczb wynosi zatem

$$\binom{50}{3} + \binom{50}{2}\binom{50}{1} = 80850.$$

C.1-5.

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} = \frac{n}{k} \cdot \frac{(n-1)!}{(k-1)!\,(n-k)!} = \frac{n}{k}\binom{n-1}{k-1}$$

C.1-6.

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} = \frac{n}{n-k} \cdot \frac{(n-1)!}{k!\,(n-k-1)!} = \frac{n}{n-k}\binom{n-1}{k}$$

C.1-7. Niech S będzie zbiorem n-elementowym oraz niech $s \in S$. Ze zbioru S można wybrać $\binom{n}{k}$ k-podzbiorów. Podzbiory te możemy podzielić na takie, które nie zawierają s i na takie, które zawierają s. Łatwo zauważyć, że tych pierwszych jest $\binom{n-1}{k}$, a drugich $\binom{n-1}{k-1}$. Stąd wnioskujemy, że

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

C.1-8. Kilka początkowych wierszy trójkąta Pascala:

$$\begin{array}{ccccccccccccc}
 & & & & & & 1 & & & & & & \\
 & & & & & 1 & & 1 & & & & & \\
 & & & & 1 & & 2 & & 1 & & & & \\
 & & & 1 & & 3 & & 3 & & 1 & & & \\
 & & 1 & & 4 & & 6 & & 4 & & 1 & & \\
 & 1 & & 5 & & 10 & & 10 & & 5 & & 1 & \\
1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1
\end{array}$$

Pierwszy wiersz składa się z elementu $\binom{0}{0} = 1$. Na krańcach kolejnych wierszy występują jedynki, a pozostałe elementy obliczane są z tożsamości z zad. C.1-7, czyli poprzez zsumowanie liczb występujących bezpośrednio nad wyznaczanym elementem.

C.1-9. Z definicji współczynnika dwumianowego i z tożsamości (A.1) mamy

$$\binom{n+1}{2} = \frac{(n+1)!}{2!\,(n-1)!} = \frac{n(n+1)}{2} = \sum_{i=1}^{n} i.$$

C.1-10. Potraktujmy współczynnik dwumianowy jak funkcję $b_n(k) = \binom{n}{k}$ dla $k = 0, 1, \ldots, n$ i sprawdźmy, dla jakich k wartość $b_n(k)$ jest największa. Badamy stosunek

$$\frac{b_n(k+1)}{b_n(k)} = \frac{\binom{n}{k+1}}{\binom{n}{k}} = \frac{n!}{(k+1)!\,(n-k-1)!} \cdot \frac{k!\,(n-k)!}{n!} = \frac{n-k}{k+1}.$$

Jeśli $n-k \geqslant k+1$, czyli $k \leqslant (n-1)/2$, to funkcja b_n jest niemalejąca. Z kolei gdy $k \geqslant (n-1)/2$, to b_n jest nierosnąca. A zatem gdy n jest nieparzyste, to b_n przyjmuje największą wartość dla $k = (n-1)/2 = \lfloor n/2 \rfloor$. Ponadto ze wzoru (C.3) mamy

$$b_n((n-1)/2) = \binom{n}{(n-1)/2} = \binom{n}{n-(n-1)/2} = \binom{n}{(n+1)/2} = b_n((n+1)/2),$$

a więc wartość największa jest przyjmowana również dla $k = (n+1)/2 = \lceil n/2 \rceil$.

W przypadku gdy n jest liczbą parzystą, maksymalna wartość funkcji b_n jest osiągana dla $k = n/2$ lub $k = n/2 - 1$. Sprawdźmy, która wartość jest większa:

$$\frac{b_n(n/2)}{b_n(n/2-1)} = \frac{\binom{n}{n/2}}{\binom{n}{n/2-1}} = \frac{n!}{(n/2)!\,(n/2)!} \cdot \frac{(n/2-1)!\,(n/2+1)!}{n!} = \frac{n/2+1}{n/2} > 1.$$

Mamy więc, że b_n przyjmuje maksimum dla $k = n/2 = \lfloor n/2 \rfloor = \lceil n/2 \rceil$.

C.1-11. Dla liczb naturalnych n, j, k takich, że $j + k \leqslant n$, sprawdzamy zachowanie stosunku

$$\frac{\binom{n}{j+k}}{\binom{n}{j}\binom{n-j}{k}} = \frac{n!}{(j+k)!\,(n-j-k)!} \cdot \frac{j!\,(n-j)!}{n!} \cdot \frac{k!\,(n-j-k)!}{(n-j)!} = \frac{j!\,k!}{(j+k)!} = \frac{\prod_{i=1}^{k} i}{\prod_{i=1}^{k}(j+i)}.$$

Oczywistym jest, że iloczyn kolejnych liczb całkowitych od 1 do k nie przekracza iloczynu kolejnych liczb całkowitych od $j + 1$ do $j + k$, zatem badany stosunek nie przekracza 1. Równość zachodzi tylko w przypadku, gdy $j = 0$ lub $k = 0$.

Lewą stronę nierówności (C.9) można zinterpretować jako liczbę możliwych sposobów wyboru $j+k$ przedmiotów ze zbioru n-elementowego, prawą zaś jako liczbę możliwości wyboru najpierw j przedmiotów spośród n, a następnie k przedmiotów spośród $n-j$ pozostawionych po pierwszym wyborze. W obu strategiach wyznaczony zostaje zbiór A złożony z $j+k$ przedmiotów. Jest tylko jeden sposób wybrania zadanego zbioru A przy pierwszej strategii i o wiele więcej, jeśli zastosuje się drugie podejście. Można bowiem dowolnie podzielić elementy z A na j takich, które będą wybierane w pierwszym kroku i k pozostałych, które wybierzemy w drugim kroku.

C.1-12. Łatwo sprawdzić, że dla $k = 0$ nierówność zachodzi. Przyjmijmy zatem, że $k \geqslant 1$ i załóżmy, że

$$\binom{n}{k-1} \leqslant \frac{n^n}{(k-1)^{k-1}(n-k+1)^{n-k+1}}.$$

Mamy teraz z zad. C.1-5 i zad. C.1-6 oraz z założenia indukcyjnego:

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} = \frac{n-k+1}{k}\binom{n}{k-1} \leqslant \frac{n^n}{k(k-1)^{k-1}(n-k+1)^{n-k}}.$$

Wystarczy wykazać nierówność

$$\frac{n^n}{k(k-1)^{k-1}(n-k+1)^{n-k}} \leqslant \frac{n^n}{k^k(n-k)^{n-k}},$$

co zrobimy poprzez sprawdzenie ilorazu

$$\frac{\frac{n^n}{k(k-1)^{k-1}(n-k+1)^{n-k}}}{\frac{n^n}{k^k(n-k)^{n-k}}} = \frac{k^{k-1}(n-k)^{n-k}}{(k-1)^{k-1}(n-k+1)^{n-k}} = \frac{\left(\frac{k}{k-1}\right)^{k-1}}{\left(\frac{n-k+1}{n-k}\right)^{n-k}} = \frac{\left(1+\frac{1}{k-1}\right)^{k-1}}{\left(1+\frac{1}{n-k}\right)^{n-k}}.$$

Ciąg $e_n = (1 + 1/n)^n$ jest rosnący, skąd dostajemy, że powyższy iloraz nie przekracza 1, o ile $k - 1 \leqslant n - k$, czyli $k \leqslant (n + 1)/2$, a więc tym bardziej, gdy $k \leqslant n/2$. Pokazaliśmy tym samym, że nierówność (C.6) jest spełniona dla $k \leqslant n/2$.

Ze wzoru (C.3) mamy $\binom{n}{k} = \binom{n}{n-k}$. Gdy $n/2 < k \leqslant n$, to $0 \leqslant n - k < n/2$ i dowód sprowadza się do pokazania, że

$$\binom{n}{n-k} \leqslant \frac{n^n}{k^k(n-k)^{n-k}}.$$

Można to zrobić, wykorzystując rozumowanie z poprzedniego paragrafu po zamianie zmiennej k na wyrażenie $n - k$.

C.1-13. Wykorzystując wzór Stirlinga, mamy

$$\binom{2n}{n} = \frac{(2n)!}{(n!)^2} = \frac{\sqrt{4\pi n}\left(\frac{2n}{e}\right)^{2n}(1+\Theta(1/n))}{2\pi n\left(\frac{n}{e}\right)^{2n}(1+\Theta(1/n))^2} = \frac{2^{2n}}{\sqrt{\pi n}} \cdot \frac{1+\Theta(1/n)}{(1+\Theta(1/n))^2}.$$

Zajmiemy się teraz ostatnim ułamkiem, którego celowo nie skracaliśmy, ponieważ funkcja w mianowniku nie musi być równa kwadratowi funkcji z licznika. Niech c, d będą stałymi takimi, że $c \geqslant d > 0$ i funkcja $1 + c/n$ ogranicza licznik ułamka od góry, a funkcja $1 + d/n$ – mianownik ułamka od dołu. Mamy

$$\frac{1+\Theta(1/n)}{(1+\Theta(1/n))^2} \leqslant \frac{1+c/n}{(1+d/n)^2} < \frac{1+c/n}{1+d/n} = \frac{n+c}{n+d} = 1 + \frac{c-d}{n+d} = 1 + O(1/n).$$

Otrzymujemy ostatecznie

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}\,(1 + O(1/n)).$$

C.1-14. Obliczamy pierwszą pochodną funkcji entropii H:

$$\begin{aligned}\frac{dH}{d\lambda}(\lambda) &= -\left(\lg\lambda + \lambda \cdot \frac{1}{\lambda \ln 2}\right) - \left(-\lg(1-\lambda) + (1-\lambda)\cdot\frac{-1}{(1-\lambda)\ln 2}\right)\\ &= -\lg\lambda - \lg e + \lg(1-\lambda) + \lg e\\ &= \lg\frac{1-\lambda}{\lambda}.\end{aligned}$$

Powyższe wyrażenie przyjmuje wartość 0, gdy $\lambda = 1/2$. Należy jeszcze zbadać drugą pochodną:

$$\frac{d^2H}{d\lambda^2}(\lambda) = \frac{\lambda}{(1-\lambda)\ln 2} \cdot \frac{-\lambda - (1-\lambda)}{\lambda^2} = -\frac{\lg e}{\lambda(1-\lambda)}.$$

W punkcie $\lambda = 1/2$ druga pochodna jest ujemna, czyli binarna funkcja entropii H osiąga maksimum wynoszące $H(1/2) = 1$.

C.1-15. Dla $n = 0$ równość jest trywialna, przyjmijmy więc, że $n \geqslant 1$. Wówczas

$$\sum_{k=1}^{n}\binom{n}{k}k = \sum_{k=1}^{n}\binom{n-1}{k-1}n = n\sum_{k=0}^{n-1}\binom{n-1}{k} = n2^{n-1}.$$

Skorzystaliśmy ze wzoru (C.8), a następnie z (C.4) dla $x = y = 1$.

C.2. Prawdopodobieństwo

C.2-1. Utwórzmy skończoną lub przeliczalną rodzinę zdarzeń $\{C_1, C_2, \dots\}$, gdzie

$$C_i = A_i \setminus \bigcup_{j=1}^{i-1} A_j$$

dla każdego $i = 1, 2, \ldots$. Korzystając z faktu, że $\bigcup_i A_i = \bigcup_i C_i$ oraz z tego, że każde dwa zdarzenia z rodziny $\{C_1, C_2, \ldots\}$ wzajemnie się wykluczają, otrzymujemy

$$\Pr\left(\bigcup_i A_i\right) = \Pr\left(\bigcup_i C_i\right) = \sum_i \Pr(C_i).$$

Ponieważ $\Pr(C_i) \leqslant \Pr(A_i)$ dla każdego $i = 1, 2, \ldots$, to prawdą jest, że

$$\Pr\left(\bigcup_i A_i\right) \leqslant \sum_i \Pr(A_i).$$

C.2-2. Zdefiniujmy 3-słowo nad alfabetem $\{\text{O}, \text{R}\}$ w następujący sposób. Pierwszy symbol tego słowa będzie oznaczać wynik rzutu monetą profesora Rosencrantza, drugi symbol – wynik rzutu pierwszą monetą profesora Guildensterna, a trzeci – wynik rzutu jego drugą monetą, przy czym O oznacza uzyskanie orła, a R – uzyskanie reszki. Tworzymy przestrzeń zdarzeń elementarnych:

$$S = \{\text{OOO}, \text{OOR}, \text{ORO}, \text{ORR}, \text{ROO}, \text{ROR}, \text{RRO}, \text{RRR}\}.$$

Każde ze zdarzeń z S zachodzi z prawdopodobieństwem równym $1/8$, w szczególności zdarzenie ORR oznaczające wyrzucenie przez profesora Rosencrantza większej ilości orłów od przeciwnika.

C.2-3. Oznaczmy zdarzenia:

A – numer drugiej karty jest większy od numeru pierwszej karty,

B – numer trzeciej karty jest większy od numeru drugiej karty.

Jeśli numer drugiej wybranej karty wynosi k, to liczba zdarzeń sprzyjających A wynosi $k - 1$, a liczba zdarzeń sprzyjających B wynosi $10 - k$. Mamy obliczyć $\Pr(A \cap B)$. Korzystając z reguły iloczynu, dostajemy, że liczba zdarzeń sprzyjających $A \cap B$ wynosi

$$\sum_{k=1}^{10} (k-1)(10-k) = 120.$$

Liczbą możliwych trójek kart wybranych spośród dziesięciu jest $10!/7! = 720$ (liczba wszystkich 3-permutacji zbioru 10-elementowego). Dostajemy zatem $\Pr(A \cap B) = 120/720 = 1/6$.

C.2-4. Ponieważ $0 < a < b$, to $0 < a/b < 1$, więc część ułamkowa rozwinięcia binarnego ilorazu a/b jest ciągiem zer i jedynek $\beta_1, \beta_2, \ldots$ zawierającym co najmniej jedno zero i co najmniej jedną jedynkę oraz spełniającym równość

$$\frac{a}{b} = \sum_{i=1}^{\infty} \frac{\beta_i}{2^i}.$$

Będziemy rzucać monetą i tworzyć nowy ciąg zer i jedynek $\alpha_1, \alpha_2, \ldots$, w zależności od wyniku i-tego rzutu, dla orła przyjmując $\alpha_i = 0$, a dla reszki $\alpha_i = 1$. Rzuty wykonujemy, dopóki zachodzi $\alpha_i = \beta_i$. Natychmiast po wystąpieniu pierwszej różnicy kończymy proces i zwracamy wynik ostatnio wykonanego rzutu. Dokładniej, jeśli k jest najmniejszym indeksem, dla którego $\alpha_k \neq \beta_k$, to zwrócimy orła, jeżeli $\alpha_k = 0$, a reszkę w przeciwnym przypadku.

Sprawdzimy teraz, ile wynosi prawdopodobieństwo zwrócenia orła. Wynik każdego rzutu monetą nie zależy od wyników pozostałych rzutów, zatem szanse uzyskania dokładnie $k - 1$

rezultatów zgodnych z kolejnymi bitami części ułamkowej a/b, po czym jednej niezgodności, są równe $1/2^{k-1} \cdot 1/2 = 1/2^k$. Stąd mamy, że orzeł zostanie zwrócony z prawdopodobieństwem równym

$$\sum_{\substack{k=1,2,\ldots \\ \beta_k=1}} \frac{1}{2^k} = \sum_{\substack{k=1,2,\ldots \\ \beta_k=0}} \frac{0}{2^k} + \sum_{\substack{k=1,2,\ldots \\ \beta_k=1}} \frac{1}{2^k} = \sum_{k=1}^{\infty} \frac{\beta_k}{2^k} = \frac{a}{b}.$$

Oczekiwana liczba rzutów monetą potrzebnych do wyznaczenia wyniku jest oczekiwaną liczbą rzutów, aż do uzyskania pierwszej niezgodności między wynikiem rzutu a odpowiednim bitem rozwinięcia binarnego części ułamkowej liczby a/b. Jeśli przyjmiemy, że sukcesem jest wynik rzutu monetą niezgodny z bieżącym bitem rozwinięcia, to liczba potrzebnych rzutów n zanim pojawi się pierwszy sukces jest zmienną losową X o rozkładzie geometrycznym. Prawdopodobieństwo sukcesu wynosi $p = 1/2$, więc ze wzoru (C.31) otrzymujemy $\mathrm{E}(X) = 1/p = 2$. Widać zatem, że oczekiwana liczba rzutów monetą w opisanej procedurze jest stała.

C.2-5. Korzystając z obserwacji, że $(A \cap B) \cup (\overline{A} \cap B) = B$ oraz z tego, że zdarzenia $A \cap B$ i $\overline{A} \cap B$ wzajemnie się wykluczają, otrzymujemy

$$\Pr(A \mid B) + \Pr(\overline{A} \mid B) = \frac{\Pr(A \cap B)}{\Pr(B)} + \frac{\Pr(\overline{A} \cap B)}{\Pr(B)} = \frac{\Pr(B)}{\Pr(B)} = 1.$$

C.2-6. Dowodzimy przez indukcję względem liczby zdarzeń n. Dla $n = 1$ dowód jest trywialny, załóżmy zatem, że $n \geqslant 2$. Otrzymujemy

$$\begin{aligned}
\Pr\left(\bigcap_{i=1}^{n} A_i\right) &= \Pr\left(A_n \cap \bigcap_{i=1}^{n-1} A_i\right) \\
&= \Pr\left(\bigcap_{i=1}^{n-1} A_i\right) \Pr\left(A_n \;\middle|\; \bigcap_{i=1}^{n-1} A_i\right) \\
&= \Pr(A_1) \Pr(A_2 \mid A_1) \Pr(A_3 \mid A_1 \cap A_2) \ldots \Pr\left(A_n \;\middle|\; \bigcap_{i=1}^{n-1} A_i\right).
\end{aligned}$$

Druga równość wynika z definicji prawdopodobieństwa warunkowego, a trzecia – z założenia indukcyjnego.

C.2-7. Niech $n \geqslant 3$ i niech $S = \{s_1, s_2, \ldots, s_{n^2}\}$ będzie przestrzenią zdarzeń, przy czym $\Pr(s_i) = 1/n^2$ dla każdego $i = 1, 2, \ldots, n^2$. Pokażemy teraz, jak skonstruować zdarzenia $A_1, A_2, \ldots, A_n \subseteq S$, które spełniają warunek z treści zadania.

Będziemy dążyć do tego, by dla każdych indeksów $1 \leqslant i_1, i_2, i_3 \leqslant n$, zachodziło $|A_{i_1} \cap A_{i_2}| = 1$ oraz $|A_{i_1} \cap A_{i_2} \cap A_{i_3}| = 0$. Jest dokładnie $n(n-1)/2$ przecięć $A_i \cap A_j$, gdzie $1 \leqslant i < j \leqslant n$. Wybierzmy zatem tyle samo zdarzeń elementarnych i umieśćmy po jednym w każdym takim przecięciu, tzn. dane zdarzenie elementarne będzie wchodzić w skład obu zdarzeń tworzących dane przecięcie. Każde zdarzenie A_i składa się teraz z $n-1$ zdarzeń elementarnych i spełnione są powyższe warunki dotyczące rozmiarów przecięć. Spośród pozostałych $n(n+1)/2$ zdarzeń elementarnych wybieramy jeszcze n i umieszczamy po jednym w każdym zdarzeniu A_i.

Z opisanej konstrukcji zdarzeń $A_1, A_2, \ldots, A_n$ wynika, że $|A_i| = n$, czyli $\Pr(A_i) = 1/n$ dla $i = 1, 2, \ldots, n$, a zatem

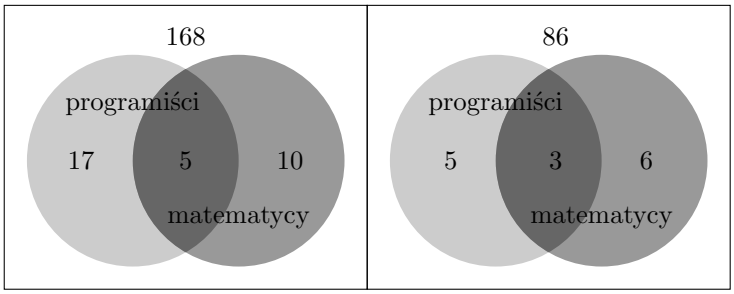
$$\Pr(A_{i_1} \cap A_{i_2}) = \frac{1}{n^2} = \Pr(A_{i_1}) \Pr(A_{i_2})$$

dla wszystkich $1 \leqslant i_1 < i_2 \leqslant n$ oraz

$$\Pr\left(\bigcap_{j=1}^{k} A_{i_j}\right) = 0 \neq \prod_{j=1}^{k} \Pr(A_{i_j})$$

dla każdego $k > 2$ i dowolnych, parami różnych indeksów $1 \leqslant i_1, \ldots, i_k \leqslant n$. Zdarzenia A_1, A_2, $\ldots$, A_n są więc parami niezależne, ale żaden ich k-podzbiór, gdzie $k > 2$, nie jest wzajemnie niezależny.

C.2-8. Rozważmy pewną grupę 300 osób, z których 100 jest w wieku powyżej 50 lat. Wśród młodszych osób znajduje się 22 programistów i 15 matematyków, przy czym tylko 5 osób jednocześnie programuje i zajmuje się matematyką. W skład grupy seniorów wchodzi 8 programistów oraz 9 matematyków, a 3 jednocześnie jest programistami i matematykami. Rys. 41 stanowi ilustrację przedstawionego opisu.



Rysunek 41: Diagramy Venna ilustrujące rozważaną grupę ludzi. Liczby w poszczególnych obszarach oznaczają liczby osób o danej profesji w danej grupie wiekowej.

Wybieramy losowo jedną osobę z całej grupy. Oznaczmy następujące zdarzenia:

A – wybrano osobę w wieku powyżej 50 lat,

B – wybrano programistę,

C – wybrano matematyka.

Ich prawdopodobieństwa wynoszą:

$$\Pr(A) = \frac{1}{3}, \quad \Pr(B) = \frac{1}{10}, \quad \Pr(C) = \frac{2}{25}.$$

Zauważmy, że zdarzenia A i B nie są niezależne, ponieważ

$$\Pr(A \cap B) = \frac{2}{75} \neq \frac{1}{30} = \Pr(A)\Pr(B).$$

Są jednak warunkowo zależne od zdarzenia C:

$$\Pr(A \cap B \mid C) = \frac{1}{8} = \frac{3}{8} \cdot \frac{1}{3} = \Pr(A \mid C)\Pr(B \mid C).$$

C.2-9. Rozważmy zdarzenie A oznaczające, że wybraliśmy zasłonę, za którą znajduje się nagroda. Oczywiście $\Pr(A) = 1/3$ i ze wzoru Bayesa (C.17) wynika, że prawdopodobieństwo wygranej W wynosi

$$\Pr(W) = \Pr(W \mid A)\Pr(A) + \Pr(W \mid \overline{A})\Pr(\overline{A}).$$

Obliczmy wartość powyższego prawdopodobieństwa w zależności od podjętej decyzji po podniesieniu przez prowadzącego jednej z zasłon. W pierwszej strategii decydujemy się na pozostanie przy aktualnym wyborze, więc mamy $\Pr(W \mid A) = 1$ i $\Pr(W \mid \overline{A}) = 0$ i wygrywamy z prawdopodobieństwem $\Pr(W) = 1/3$. Jeśli teraz rozważymy drugą strategię, to będziemy mieć $\Pr(W \mid A) = 0$ i $\Pr(W \mid \overline{A}) = 1$, ponieważ zamieniliśmy wybraną zasłonę na inną, dotychczas nieodkrytą. W tym przypadku prawdopodobieństwo wygranej wynosi $\Pr(W) = 2/3$. Widać zatem, że zmiana decyzji jest opłacalna, ponieważ podwaja szanse na wygraną.

C.2-10. Niech A_X, A_Y i A_Z będą zdarzeniami oznaczającymi wyjście na wolność, odpowiednio, więźnia X, Y i Z. Ponadto niech B będzie zdarzeniem polegającym na tym, że strażnik wskazał więźnia Y jako tego, który zostanie ścięty. Strażnik nie może zdradzić, który z więźniów będzie wolny, mamy więc $\Pr(B \mid A_X) = 1/2$, $\Pr(B \mid A_Y) = 0$ i $\Pr(B \mid A_Z) = 1$. Przed rozmową ze strażnikiem prawdopodobieństwo, że więzień X będzie wolny, wynosi $\Pr(A_X) = 1/3$. Jego sytuację po rozmowie opisuje zdarzenie $A_X \mid B$. Korzystając z obserwacji, że $B = (B \cap A_X) \cup (B \cap A_Y) \cup (B \cap A_Z)$ i ze wzoru Bayesa (C.17), mamy

$$\begin{aligned}
\Pr(A_X \mid B) &= \frac{\Pr(A_X)\Pr(B \mid A_X)}{\Pr(A_X)\Pr(B \mid A_X) + \Pr(A_Y)\Pr(B \mid A_Y) + \Pr(A_Z)\Pr(B \mid A_Z)} \\
&= \frac{\frac{1}{3} \cdot \frac{1}{2}}{\frac{1}{3} \cdot \frac{1}{2} + \frac{1}{3} \cdot 0 + \frac{1}{3} \cdot 1} \\
&= \frac{1}{3}.
\end{aligned}$$

A zatem informacja, jaką uzyskał od strażnika więzień X, nie zmienia jego szans na wyjście na wolność, które nadal wynoszą $1/3$.

C.3. Dyskretne zmienne losowe

C.3-1. Niech X będzie zmienną losową oznaczającą sumę oczek na obu kostkach. Mamy

$$\begin{aligned}
\mathrm{E}(X) &= \sum_{x=2}^{12} x \Pr(X = x) \\
&= 2 \cdot \frac{1}{36} + 3 \cdot \frac{2}{36} + 4 \cdot \frac{3}{36} + 5 \cdot \frac{4}{36} + 6 \cdot \frac{5}{36} + 7 \cdot \frac{6}{36} \\
&\quad + 8 \cdot \frac{5}{36} + 9 \cdot \frac{4}{36} + 10 \cdot \frac{3}{36} + 11 \cdot \frac{2}{36} + 12 \cdot \frac{1}{36} \\
&= 7.
\end{aligned}$$

Niech Y będzie zmienną losową oznaczającą większą z liczb oczek na obu kostkach. Zachodzi

$$\mathrm{E}(Y) = \sum_{y=1}^{6} y \Pr(Y = y) = 1 \cdot \frac{1}{36} + 2 \cdot \frac{3}{36} + 3 \cdot \frac{5}{36} + 4 \cdot \frac{7}{36} + 5 \cdot \frac{9}{36} + 6 \cdot \frac{11}{36} \approx 4{,}47.$$

C.3-2. Niech X będzie zmienną losową przyjmującą wartość indeksu największego elementu tablicy A. Jeśli w tej tablicy znajduje się n liczb, to $\Pr(X = i) = 1/n$ dla każdego $i = 1, 2, \ldots, n$, a zatem

$$\mathrm{E}(X) = \sum_{i=1}^{n} i \Pr(X = i) = \frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

W rzeczywistości wynik ten jest identyczny dla każdego elementu tablicy, w szczególności dla elementów największego i najmniejszego.

C.3-3. Zdefiniujmy zmienną losową X przyjmującą wielkość wygranej w opisanej grze. Mamy obliczyć

$$\mathrm{E}(X) = -\Pr(A_0) + \Pr(A_1) + 2\Pr(A_2) + 3\Pr(A_3),$$

przy czym A_k, dla $k = 0, 1, 2, 3$, oznacza zdarzenie, że obstawiona przez gracza liczba oczek pojawiła się dokładnie na k kostkach. Prawdopodobieństwa tych zdarzeń wynoszą:

$$\begin{aligned}
\Pr(A_0) &= \frac{5^3}{6^3} = \frac{125}{216}, \\
\Pr(A_1) &= \frac{3 \cdot 5^2}{6^3} = \frac{75}{216}, \\
\Pr(A_2) &= \frac{3 \cdot 5^1}{6^3} = \frac{15}{216}, \\
\Pr(A_3) &= \frac{1}{6^3} = \frac{1}{216}.
\end{aligned}$$

Dostajemy zatem

$$\mathrm{E}(X) = -\frac{17}{216} \approx -0{,}08,$$

a więc gracz straci w tej grze średnio około 8 groszy.

C.3-4. Niech S będzie przestrzenią zdarzeń, w której określone są zmienne losowe X i Y. Załóżmy, że w pewnym podzbiorze zbioru S zachodzi $X \geqslant Y$. Ponieważ $Y \geqslant 0$, więc także $\mathrm{E}(Y) \geqslant 0$, skąd

$$\mathrm{E}(\max(X, Y)) = \mathrm{E}(X) \leqslant \mathrm{E}(X) + \mathrm{E}(Y).$$

Przypadek, gdy $Y \geqslant X$, dowodzi się analogicznie.

C.3-5. Zgodnie z definicją niezależnych zmiennych losowych X i Y mamy

$$\Pr(X = x \text{ i } Y = y) = \Pr(X = x) \Pr(Y = y).$$

Niech f i g będą dowolnymi funkcjami rzeczywistymi. Jeśli X przyjmuje wartość x, to zmienna losowa $f(X)$ przyjmuje wartość $f(x)$. Analogicznie dla Y: jeśli $Y = y$, to $g(Y) = g(y)$. Powyższy wzór możemy więc zapisać w postaci

$$\Pr\big(f(X) = f(x) \text{ i } g(Y) = g(y)\big) = \Pr\big(f(X) = f(x)\big) \Pr\big(g(Y) = g(y)\big),$$

na podstawie której wnioskujemy, że $f(X)$ i $g(Y)$ są niezależnymi zmiennymi losowymi.

C.3-6. Niech $t \in \mathbb{R}$. Wartość oczekiwaną $\mathrm{E}(X)$ zapisujemy w następujący sposób:

$$\mathrm{E}(X) = \sum_x x \Pr(X = x) = \sum_{x<t} x \Pr(X = x) + \sum_{x \geqslant t} x \Pr(X = x).$$

Ponieważ zmienna losowa X jest nieujemna, to można ograniczyć $E(X)$ od dołu, opuszczając pierwszą sumę w ostatnim wyrażeniu. Zachodzi więc

$$\mathrm{E}(X) \geqslant \sum_{x \geqslant t} x \Pr(X = x) \geqslant t \sum_{x \geqslant t} \Pr(X = x) = t \Pr(X \geqslant t).$$

C.3-7. Ustalmy $t \in \mathbb{R}$. Niech $A = \{\, s \in S : X(s) \geqslant t \,\}$ oraz $A' = \{\, s \in S : X'(s) \geqslant t \,\}$. Wprost z definicji zmiennej losowej mamy

$$\Pr(X \geqslant t) = \sum_{s \in A} \Pr(s) \quad \text{oraz} \quad \Pr(X' \geqslant t) = \sum_{s \in A'} \Pr(s).$$

Dowodzimy, że

$$\sum_{s \in A} \Pr(s) \geqslant \sum_{s \in A'} \Pr(s).$$

Z założenia wynika, że jeśli $X'(s) \geqslant t$, to $X(s) \geqslant t$, lub równoważnie, $A' \subseteq A$. A zatem suma po lewej stronie powyższej nierówności zawiera o $|A \setminus A'|$ więcej składników niż suma po prawej stronie. Wszystkie te składniki są liczbami nieujemnymi, więc nierówność jest spełniona.

C.3-8. Wariancja zmiennej losowej X jest liczbą nieujemną, a więc po skorzystaniu ze wzoru (C.26) otrzymujemy, że $\mathrm{E}(X^2) \geqslant \mathrm{E}^2(X)$.

C.3-9. Niech $\Pr(X = 0) = 1 - p$ oraz $\Pr(X = 1) = p$. Stąd

$$\mathrm{E}(X) = 0 \cdot (1 - p) + 1 \cdot p = p.$$

Zauważmy ponadto, że $\mathrm{E}(X^2) = \mathrm{E}(X)$. Wariancja zmiennej losowej X jest równa

$$\mathrm{Var}(X) = \mathrm{E}(X^2) - \mathrm{E}^2(X) = p(1 - p) = \mathrm{E}(X)(1 - \mathrm{E}(X)) = \mathrm{E}(X)\,\mathrm{E}(1 - X),$$

przy czym ostatnia równość zachodzi dzięki liniowości wartości oczekiwanej.

C.3-10. Wprost z definicji wariancji oraz ze wzoru (C.21):

$$\mathrm{Var}(aX) = \mathrm{E}(a^2X^2) - \mathrm{E}^2(aX) = a^2\,\mathrm{E}(X^2) - (a\,\mathrm{E}(X))^2 = a^2\big(\mathrm{E}(X^2) - \mathrm{E}^2(X)\big) = a^2\,\mathrm{Var}(X).$$

C.4. Rozkłady: geometryczny i dwumianowy

C.4-1. Rodzina zdarzeń elementarnych S składa się ze zdarzeń oznaczających, że należy wykonać k prób, zanim nastąpi pierwszy sukces, przy czym $k = 1, 2, \ldots$. Niech zmienna losowa X będzie liczbą prób potrzebnych do osiągnięcia sukcesu. Wówczas

$$\Pr(S) = \sum_{k=1}^{\infty} \Pr(X = k) = \sum_{k=1}^{\infty} q^{k-1}p = p \sum_{k=0}^{\infty} (1 - p)^k = \frac{p}{1 - (1 - p)} = 1.$$

Wykorzystano wzór (A.6) przy założeniu, że $p > 0$.

C.4-2. Niech sukcesem w tym doświadczeniu będzie uzyskanie trzech orłów i trzech reszek, a porażką – każdy inny wynik. Spośród wszystkich sześciu monet możemy wybrać dowolne trzy takie, na których wypadnie orzeł – mamy zatem $\binom{6}{3} = 20$ sposobów osiągnięcia sukcesu. Jest $2^6 = 64$ wszystkich możliwych wyników, zatem prawdopodobieństwo sukcesu wynosi $p = 20/64 = 5/16$. Ze wzoru (C.31) otrzymujemy, że musimy wykonać średnio $1/p = 3{,}2$ rzutów.

C.4-3. Z definicji rodziny rozkładów dwumianowych mamy

$$b(k;n,p) = \binom{n}{k} p^k (1-p)^{n-k} \quad \text{oraz} \quad b(n-k;n,q) = \binom{n}{n-k} q^{n-k}(1-q)^k.$$

Ponieważ $\binom{n}{n-k} = \binom{n}{k}$ (ze wzoru (C.3)) oraz $q = 1-p$, to zachodzi $b(k;n,p) = b(n-k;n,q)$.

C.4-4. Rozkład dwumianowy przyjmuje maksimum dla pewnego k, gdzie $np - q \leqslant k \leqslant (n+1)p$, więc dobrym przybliżeniem wartości maksymalnej jest wartość przyjmowana dla $k = np$. Ponieważ nie interesuje nas dokładny wynik, to pozwolimy sobie na pewną niedbałość w rachunkach (wartość np oraz argumenty silni mogą być niecałkowite). Mamy

$$b(np;n,p) = \binom{n}{np} p^{np}(1-p)^{n-np} = \frac{n!}{(np)!\,(n-np)!}\, p^{np}(1-p)^{n-np} = \frac{n!}{(np)!\,(nq)!}\, p^{np}q^{nq}.$$

Wykorzystując wzór Stirlinga do przybliżenia silni, możemy uprościć pierwszy czynnik ostatniego wyrażenia:

$$\frac{n!}{(np)!\,(nq)!} \approx \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\sqrt{2\pi np}\left(\frac{np}{e}\right)^{np}\sqrt{2\pi nq}\left(\frac{nq}{e}\right)^{nq}} = \frac{\left(\frac{n}{e}\right)^n\left(\frac{e}{np}\right)^{np}\left(\frac{e}{nq}\right)^{nq}}{\sqrt{2\pi npq}} = \frac{1}{p^{np}q^{nq}\sqrt{2\pi npq}}.$$

Stąd dostajemy przybliżenie

$$b(np;n,p) \approx \frac{1}{\sqrt{2\pi npq}}$$

na maksymalną wartość rozkładu dwumianowego $b(k;n,p)$.

C.4-5. Niech X będzie zmienną losową przyjmującą liczbę sukcesów w n próbach Bernoulliego. Wówczas

$$\Pr(X=0) = b(0;n,1/n) = \left(1 - \frac{1}{n}\right)^n.$$

Wyrażenie to dąży do $1/e$ wraz ze wzrostem n, ponieważ dla dowolnej liczby rzeczywistej a ciąg $e_n = (1 + a/n)^n$ ma granicę równą e^a przy n dążącym do ∞.

Analogicznie,

$$\Pr(X=1) = b(1;n,1/n) = \frac{\left(1-\frac{1}{n}\right)^n}{1-\frac{1}{n}},$$

co także dąży do $1/e$, bo mianownik zbliża się do 1 wraz ze wzrostem n.

C.4-6. Obliczymy dwoma sposobami prawdopodobieństwo uzyskania przez profesorów równej liczby orłów. W pierwszym z nich będziemy traktować wynik każdego doświadczenia jako $2n$-elementowy ciąg orłów i reszek taki, że jego początkowych n wyrazów oznacza wyniki uzyskane przez profesora Rosencrantza, a n końcowych – wyniki profesora Guildensterna. Wszystkich takich ciągów jest $2^{2n} = 4^n$. Niech sukcesem dla profesora Rosencrantza będzie uzyskanie orła,

a dla profesora Guildensterna – uzyskanie reszki. Zauważmy, że wyrzucenie równej liczby orłów przez obu profesorów jest równoważne z osiągnięciem przez nich w sumie n sukcesów. Liczba sposobów, na jakie można to zrobić, jest liczbą możliwości wyboru spośród $2n$ pozycji ciągu n odpowiedzialnych za sukces. Wartość ta wynosi $\binom{2n}{n}$, a zatem szukane prawdopodobieństwo jest równe

$$\frac{\binom{2n}{n}}{4^n}.$$

Drugi sposób polega na zdefiniowaniu X i Y jako zmiennych losowych przyjmujących liczby orłów uzyskane kolejno przez obu profesorów. Obie te zmienne są rozkładu dwumianowego $b(k; n, 1/2)$. Zdarzenia $X = k$ i $Y = k$ są niezależne, zatem prawdopodobieństwo uzyskania przez obu profesorów równej ilości orłów wynosi

$$\begin{aligned}\sum_{k=0}^{n} \Pr(X = k \text{ i } Y = k) &= \sum_{k=0}^{n} \Pr(X = k) \Pr(Y = k) \\ &= \sum_{k=0}^{n} \binom{n}{k}\left(\frac{1}{2}\right)^n \binom{n}{k}\left(\frac{1}{2}\right)^n \\ &= \frac{\sum_{k=0}^{n} \binom{n}{k}^2}{4^n}.\end{aligned}$$

Przyrównując do siebie wyniki otrzymane w obu sposobach, dostajemy tożsamość

$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n}.$$

C.4-7. Niech $0 \leqslant \lambda \leqslant 1$. Wykorzystując nierówność

$$\binom{n}{\lambda n} \leqslant 2^{nH(\lambda)},$$

wynikającą ze wzoru (C.6), otrzymujemy

$$b(k; n, 1/2) = \binom{n}{k}\left(\frac{1}{2}\right)^n \leqslant \frac{2^{nH(k/n)}}{2^n} = 2^{nH(k/n)-n}.$$

C.4-8. *Znak nierówności, którą należy udowodnić w tym zadaniu, powinien być skierowany przeciwnie.*

Niech X' będzie zmienną losową przyjmującą liczbę sukcesów w serii prób Bernoulliego, każda o prawdopodobieństwie sukcesu równym p. Z zad. C.4-9, po zdefiniowaniu $p'_i = p$ dla każdego $i = 1, 2, \ldots, n$, wynika wzór

$$\Pr(X \geqslant k) \leqslant \Pr(X' \geqslant k).$$

Zmienna losowa X jest rozkładu dwumianowego, zatem powyższą nierówność można zapisać w postaci

$$\Pr(X \geqslant k) \leqslant \sum_{i=k}^{n} b(i; n, p),$$

skąd, po wykorzystaniu wzoru (C.35), dostajemy

$$1 - \Pr(X < k) \leqslant 1 - \sum_{i=0}^{k-1} b(i; n, p).$$

Aby dokończyć dowód, wystarczy od obu stron nierówności odjąć jedynki i obustronnie pomnożyć przez -1.

C.4-9. Niech S będzie przestrzenią zdarzeń złożoną z wszystkich możliwych n-słów nad alfabetem $\{\text{P}, \text{S}\}$, gdzie P oznacza porażkę, a S – sukces. Ciągowi prób A odpowiada zatem pewne słowo $s \in S$, przy czym $X(s)$ to liczba wystąpień S w s. Utwórzmy teraz nowy ciąg prób Bernoulliego A' poprzez doświadczenie na próbach z ciągu A. Przez A_i będziemy rozumieć zdarzenie oznaczające wystąpienie sukcesu w i-tej próbie w ciągu A, a przez A'_i – zdarzenie oznaczające sukces w i-tej próbie w ciągu A'. Jeśli zachodzi A_i, to przyjmiemy, że zachodzi także A'_i. W przeciwnym przypadku będziemy generować sukces w i-tej próbie ciągu A' z pewnym prawdopodobieństwem r_i. Ze wzoru Bayesa (C.17) wynika, że

$$p'_i = \Pr(A'_i) = \Pr(A_i)\Pr(A'_i \mid A_i) + \Pr(\overline{A_i})\Pr(A'_i \mid \overline{A_i}) = p_i \cdot 1 + (1 - p_i) \cdot r_i,$$

więc

$$r_i = \frac{p'_i - p_i}{1 - p_i}.$$

Operując na tej samej przestrzeni S, przyjmujemy, że $X'(s)$ oznacza liczbę sukcesów w serii n prób otrzymanych powyższą procedurą z ciągu A na podstawie przyjmowanych sukcesów opisanych przez słowo s. Dla dowolnego $s \in S$ oczywistym jest, że przy takiej konstrukcji ciągu A' nie zdarzy się, aby w jego próbach było sumarycznie mniej sukcesów niż w początkowym ciągu prób A, to znaczy $X'(s) \geqslant X(s)$. Korzystając teraz z zad. C.3-7, otrzymujemy żądany wynik.

C.5. Krańce rozkładu dwumianowego

C.5-1. Zanim przejdziemy do porównywania prawdopodobieństw, udowodnimy pomocniczą nierówność

$$\binom{4n}{n-1} > 8^n$$

dla $n \geqslant 20$, stosując indukcję matematyczną. Przypadek bazowy indukcji można zweryfikować, obliczając wartości obu stron nierówności. Załóżmy zatem, że $n > 20$ i skorzystajmy z zad. C.1-5, a następnie trzykrotnie z zad. C.1-6:

$$\binom{4n}{n-1} = \frac{4n}{n-1} \cdot \binom{4n-1}{n-2} = \frac{4n}{n-1} \cdot \frac{4n-1}{3n+1} \cdot \frac{4n-2}{3n} \cdot \frac{4n-3}{3n-1} \cdot \binom{4(n-1)}{n-2}.$$

Na mocy założenia indukcyjnego zachodzi

$$\binom{4n}{n-1} > \frac{(4n)(4n-1)(4n-2)(4n-3)}{(n-1)(3n+1)(3n)(3n-1)} \cdot 8^{n-1} = \frac{(4n-1)(2n-1)(4n-3)}{3(n-1)(3n+1)(3n-1)} \cdot 8^n.$$

Okazuje się, że dla $n \geqslant 2$, a więc w szczególności dla $n \geqslant 20$ spełnione jest

$$\frac{(4n-1)(2n-1)(4n-3)}{3(n-1)(3n+1)(3n-1)} \geqslant 1,$$

co dowodzi tezy.

Zdefiniujmy teraz X i Y jako zmienne losowe przyjmujące liczby uzyskanych orłów, odpowiednio, w pierwszym i w drugim doświadczeniu. Mamy zatem

$$\Pr(X=0) = b(0;n,1/2) = \binom{n}{0}\left(\frac{1}{2}\right)^0\left(\frac{1}{2}\right)^n = \left(\frac{1}{2}\right)^n,$$
$$\Pr(Y<n) = \sum_{i=0}^{n-1} b(i;4n,1/2) = \sum_{i=0}^{n-1}\binom{4n}{i}\left(\frac{1}{2}\right)^i\left(\frac{1}{2}\right)^{4n-i} = \left(\frac{1}{2}\right)^{4n}\sum_{i=0}^{n-1}\binom{4n}{i}.$$

Dzięki uprzednio udowodnionej nierówności pokazujemy, że jeśli $n \geqslant 20$, to

$$\frac{\Pr(Y<n)}{\Pr(X=0)} = \frac{\left(\frac{1}{2}\right)^{4n}\sum_{i=0}^{n-1}\binom{4n}{i}}{\left(\frac{1}{2}\right)^n} = \frac{\sum_{i=0}^{n-1}\binom{4n}{i}}{8^n} > \frac{\binom{4n}{n-1}}{8^n} > 1.$$

Wartości obu prawdopodobieństw w przypadku, gdy $n < 20$, obliczamy bezpośrednio. Otrzymujemy ostatecznie, że $\Pr(X=0) > \Pr(Y<n)$, gdy $n \leqslant 17$, oraz $\Pr(X=0) < \Pr(Y<n)$ w przeciwnym przypadku. A zatem, dla odpowiednio dużych n, prawdopodobieństwo uzyskania mniej niż n orłów w $4n$ rzutach monetą jest większe niż nieuzyskanie żadnego orła w n rzutach.

C.5-2.

Dowód wniosku C.6. Ponieważ uzyskanie więcej niż k sukcesów w n próbach jest równoważne uzyskaniu mniej niż $n-k$ porażek, to zachodzi $\Pr(X>k) = \Pr(Y<n-k)$, gdzie Y jest zmienną losową oznaczającą liczbę porażek uzyskanych w tym doświadczeniu. Mamy $np < k < n$, skąd $0 < n-k < nq$, a zatem możemy zastosować tw. C.4 dla zmiennej losowej Y, zamieniając ze sobą role sukcesu i porażki, dzięki czemu uzyskujemy żądaną nierówność. □

Dowód wniosku C.7. Jeśli $k = n$, to wniosek jest oczywiście prawdziwy, bo szanse uzyskania więcej niż n sukcesów są zerowe. Załóżmy więc, że $k < n$. Analogicznie do poprzedniego dowodu potraktujmy prawdopodobieństwo uzyskania więcej niż k sukcesów jako prawdopodobieństwo uzyskania mniej niż $n-k$ porażek. Ponieważ $(np+n)/2 < k < n$, to $0 < n-k < nq/2$ i po zamianie sukcesu z porażką stosujemy wniosek C.5. □

C.5-3. *W rzeczywistości nierówność nie zachodzi dla podanego warunku. Zmienna k powinna spełniać nierówności $0 < k < \frac{a}{a+1}n$, bo tylko wtedy można zastosować twierdzenie C.4.*

Niech $p = \frac{a}{a+1}$, skąd mamy, że $q = \frac{1}{a+1}$ oraz $a = \frac{p}{1-p}$. Zachodzi

$$\sum_{i=0}^{k-1}\binom{n}{i}a^i = \sum_{i=0}^{k-1}\binom{n}{i}\left(\frac{p}{1-p}\right)^i = \frac{\sum_{i=0}^{k-1}\binom{n}{i}p^i(1-p)^{n-i}}{(1-p)^n} = \frac{\sum_{i=0}^{k-1} b(i;n,p)}{q^n},$$

zatem z tw. C.4 otrzymujemy

$$\sum_{i=0}^{k-1}\binom{n}{i}a^i < \frac{\frac{k}{a+1}}{\left(\frac{na}{a+1}-k\right)\left(\frac{1}{a+1}\right)^n}\, b(k;n,a/(a+1)) = (a+1)^n\frac{k}{na-k(a+1)}\, b(k;n,a/(a+1)).$$

C.5-4. Wykorzystując obserwację, że $\binom{n}{i} \geqslant 1$ dla $i = 0, 1, \ldots, n$ oraz tw. C.4, mamy

$$\sum_{i=0}^{k-1} p^i q^{n-i} \leqslant \sum_{i=0}^{k-1} \binom{n}{i} p^i q^{n-i} = \sum_{i=0}^{k-1} b(i; n, p) < \frac{kq}{np-k}\, b(k; n, p).$$

Z kolei na mocy lematu C.1 dostajemy

$$\frac{kq}{np-k}\, b(k; n, p) \leqslant \frac{kq}{np-k} \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}.$$

C.5-5. *Aby udowodnić nierówność, wymagane jest założenie, że $r > n-\mu$, którego brak w treści zadania (także w oryginale).*

Niech ν będzie wartością oczekiwaną zmiennej losowej $Y = n - X$. Wówczas

$$\nu = \mathrm{E}(Y) = \mathrm{E}(n - X) = n - \mathrm{E}(X) = n - \mu.$$

Na mocy tw. C.8 mamy, że

$$\Pr(Y - \nu \geqslant r) \leqslant \left(\frac{\nu e}{r}\right)^r,$$

skąd dostajemy nierówność

$$\Pr(\mu - X \geqslant r) \leqslant \left(\frac{(n-\mu)e}{r}\right)^r.$$

W dowodzie nierówności z drugiej części zadania zauważmy, że $\mu = \mathrm{E}(X) = np$ i na mocy nierówności udowodnionej w pierwszej części otrzymujemy

$$\Pr(np - X \geqslant r) \leqslant \left(\frac{(n-np)e}{r}\right)^r = \left(\frac{nqe}{r}\right)^r.$$

C.5-6. *W treści zadania występuje błąd. Prawa strona nierówności ze wskazówki powinna mieć postać $e^{\alpha^2/2}$.*

Lemat. *Dla dowolnych $\alpha > 0$, p, $q \geqslant 0$, spełniających $p + q = 1$, prawdziwa jest nierówność*

$$pe^{\alpha q} + qe^{-\alpha p} \leqslant e^{\alpha^2/2}.$$

Dowód. Przekształćmy nierówność do alternatywnej postaci:

$$\begin{aligned} pe^{\alpha q} + qe^{-\alpha p} &\leqslant e^{\alpha^2/2}, \\ pe^{\alpha(1-p)} + (1-p)e^{-\alpha p} &\leqslant e^{\alpha^2/2}, \\ pe^{\alpha} - p + 1 &\leqslant e^{\alpha^2/2 + \alpha p}, \\ \ln(pe^{\alpha} - p + 1) &\leqslant \alpha^2/2 + \alpha p, \\ \ln(pe^{\alpha} - p + 1) - \alpha^2/2 - \alpha p &\leqslant 0. \end{aligned}$$

Ustalmy p i potraktujmy wyrażenie po lewej stronie znaku ostatniej nierówności jako funkcję f_p zmiennej α. Zauważmy, że granicą prawostronną tej funkcji w punkcie 0 dla dowolnego $0 \leqslant p \leqslant 1$

jest 0. Udowodnimy, że wraz ze wzrostem α funkcja f_p maleje, co będzie oznaczać, że przyjmuje ona wyłącznie wartości niedodatnie i uzasadni nierówność. Obliczmy w tym celu pochodną f_p:

$$\frac{df_p}{d\alpha}(\alpha) = \frac{pe^\alpha}{pe^\alpha - p + 1} - \alpha - p = \frac{pe^\alpha(1-\alpha-p) + (\alpha+p)(p-1)}{pe^\alpha - p + 1}.$$

Mianownik ostatniego ułamka jest dodatni. Wystarczy wykazać, że licznik jest ujemny dla każdych $\alpha > 0$, $0 \leqslant p \leqslant 1$, czyli, równoważnie,

$$(\alpha+p)(p-1) < pe^\alpha(\alpha+p-1).$$

Rozważmy dwa przypadki w zależności od znaku wyrażenia $\alpha+p-1$. Jeśli $\alpha+p-1 \geqslant 0$, to $\alpha \geqslant 1-p$. Zachodzi

$$(\alpha+p)(p-1) - p(1+\alpha)(\alpha+p-1) = -\alpha(\alpha p + p^2 - p + 1) < 0.$$

Korzystając ze wzoru (3.12) i z tego, że $\alpha > 0$, mamy $1+\alpha < e^\alpha$, zatem

$$(\alpha+p)(p-1) < p(1+\alpha)(\alpha+p-1) < pe^\alpha(\alpha+p-1).$$

W drugim przypadku, czyli gdy $\alpha+p-1 < 0$, mamy $0 < \alpha < 1-p$ oraz

$$(\alpha+p)(p-1) - p(1+\alpha+\alpha^2)(\alpha+p-1) = -\alpha(\alpha^2 p + \alpha p^2 + p^2 - p + 1) < 0.$$

Wykorzystując ponownie wzór (3.12), dostajemy $e^\alpha < 1+\alpha+\alpha^2$, więc

$$(\alpha+p)(p-1) < p(1+\alpha+\alpha^2)(\alpha+p-1) < pe^\alpha(\alpha+p-1).$$

Otrzymaliśmy ostatecznie, że pochodna funkcji f_p, gdzie $0 \leqslant p \leqslant 1$, jest ujemna dla każdego $\alpha > 0$. A zatem funkcja f_p jest malejąca. □

Początek głównego rozumowania prowadzimy w oparciu o dowód twierdzenia C.8:

$$\Pr(X - \mu \geqslant r) = \Pr\bigl(e^{\alpha(X-\mu)} \geqslant e^{\alpha r}\bigr) \leqslant \mathrm{E}\bigl(e^{\alpha(X-\mu)}\bigr)e^{-\alpha r}.$$

Następnie, przy tych samych oznaczeniach jak w oryginalnym dowodzie, zachodzi

$$\mathrm{E}\bigl(e^{\alpha(X-\mu)}\bigr) = \prod_{i=1}^{n} \mathrm{E}\bigl(e^{\alpha(X_i-p_i)}\bigr).$$

Wykorzystując udowodnioną w lemacie nierówność dla $p = p_i$ oraz $q = q_i$, otrzymujemy

$$\mathrm{E}\bigl(e^{\alpha(X_i-p_i)}\bigr) = e^{\alpha(1-p_i)}p_i + e^{\alpha(0-p_i)}q_i = p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leqslant e^{\alpha^2/2}$$

i dalej mamy

$$\mathrm{E}\bigl(e^{\alpha(X-\mu)}\bigr) = \prod_{i=1}^{n} \mathrm{E}\bigl(e^{\alpha(X_i-p_i)}\bigr) \leqslant \prod_{i=1}^{n} e^{\alpha^2/2} = \exp(\alpha^2 n/2).$$

Wobec tego

$$\Pr(X - \mu \geqslant r) \leqslant \mathrm{E}\bigl(e^{\alpha(X-\mu)}\bigr)e^{-\alpha r} \leqslant \exp(\alpha^2 n/2 - \alpha r).$$

Należy teraz wybrać taką wartość α, która minimalizuje ostatnie wyrażenie. Argumentem funkcji wykładniczej w tym wyrażeniu jest funkcja kwadratowa zmiennej α, w prosty sposób można więc sprawdzić, że osiąga ona minimum dla argumentu $\alpha = r/n$. Dostajemy ostatecznie

$$\Pr(X - \mu \geqslant r) \leqslant \exp\bigl((r/n)^2 n/2 - (r/n)r\bigr) = e^{-r^2/2n}.$$

C.5-7. Potraktujmy wyrażenie jak funkcję f zmiennej $\alpha > 0$, $f(\alpha) = \exp(\mu e^\alpha - \alpha r)$. W celu wyznaczenia jej minimum obliczmy pierwszą i drugą pochodną:

$$\frac{df}{d\alpha}(\alpha) = (\mu e^\alpha - r)\exp(\mu e^\alpha - \alpha r),$$
$$\frac{d^2f}{d\alpha^2}(\alpha) = \left(\mu e^\alpha + (\mu e^\alpha - r)^2\right)\exp(\mu e^\alpha - \alpha r).$$

Pierwsza pochodna zeruje się dla $\alpha = \ln(r/\mu)$, co wymaga założenia $r > \mu$, ale warunek ten stanowi jedno z założeń tw. C.8. Po obliczeniu wartości drugiej pochodnej w tym punkcie, dostajemy

$$\frac{d^2f}{d\alpha^2}(\ln(r/\mu)) = r\exp(r - r\ln(r/\mu)) > 0,$$

ponieważ funkcja wykładnicza jest dodatnia oraz $r > \mu \geqslant 0$. A zatem w punkcie $\alpha = \ln(r/\mu)$ istnieje minimum funkcji f.

Problemy

C-1. Kule i urny

(a) Każda kula trafia do jednej z b urn. Jest b sposobów umieszczenia pierwszej kuli, na każdy z nich przypada b sposobów umieszczenia drugiej kuli itd. Jest zatem b^n sposobów rozmieszczenia n różnych kul w b różnych urnach.

(b) Ponieważ dysponujemy n rozróżnialnymi kulami oraz b nierozróżnialnymi urnami, to nasz problem jest równoważny policzeniu wszystkich możliwych ciągów n różnych kul i $b-1$ identycznych patyków. Patyki dzielą ciąg kul na spójne podciągi, z których każdy odpowiada ciągowi kul w kolejnej urnie, reprezentując wzajemne uporządkowanie kul wewnątrz urny.

Wszystkich takich ciągów jest $(b+n-1)!$, ale ponieważ nie rozróżniamy urn, to musimy podzielić tę liczbę przez liczbę możliwych rozmieszczeń urn między sobą, czyli $(b-1)!$. Istnieje zatem $\frac{(b+n-1)!}{(b-1)!}$ różnych rozmieszczeń kul w urnach.

(c) Sytuacja jest podobna jak w punkcie (b) z tą różnicą, że nie rozróżniamy kul między sobą, a więc również każda permutacja n kul między sobą opisuje ten sam sposób rozmieszczenia kul w urnach. Mamy zatem $\frac{(b+n-1)!}{n!\,(b-1)!} = \binom{b+n-1}{n}$ możliwości rozmieszczenia kul.

(d) Zakładając, że $n \leqslant b$, wybieramy spośród b urn n takich, które będą zawierać po jednej kuli. Jest $\binom{b}{n}$ sposobów ich wyboru.

(e) Zakładamy, że $n \geqslant b$. Najpierw umieszczamy po jednej kuli w każdej z b urn, dzięki czemu żadna urna nie jest pusta. Na mocy punktu (c) pozostałe $n-b$ kul możemy umieścić w b urnach na $\binom{b+(n-b)-1}{n-b} = \binom{n-1}{n-b} = \binom{n-1}{b-1}$ sposobów.

Bibliografia

[1] Robert M. Corless, Gaston H. Gonnet, Dave E. G. Hare, David J. Jeffrey, Donald E. Knuth. On the Lambert W function. *Advances in Computational Mathematics*, 5(1):329–359, 1996.

[2] Thomas H. Cormen, Clara Lee, Erica Lin. *Instructor's Manual to Accompany Introduction to Algorithms Second Edition*, 2002.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, wydanie 2, 2001.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Wprowadzenie do algorytmów*. Klasyka informatyki. Wydawnictwa Naukowo-Techniczne, Warszawa, wydanie 6, 2004.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, wydanie 3, 2009.

[6] Ronald L. Graham, Donald E. Knuth, Oren Patashnik. *Matematyka konkretna*. Wydawnictwo Naukowe PWN, Warszawa, wydanie 4, 2009.

[7] Donald E. Knuth. *The TEXbook*. Addison-Wesley, Massachusetts, 1984.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, wydanie 3, 1997.

[9] Leslie Lamport. *LATEX: A Document Preparation System*. Addison-Wesley, wydanie 2, 1994.

[10] Till Tantau. *The TikZ and PGF Packages – Manual for version 3.0.1a*, sierpień 2015. (`http://mirrors.ctan.org/graphics/pgf/base/doc/pgfmanual.pdf`).

[11] Stanisław M. Ulam, Myron L. Stein, Mark B. Wells. A visual display of some properties of the distribution of primes. *American Mathematical Monthly*, 71(5):516–520, 1964.

[12] Eric W. Weisstein. Dirichlet's box principle. From MathWorld – A Wolfram Web Resource. (`http://mathworld.wolfram.com/DirichletsBoxPrinciple.html`).

[13] Eric W. Weisstein. Ramsey number. From MathWorld – A Wolfram Web Resource. (`http://mathworld.wolfram.com/RamseyNumber.html`).