Krzysztof Diks, Wojciech Śmietanka
Instytut Informatyki
Uniwersytet Warszawski

Kubełkowe struktury danych

W niniejszym rozdziale przedstawiamy struktury danych, które są proste koncepcyjnie, stosunkowo łatwe w implementacji i nieźle zachowują się w praktyce. Pomimo swojej prostoty i dobrego zachowania, opis tych struktur trudno znaleźć w podręcznikach algorytmiki, dlatego idee przedstawione tutaj powinny zainteresować każdego, kto chciałby poszerzyć swoją wiedzę algorytmiczną. Dla każdego z problemów omawianych w tym rozdziale względnie łatwo zaprojektować algorytm działający w czasie kwadratowym ze względu na rozmiar danych. Idee tutaj zaprezentowane umożliwiają otrzymanie algorytmów działających o czynnik pierwiastkowy szybciej. W szczególności pokazujemy, w jaki sposób obliczyć w czasie $O\left(n\sqrt{n}\right)$ wektor inwersji dla zadanej n-elementowej permutacji oraz jak zaimplementować algorytm Dijkstry, żeby działał w czasie $O\left(m+n\sqrt{c}\right)$, gdzie n to liczba wierzchołków grafu, m liczba jego krawędzi, a c jest całkowitoliczbowym ograniczeniem górnym na długości krawędzi, przy czym zakładamy, że te długości są także całkowitoliczbowe. Oprócz przedstawienia ciekawych struktur danych i ich zastosowań, nie mniej ważny jest proponowany sposób rozwiązywania problemów algorytmicznych, polegający na formułowaniu najpierw algorytmów jak najbardziej abstrakcyjnie, a następnie poszukiwaniu możliwie najlepszych metod implementacji wykorzystywanych w opisie algorytmów abstrakcyjnych konstrukcji.

1. Wstęp

Z pojęciem **kubełków** każdy interesujący się programowaniem i algorytmiką miał pewnie okazję się spotkać przy okazji poznawania algorytmów sortowania. **Sortowanie kubełkowe** [3, 8] stosuje się wtedy, gdy o elementach porządkowanego ciągu wiemy, że są liczbami całkowitymi z przedziału $[0, c-1]$ (lub danymi, którym można przypisać liczby całkowite, uwzględniając ich względny porządek), dla znanej z góry wartości parametru c. Sortowanie wówczas polega na rozrzuceniu sortowanych liczb do kubełków indeksowanych liczbami z przedziału $[0, c-1]$ w taki sposób, że liczby o wartości i wpadają do kubełka z indeksem i. Żeby otrzymać posortowany ciąg wystarczy teraz przejrzeć kubełki w kolejności rosnących indeksów i wypisać ich zawartości. Taki algorytm działa w czasie $O(n+c)$. Jeśli c jest rzędu co najwyżej n, to otrzymujemy algorytm sortowania działający w czasie liniowym. Niektórzy mogą poczuć się zaskoczeni, przypominając sobie, że każdy algorytm sortowania wymaga czasu rzędu co najmniej $n\log n$ [3, 8]. Trzeba jednak pamiętać, że ta dolna granica obowiązuje dla sortowania, w którym informację o względnym porządku elementów uzyskuje się tylko porównując porządkowane elementy między sobą. W sortowaniu kubełkowym elementy nie są ze sobą porównywane i wykorzystuje się wiedzę na temat natury ich wartości. W szczególności to, że te wartości mogą być indeksami tablicy (kubełków). Jeśli zakres tych indeksów jest nieduży, to otrzymujemy bardzo szybki algorytm sortowania, który może być alternatywą dla znanych, najszybszych algorytmów sortowania z użyciem porównań.

Jednym z najważniejszych praktycznych zastosowań kubełków jest **haszowanie** [3], wykorzystywane w bazach danych do szybkiego wyszukiwania interesujących nas danych. Dane przechowujemy w kubełkach poindeksowanych od 0 do $c-1$, dla pewnego parametru c. Z każdymi danymi związany jest klucz jednoznacznie je identyfikujący. Dane o kluczu k przechowywane są w kubełku o indeksie $h(k)$, gdzie h jest funkcją odwzorowującą klucze w przedział indeksów $[0, c-1]$. Jeśli funkcję h dobierzemy w taki sposób, że jest ona szybko obliczalna i równomiernie rozrzuca dane po kubełkach, to dla dobrze dobranego parametru c do każdego kubełka wpadnie niewiele danych, co gwarantuje, że każde interesujące nas dane można odzyskać w czasie stałym, czyli niezależnym od liczby elementów w bazie danych.

W tym rozdziale idea kubełków jest wykorzystana w efektywnej implementacji algorytmów dla trzech problemów, opisanych kolejno w punktach 2, 3 i 4. Zaproponowane struktury danych są proste koncepcyjnie, łatwe w implemen-

tacji, dobrze zachowują się w praktyce i są alternatywą dla struktur danych polegających na (często niełatwej) adaptacji zrównoważonych drzew wyszukiwań binarnych lub wykorzystaniu złożonych implementacji kolejek priorytetowych. Oprócz przedstawienia konkretnych rozwiązań, które mogą znaleźć zastosowania także w innych problemach, proponujemy też pewną metodologię atakowania problemów algorytmicznych, która polega na zapisania algorytmu w sposób jak najbardziej abstrakcyjny, operując takimi pojęciami jak zbiór, ciąg, funkcja itp., a dopiero potem na podawaniu szczegółów implementacyjnych odnoszących się do używanych obiektów abstrakcyjnych i wykonywanych na nich operacjach.

2. Kodowanie permutacji

Nasze rozważania rozpoczniemy od rozwiązania nietrudnego, ale użytecznego zadania, które to rozwiązanie wprowadzi nas w świat kubełkowych struktur danych.

Zadanie 2.1. Kodowanie permutacji

Każdą permutację $A = [0..n-1]$ liczb $0, 1,..., n-1$ można zakodować za pomocą tablicy $B = [0..n-1]$, w której dla każdego $i = 0, 1,..., n-1$ $B[i]$ jest równe liczbie wszystkich takich j, że $0 \le j < i$ oraz $A[j] > A[i]$. Należy zaprojektować algorytm, który dla danej permutacji A znajduje jej kod B.

Przykład 2.1.

Dla $n = 10$ i $A = [2, 6, 0, 9, 7, 3, 1, 5, 4, 8]$ kodem permutacji A jest tablica $B = [0, 0, 2, 0, 1, 3, 5, 3, 4, 1]$.

W literaturze kombinatorycznej kod B jest nazywany **wektorem inwersji permutacji** A [6]. Suma elementów wektora inwersji mówi, ile jest wszystkich nieuporządkowanych (rosnąco) par elementów w permutacji, którą ten wektor koduje. Taka liczba jest pewną miarą uporządkowania tablicy A i wiadomo na przykład, że algorytm sortowania przez wstawianie (w implementacji ze strażnikiem) wykonuje $n-1+\sum B[i]$ porównań, żeby posortować rosnąco permutację A. A zatem, im ciąg ma mniej inwersji, tym szybciej algorytm sortowania przez wstawianie poradzi sobie z jego uporządkowaniem.

Wymyślenie algorytmu dla naszego zadania nie jest specjalnie trudne. Jest on ukryty w samej definicji problemu. Zapiszmy go jednak. Pamiętajmy przy tym, żeby pierwszy algorytm był jak najbardziej abstrakcyjny, operował pojęciami matematycznymi takimi jak zbiór, ciąg, funkcja oraz operacjami na tych obiektach. Przy pierwszym podejściu nie myślmy nad szczegółami implementacyjnymi. Im ogólniejszy zapis algorytmu, tym większa potem swoboda w doborze tych szczegółów. Oto pierwszy algorytm:

```
Algorytm Kodowanie_1
    for i := 0 to n-1 do
        B[i] := liczba takich j, że 0 ≤ j < i orazA[j]>A[i];  (**)
```

Bardzo łatwo zaimplementować ten algorytm wykonując operację z wnętrza pętli **for** w następujący sposób: przeglądamy po kolei elementy $A[0], A[1],..., A[i-1]$ i każdy z nich porównujemy z $A[i]$, zliczając te $A[j]$, które są większe od $A[i]$:

```
licz := 0;
e := A[i];
for j := 0 to i-1 do
    if (A[j] > e) then
        licz := licz + 1;
    B[i] := licz;
```

Wyznaczenie wartości $B[i]$ wymaga i porównań elementu $A[i]$ z elementami go poprzedzającymi. Zatem łączna liczba porównań konieczna do wyznaczenia kodu B zaproponowanym algorytmem wynosi $\sum_{i=0}^{n-1} i = n(n-1)/2$, niezależnie od zawartości tablicy A. Algorytm ten jest algorytmem kwadratowym, czyli o złożoności $O(n^2)$.

Spróbujmy teraz znaleźć szybsze rozwiązanie. W takim przypadku zazwyczaj mamy dwie możliwości. Możemy zastanowić się nad zupełnie nowym algorytmem albo przyśpieszyć działanie algorytmu, który znamy, poprzez odpowiedni dobór struktur danych. Zastosujemy to drugie podejście. Ponieważ ten rozdział adresujemy do początkujących algorytmików, nasze propozycje nie będą najbardziej optymalne, ale ich zaletą będzie prostota oraz stosunkowo niezłe zachowanie się w praktyce.

Spróbujmy wyrazić operację (**) trochę inaczej. Przez $S_i = \{A[0],..., A[i-1]\}$ oznaczmy zbiór elementów permutacji A z pozycji $0,..., i-1$. Zbiór S_i jest i-elementowym podzbiorem zbioru $S = \{0,..., n-1\}$, ale nie zawiera $A[i]$. Każdy podzbiór $S' \subseteq S$ można reprezentować jako zero-jedynkową tablicę $V[0..n-1]$ taką, że $V[i] = 1$, jeśli tylko $i \in S'$, natomiast $V[i] = 0$, gdy $i \notin S'$.

Przykład 2.2.

Dla $n = 10$ i $S' = \{1, 3, 6, 7, 9\}$ mamy $V = [0, 1, 0, 1, 0, 0, 1, 1, 0, 1]$.

Jeśli V reprezentuje zbiór S_{i-1}, to $B[i]$ jest po prostu równe liczbie jedynek w tablicy V na prawo od pozycji o indeksie równym $A[i]$, czyli znajdujących się w V na pozycjach $A[i] + 1, A[i] + 2,..., n - 1$. Przejście od reprezentacji zbioru S_{i-1} do reprezentacji zbioru S_i polega po prostu na ustawieniu jedynki w tablicy V na pozycji $A[i]$, tzn. $V[A[i]]:= 1$. Zauważmy jeszcze, że S_0 jest zbiorem pustym, a jego reprezentacją jest $V = [0, 0,..., 0]$. Nasz algorytm można teraz przepisać następująco:

```
Algorytm Kodowanie_2
    {* inicjowanie pustego zbioru S_0 *}
    for i := 0 to n – 1 do
        V[i] := 0;
    (* obliczanie kodu B *)
    for i := 0 to n – 1 do
        B[i] := Jedynki(A[i]);
        Ustaw(A[i]);
```

W tym algorytmie funkcja `Jedynki(k)` oblicza liczbę jedynek w tablicy V na pozycjach o indeksach większych od k, natomiast procedura `Ustaw(k)` ustawia jedynkę w tablicy V na pozycji k-tej. Jeśli zliczanie jedynek zaimplementujemy w naiwny sposób polegający na przeglądaniu wszystkich pozycji o indeksach większych od k, to złożoność algorytmu nadal pozostanie kwadratowa. Pokażemy teraz, w jaki sposób znacząco przyśpieszyć obliczenia, stosując chwyt, którego opis i zastosowania są przedmiotem rozważań w tym rozdziale.

Niech m będzie dodatnią liczbą całkowitą nie większą od n i niech $s = \lfloor (n-1)/m \rfloor + 1$. Zauważmy, że jeśli m dzieli całkowicie n, to $s = n/m$, a w przeciwnym razie $s = \lceil n/m \rceil$. Na przykład dla $n = 10$ i $m = 2$ mamy $s = 5$, natomiast dla $n = 10$ i $m = 3$ mamy $s = 4$.

Aby przyspieszyć algorytm, podzielimy tablicę V na s rozłącznych bloków $V_0, V_1, ..., V_{s-1}$, gdzie $V_i = V[im..\min((i+1)m - 1, n)]$. W naszym przykładzie dla $n = 10$ i $m = 3$ mamy $V_0 = V[0..2]$, $V_1 = V[3..5]$, $V_2 = V[6..8]$, $V_3 = V[9..9]$. Zauważmy, że dla $i < s - 1$ każdy blok V_i ma długość m, natomiast długość bloku V_{i-1} wynosi $n - (s-1)m$ i jest nie większa niż m. Wprowadźmy dodatkową tablicę $K[0..s-1]$, która będzie zawierała informacje o liczbach jedynek w blokach V_i. To znaczy chcemy, żeby $K[i]$ było równe liczbie jedynek w bloku V_i. Dla $n = 10$, $m = 3$ i $V = [0, 1, 0, 1, 0, 0, 1, 1, 0, 1]$, mamy $K = [1, 1, 2, 1]$. Korzystając z tablicy K, wartość funkcji `Jedynki(j)` można teraz obliczyć następująco. Najpierw obliczamy indeks bloku zawierającego pozycję j. Łatwo przekonać się, że wynosi on $l = \lfloor j/m \rfloor$. Teraz sumując wartości $K[l+1], ..., K[s-1]$, otrzymujemy liczbę wszystkich jedynek w blokach na prawo od bloku l, czyli bloku zawierającego pozycję j. Niech tą liczbą będzie x. To, co nam jeszcze pozostaje, to obliczyć liczbę jedynek na prawo od j w bloku V_l. W tym celu przeglądamy kolejne pozycje $j+1, j+2..., \min((l+1)m - 1, n)$. Takich pozycji jest mniej niż m.

Zastanówmy się teraz, ile nas kosztuje obliczenie wartości `Jedynki(j)`. Sumowanie wartości z tablicy K zabiera mniej niż s dodawań, natomiast zliczanie jedynek w bloku V_i wymaga przejrzenia mniej niż m elementów w tym bloku. Zatem łączny koszt wykonania algorytmu jest rzędu co najwyżej $O(m + s)$. Zapytajmy teraz, w jaki sposób dobrać m, żeby osiągnąć jak najlepszy efekt złożonościowy? Ponieważ $s = \lfloor (n-1)/m \rfloor + 1$, to widać, że najszybszy algorytm dosta-

niemy wtedy, gdy m i s są mniej więcej takie same. Zatem za m najlepiej wziąć $\lfloor\sqrt{n}\rfloor$, a wówczas funkcja `Jedynki` jest wykonywana w czasie $O(\sqrt{n})$.

Procedurę `Ustaw(j)` zaimplementować bardzo łatwo. Wystarczy ustawić j-tą pozycję w tablicy V na 1 oraz zwiększyć liczbę jedynek w bloku zawierającym j o 1, $K[\lfloor j/m \rfloor] := K[\lfloor j/m \rfloor] + 1$.

Podsumowując, wykonujemy n operacji obliczania liczby jedynek i n operacji ustawienia nowej jedynki, zatem obliczanie kodu permutacji można wykonać w czasie $O\left(n\sqrt{n}\right)$, czyli zdecydowanie szybciej niż w czasie $O(n^2)$. Dla przykładu, jeśli przyjmiemy realistycznie, że stałe ukryte w notacji $O(.)$ wynoszą odpowiednio 2 i 1/2, to dla $n = 1\,000\,000$, $2n\sqrt{n} = 2\,000\,000\,000$, natomiast $n^2/2 = 1\,000\,000\,000\,000/2 = 500\,000\,000\,000 = 250*2\,000\,000\,000$.

Pozostaje formalnie zapisać poszczególne operacje. Niech $m = \lfloor\sqrt{n}\rfloor$, $s = \lfloor (n-1)/m \rfloor + 1$. Poniżej przedstawiamy zapis omówionych właśnie funkcji. Pamiętajmy tylko, że tablica K musi być wcześniej zainicjowana zerami.

```
Algorytm Jedynki(j)
    l := ⌊j / m⌋;
    x := 0;
    for i := l + 1 to s - 1 do
        x := x + K[i];
    for i := j + 1 to min((l + 1) * m - 1, n) do
        if V[i] = 1 then x := x + 1;
    return x;

Algorytm Ustaw(j)
    V[j] := 1; l := ⌊j / m⌋;
    K[l] := K[l] + 1 ;
```

3. Odwracanie

W tym punkcie zajmiemy się rozwiązaniem zadania często spotykanego w praktyce. Operujemy na zbiorze danych, które podlegają różnorodnym modyfikacjom. Należy tak zorganizować ten zbiór, żeby móc szybko odpowiadać na pytania dotyczące zawartych w nim elementów. Nasze kolejne zadanie będzie właśnie takiego typu.

Zadanie 3.1. Odwracanie bloków

Dana jest dodatnia liczba całkowita n i ciąg elementów $C = [c_1, c_2, ..., c_n]$ dowolnego, ale ustalonego typu. Dla ustalenia uwagi przyjmijmy, że elementy ciągu C to małe litery alfabetu angielskiego. Blokiem nazwiemy każdy podciąg ciągu C złożony z kolejnych elementów. Na ciągu C wykonujemy następujące operacje:

1. `Odwróć(i,j)` – odwróć kolejność elementów w ciągu C na pozycjach od i do j, dla $1 \leq i \leq j \leq n$;
2. `El(j)` – podaj wartość j-tego elementu w aktualnym ciągu C, gdzie $1 \leq j \leq n$.

Należy zaprojektować strukturę danych, która służyć będzie do szybkiego wykonania m operacji powyższego typu na dynamicznym ciągu C. Zauważmy, że z punktu widzenia użytkownika ważne są pytania o elementy ciągu. Tak więc operacji `Odwróć` nie musimy fizycznie wykonywać, ważne jest tylko, żeby odpowiedzi na pytania o elementy ciągu były takie, jak byśmy wykonali wszystkie poprzedzające je odwrócenia.

Przykład 3.1.

Załóżmy, że na początku ciąg C ma postać [a, b, c, d, e, f, g, h], $n = 8$ i na C wykonujemy kolejno operacje: `Odwróć(2,4), El(2), El(5), Odwróć(4,7), El(5)`. Wówczas odpowiedziami dla kolejnych wywołań funkcji `El` są odpowiednio d, e, f.

Dla dalszych rozważań przyjmijmy, że przetwarzany ciąg jest zapisany w tablicy $C[1..n]$. Pierwsze narzucające się rozwiązanie jest oczywiste. Każde wywołanie operacji `Odwróć(i,j)` pociąga za sobą fizyczne odwrócenie zawartości podtablicy $C[i..j]$. Ponieważ odwrócenie kolejności elementów w podciągu o długości $j - i + 1$ pociąga za sobą $\lfloor (j - i + 1)/2 \rfloor$ zamian elementów parami, to w pesymistycznym przypadku wykonanie całego ciągu m operacji zabrałoby czas $O(mn)$.

Nasze rozwiązanie będzie polegało na odwlekaniu fizycznej reorganizacji tablicy C – wykonaniu fizycznych odwróceń – tak długo, jak to tylko możliwe. Czy rzeczywiście, żeby odpowiadać na pytanie o elementy ciągu C musimy wykonać fizycznie wszystkie wcześniejsze odwrócenia? Gdyby na przykład przedziały, w których dokonujemy odwrócenia były parami rozłączne, wystarczyłoby tablicę C podzielić na spójne bloki, które odpowiadają maksymalnym fragmentom, które są w całości odwrócone lub w całości nieodwrócone, i dla każdego bloku mieć znacznik informujący o jego statusie: odwrócony, nieodwrócony. Kiedy pytamy o k-ty element ciągu, który wpada do bloku o indeksach od i do j, odpowiedzią jest element $C[k]$, gdy blok nie jest odwrócony, natomiast element $C[j - (k - i)]$, gdy blok został odwrócony. Niestety odwrócenia mogą być wykonywane na blokach, które nie są rozłączne. Szczęśliwie, idee opisane wcześniej znajdują zastosowanie w ogólnym przypadku.

Zawartość tablicy C po wykonaniu pewnej liczby operacji `Odwróć` będziemy reprezentowali jako ciąg parami rozłącznych, pokrywających ją bloków $B_1, B_2, \ldots, B_k$, dla pewnego $k \geq 1$. Każdy blok jest zadany jako trójka (a, b, *odwr*), gdzie a jest indeksem początku bloku, b indeksem jego końca, natomiast *odwr* jest logicznym wskaźnikiem informującym o tym, czy elementy w bloku są odwrócone (*odwr* = *TRUE*), czy też nie (*odwr* = *FALSE*). Żeby odczytać ciąg w aktualnej postaci,

należy przejść kolejno po wszystkich blokach, a w każdym bloku odczytywać elementy w C z lewa na prawo, gdy blok nie jest odwrócony, natomiast z prawa na lewo, gdy jest odwrócony. Na samym początku mamy tylko jeden blok o parametrach (1, n, *FALSE*).

Przykład 3.2.

Niech $C = [a, b, c, d, e, f, g, h]$ i niech (5, 7, T), (2, 4, F), (8, 8, T), (1, 1, T) będzie ciągiem czterech bloków. Zawartość tablicy C po fizycznym wykonaniu odwróceń miałaby postać $[g, f, e, b, c, d, h, a]$.

Zacznijmy od operacji `El(j)`. Oznaczmy przez s_i rozmiar i-tego bloku. Oczywiście $s_i = b_i - a_i + 1$. Żeby wyznaczyć j-ty element w aktualnym ciągu, przeglądamy bloki po kolei w poszukiwaniu pierwszego takiego, dla którego suma rozmiarów wszystkich bloków go poprzedzających oraz jego własnego rozmiaru wynosi co najmniej j. Niech tym blokiem będzie B_i i niech suma rozmiarów wszystkich bloków go poprzedzających wynosi s. Wówczas poszukiwanym elementem jest element na pozycji $j - s$ w tym bloku, gdy blok nie jest odwrócony. W niezmienionej tablicy C ten element znajduje się na pozycji $a_i + j - s - 1$. Jeśli blok jest odwrócony, to poszukiwanym elementem jest ten z pozycji $s_i + 1 - (j - s)$ w bloku B_i, czyli $C[a_i + s_i - (j - s)]$. Oto formalny zapis funkcji `El`.

```
Algorytm El(j)
    s := 0;
    i := 1;
    while (s + s_i < j) do
        i := i + 1;
        if (odwr_i) then
            return C[a_i + s_i - (j - s)]
        else
            return C[a_i + j - s - 1];
```

Zauważmy, że koszt pesymistyczny tego algorytmu jest rzędu liczby wszystkich bloków k.

Trochę trudniejsza do wykonania jest operacja `Odwróć(i,j)`. Jak się za chwilę okaże, najbardziej problematyczny jest przypadek, gdy fragment, który chcemy odwrócić, wpada do jednego bloku. Jeśli jest to cały blok, to wystarczy tylko zmienić jego wskaźnik odwrócenia na przeciwny. Załóżmy zatem, że ten fragment jest podblokiem pewnego bloku B zadanego przez trójkę (a, b, *odwr*). Najpierw obliczamy początek c i koniec d podbloku bloku B, który odpowiada przedziałowi $[i, j]$ i którego elementy chcemy odwrócić.

Pozycje c i d obliczamy w sposób opisany w funkcji `El`. Teraz blok B dzielimy na (co najwyżej) trzy bloki $X = B[a..c - 1]$, $Y = B[c..d]$, $Z = B[d + 1..b]$. Przedziały

X i *Z* mogą być puste i wtedy pomijamy je w naszych rozważaniach. Wskaźnikom odwrócenia bloków *X* i *Z* nadajemy taką samą wartość, jaką ma wskaźnik bloku *B*, natomiast wskaźnik odwrócenia bloku *Y* przyjmuje wartość przeciwną. Po dokonaniu tych czynności w ciągu bloków $B_1, B_2,..., B_k$, w miejsce bloku *B*, wstawiamy kolejno (tylko niepuste) bloki *X*, *Y*, *Z*, gdy wartość wskaźnika odwrócenia bloku *B* wynosi *FALSE*, natomiast *Z*, *Y*, *X*, gdy tą wartością jest *TRUE*.

A co, gdy przedział [*i*, *j*] zawiera więcej niż jeden blok? Niech $B_l, B_{l+1},..., B_p$ będą tymi blokami. Niech *c* będzie pozycją w bloku B_l odpowiadającą początkowi przedziału, który chcemy odwrócić, czyli *i*. W zależności od wskaźnika odwrócenia $odwr_l$ dla bloku B_l dzielimy ten blok na dwa podbloki $B_l^{(1)}$ i $B_l^{(2)}$ w następujący sposób:

- jeśli $odwr_l = FALSE$, to $B_l^{(1)} = B[a..c-1]$, $B_l^{(2)} = B[c..b]$ i dla obu bloków wskaźniki odwrócenia ustawiamy na $FALSE$;
- jeśli $odwr_l = TRUE$, to $B_l^{(1)} = B[c+1..b]$, $B_l^{(2)} = B[a..c]$ i dla obu bloków wskaźniki odwrócenia ustawiamy na $TRUE$.

Jeśli teraz w miejsce bloku B_l wstawimy bloki $B_l^{(1)}$ i $B_l^{(2)}$ (tylko niepuste), to możemy dalej pracować przy założeniu, że blok B_l (teraz jest to blok $B_l^{(2)}$) całkowicie wpada w przedział [*i*, *j*]. Podobnie możemy postąpić z blokiem B_p. Zostawiamy to jako ćwiczenie dla Czytelnika.

W tym momencie możemy założyć, że cały przedział [*i*, *j*] to tak naprawdę suma bloków $B_l,..., B_p$, być może z odwróconymi kolejnościami wystąpień elementów w pewnych z nich. Aktualizacja struktury jest teraz bardzo prosta. W miejsce ciągu $B_l, B_{l+1},..., B_p$ w naszej strukturze wstawiamy ciąg $B_p, B_{p-1},..., B_l$ pamiętając przy tym, żeby wskaźnik odwrócenia każdego bloku w tym ciągu zamienić na przeciwny. Nietrudno zauważyć, że koszt operacji aktualizacji naszej struktury jest rzędu co najwyżej liczby bloków w niej zawartych, czyli wynosi $O(k)$.

Pokazaliśmy, że obie operacje `Odwróć` i `El` w pesymistycznym przypadku wykonują się w czasie proporcjonalnym do liczby bloków w strukturze. Duża liczba operacji odwracania może doprowadzić do dużej liczby bloków, w pesymistycznym przypadku nawet liniowej ze względu na *n*. Dlatego warto dbać o to, żeby liczba bloków nie była za duża. Przyjmiemy, że gdy ta liczba przekroczy $\sqrt{n}$, to odtworzymy aktualny ciąg *C* na podstawie zawartości bloków. To nie zajmie więcej niż czas liniowy i znowu będziemy mogli wystartować z jednym blokiem z parametrami (1, *n*, *FALSE*). Zatem koszt wykonania ciągu *m* operacji wynosi co najwyżej $O(m\sqrt{n})$. Pomijamy tu koszt inicjacji danych, który nie przekracza $O(n)$ – zainicjowanie tablicy *C* i jednoelementowej struktury bloków.

Zadanie omawiane w tym rozdziale pochodzi z zawodów programistycznych CERC 2007 [9], a autorem pomysłu przedstawionego rozwiązania jest reprezentant Uniwersytetu Warszawskiego na tych zawodach Piotr Niedźwiedź.

4. Algorytm Dijkstry

Edsger W. Dijkstra (1930-2002), światowej sławy holenderski informatyk. Jak sam wspominał, algorytm dla problemu najkrótszych ścieżek z jednym źródłem wymyślił w 20 minut podczas zakupów w centrum handlowym w Amsterdamie w roku 1956. Publikacja zawierająca opis algorytmu ukazała się trzy lata później: Dijkstra, E. W., *A note on two problems in connexion with graphs*, **„Num. Math." 1(1959), 269-271. Sam problem pojawił się jako ilustracja zastosowania komputera ARMAC dla niespecjalistów. Dijkstra pokazywał, w jaki sposób najszybciej przejechać z Rotterdamu to Groningen. Na owe czasy takie obliczenia z użyciem komputera były prawdziwym wyzwaniem. Warto tutaj dodać, że w tej samej pracy Dijkstra zamieścił również opis algorytmu dla problemu najkrótszego drzewa rozpinającego, bazujący na podobnej idei zachłanności, w myśl której w kolejnym kroku algorytm przemieszcza się do najbliższego wierzchołka wśród jeszcze nieodwiedzonych. Warto jednak zauważyć różnice między tymi dwoma problemami i algorytmami – pozostawiamy to jako ćwiczenie dla własnych przemyśleń** [7].

W tej części zajmiemy się efektywną implementacją jednego z najsłynniejszych algorytmów – algorytmu Dijkstry dla problemu znajdowania najkrótszych ścieżek z jednym źródłem [3, 6]. Krótko sformułujemy problem i podamy dla niego abstrakcyjny algorytm. Nie będziemy się zajmować jego poprawnością, ponieważ opis algorytmu Dijkstry można znaleźć w każdym porządnym podręczniku algorytmiki. Podamy za to jego podstawowe charakterystyki, które mają wpływ na implementację, a następnie zaproponujemy struktury danych, które są nie tylko łatwe w realizacji, ale także efektywne w praktyce.

Załóżmy, że mamy dość szczegółową mapę drogową Polski. Szczegółowość oznacza tu, że odległości między sąsiednimi miejscowościami na mapie (bezpośrednio połączone drogą, nieprzechodzącą przez żadną inną miejscowość uwzględnioną na mapie) nie przekraczają powiedzmy 10 km. Naszym celem jest obliczenie długości najkrótszych tras prowadzących z Warszawy do wszystkich miejscowości uwzględnionych na mapie. Dla uproszczenia przyjmijmy, że wszystkie drogi są dwukierunkowe.

Mapę można modelować za pomocą grafu nieskierowanego, w którym wierzchołki odpowiadają miejscowościom na mapie, natomiast krawędzie bezpośrednim drogom łączącym sąsiednie miejscowości. Z każdą drogą (krawędzią) wiążemy dodatnią liczbę całkowitą równą długości tej drogi. W naszym przypadku przyjmujemy, że długości są dodatnimi liczbami całkowitymi nie większymi od zadanej, dodatniej liczby całkowitej c. Dodatkowo w grafie (na mapie)

wyróżniamy jeden wierzchołek (np. stolicę), dla którego chcemy policzyć odległości (długości najkrótszych tras) do wszystkich pozostałych wierzchołków (miejscowości). Problem, o którym mówimy, w literaturze spotyka się pod nazwą **problemu najkrótszych ścieżek** z jednym źródłem i jest jednym z najczęściej badanych algorytmicznych problemów optymalizacyjnych. Istnieją setki prac na temat rozwiązania tego problemu, a większość z nich to wariacje na temat algorytmu zaproponowanego przez wybitnego holenderskiego informatyka Edsgara Dijkstrę. Należy tutaj zaznaczyć, że w ogólnym sformułowaniu problemu, o długościach krawędzi zakłada się tylko, że są dodatnie i nie narzuca się na te długości żadnego górnego ograniczenia. Sformułujemy teraz nasz problem w oderwaniu od terminologii grafowej, ale w sposób, który będzie przydatny w naszych rozważaniach.

Zadanie 4.1. Algorytm Dijkstry

Danych jest n obiektów ponumerowanych od 1 do n. Obiekty utożsamiamy z ich numerami. Z każdym obiektem i związany jest podzbiór $N(i)$ obiektów różnych od i. Elementy zbioru $N(i)$ nazywamy **sąsiadami** obiektu i. Dla każdego obiektu-sąsiada $j \in N(i)$ znamy odległość $d_i[j]$ pomiędzy obiektami i oraz j. Każda odległość jest dodatnią liczbą całkowitą, nie większą niż zadana z góry dodatnia liczba całkowita c. Naszym zadaniem jest zaprojektowanie „szybkiego" obliczania tablicy $d[1..n]$, zgodnie z następującym algorytmem:

```
Algorytm AD
    (* Inicjacja *)
    d[1] := 0;
    for i := 2 to n do
        if (i ∈ N(1)) then
            d[i] := d1[i];
        else
            (* górne ograniczenie na długość najdłuższej trasy *)
            d[i] := c * (n - 1);
    S := {2,3,...,n};
    (* główne obliczenia *)
    for i := 1 to n - 1 do
        j := obiekt w zbiorze S z najmniejszym d[j]; (* Min *)
        S := S\{j}; (* Usuń Min *)
        for k ∈ N(j) do
            (* Zmniejsz Priorytet *)
            if (d[k] > (d[j] + dj[k])) then d[k] := d[j] + dj[k];
```

Algorytm AD jest tak naprawdę algorytmem Dijkstry zapisanym w abstrakcyjny sposób. Poniżej podajemy kilka ważnych własności tego algorytmu, w których uwzględniamy ograniczenie c na odległości. Będą one przydatne w jego wydajnej implementacji.

1. Niech $j_1, j_2, ..., j_{n-1}$ będą kolejnymi obiektami obliczanymi w wierszu (* `Min` *) i weźmy $j_0 = 1$. Wówczas $d[j_0] = 0 < d[j_1] \le d[j_2] \le ... \le d[j_{n-1}]$. Innymi słowy w algorytmie Dijkstry obliczamy odległości wierzchołków od źródła w kolejności od najbliższych do najdalszych.
2. Dla każdego $k = 1, 2, ..., n-1$, $d[j_k] - d[j_{k-1}] \le c$ – kolejny wierzchołek jest odległy od źródła o co najwyżej c dalej, niż wierzchołek go poprzedzający.
3. $d[j_{n-1}] \le c*(n-1)$ – najbardziej odległy wierzchołek jest położony nie dalej niż $c*(n-1)$.

Przyjrzyjmy się teraz, co wpływa na koszt wykonania algorytmu. Główne operacje są wykonywane na zmieniającym się zbiorze S. Każdy element j w tym zbiorze ma przypisaną liczbę całkowitą $d[j]$, którą nazwiemy **priorytetem**. Wiemy, że każdy priorytet jest liczbą całkowitą z przedziału $[0, c(n-1)]$. Na zbiorze S wykonujemy następujące operacje:

`Inicjacja::` utwórz zbiór S z obiektami 2, 3,..., n o priorytetach zdefiniowanych w części `Inicjacja` algorytmu `AD`.

`Min::` wskaż w zbiorze S obiekt z najmniejszym priorytetem; w przypadku wielu obiektów z takim samym, najmniejszym priorytetem, wskaż dowolny z nich.

`Usuń_Min::` usuń ze zbioru S obiekt wskazany w wyniku wykonania operacji `Min`.

`Zmniejsz_Priorytet(`***k,p***`)::` zmniejsz priorytet obiektu k do nowej wartości p.

W naszym algorytmie wykonujmy $n-1$ operacji `Min`, $n-1$ operacji `Usuń_Min` oraz co najwyżej m operacji `Zmniejsz_Priorytet`, gdzie m jest równe sumie rozmiarów zbiorów sąsiadów $N(i)$ (co odpowiada liczbie krawędzi w grafie). W analizie czasu działania algorytmu musimy też uwzględnić inicjację zbioru S.

W jednej z najbardziej zaawansowanych implementacji algorytmu Dijkstry wykorzystuje się kopce Fibonacciego [3] do reprezentacji zbioru S. W implementacji tej nie zakłada się ograniczenia na długości krawędzi. Algorytm Dijkstry z kopcami Fibonacciego działa w czasie $O(m + n\log n)$. Niestety stopień złożoności tego algorytmu jest tak duży, że jest on trudny w analizie i słabo zachowuje się w praktyce. Najtrudniejsze w tej implementacji jest zapewnienie takiej organizacji zbioru S, żeby operację zmniejszenia priorytetu móc wykonywać w stałym (zamortyzowanym) czasie. Z drugiej strony, w ogólnym przypadku składnika $n\log n$ nie można zmniejszyć, ponieważ algorytm Dijkstry można użyć do sortowania. To ostatnie stwierdzenie pozostawiamy do uzasadnienia Czytelnikowi.

Zapoznamy się teraz z implementacją zbioru S, która jest nie tylko prosta, ale też nieźle zachowuje się w praktyce. Więcej, jej sprytna modyfikacja znajduje zastosowania w analizie różnego rodzaju sieci. W naszych rozwiązaniach istotnie wykorzystamy fakt, że odległości są liczbami całkowitymi ograniczonymi przez stałą c.

4.1. Rozwiązanie 1

Do reprezentacji zbioru S wykorzystamy tablicę $K[0..c(n-1)]$. Elementy tablicy K nazywamy **kubełkami**. Kubełek o indeksie i będzie przechowywał wszystkie obiekty, które aktualnie znajdują się w zbiorze S i których priorytety są równe i. Każdy kubełek reprezentujemy jako listę dwukierunkową po to, żeby nowe obiekty można wrzucać do kubełków w czasie stałym oraz by usuwać z nich w czasie stałym wskazane (na liście) obiekty. Dodatkowo zakładamy, że dana jest tablica $G[1..n]$ taka, że dla każdego obiektu $i \in S$, $G[i]$ wskazuje miejsce wystąpienia obiektu i (na liście) w kubełku $K[d[i]]$. Umożliwia to usuwanie w czasie stałym obiektu z zawierającego go kubełka.

Zastanówmy się teraz, w jaki sposób zaimplementować poszczególne operacje.

`Inicjacja::`

Wszystkie kubełki inicjujemy jako puste. Następnie każdego sąsiada j wierzchołka 1 wrzucamy do kubełka $K[d_1[j]]$, natomiast wszystkie pozostałe wierzchołki wrzucamy do kubełka o numerze $c(n-1)$. Wprowadzamy dodatkową zmienną *ost*, która będzie wskazywała, który z kubełków był oglądany jako ostatni. Zmiennej *ost* na początku nadajemy wartość 0, ponieważ do kubełka o numerze 0 wpadłby wierzchołek 1, choć tego nie czynimy. Koszt inicjacji wynosi zatem $O(cn)$.

`Min::`

Operacja `Min` polega na znalezieniu pierwszego niepustego pudełka z lewej strony, poczynając od pudełka $K[ost]$, i wskazaniu w nim jednego obiektu, np. pierwszego na liście obiektów umieszczonych w tym pudełku. Własność 1 gwarantuje, że do pominiętych kubełków nigdy nie będziemy wracać. Ponadto wskazany obiekt z najmniejszym priorytetem zostanie za chwilę usunięty ze zbioru S, a tym samym z kubełków, i nigdy do nich nie wróci.

`Usuń_Min::`

Operacja ta usuwa obiekt wskazany przez `Min`. Po prostu należy usunąć pierwszy obiekt z listy obiektów w kubełku $K[ost]$. W oczywisty sposób tę operację można wykonać w czasie stałym.

Możemy już podsumować łączny koszt wykonania operacji `Min` i `Usuń_Min`. Wynosi on $O(cn)$. Każda operacja `Usuń_Min` zabiera czas stały. Obiekt usunięty nigdy nie wraca do kubełków. W każdym kubełku spędzamy czas proporcjonalny do liczby zawartych w nim obiektów plus jeden. Ponieważ w poszukiwaniu niepustych kubełków przesuwamy się zawsze w prawo, to łączny koszt wykonania obu operacji jest równy $O(cn)$.

`Zmniejsz_Priorytet(k,p)::`

Operacja ta jest niesłychanie prosta. Wystarczy usunąć obiekt k z kubełka $K[d[k]]$ i przesunąć go do kubełka $K[p]$, zmieniając jednocześnie wartość $d[k]$

na p. Koszt wykonania tej operacji jest stały, a wykonanie m takich operacji zajmuje czas $O(m)$.

Podsumowując, łączny koszt wykonania algorytmu w implementacji rozwiązania 1 wynosi $O(m + cn)$. Algorytm w tej postaci jest znany pod nazwą algorytmu Diala [5]. To rozwiązanie dla małych c jest dość szybkie. Jego główną wadą jest tablica kubełków, która przy dużym, ale ciągle rozsądnym c, może być nieakceptowalna ze względu na swój rozmiar. Pokażemy teraz, w jaki sposób znacząco zmniejszyć rozmiar pamięci potrzebnej do implementacji zbioru S [5].

4.2. Rozwiązanie 2

Dla pokazania, że do implementacji kubełków nie jest do niczego potrzebna tak duża pamięć, wykorzystamy drugą własność algorytmu. Wiemy, że jeśli $K[ost]$ jest ostatnio oglądanym, niepustym kubełkiem, to następny niepusty kubełek będzie na pozycji o numerze co najwyżej $ost + c$. Więcej, jeśli obiekt k jest usuwany z $K[ost]$, to po wykonaniu operacji zmniejszenia priorytetów, wszyscy jego sąsiedzi ze zbioru S znajdą się także w kubełkach o numerach od ost do $ost + c$. Te fakty wykorzystamy do implementacji naszego algorytmu przy użyciu tylko $c + 1$ kubełków $K[0..c]$, po których będziemy wędrować cyklicznie. Jeśli ost jest indeksem ostatnio odwiedzanego, niepustego kubełka, to możemy przyjąć, że wszystkie obiekty ze zbioru S, które sąsiadują z jakimś obiektem spoza S, znajdują się już w kubełkach o numerach ost, $ost + 1 \bmod (c + 1)$,..., $ost + c \bmod (c + 1)$. Ponadto wiemy, że jeśli priorytety obiektów w kubełku $K[ost]$ są równe p, to w kolejnych (cyklicznie) kubełkach znajdują się obiekty o priorytetach odpowiednio $p + 1$, $p + 2$,..., $p + c$. Pozostaje jeszcze jeden mały problem do rozwiązania. W pierwszym podejściu, podczas inicjacji do kubełków wrzucono wszystkie obiekty. Teraz chcemy, żeby w kubełkach były tylko obiekty sąsiadujące z tymi spoza S. W nowym podejściu obiekt będzie się pojawiał w kubełkach dopiero wtedy, gdy wykryjemy po raz pierwszy, że sąsiaduje on z obiektem usuwanym z kubełków w wyniku operacji `Usuń_Min`. Musimy tylko zaznaczyć w jakiś sposób, że obiekt jest w S, ale poza kubełkami. Do tego celu wykorzystamy tablicę odległości d. Wartość – 1 w tablicy będzie oznaczała, że odpowiadający jej obiekt jest poza kubełkami. Oto zapis zaproponowanego rozwiązania.

```
Algorytm AD w małej pamięci
    (* Inicjacja *)
    zainicjuj kubełki K[0..c] jako puste;
    d[1] := 0;
    ost := 0;
    for i:= 2 to n do
        if (i ∈ N(1)) then
            d[i] := d1[i];
            umieść obiekt i w kubełku K[d[i]];
        else
```

```
        d[i] := -1;    (* d[i] = -1 oznacza obiekt zbioru S
                          nieumieszczony jeszcze w żadnym kubełku *)
(* główne obliczenia *)
for i := 1 to n - 1 do
    (* cykliczne poszukiwanie pierwszego niepustego kubełka *)
    while (kubełek K[ost] jest pusty) do
        ost := (ost+1) mod (c+1);
    (* Min *)
    j := obiekt z kubełka K[ost];
    (* Usuń Min *)
    usuń obiekt j z kubełka K[ost];
    for k ∈ N(j) do
        if (d[k] = -1) then
            umieść k w kubełku K[(ost+d_j[k]) mod (c+1)];
        else
            if (d[k] > (d[j] + d_j [k])) then
                (* zmniejsz priorytet *)
                usuń obiekt k zkubełka K[(ost+(d[k]-d[j])) mod (c+1)];
                d[k] := d[j] + d_j[k];
                umieść obiekt k w kubełku K[(ost+d_j[k]) mod (c+1)];
```

W ten sposób zmniejszyliśmy liczbę kubełków z $cn - 1$ do $c + 1$, zachowując (asymptotyczny) czas działania samego algorytmu. W następnym kroku pokażemy, w jaki sposób przyśpieszyć sam algorytm.

4.3. Rozwiązanie 3

Wróćmy do rozwiązania 1. Pokażemy teraz, że można przyśpieszyć opisany w nim algorytm, wykorzystując chwyt z zadania o kodowaniu permutacji. Dla prostoty opisu przyjmijmy, że liczba c jest kwadratem liczby naturalnej a, czyli $c = aa$. Niech $s = an$. Podzielmy kubełki $K[0..cn - 1]$ na s bloków $K_0, K_1,..., K_{s-1}$, każdy o długości a, gdzie $K_i = K[ia..ia + a - 1]$. Z każdym blokiem K_i wiążemy superkubełek S_i, który będzie służył do przechowywania obiektów z odległościami $d[i]$ wpadającymi do przedziału związanego z tym kubełkiem, czyli $[ia, ia + a - 1]$. W poszukiwaniu obiektu z najmniejszą odległością d przeglądamy superkubełki z lewa na prawo, aż znajdziemy pierwszy niepusty superkubełek. Przyjmijmy, że jest to kubełek S_i. Po znalezieniu niepustego superkubełka S_i, wszystkie zawarte w nim obiekty umieszczamy w kubełkach drugiego poziomu $L[0..a - 1]$, które w tym celu inicjujemy jako puste. W tym przypadku kubełek $L[0]$ służy do przechowywania obiektów z priorytetami równymi ia, kubełek $L[1]$ zawiera obiekty z priorytetami równymi $ia + 1$ itd. Teraz poszukiwania obiektów z najmniejszymi priorytetami dokonujemy w kubełkach L, aż do ich wyczerpania. Nowe obiekty wstawiamy albo do kubełków z L, jeśli związane z nimi odległości są z przedziału $[ia, ia + a - 1]$, albo do superkubełków z numerami większymi od i, gdy te odległości są większe od $ia + a - 1$. Podobnie postępujemy z obiektami, dla których wykonujemy operację zmniejszenia priorytetów. Zauważmy, że kubełki pomocnicze L są przetwarzane co najwyżej

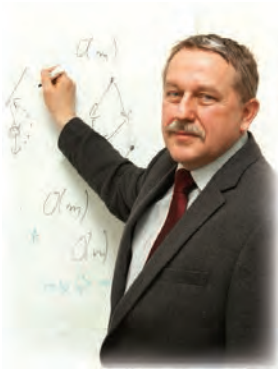
n razy – używamy ich tylko wtedy, gdy superkubełek jest niepusty, a to może się zdarzyć nie więcej niż n razy, bo tyle jest wszystkich obiektów. Zatem łączny koszt wykonania algorytmu przy użyciu dwupoziomowej struktury kubełków wynosi $O(m + na)$. Przeglądamy na superkubełków. Jeśli superkubełek jest pusty, to przechodzimy do następnego superkubełka. Jeśli superkubełek nie jest pusty, to inicjujemy kubełki L jako puste, umieszczamy w nich obiekty z superkubełka i przeglądamy kubełki L po kolei w poszukiwaniu obiektu z najmniejszym priorytetem. Kubełki L przetwarzamy nie więcej niż n razy, a czas potrzebny na ich przetworzenie jest rzędu an plus $n - 1$, bo $n - 1$ razy obiekty zostaną usunięte z kubełków L. Operację `Zmniejsz_Priorytet` wykonujemy w czasie stałym, podobnie jak w rozwiązaniu 1. Należy tylko uwzględnić dwupoziomową strukturę kubełków. Zauważmy na koniec, że możemy użyć tylko $O(a)$ kubełków, jeśli zastosujemy metodę z rozwiązania 2. Podsumowując, przedstawiliśmy implementację algorytmu Dijkstry działającą w czasie $O\left(m + n\sqrt{c}\right)$ i przy użyciu tylko $O\left(\sqrt{c}\right)$ kubełków. Ta implementacja pochodzi od Denardo i Foxa [4]. Dalsze rozwijanie opisanych tu pomysłów oraz zastosowanie wielopoziomowej struktury kubełków prowadzi do implementacji algorytmu Dijkstry działającej w czasie $O(m + n\log c)$ i przy użyciu tylko $O(\log c)$ kubełków [1].

5. Podsumowanie

W tym rozdziale przedstawiliśmy rozwiązania trzech problemów algorytmicznych z wykorzystaniem struktur kubełkowych. Dla każdego z tych problemów znane są asymptotycznie szybsze algorytmy, ale struktury danych w nich wykorzystywane są znacznie bardziej złożone, zarówno jeśli chodzi o ich implementację, jak i analizę. W przypadku kodowania permutacji są to wzbogacone drzewa wyszukiwań binarnych lub drzewa przedziałowe [3, 6], w przypadku zadania odwracanie – są to drzewa typu „splay" [2], a przypadku algorytmu Dijkstry – znajdują zastosowanie kopce Fibonacciego [3]. Każda z tych struktur danych wymagałaby odrębnego artykułu na jej opisanie. Co więcej, analiza ich zachowania wymaga dość złożonych technik analizy algorytmów. Struktury przedstawione w tym rozdziale wymagają tylko znajomości tablic oraz list, z którymi to strukturami ma szanse zapoznać się każdy początkujący programista. Mamy nadzieję, że metody projektowania algorytmów zaproponowane tutaj okażą się Czytelnikowi przydatne.

Literatura

1. Ahuja R.K., Mehlhorn K., Orlin J.B., Tarjan R.E., *Faster Algorithms for the Shortest Path Problem*, „J. ACM" 37(1990), 213-223
2. Banachowski L., Diks K., Rytter W., *Algorytmy i struktury danych*, WNT, Warszawa 2001
3. Cormen T.H., Leiserson Ch.E., Rivest R.L., Stein C., *Wprowadzenie do algorytmów*, WN PWN, Warszawa 2012
4. Denardo E.V., Fox F.L., *Shortest-route methods: Reaching, pruning, and buckets*, „Op. Res." 27(1979), 131-186
5. Dial R., *Algorithm 360: Shortest path forest with topological ordering*, „Comm. ACM" 12(1969), 632-633
6. Diks K., Malinowski A., Rytter W., Waleń T., *Moduł Algorytmy i struktury danych*, portal wazniak.mimuw.edu.pl
7. Misa T.J., *An Interview With Edsger W. Dijkstra*, „Comm. ACM" 53(2010), 41-47
8. Sysło M.M., *Algorytmy*, WSiP, Warszawa 1997
9. Zadania z CERC 2007, http://contest.felk.cvut.cz/07cerc/home/cteam06.zip, Praga 2007



autor zdjecia: Adrian Krawczyk

Prof. dr hab. Krzysztof Diks

jest pracownikiem Instytutu Informatyki Uniwersytetu Warszawskiego. Specjalizuje się w algorytmice, prowadząc przedmiot Algorytmy i struktury danych. Od roku 1994 jest związany z Olimpiadą Informatyczną, a od 1999 roku jest przewodniczącym Komitetu Głównego Olimpiady. W roku 2005 był przewodniczącym Komitetu Organizacyjnego Międzynarodowej Olimpiady Informatycznej, która odbyła się w Polsce. W roku 2012 współorganizował Finały Akademickich Mistrzostw Świata w Programowaniu Zespołowym. Jest pomysłodawcą i współorganizatorem Potyczek Algorytmicznych – najpopularniejszego konkursu algorytmicznego w Polsce. Autor wielu zadań konkursowych i popularyzator informatyki. Od roku 1999 wspólnie z prof. Janem Madeyem jest opiekunem reprezentacji Uniwersytetu Warszawskiego w konkursach programistycznych. Był współopiekunem mistrzów świata w programowaniu zespołowym w latach 2003 i 2007, oraz wicemistrzów świata z roku 2012.

diks@mimuw.edu.pl

Wojciech Śmietanka

w 2007 roku rozpoczął Jednoczesne Studia Informatyczno-Matematyczne na Uniwersytecie Warszawskim. Od początku studiów startował w zawodach programistycznych ACM ICPC zostając trzykrotnym drużynowym mistrzem Polski (2009, 2010, 2011), dwukrotnym mistrzem Europy Środkowej (2010, 2011) i wicemistrzem świata (2012). Te sukcesy, z wyjątkiem mistrzostwa z 2009 roku, odniósł wspólnie z Tomkiem Kulczyńskim i Kubą Pachockim, a nad przygotowaniami do zawodów ACM ICPC czuwali profesorowie Krzysztof Diks i Jan Madey oraz doktorzy Marek Cygan i Jakub Radoszewski. W czasie studiów zajmował się siatkówką, koszykówką i tańcem towarzyskim. Oprócz sukcesów drużynowych, kwalifikował się w roku 2010 do finałów konkursów Google Code Jam i TopCoder Open. Był także stażystą w firmach Google (Kraków) i Facebook (Palo Alto, CA).

wojciech.smietanka@gmail.com