Elementarz chakiera

Bartosz Szreder

Ostatnia aktualizacja 10 listopada 2012

Spis treści

0.1 Słowo na niedzielę

README

Niniejszy skrypt został napisany celem wyjaśnienia początkującym uczestnikom obozów informatycznych podstawowych algorytmów i sposobów ich implementacji. Założeniem autora tekstu jest pomoc w zrozumieniu i implementacji oraz udzielenie wskazówek dot. przydatności różnych metod w rozwiązywaniu przykładowych problemów algorytmicznych. Nie należy spodziewać się przesadnego nagromadzenia formalizmów, oczekiwać pełnych dowodów poprawności i złożoności. Działanie algorytmów i struktur będę próbował tłumaczyć intuicyjnie, nierzadko z wykorzystaniem konkretnych przykładów lub całych zadań.

Mimo częstego pojawiania się kodów źródłowych, skrypt **nie jest** przeznaczony do nauki języka jako takiego. Jedynym wyjątkiem od tej zasady może być tłumaczenie funkcjonowania elementów STL.

Zawarte kody nie są jedynie słuszną wersją implementacji algorytmu w żadnym z aspektów kodowania, od stylu indentacji i nazewnictwa począwszy, na modularyzacji skończywszy.

Wszelkie requesty, zauważone błędy (w tym literówki i niespójny styl programowania), propozycje, ciekawe pomysły i tym podobne proszę słać na adres `szreder [at] mimuw.edu.pl`. Mile widziani ochotnicy do tworzenia rysunków[1] i zgłaszania przykładowych zadań (mogą być wraz z rozwiązaniami jeśli nie są trywialne).

Rozpowszechnianie

. . . jest nieograniczone z zastrzeżeniem o niemodyfikowalności treści. Jeśli uważasz, że coś jest schrzanione i należy poprawić – napisz do mnie. Wszelkie inne czynności związane z kopiowaniem, drukowaniem, kserowaniem i tapetowaniem ścian są dozwolone, a nawet zachęcane. Tym niemniej sugerowany sposób rozpowszechniania elektronicznego to linkowanie adresu:

`http://students.mimuw.edu.pl/~szreder/skrypt.pdf`

Motywacja: dostęp do najświeższej wersji z wszystkimi aktualizacjami i poprawkami.

[1] Wymagania: kod kompatybilny z pakietem TikZ.

Część I

Trochę matematyki

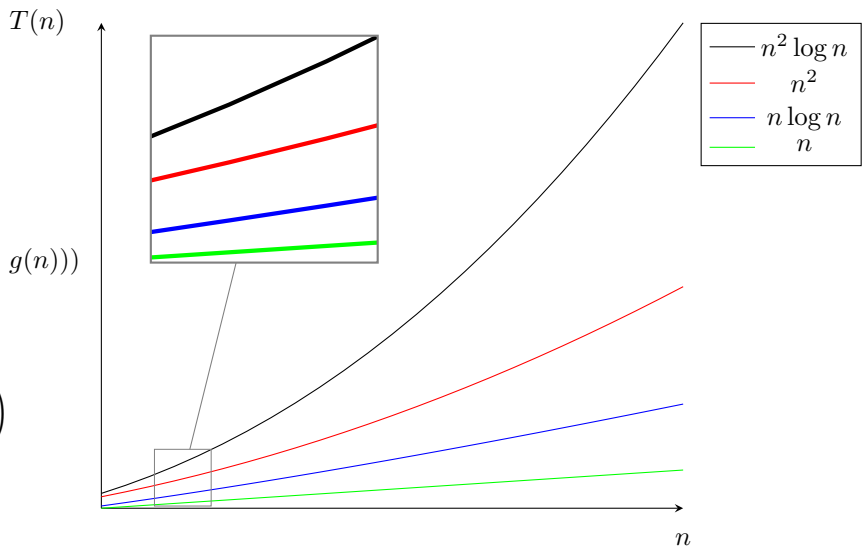
$$\log_a xy = \log_a x + \log_a y$$
$$\log_a \frac{x}{y} = \log_a x - \log_a y$$
$$\log_a x^k = k \cdot \log_a x$$
$$\log_b x = \frac{\log_a x}{\log_a b}$$

$$x^k = \begin{cases} x & k = 1 \\ (x^{\frac{k}{2}})^2 & 2 \mid k \\ (x^{\frac{k-1}{2}})^2 \cdot x & 2 \nmid k \end{cases}$$

$$\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max(f(n), g(n)))$$
$$\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$$
$$\mathcal{O}(\log n!) = \mathcal{O}(n \log n)$$

$$\begin{pmatrix} F_k \\ F_{k-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

$$\varphi\left(p^k\right) = p^k - p^{k-1}$$
$$a \perp b \Rightarrow \varphi(ab) = \varphi(a) \cdot \varphi(b)$$



1 Wszędobylski logarytm

Logarytm jest operacją odwrotną do potęgowania, tak jak dzielenie jest odwrotnością mnożenia, a odejmowanie odwrotnością dodawania. Zapis $\log_a b$ oznacza *logarytm o podstawie a z liczby b*. Jesli wynikiem takiego logarytmu jest pewna liczba c, to znaczy, że podstawa logarytmu podniesiona do potęgi c daje liczbę b, czyli:

$$\log_a b = c \iff a^c = b$$

Na przykład $\log_2 1024 = 10$, bo $2^{10} = 1024$. Czasami używa się skróconych oznaczeń na logarytmy o specjalnych podstawach, np. logarytm o podstawie 2 zapisuje się przez „lg", natomiast logarytm o podstawie e (jedna z najpopularniejszych stałych w przyrodzie, zaraz obok π) zapisuje się „ln" (tzw. *logarytm naturalny*). Obliczenie przybliżonego logarytmu można wykonać w drodze wielokrotnego dzielenia przez podstawę, np. $\log_{15} 750 = 1 + \log_{15} 50 = 2 + \log_{15} 3\frac{1}{3}$. Zatem $2 < \log_{15} 750 < 3$.

Pojęcie logarytmu będzie się często pojawiało w opisach algorytmów, zazwyczaj w kontekście czasu ich działania. Poniższy algorytm prezentuje pewną ideę obrazującą dlaczego tak się dzieje.

1.1 Wyszukiwanie binarne

Załóżmy, że mamy tablicę zawierającą n różnych wartości liczbowych, posortowanych rosnąco. Chcemy stwierdzić, czy w tablicy znajduje się pewna wartość x. Rozwiązanie brutalne polegałoby na sprawdzaniu liczb po kolei i przerwaniu w momencie, gdy albo znajdziemy x, albo dojdziemy bez sukcesu do końca tablicy. Nie wykorzystaliśmy w żaden sposób faktu, że zawartość tablicy jest posortowana. Możemy odrobinę poprawić algorytm przerywając jego działanie w chwili gdy stwierdzimy, że na pewno nie znajdziemy x, bo przekroczyliśmy już jego wartość i aż do końca tablicy będziemy mieli tylko elementy większe.

Nadal jednak mamy dużo miejsca na poprawę. Zróbmy tak, że weźmiemy **środkowy** element tablicy (lub ± 1 jeśli liczba elementów jest parzysta) i oznaczmy go y. Mamy teraz trzy możliwości:

1. $x = y$, więc wygraliśmy i koniec pracy.
2. $x < y$, wtedy wszystkie elementy na prawo od y także są większe od x, a zatem nas nie interesują – możemy o nich zapomnieć i skupić się na szukaniu x w lewej połowie tablicy.
3. $x > y$ to sytuacja analogiczna do powyższej, tylko teraz odrzucamy lewą połowę, bo tam na pewno nie ma x i kontynuujemy w prawej połowie.

W ten sposób odrzuciliśmy z rozważań połowę elementów tablicy. Jeśli teraz wykonamy identyczne działanie dla pozostałej części tablicy, to znowu odrzucimy połowę z pozostałych elementów itd. Każdy krok algorytmu dzieli dwukrotnie pozostały do sprawdzenia przedział w tablicy, a zatem liczba kroków algorytmu jest ograniczona przez logarytm o podstawie 2 z liczby elementów tablicy. Żeby docenić przyspieszenie wynikające z takiego działania należy dostrzec jak powoli rosną funkcje logarytmiczne:

- $\log_2 32 = 5$, mamy więc 5 kroków algorytmu zamiast sprawdzenia 32 elementów tablicy.
- $\log_2 1024 = 10$, czyli 10 kroków zamiast 1024 sprawdzeń.
- $\log_2 1\,048\,576 = 20$, czyli 20 kroków zamiast ponad miliona sprawdzeń!

To są jedynie logarytmy dwójkowe. Logarytmy o większych podstawach rosną dużo wolniej! Pokazuje to jak potężne są metody działające w oparciu o powyższą zasadę *wyszukiwania binarnego*.

1.2 Szybkie potęgowanie

Analogiczny pomysł do wyszukiwania binarnego możemy zastosować przy podnoszeniu liczb do wysokich potęg. Przyjmijmy, że chcemy obliczyć wartość x^n. Standardowa metoda polega na wykonaniu wielokrotnego mnożenia przez wartość x, zgodne ze schematem $x^n = x^{n-1} \cdot x$. Wykorzystajmy jednak sympatyczną właściwość mnożenia dwóch liczb o tych

samych podstawach potęgi: $x^n \cdot x^m = x^{nm}$, a w szczególności $x^n \cdot x^n = x^{2n}$. Możemy teraz zapisać algorytm potęgowania w formie rekurencyjnej:

$$x^n = \begin{cases} x & \text{dla } n = 1 \\ x^k \cdot x^k & \text{dla } n = 2k \\ x^k \cdot x^k \cdot x & \text{dla } n = 2k+1 \end{cases}$$

W oczywisty sposób rekurencja zbiegnie do wartości brzegowej $n = 1$ w logarytmicznej liczbie kroków.

1.3 Wyszukiwanie binarne po wyniku

Weźmy n-elementowy ciąg liczb naturalnych, który mamy podzielić na k spójnych przedziałów w taki sposób, aby każda liczba z ciągu znalazła się w dokładnie jednym przedziale. Oprócz tego chcemy, aby przedział o największej sumie elementów miał tę sumę możliwie najniższą.

Powyższe zadanie należy do problemów, które da się rozwiązać techniką wyszukiwania binarnego po wyniku. Idea jest następująca. Chcemy rozwiązać problem optymalizacyjny (znalezienie najmniejszej albo największej wartości spełniającej warunki zadania), jednak umiemy rozwiązać jedynie problem decyzyjny (czy określona liczba x spełnia warunki zadania). W postawionym zadaniu możemy „strzelić" w jakąś liczbę, stanowiącą górne ograniczenie na sumę elementów w pojedynczym przedziale. Następnie przechodzimy przez kolejne elementy ciągu i zachłannie przydzielamy je do bieżącego przedziału tak długo, jak to jest możliwe, tzn. dopóki biorąc kolejny element nie przekroczymy wybranego ograniczenia x. Jeśli przekroczymy ograniczenie, to w tym miejscu kończymy przedział i rozpoczynamy nowy. Na końcu sprawdzamy, czy w ten sposób „zużyliśmy" co najwyżej k przedziałów (wtedy wybrane ograniczenie x jest poprawne), czy może więcej (wtedy jest niepoprawne).

Jeśli strzeliliśmy w poprawną wartość x, to teraz strzelamy w jakąś mniejszą (bo może uda się poprawić wynik), w przeciwnym wypadku strzelamy w większą. Możemy w ten sposób znaleźć rozwiązanie problemu optymalizacyjnego poprzec sukcesywne rozwiązywanie problemu decyzyjnego, aż do zbiegnięcia wybieranych ograniczeń do pewnego punktu granicznego, tzn. takiej liczby x, że stanowi ona poprawne ograniczenie, ale $x - 1$ jest już niepoprawne.

Wyszukiwanie binarne po wyniku można stosować w miejscach, w których funkcja wyniku od przyjętej wartości rozwiązania jest monotoniczna. Jeśli nie mamy takiej gwarancji, to wyszukiwanie binarne może zgłupieć, odcinając przedziały zawierające optymalny wynik. Opisane wyżej przykładowe zadanie spełnia ten warunek – jeśli zwiększymy dopuszczalne ograniczenie na sumę elementów w pojedynczym przedziale, to liczba przedziałów potrzebnych do podzielenia wejściowego ciągu nigdy nie wzrośnie.

2 Asymptotyka

Analiza czasu działania programu komputerowego oraz ilości zużywanej pamięci pozwala na dobór odpowiedniego rozwiązania do postawionego problemu. Czasami mamy do dyspozycji więcej czasu, jednak musimy się zmieścić w ograniczonej pamięci lub odwrotnie. Chcielibyśmy określić pewną miarę dla zasobów wymaganych do działania algorytmu, żeby móc porównywać ze sobą różne rozwiązania i dowiedzieć się, które wybrać najlepiej.

Niestety dokładne wyliczenie liczby operacji jest zazwyczaj niemożliwe, ponieważ dla większości zadań jest ona zależna od danych wejściowych. Ponadto różne implementacje tego samego algorytmu mogą się znacząco różnić, wymuszając wielokrotne analizowanie kodów źródłowych. W większości wypadków wystarczy jednak pewne oszacowanie na rząd wielkości liczby najważniejszych operacji w algorytmie.

- *Operacja dominująca* jest pojedynczą operacją lub niezmiennym zestawem operacji, stanowiących trzon algorytmu. Dla przykładu operacją dominującą w większości algorytmów sortujących jest porównanie dwóch elementów.
- *Złożoność obliczeniowa*, zamiennie nazywana czasową, to liczba operacji dominujących wykonywanych przez wybrany algorytm do rozwiązania pewnego problemu dla danych wejściowych rozmiaru n. Ponieważ ten sam algorytm może wykonywać się diametralnie różnie dla różnych danych wejściowych tego rozmiaru, na ogół będziemy rozpatrywać złożoności czasową w dwóch wariantach: *pesymistycznym* (najbardziej złośliwe dane wejściowe) i *oczekiwanym* (przypadek średni).
- Analogicznie *złożonością pamięciową* nazywamy ilość pamięci wymaganej przez zastosowany algorytm do rozwiązania problemu dla danych wejściowych rozmiaru n. Przez większość tekstu omawiany będzie jedynie problem określania złożoności czasowej.

2.1 Notacja wielkiego 'O'

Rząd wielkości można pojmować intuicyjnie jako tempo wzrostu wykonywanych operacji dominujących wraz ze wzrostem rozmiaru danych wejściowych. Dla przykładu rozpatrzmy sortowanie bąbelkowe zaimplementowane następująco:

```
001 for (int i = 0; i < n; ++i)
002     for (int j = 1; j < n; ++j)
003         if (tab[j - 1] > tab[j]) {
004             int temp = tab[j];
005             tab[j] = tab[j - 1];
006             tab[j - 1] = temp;
007         }
```

Operacją dominującą jest porównanie dwóch elementów w tablicy `tab`. Zewnętrzna pętla wykonuje n obiegów, wewnętrzna $n-1$, co łącznie daje $n^2 - n$ porównań. Interesuje nas jedynie oszacowanie liczby wykonywanych operacji dominujących, a ponieważ wyraz n^2 zdecydowanie przeważa n już dla małych wartości naturalnych, odrzucamy ten drugi czynnik jako nieznaczący. Możemy teraz powiedzieć, że złożoność algorytmu sortowania bąbelkowego jest kwadratowa.

Do oznaczania rzędu wielkości złożoności obliczeniowej algorytmu posłużymy się symbolem wielkiej litery 'O'. $\mathcal{O}(f(n))$ oznacza, że algorytm ma złożoność **nie większą niż** $f(n)$. Bardziej formalnie: $g(n)$ jest rzędu nie większego, niż $f(n)$, jeśli istnieje taka stała c, że zachodzi $g(n) \leqslant c \cdot f(n)$ przy wystarczająco dużych wartościach n.

Dla uproszczenia stosuje się zapis $g(n) = \mathcal{O}(f(n))$ jeśli $g(n)$ jest rzędu nie większego, niż $f(n)$. Zastosowanie znaku równości jest pewnym nadużyciem — możemy zapisać $f(n) = \mathcal{O}(h(n))$ i $g(n) = \mathcal{O}(h(n))$, co jednak nie oznacza, że $f(n)$ i $g(n)$ są tego samego rzędu.

Przyjmijmy, że $f(n) = \mathcal{O}(h(n))$ i $g(n) = \mathcal{O}(h(n))$. Można wtedy określić kilka przydatnych własności wielkiego 'O':

- $f(x) + g(x) = \mathcal{O}(h(n))$
- $f(x) - g(x) = \mathcal{O}(h(n))$
- $f(x) \cdot g(x) = \mathcal{O}(h^2(n))$

Następujące formuły są zatem prawdziwe:

- $n^2 = \mathcal{O}(n^2)$ (oczywiste)
- $n = \mathcal{O}(n^2)$ (ale nieprawdziwe jest $n^2 = \mathcal{O}(n)$)
- $n^2 - n = \mathcal{O}(n^2)$ (bo $n = \mathcal{O}(n^2)$)
- $n^2 - \left(n^2 - n\right) = n = \mathcal{O}(n)$

Proste przykłady

Poniższe złożoności obliczeniowe uszeregowane są w kolejności rosnącej.

- $\mathcal{O}(1)$ oznacza algorytm wykonujący się *w czasie stałym*, czyli niezależnym od rozmiaru danych wejściowych. Rozwiązanie działające w czasie stałym istnieje dla bardzo małego zbioru problemów obliczeniowych, na przykład określenie parzystości liczby całkowitej lub znalezienie cyfry jedności operacji $n!$.
- $\mathcal{O}(\log n)$ oznacza algorytm wykonujący się *w czasie logarytmicznym*. W poprzednim rozdziale zostały przedstawione przykładowe algorytmy działające w takim czasie. Należy zwrócić uwagę na brak podstawy obok symbolu logarytmu – wiążę się to z tzw. *pominięciem stałego czynnika* w zapisie złożoności i zostanie opisane w dalszej części rozdziału.
- $\mathcal{O}(n)$ oznacza algorytm wykonujący się *w czasie liniowym* względem rozmiaru danych wejściowych, na przykład poszukiwanie konkretnej wartości w nieuporządkowanym n–elementowym ciągu liczb.
- $\mathcal{O}(n \log n)$ oznacza algorytm wykonujący się *w czasie liniowo-logarytmicznym*, czasami zwanym także *pseudoliniowym*. Większość algorytmów sortowania działa w takim czasie pesymistycznym albo oczekiwanym.
- $\mathcal{O}(n^2)$ oznacza algorytm wykonujący się *w czasie kwadratowym*. Proste algorytmy sortujące przez porównania działają w takim czasie: sortowanie bąbelkowe, sortowanie przez wybór, sortowanie przez wstawianie. Co ciekawe, jeden z najpopularniejszych algorytmów sortowania, czyli *sortowanie szybkie* (*QuickSort*) działa w pesymistycznym czasie kwadratowym, chociaż w przypadku średnim jest dużo szybszy od wielu algorytmów sortowania o pesymistycznej złożoności liniowo-logarytmicznej.

2.2 Pominięcie stałej

Załóżmy, że w pewnym algorytmie wykonujemy $g(n) = 3n^2 + n$ operacji dominujących. Zgodnie z opisem notacji wielkiego 'O', jeśli dla pewnej funkcji $f(n)$ istnieje stała c, taka że $g(n) \leqslant c \cdot f(n)$, to mamy $g(n) = \mathcal{O}(f(n))$. Wobec tego ustalmy $f(n) = n^2$ i zauważmy, że $3n^2 + n \leqslant 4 \cdot n^2$. Możemy wobec tego zapisać, że $g(n) = \mathcal{O}(n^2)$. Intuicyjne uzasadnienie jest takie, że przy zwiększaniu rozmiaru danych wejściowych liczba koniecznych operacji wciąż wzrasta kwadratowo, niezależnie od stałej stojącej przy wyrazie n^2.

Następujące formuły są zatem prawdziwe:

- $n^3 - 100n^2 = \mathcal{O}(n^3)$
- $\frac{3}{2}n + 25000 = \mathcal{O}(n)$
- $2147483647 = \mathcal{O}(1)$
- $0.00003n^2 = \mathcal{O}(n^2)$

Na tych przykładach można dostrzec pewne pułapki kryjące się w ocenianiu algorytmu jedynie według jego złożoności. Teoretycznie przykład czwarty jest dużo gorszy od trzeciego. W praktyce, dla dostatecznie małych wartości n, bardziej opłacalny jest algorytm opisany przykładem czwartym. Należy więc mieć na uwadze istnienie stałego mnożnika przy rozpatrywaniu kilku algorytmów dla pewnego problemu, ponieważ nie zawsze warto wybrać ten najszybszy „na papierze".

Należy tutaj wyjaśnić dwie kwestie związane ze złożonościami, w których pojawia się logarytm. Po pierwsze, standardowo pomija się w notacji podstawę logarytmu. Wynika to z tego, że przejście z jednej podstawy logarytmu do drugiej jest relatywnie prostą operacją. Jeśli chcemy sprowadzić logarytm o podstawie a z liczby x do logarytmu o podstawie b:

$$\log_b x = \frac{\log_a x}{\log_a b} = \log_a x \cdot \frac{1}{\log_a b}$$

Wartość $\frac{1}{\log_a b}$ jest po prostu stałą, którą pomijamy w zapisie złożoności, z czego wynika, że wszystkie logarytmy należą do tej samej klasy asymptotycznej. Trik związany z zamianą podstaw przydaje się czasami w życiu codziennym – większość podręcznych kalkulatorów potrafi obliczać jedynie logarytmy naturalne, zatem stosując zmianę podstawy potrafimy pośrednio obliczać logarytmy o dowolnych podstawach.

Druga kwestia związana jest z obliczaniem logarytmów z wartości będących potęgami. Łatwo sprawdzić, że zachodzi $\log_a n^c = c \cdot \log_a n$. Ale wtedy w zapisie złożoności mamy $\mathcal{O}(\log n^c) = \mathcal{O}(c \cdot \log n)$, co pozwala pominąć czynnik c o ile jest on stały (czyli nie jest parametrem algorytmu), sprowadzając w ten sposób złożoność do $\mathcal{O}(\log n)$. Nie należy jednak mylić takiego zapisu ze złożonością $\mathcal{O}((\log n)^2) = \mathcal{O}(\log^2 n)$.

3 Teoria liczb

3.1 Pierwszość i względna pierwszość

Liczba naturalna x jest *pierwsza* wtedy i tylko wtedy, gdy ma dokładnie dwa różne dzielniki (jedynkę oraz x). Liczby, które mają więcej niż dwa dzielniki nazywamy *złożonymi*. Sprawdzenie pierwszości najprościej wykonać brutalnie, tzn. testując czy $\forall_{1<i<x}\ x - i \cdot \left\lfloor \frac{x}{i} \right\rfloor \neq 0$. Ten bizantyjski zapis sprawdza, czy reszta z dzielenia x/i jest różna dla wszystkich wartości i z zakresu $[2, x-1]$ (o resztach z dzielenia więcej w rozdziale 3.2). Zapis ten jednak da się od razu przełożyć na prostą pętlę sprawdzającą pierwszość:

```
001 bool is_prime(int x)
002 {
003     if (x == 1)
004         return false;
005     if (x == 2)
006         return true;
007
008     for (int i = 2; i < x; ++i)
```

```
009         if (x - i * (x / i) == 0)
010             return false;
011     return true;
012 }
```

Takie podejście jest mocno głupawe i kosztowne. Możemy w zasadzie za darmo zaoszczędzić połowę pracy sprawdzając tylko liczby nie większe niż $x/2$, bo dalej i tak nie będzie żadnych dzielników. Kolejną połowę kandydatów do bycia dzielnikiem możemy odrzucić zauważając, że jedyną parzystą liczbą pierwszą jest dwójka. Zatem wszystkie liczby parzyste różne od 2 od razu odrzucamy, a pętlę zapuszczamy jedynie po liczbach nieparzystych. Oczywiście nie będziemy się wygłupiać i zamiast kombinować z zapisem z poprzedniego fragmentu kodu użyjemy operacji *modulo* (`%`), czyli reszty z dzielenia.

```
001 bool is_prime(int x)
002 {
003     if (x == 1)
004         return false;
005     if (x == 2)
006         return true;
007     if (x % 2 == 0)
008         return false;
009
010     for (int i = 3; i < x / 2; i += 2)
011         if (x % i == 0)
012             return false;
013     return true;
014 }
```

Dużo lepiej, ale nadal mamy czas $\mathcal{O}(n)$. Pora na kolejną, tym razem dużo silniejszą obserwację. Dzielniki pewnej liczby x można pogrupować w pary (a, b) w taki sposób, że $\frac{x}{a} = b$, a zatem jednocześnie $\frac{x}{b} = a$. Na przykład dla $x = 36$ mamy następujące pary: (1, 36), (2, 18), (3, 12), (4, 9), (6, 6), (9, 4), (12, 3), (18, 2), (36, 1). Uporządkowanie po rosnącej wartości a jest celowe, pozwala bowiem dostrzec bardzo przydatną prawidłowość: w momencie gdy $a > \sqrt{x}$, pary dzielników zaczną się powtarzać. Możemy zatem wszystkie dzielniki dowolnej liczby x odnaleźć (lub stwierdzić ich brak) sprawdzając jedynie liczby nie większe niż $\sqrt{x}$, a zatem mamy algorytm działający w czasie $\mathcal{O}(\sqrt{n})$.

Znajdowanie liczb pierwszych

Jeśli chcemy wyznaczyć liczby pierwsze z jakiegoś zakresu, to najprościej użyć powyższej funkcji `is_prime` dla kolejnych wartości x i zapisywać gdzieś na boku te liczby, dla których funkcja zwróciła `true`. Dla przyspieszenia działania możemy przerobić pętlę wewnątrz tej funkcji w taki sposób, żeby nie przebiegała przez wszystkie liczby nieparzyste od 3 do $\sqrt{x}$, a jedynie przez wcześniej znalezione liczby pierwsze.

Jeśli jednak zakres, w którym potrzebujemy liczb pierwszych jest na tyle niewielki, że zmieścimy w pamięci tablicę wartości logicznych odpowiadających na pytanie „czy x jest pierwsze”, to możemy zastosować sito Eratostenesa:

1. Ustalamy górne ograniczenie na szukane liczby pierwsze, oznaczmy je n.
2. Oznaczamy wszystkie liczby od 2 do n jako pierwsze. Później niektóre z nich będziemy wykreślać jako złożone.
3. Przebiegamy po tablicy w kolejności od 2 do n. Jeśli trafimy na liczbę x widniejącą jako pierwsza, to wykreślamy z tablicy wszystkie jej dalsze wielokrotności, czyli liczby postaci $2x$, $3x\ldots$

```
001 int primes[MAX], prime_cnt;
002 bool sieve[N]; //jeśli sieve[i] == true, to liczba i jest wykreślona
003
004 void calc_primes()
005 {
006     for (int i = 2; i < N; ++i)
007         if (!sieve[i]) {
008             //znaleźliśmy liczbę pierwszą
009             primes[prime_cnt++] = i;
010
011             //wykreślamy wszystkie jej wielokrotności
012             for (int j = i + i; j < N; j += i)
013                 sieve[j] = true;
```

```
014         }
015 }
```

Rozkład na czynniki pierwsze

Każdą liczbę naturalną można zapisać w postaci iloczynu liczb pierwszych, np. $300 = 2 \cdot 2 \cdot 3 \cdot 5 \cdot 5 = 2^2\, 3^1\, 5^2$. Ogólniej, jeśli przez p_i oznaczymy jakąś liczbę pierwszą, to możemy przedstawić każdą liczbę jako $x = p_1^{\alpha_1}\, p_2^{\alpha_2}\, \ldots\, p_k^{\alpha_k}$. Korzystając z tego zapisu można obliczyć kilka rzeczy.

- Każdy dzielnik liczby x to liczba postaci $p_1^{\gamma_1}\, p_2^{\gamma_2}\, \ldots\, p_k^{\gamma_k}$, gdzie $\forall_{1 \leqslant i \leqslant k}\ \gamma_i \leqslant \alpha_i$. Zatem możemy na $\alpha_1 + 1$ sposobów wybrać wykładnik przy liczbie p_1, niezależnie na $\alpha_2 + 1$ sposobów wybrać wykładnik przy liczbie p_2 itd., zawsze otrzymując w ten sposób liczbę będącą dzielnikiem x. Zatem liczba różnych dzielników x wynosi

$$\prod_{i=1}^{k} (\alpha_i + 1)$$

- Największy wspólny dzielnik liczb $a = p_1^{\alpha_1}\, p_2^{\alpha_2}\, \ldots\, p_k^{\alpha_k}$ i $b = p_1^{\beta_1}\, p_2^{\beta_2}\, \ldots\, p_k^{\beta_k}$ to

$$c = p_1^{\min(\alpha_1,\, \beta_1)}\, p_2^{\min(\alpha_2,\, \beta_2)}\, \ldots\, p_k^{\min(\alpha_k,\, \beta_k)}$$

- Analogicznie najmniejsza wspólna wielokrotność

$$c = p_1^{\max(\alpha_1,\, \beta_1)}\, p_2^{\max(\alpha_2,\, \beta_2)}\, \ldots\, p_k^{\max(\alpha_k,\, \beta_k)}$$

Algorytm Euklidesa

Największy wspólny dzielnik dwóch liczb można szybko znaleźć nie korzystając z rozkładu na czynniki pierwsze. Korzystamy z następujących własności podzielności liczb:

- 0 jest podzielne przez każdą liczbę niezerową, zatem $\mathrm{NWD}(a,\, 0) = a$ dla $a > 0$.
- jeśli $d \mid a$ i $d \mid b$, to $d \mid (a - b)$
- $\mathrm{NWD}(a,\, b) = \mathrm{NWD}(a - b,\, b)$

W najprostszym ujęciu powyższych własności, możemy zapisać algorytm znajdowania największego wspólnego dzielnika (*algorytm Euklidesa*) następująco:

```
001 //założenie: a >= b >= 0
002 int gcd(int a, int b)
003 {
004     while (b != 0) {
005         while (a >= b)
006             a -= b;
007 
008         int c = a;
009         a = b;
010         b = c;
011     }
012     return a;
013 }
```

Nazwa funkcji `gcd` pochodzi od angielskiego określenia największego wspólnego dzielnika – *greatest common divisor*. Powyższa implementacja jest dość wolna, możemy ją jednak znacząco przyspieszyć zamieniając wielokrotne odejmowanie na pojedynczą operację modulo.

```
001 int gcd(int a, int b)
002 {
003     while (b != 0) {
004         int c = a % b;
005         a = b;
006         b = c;
```

```
007     }
008     return a;
009 }
```

Względna pierwszość

Dwie liczby a i b nazywamy względnie pierwszymi wtedy i tylko wtedy, gdy NWD$(a, b) = 1$. Często jest to skrótowo oznaczane $a \perp b$. Dla liczb całkowitych dodatnich określamy funkcję φ (tzw. *funkcja Eulera*), zdefiniowaną następująco:

$$\varphi(x) = |\{0 < n \leqslant x \colon n \perp x\}|$$

Innymi słowy $\varphi(x)$ oznacza liczbę wartości nie większych od x i względnie pierwszych z x. Warunek $n \leqslant x$ (zamiast ostrzejszego $n < x$) może się wydawać dziwny, ponieważ wydaje się, że zawsze jeśli $n = x$, to $n \not\perp x$. Jest jednak istotny wyjątek w postaci liczby 1, bowiem zgodnie z definicją względnej pierwszości mamy NWD$(1, 1) = 1$. Zatem 1 jest względnie pierwsze z 1.

Obliczanie funkcji Eulera jest związane z rozkładem na czynniki pierwsze liczb. Wiąże się to z kilkoma obserwacjami dotyczącymi wartości funkcji Eulera:

- Jeśli x jest liczbą pierwszą, to wszystkie liczby mniejsze od x są względnie pierwsze, zatem $\varphi(x) = x - 1$.
- Wzmocnienie powyższej reguły, jeśli x jest liczbą pierwszą, to $\varphi(x^k) = x^k - x^{k-1}$. Wynika to z faktu, że wszystkie liczby nie większe od x^k, dla których NWD z tą wartością jest różne od jedynki, to wielokrotności liczby x: x, $2x$, $3x$, $\ldots$, $x^{k-1}x = x$. Jak widać jest ich dokładnie x^{k-1}.
- Funcja Eulera jest tzw. *funkcją multiplikatywną*, tzn. jeśli $a \perp b$, to $\varphi(ab) = \varphi(a) \cdot \varphi(b)$. Możemy skorzystać z tej własności aby obliczyć φ dla dowolnej liczby. Wystarczy dokonać rozkładu na czynniki pierwsze, a potem idzie już łatwo:

$$x = p_1^{\alpha_1}\, p_2^{\alpha_2}\, \ldots\, p_k^{\alpha_k}, \quad \varphi(x) = \prod_{i=1}^{k} (p_i^{\alpha_i} - p_i^{\alpha_i - 1})$$

3.2 Arytmetyka modularna

Operacja *modulo* oznacza resztę z dzielenia dwóch liczb całkowitych. Mówimy, że dwie liczby a i b są *przystające* modulo n wtedy i tylko wtedy, gdy reszta z dzielenia a/n (czyli $a \bmod n$) równa jest reszcie z dzielenia b/n (czyli $b \bmod n$). Przystawanie oznaczamy znakiem równości z trzema kreskami i podając na boku wartość n względem której obliczamy resztę:

$$a \equiv b \pmod{n} \iff a \bmod n = b \bmod n$$

Mówimy, że działamy w *arytmetyce modulo* n gdy wynik wszystkich obliczeń jakie wykonujemy traktujemy na końcu operacją wzięcia reszty z dzielenia przez n. Tak naprawdę sprowadzamy wtedy każdą liczbę całkowitą do równoważnej jej (przystającej) liczby ze zbioru $\{0, 1, \ldots, n-1\}$. Liczby ujemne sprowadzamy do nieujemnych wiedząc, że $a \equiv a + n \pmod{n}$, czyli na przykład $-2 \equiv 11 \pmod{13}$. Łatwo sprawdzić, że zachodzą następujące tożsamości:

- $(a \pm b) \bmod n = ((a \bmod n) \pm (b \bmod n)) \bmod n$
- $(a \cdot b) \bmod n = ((a \bmod n) \cdot (b \bmod n)) \bmod n$

Problem pojawia się dopiero, gdy chcemy dokonać dzielenia. Dzielenie jest odwrotnością mnożenia – każde dzielenie możemy zapisać jako mnożenie przez liczbę odwrotną:

$$a \div b = a \cdot \frac{1}{b} = ab^{-1}$$

Oczywiście przez liczbę odwrotną do b rozumiemy taką liczbę b^{-1}, że $b \cdot b^{-1} = 1$. Będziemy trzymać się tej definicji także w arytmetyce modulo n: $b \cdot b^{-1} \equiv 1 \pmod{n}$. Działając w arytmetyce modulo n wykonanie dzielenia przez dowolną liczbę x będzie zatem równoważne wykonaniu mnożenia przez x^{-1}. Znalezienie odwrotności jest czasami niemożliwe, na przykład w modulo 9:

1. $1 \cdot 1 \equiv 1 \pmod{9}$
2. $2 \cdot 5 \equiv 1 \pmod{9}$
3. nie ma odwrotności
4. $4 \cdot 7 \equiv 1 \pmod{9}$

5. $5 \cdot 2 \equiv 1 \pmod 9$
6. nie ma odwrotności
7. $7 \cdot 4 \equiv 1 \pmod 9$
8. $8 \cdot 8 \equiv 1 \pmod 9$

Nietrudno zauważyć, że istnienie odwrotności liczby x w arytmetyce modulo n jest równoważne $x \perp n$.

Równania liniowe

Znajdowanie odwrotności jest specjalnym przypadkiem rozwiązywania równań liniowych w arytmetyce modularnej. Równania takie mają następującą postać:

$$ax \equiv b \pmod n$$

Jeśli szukamy odwrotności liczby a, to rozwiązujemy takie równanie dla $b = 1$. Modularne równania liniowe mają rozwiązanie tylko, gdy $\mathrm{NWD}(a, n) \mid b$. Istnieje wtedy dokładnie $\mathrm{NWD}(a, n)$ różnych rozwiązań w zbiorze liczb $\{0, 1, \ldots, n-1\}$. Gdy znajdziemy przynajmniej jedno rozwiązanie s_0, to pozostałe rozwiązania możemy wyliczyć korzystając z wzoru

$$s_i = \left(s_0 + i \cdot \frac{n}{\mathrm{NWD}\,(a, n)} \right) \bmod n$$

Użyjemy *rozszerzonego algorytmu Euklidesa* do znalezienia interesujących nas liczb. Rozszerzony algorytm Euklidesa oprócz znalezienia największego wspólnego dzielnika liczb a i n potrafi również znaleźć parę liczb p, q spełniających równanie:

$$ap + nq = \mathrm{NWD}(a, n)$$

Mając taką parę możemy wyznaczyć rozwiązanie korzystając z wzoru

$$s_0 = \frac{b \cdot p}{\mathrm{NWD}(a, n)} \bmod n$$

Z warunku $\mathrm{NWD}(a, n) \mid b$ wynika, że powyższa liczba będzie zawsze całkowita.

Rozszerzony algorytm Euklidesa

Działanie algorytmu polega na obliczaniu x_i i y_i w równaniu $r_i = ax_i + by_i$ dla kolejnych wartości i, gdzie r_i oznacza resztę po i-tym kroku algorytmu Euklidesa. Kończymy, gdy reszta spadnie do zera (por. „zwykła" wersja algorytmu w p. 3.1). Zaczynamy od ustalenia $r_1 = a \cdot 1 + b \cdot 0$ i $r_2 = a \cdot 0 + b \cdot 1$ (zmienne `x`, `y`, `px` i `py` w poniższym programie przechowują odpowiednio bieżące wartości x_i, y_i oraz poprzednie x_{i-1}, y_{i-1}).

Uwaga. Poniższa implementacja rozszerzonego algorytmu Euklidesa nie zwraca żadnej wartości w wyniku wywołania funkcji. Wynika to z faktu, iż w zależności od okoliczności możemy jako „wynik" uznać zupełnie różne końcowe liczby. Największy wspólny dzielnik znajduje się na końcu w zmiennej `a`, natomiast zmienne `px` i `py` przechowują takie wartości x, y, że zachodzi $\mathrm{NWD}(a, b) = ax + by$. Wynikiem odpalenia rozszerzonego algorytmu Euklidesa będzie pewien podzbiór tych wartości – należy zdecydować które z nich są dla nas istotne i zawrzeć je w wyniku wywołania funkcji.

```
001 void extended_gcd(int a, int b)
002 {
003     int x = 0, y = 1, px = 1, py = 0;
004
005     while (b != 0) {
006         int c = a % b, d = a / b, temp;
007
008         a = b;
009         b = c;
010
011         temp = x;
012         x = px - d * x;
013         px = temp;
014
015         temp = y;
016         y = py - d * y;
017         py = temp;
018     }
```

```
019 }
```

4 Macierze

Macierz jest prostokątną tablicą o określonych wymiarach przechowującą wartości liczbowe[1]. Wymiary nazywamy wierszami i kolumnami, a same macierze zwykle oznaczamy wielkimi literami alfabetu. Pojedynczą komórkę macierzy oznaczamy taką samą literą, jak tę macierz, tylko małą. Macierz składającą się z jednej kolumny często nazywamy wektorem. Transpozycją macierzy $\mathbf{A}$ nazwiemy taką macierz $\mathbf{A^T}$, która odpowiada pierwotnej macierzy z zamienionymi wierszami i kolumnami, tzn. $\mathbf{a}_{i,j} = \mathbf{a^T}_{j,i}$ (tab. 4.1).

4.1 Macierze specjalnego typu

- Jeśli macierz ma tyle samo wierszy co kolumn, to wtedy jest *kwadratowa*.
- Macierz kwadratowa jest *symetryczna* wtedy, gdy zachodzi $\forall_i \forall_j \ \mathbf{a}_{i,j} = \mathbf{a}_{j,i}$.
- Macierz kwadratowa jest *diagonalna* wtedy, gdy wszystkie jej komórki poza tymi na przekątnej (*diagonali*) są równe zero: $\forall_i \forall_j \ i \neq j \Rightarrow \mathbf{a}_{i,j} = 0$. Analogicznie można zdefiniować macierz *antydiagonalną*, jeśli jedyne niezerowe komórki znajdują się na drugiej przekątnej, tzn. dla macierzy $n \times n$ zachodzi $\forall_i \forall_j \ i + j - 1 \neq n \Rightarrow \mathbf{a}_{i,j} = 0$.
- Macierz diagonalna o wszystkich wartościach na przekątnej równych 1 to macierz *jednostkowa*.
- Jeśli wszystkie wartości **powyżej** diagonali są zerowe, to mamy wtedy macierz *trójkątną górną*. Analogicznie definiujemy macierz *trójkątną dolną* jeżeli wszystkie wartości **poniżej** diagonali są zerowe.

4.2 Działania na macierzach

Działania na macierzach podlegają pewnym restrykcjom. Dwie macierze można dodać lub odjąć tylko jeśli są takiego samego wymiaru. Wtedy dokonujemy prostego dodawania (albo odejmowania) poszczególnych komórek:

$$\mathbf{C} = \mathbf{A} \pm \mathbf{B} \Leftrightarrow \forall_i \forall_j \ \mathbf{c}_{i,j} = \mathbf{a}_{i,j} \pm \mathbf{b}_{i,j}$$

Dużo ciekawsze jest mnożenie macierzy. Macierz $\mathbf{A}$ można pomnożyć przez macierz $\mathbf{B}$ tylko wtedy, gdy liczba kolumn $\mathbf{A}$ równa jest liczbie wierszy $\mathbf{B}$. Wynikiem mnożenia jest macierz, która ma tyle wierszy co $\mathbf{A}$ i tyle kolumn co $\mathbf{B}$. Przyjmijmy, że $\mathbf{A}$ wymiaru $p \times q$, a $\mathbf{B}$ jest wymiaru $q \times r$. Samo mnożenie zdefiowane jest następująco:

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} \Leftrightarrow \forall_{1 \leqslant i \leqslant p} \forall_{1 \leqslant j \leqslant r} \ \mathbf{c}_{i,j} = \sum_{k=1}^{q} \mathbf{a}_{i,k} \cdot \mathbf{b}_{k,j}$$

Ten dość skomplikowany zapis oznacza tyle, że obliczając wartość komórki w i-tym wierszu, j-tej kolumnie macierzy wynikowej dodajemy do siebie kolejno elementy z i-tego wiersza macierzy $\mathbf{A}$ przemnożone przez elementy j-tej kolumny macierzy $\mathbf{B}$. Łatwo zapamiętać ten sposób obliczeń, jeśli mnożąc dwie macierze jedną zapiszemy po lewej stronie, a drugą po prawej i trochę powyżej (tab. 4.2). Całą operację można zapisać za pomocą prostego kodu, który w istocie implementuje mnożenie macierzy wprost z definicji:

```
001 for (int i = 1; i <= p; ++i)
002     for (int j = 1; j <= r; ++j)
003         for (int k = 1; k <= q; ++k)
004             C[i][j] += A[i][k] * B[k][j];
```

Oczywiście złożoność takiej operacji wynosi $\mathcal{O}(pqr)$, lub $\mathcal{O}(n^3)$ jeśli mamy do czynienia z mnożeniem kwadratowych macierzy $n \times n$. Co ciekawe, taki sposób implementacji algorytmu jest wysoce nieefektywny ze względu na dość specyficzny sposób przechodzenia po pamięci. Wystarczy jednak zamienić miejscami dwie wewnętrzne pętle, aby otrzymać algorytm o równoważnym działaniu i zachowujący się dużo lepiej w praktyce.

[1] Tak naprawdę macierz może zawierać dużo różnych bytów matematycznych, ale skupiamy się tutaj tylko na liczbach rzeczywistych.

$$\mathbf{A} = \begin{pmatrix} 5 & 2 & -3 & 0 & 9 \\ 3 & 1 & 3 & -6 & 4 \\ 0 & -8 & 6 & 7 & 0 \end{pmatrix} \qquad \mathbf{A^T} = \begin{pmatrix} 5 & 3 & 0 \\ 2 & 1 & -8 \\ -3 & 3 & 6 \\ 0 & -6 & 7 \\ 9 & 4 & 0 \end{pmatrix}$$

$$\mathbf{a}_{1,1} = 5 \quad \mathbf{a}_{3.2} = -8 \quad \mathbf{a}_{2,5} = 4 \quad \mathbf{a^T}_{1,1} = 5 \quad \mathbf{a^T}_{3.2} = 3 \quad \mathbf{a^T}_{2,5} = \text{nie istnieje}$$

Tablica 4.1: Przykładowa macierz wymiaru 3×5 oraz transpozycja tej macierzy.

$$\mathbf{A} = \begin{pmatrix} 5 & 2 & -3 & 0 & 9 \\ 3 & 1 & 3 & -6 & 4 \\ 0 & -8 & 6 & 7 & 0 \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} 1 & 0 & 4 & 2 \\ 2 & 7 & 3 & 5 \\ 0 & 1 & 6 & 0 \\ -1 & -3 & 0 & 1 \\ 5 & -2 & 0 & 3 \end{pmatrix} \quad \mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} 54 & -7 & 8 & 47 \\ 31 & 20 & 33 & 17 \\ -23 & -71 & 12 & -33 \end{pmatrix}$$

$$\left(\begin{array}{c|c|cc} 1 & 0 & 4 & 2 \\ 2 & 7 & 3 & 5 \\ 0 & 1 & 6 & 0 \\ -1 & -3 & 0 & 1 \\ 5 & -2 & 0 & 3 \end{array}\right) = \mathbf{B}$$

$$\mathbf{A} = \left(\begin{array}{ccccc} 5 & 2 & -3 & 0 & 9 \\ \hline 3 & 1 & 3 & -6 & 4 \\ \hline 0 & -8 & 6 & 7 & 0 \end{array}\right) \quad \begin{pmatrix} \dots & \dots & \dots & \dots \\ \dots & \boxed{20} & \dots & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix}$$

$$\begin{aligned} \mathbf{c}_{2,2} &= \mathbf{a}_{2,1} \cdot \mathbf{b}_{1,2} + \mathbf{a}_{2,2} \cdot \mathbf{b}_{2,2} + \mathbf{a}_{2,3} \cdot \mathbf{b}_{3,2} + \mathbf{a}_{2,4} \cdot \mathbf{b}_{4,2} + \mathbf{a}_{2,5} \cdot \mathbf{b}_{5,2} \\ &= 3 \cdot 0 + 1 \cdot 7 + 3 \cdot 1 + (-6) \cdot (-3) + 4 \cdot (-2) \\ &= 0 + 7 + 3 + 18 - 8 = 20 \end{aligned}$$

Tablica 4.2: Mnożenie macierzy.

4.3 Równania rekurencyjne

W dość zaskakujący sposób można wykorzystać technikę mnożenia macierzy do szybkiego obliczania równań rekurencyjnych. Zacznijmy od prostego przykładu – liczb Fibonacciego zdefiniowanych następująco:

$$F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2} \text{ dla } n > 1$$

Obliczenie k-tego wyrazu ciągu można w oczywisty sposób przeprowadzić w czasie $\mathcal{O}(k)$. Co prawda istnieje wzór pozwalający na obliczanie dowolnej liczby Fibonacciego, jednak ma on dwie istotne wady: wymaga podnoszenia do potęgi (co wymaga czasu logarytmicznego) wartości niewymiernych (co ogólnie jest przykre).

Zapiszmy jednak pierwsze dwie wartości (F_0 i F_1) ciągu w postaci wektora, którego przemnożymy przez macierz kwadratową $\left(\begin{smallmatrix} 1 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ (oznaczmy ją $\mathbf{A}$). Współczynniki macierzy $\mathbf{A}$ są tak dobrane, aby w wyniku mnożenia otrzymać wektor wartości F_1 i F_2:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} F_1 + F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_2 \\ F_1 \end{pmatrix}$$

Jeśli będziemy kontynuować mnożenie tak otrzymanego wektora przez macierz $\mathbf{A}$, to wynikiem będą kolejne wartości ciągu Fibonacciego:

$$\underbrace{\mathbf{A} \cdot \mathbf{A} \cdots \mathbf{A}}_{k-1} \cdot \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \mathbf{A}^{k-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} F_k \\ F_{k-1} \end{pmatrix}$$

Przy takim zapisie już widać, że możemy macierz $\mathbf{A}$ podnieść do żądanej potęgi w czasie logarytmicznym, a następnie przemnożyć ją przez wektor inicjalnych wartości ciągu liczb Fibonacciego, osiągając w ten sposób dużo lepszy czas, niż standardowa, liniowa symulacja rekurencji.

Korzystając z tej techniki potrafimy radzić sobie z równaniami rekurencyjnymi takiej postaci:

$$T_n = c_1 T_{n-1} + c_2 T_{n-2} + \ldots + c_k T_{n-k} \quad \text{dla } n \geqslant k$$

oczywiście przy założeniu, że dane są wartości graniczne T_0, T_1, …, T_{k-1}. Zaczynamy od zbudowania wektora $\mathbf{v_0}$ zawierającego te wartości początkowe i konstruujemy macierz $\mathbf{A}$ w taki sposób, aby w wyniku działania $\mathbf{A} \cdot \mathbf{v_{k-1}}$

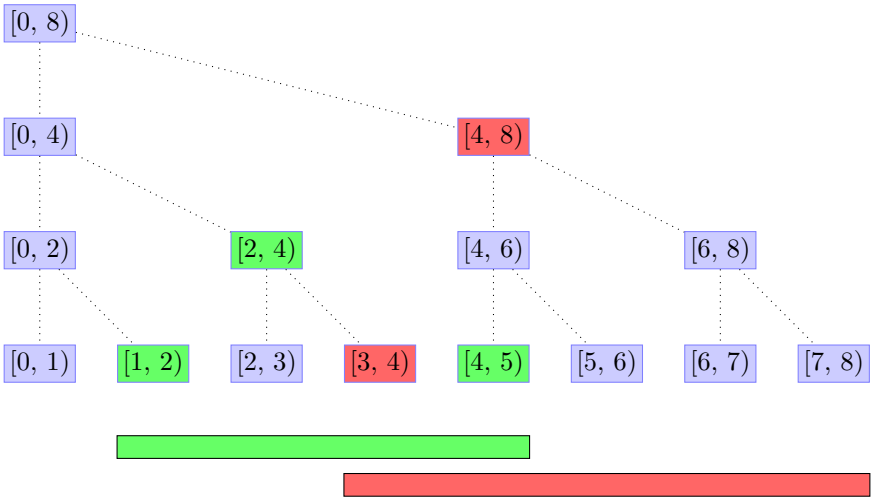
otrzymać wektor $\mathbf{v_k}$ zawierający wartości $T_1, T_2, \ldots, T_k$. Nietrudno sprawdzić, że macierz taka wygląda w ten sposób:

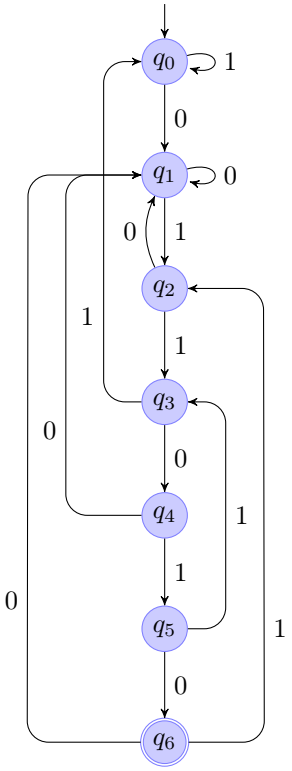
$$\mathbf{A} = \begin{pmatrix} c_1 & c_2 & \cdots & c_{k-1} & c_k \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix}$$

Teraz aby znaleźć szybko T_n musimy obliczyć $\mathbf{v_n} = \mathbf{A}^{n-k+1}\mathbf{v_{k-1}}$. Czyli potęgujemy macierz $\mathbf{A}$ w logarytmicznej liczbie kroków, przemnażamy przez wektor wartości początkowych i mamy wynik.

Część II

Struktury danych





5 Kolejki

Kolejki służą do kolejkowania (łał!) zadań, czynności itp. Kolejki dobiera się w zależności od potrzeb — zwykłe kolejki FIFO (*First In, First Out*) dobre są do przeszukiwania wszerz, ponieważ działają analogicznie do sklepów mięsnych: kto pierwszy, ten lepszy. Kolejki LIFO (*Last In, First Out*), zwane także stosem, są jak sterta talerzy do zmycia: najpierw bierzemy te z samej góry, nawet jeśli ktoś doniesie nowe, dlatego dobre są do nierekurencyjnej implementacji przeszukiwania w głąb — elementy na stosie reprezentują kolejne wywołania DFS na różnych głębokościach (czyli zdejmujemy ze stosu, kiedy rekurencyjny DFS musiałby wykonać krok w tył). Istotne są jeszcze kolejki *priorytetowe* tzn. takie, gdzie w pierwszej kolejności nie muszą być przetwarzane elementy pierwsze ani ostatnie, ale o maksymalnym priorytecie, który w dodatku może się zmieniać w czasie działania programu.

5.1 Kolejka prosta (FIFO)

Kolejkę First In, First Out (pierwszy na wejściu jest pierwszy na wyjściu) wygodnie implementuje się na bazie jednowymiarowej statycznej tablicy. W takim wypadku oprócz kolejkowanych elementów trzeba pamiętać tylko o położeniu początku i końca kolejki. Dodanie nowego elementu powoduje przesunięcie końca kolejki w przód, pobranie elementu przesuwa początek kolejki. W momencie spotkania początku kolejki z jej końcem nie ma już żadnych obiektów do przetworzenia.

Bardzo ostrożnie należy ustalić rozmiar tablicy na kolejkę obliczając wcześniej liczbę wszystkich możliwych obiektów w najbardziej pesymistycznym wypadku. Niewielkie przeoczenie może skutkować niedoborem pamięci na nowe elementy. Dobrą praktyką jest obliczenie takiej granicznej wartości i dodanie do niej jeszcze kilku elementów na „nieprzewidziane wypadki". Z drugiej strony zadeklarowanie zbyt dużej tablicy skończy się przekroczeniem dozwolonego limitu pamięci. Jeśli przy planowaniu algorytmu wygląda na to, że pamięci na potrzebną kolejkę z wszystkimi elementami będzie brakować, to prawdopodobnie należy szukać innego algorytmu. Sztuczki oszczędzające pamięć na kolejkach są rzadko wykorzystywane, chociaż całkiem przydatne.

Przykładowa implementacja

Jednowymiarowa tablica `Q` jest kolejką o początku w indeksie `head` i końcu `tail`. Stała `MAX` wyznacza górny limit liczby wszystkich elementów w kolejce (także tych już przetworzonych, ponieważ ta implementacja nie pozbywa się niepotrzebnych obiektów).

Funkcja `queue_front()` zwraca pierwszy element w kolejce i od razu przemieszcza początek kolejki do przodu (czyli jednocześnie usuwa z kolejki pobrany element); `queue_push(x)` wstawia na końcu kolejki nowy element x i przesuwa do przodu koniec kolejki; `queue_empty()` odpowiada, czy istnieje jakiś nieprzetworzony element. Czyszczenie kolejki to zwykłe wyzerowanie jej rozmiaru, realizowane w funkcji `queue_clear()`.

```
001 int Q[MAX];
002 int head = 0, tail = 0;
003
004 int queue_front()
005 {
006     return Q[head++];
007 }
008
009 void queue_push(int x)
010 {
011     Q[tail++] = x;
012 }
013
014 bool queue_empty()
015 {
016     return head == tail;
```

```
017 }
018
019 void queue_clear()
020 {
021     head = tail = 0;
022 }
```

5.2 Cykliczna FIFO

Jeśli przewidywana liczba wszystkich obiektów w kolejce jest zbyt duża na pomieszczenie w dozwolonej pamięci, ale wiemy, że nowe elementy do przetworzenia nie napływają przesadnie szybko, możemy nadpisywać już przetworzone obiekty nowymi w jednej i tej samej tablicy. W momencie kiedy kończy się miejsce w tablicy zaczynamy zapisywać nowe elementy na jej początku, wykorzystując miejsce po już niepotrzebnych, przetworzonych elementach.

Oczywiście procedurę zapisywania tablicy od początku można powtórzyć wielokrotnie w czasie działania algorytmu. Trzeba tylko pamiętać o zabezpieczeniu się przed nadpisywaniem jeszcze nieprzetworzonych elementów nowymi, bo wtedy bezpowrotnie traconych jest część obiektów w kolejce. W tym celu należy obliczyć sensowne (tzn. pozwalające na stworzenie mieszczącej się w pamięci tablicy) górne ograniczenie na największą możliwą liczbę skolejkowanych elementów.

Przykładowa implementacja

Jednowymiarowa tablica `Q` jest kolejką o początku w indeksie `head` i końcu `tail`. Stała `MAX` wyznacza górny limit liczby nieprzetworzonych elementów w kolejce.

Funkcja `queue_front()` zwraca pierwszy element w kolejce i od razu przemieszcza początek kolejki do przodu; `queue_push(x)` wstawia na końcu kolejki nowy element x i przesuwa do przodu koniec kolejki. Obie funkcje pomagają sobie dzieleniem modulo do łatwego obliczenia miejsca docelowego w tablicy reprezentującej kolejkę. Funkcja `queue_empty()` odpowiada, czy istnieje jakiś nieprzetworzony element, a `queue_clear()` czyści kolejkę — tak jak wcześniej, wystarczy wyzerowanie jej rozmiaru.

```
001 int Q[MAX];
002 int head = 0, tail = 0;
003
004 int queue_front()
005 {
006     int temp = Q[head % MAX];
007     ++head;
008     return temp;
009 }
010
011 void queue_push(int x)
012 {
013     Q[tail % MAX] = x;
014     tail++;
015 }
016
017 bool queue_empty()
018 {
019     return head == tail;
020 }
021
022 void queue_clear()
023 {
024     head = tail = 0;
025 }
```

5.3 FIFO wskaźnikowa (lista jednokierunkowa)

Jeśli górne ograniczenie na liczbę elementów jest trudne do wyznaczenia lub jest zbyt wysokie do zaalokowania statycznej tablicy, możemy zaimplementować kolejkę za pomocą wskaźników zamiast tablicy. Każdy pobrany element od razu usuwamy, zwalniając niepotrzebną pamięć.

Konstrukcja takiej struktury wymaga pamiętania wskaźników na pierwszy element w kolejce i ostatni element w kolejce, a ponadto każdy z elementów „środkowych" musi wiedzieć jaki jest kolejny element, na co potrzebujemy kolejnego wskaźnika. Nie ma oczywiście niczego za darmo — przydzielanie i zwalnianie pamięci jest dość czasochłonne, a konieczność posiadania wskaźników zwiększa zużycie pamięci.

Przykładowa implementacja

Struktura `queue_el` reprezentuje pojedynczy element w kolejce i składa się z dwóch pól: samego elementu (w tym przykładzie są to zmienne typu `int`) oraz wskaźnik na następny element. Początek i koniec kolejki wskazują odpowiednio `head` i `tail`.

Początkowo kolejka jest pusta (`head` wskazuje na `NULL`). Standardowo funkcja `queue_front()` zwraca pierwszy element w kolejce, uprzednio zwalniając zajmowane przezeń miejsce oraz przemieszcza głowę kolejki o jeden element naprzód; `queue_empty()` sprawdza, czy mamy w kolejce jakiś element. Funkcja `queue_push(x)` wstawia na końcu kolejki nowy element x i przesuwa do przodu koniec kolejki. Jeśli kolejka w momencie wywołania tej funkcji była pusta, to jest wpierw inicjowana pierwszym elementem. Czyszczenie (`queue_clear()`) wykonuje się niestety w czasie proporcjonalnym do liczby elementów w kolejce (wcześniej mieliśmy czas stały).

Można trochę oszukać odcinając głowę kolejki (`queue_dirty_clear()`), jednak powoduje to wycieki pamięci (nie usuwamy elementów, ale zapominamy o nich bezpowrotnie). Należy tego używać możliwie rzadko, w sytuacjach, gdy bardziej interesuje nas obniżenie czasu działania i mając pewność, że programowi wystarczy pamięci na dalsze alokacje.

```
001 struct queue_el {
002     int el;
003     queue_el *next;
004 };
005
006 queue_el *head = NULL, *tail = NULL;
007
008 int queue_front()
009 {
010     queue_el *temp = head;
011     int result = temp->el;
012
013     head = head->next;
014     delete temp;
015
016     return result;
017 }
018
019 bool queue_empty()
020 {
021     return head == NULL;
022 }
023
024 void queue_push(int x)
025 {
026     if (queue_empty()) {
027         //kolejka jest pusta, tworzymy jeden element,
028         //który staje się jednocześnie początkiem i końcem kolejki
029         head = new queue_el;
030         head->next = NULL;
031         head->el = x;
032         tail = head;
033     } else {
034         //tworzymy nowy element na końcu kolejki
035         tail->next = new queue_el;
036         tail = tail->next;
037         tail->next = NULL;
038         tail->el = x;
039     }
040 }
041
```

```
042 void queue_clear()
043 {
044     while (!queue_empty())
045         queue_front();
046 }
047
048 void queue_dirty_clear()
049 {
050     head = NULL;
051 }
```

5.4 LIFO (stos)

Kolejka Last In, First Out (ostatni na wejściu jest pierwszy na wyjściu), zwana często stosem, jest równie prosta w działaniu, co zwykła kolejka FIFO. Jedyna istotna różnica zawiera się w kolejności zdejmowania elementów ze stosu — nie zachodzi na początku, lecz na końcu. W ten sposób w pierwszej kolejności pobrane zostaną elementy najmłodsze. Z tej własności wynika najczęstsze zastosowanie stosu — nierekurencyjne implementowanie wielu operacji intuicyjnie rekurencyjnych, np. przeszukiwania w głąb.

Przykładowa implementacja

Jednowymiarowa tablica `S` jest stosem o początku w indeksie `0` i końcu (wysokości) `height`. Stała `MAX` wyznacza górny limit ilości nieprzetworzonych elementów na stosie.

Funkcja `stack_pop()` zwraca pierwszy element na stosie i od razu usuwa go oraz zmniejsza wysokość stosu. Funkcja `stack_push(x)` wstawia na stos nowy element x i zwiększa wysokość stosu; `stack_empty()` odpowiada, czy istnieje jakiś element na stosie. Czyszczenie analogicznie jak dla kolejek prostych na tablicach za pomocą wyzerowania rozmiaru w funkcji `stack_clear()`.

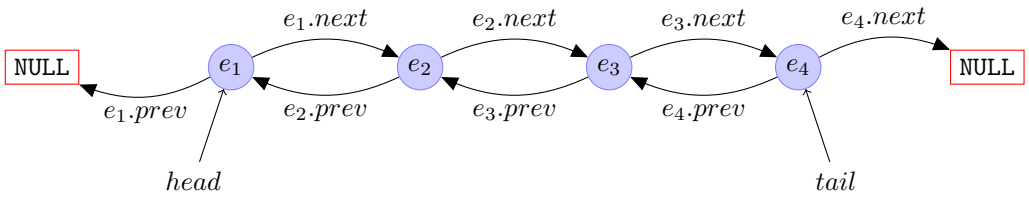
```
001 int S[MAX];
002 int height = 0;
003
004 int stack_pop()
005 {
006     return S[--height];
007 }
008
009 void stack_push(int x)
010 {
011     S[height++] = x;
012 }
013
014 bool stack_empty()
015 {
016     return height == 0;
017 }
018
019 void stack_clear()
020 {
021     height = 0;
022 }
```

5.5 Lista dwukierunkowa

Dotychczasowe kolejki służyły do reprezentowania ciągów, w których na określonym końcu wstawiamy kolejne elementy i z określonego (być może tego samego) końca możemy je zdejmować. Dość naturalnym rozwinięciem tej idei są *kolejki dwustronne* czyli takie, w których po obu stronach możemy wstawiać i usuwać elementy. Zamiast rozmieniać się na drobne pójdziemy od razu o krok dalej i zajmiemy się *listami dwukierunkowymi*, czyli strukturami, które umożliwiają szybkie wstawienie i usunięcie elementu w dowolnym miejscu ciągu, a nie tylko na końcu i początku. Przedstawiona dalej implementacja listy dwukierunkowej oparta będzie o wskaźniki i alokację potrzebnej pamięci w locie. Można zamiast

tego wykorzystać technikę analogiczną do pokazanej w 6.1 i uniknąć potencjalnie kosztownych dynamicznych alokacji pamięci.

Każdy element listy będzie zawierał dwa wskaźniki, odnoszące się do poprzedniego (*prev*) i następnego (*next*) elementu w liście. Podobnie jak w przypadku kolejki wskaźnikowej, gdzieś na boku zapamiętamy także wskaźniki na pierwszy (głowa – *head*) i ostatni (ogon – *tail*) element w liście. Dzięki temu będziemy potrafili dodawać i usuwać elementy na początku i końcu. Wstępnie ustalmy, że wskaźnik na poprzedni element w głowie listy oraz wskaźnik na następny element w ogonie mają wartość `NULL` (rys. 5.1). Ustalmy także, że lista pusta reprezentowana jest przez wskaźnik *head* wskazujący `NULL`, czyli tak samo jak w kolejce wskaźnikowej z rozdziału 5.3.



Rysunek 5.1: Schemat działania dwukierunkowej listy wskaźnikowej.

Skupimy się na operacjach działających na głowie listy. Odpowiadające im operacje działające na ogonie są symetryczne. Wstawienie nowego elementu e_0 przed element e_1 (wskazywany przez *head*) wymaga następujących aktualizacji:

- $e_0.prev$ ustawiamy na `NULL` (nie ma niczego przed e_0),
- $e_0.next$ ustawiamy na *head* (czyli $e_0.next$ wskazuje na e_1),
- *head.prev* (czyli $e_1.prev$) ustawiamy na e_0,
- zapamiętujemy, że nową głową kolejki jest e_0.

Jeśli natomiast lista była początkowo pusta, to ustawiamy wskaźniki $e_0.prev$ i $e_0.next$ na `NULL` oraz *head* i *tail* na e_0.

Usunięcie głowy listy (czyli tak naprawdę elementu e_1), po której występuje element e_2 wygląda następująco:

- ustawiamy głowę na *head.next* (czyli na e_2),
- zwalniamy pamięć pod adresem *head.prev* (czyli usuwamy $e_2.prev$, czyli e_1),
- *head.prev* ustawiamy na `NULL`.

Należy uważać na sytuację, w której usuwamy jedyny element listy, bo wtedy *head.next* przed wykonaniem pierwszego kroku będzie miało wartość `NULL`. A zatem w drugim kroku odwołanie *head.prev* spowoduje nielegalne odwołanie do pamięci i w efekcie zabicie programu. Przypadek jednoelementowej listy wykrywamy sprawdzając czy *head* i *tail* wskazują na ten sam adres.

Jak już wcześniej zostało wspomniane, nie ograniczymy się do wstawiania i usuwania elementów jedynie na początku lub końcu listy. Mając wiedzę o elemencie poprzednim i następnym możemy w czasie stałym dokonywać modyfikacji w dowolnym miejscu listy. Przyjrzyjmy się najpierw wstawieniu: powiedzmy, że pomiędzy e_1 i e_2 chcemy wstawić pewien element v. Kroki, które musimy wykonać układają się następująco:

- Powiązujemy e_1 z v, a zatem ustawiamy $e_1.next$ na v oraz $v.prev$ na e_1.
- Analogicznie powiązujemy e_2 z v, czyli ustawiamy $e_2.prev$ na v oraz $v.next$ na e_1.

Usuwanie wygląda bardzo podobnie do wyrzucania elementów z krańców listy. Przyjmijmy, że element v wstawiony pomiędzy e_1 i e_2 chcemy teraz usunąć. Po kolei:

- Wiążemy ze sobą e_1 i e_2 z pominięciem v, czyli $e_1.next$ ustawiamy na e_2 oraz $e_2.prev$ ustawiamy na e_1.
- Sprzątamy pamięć po v.

Przykładowa implementacja #1

Definiujemy typ złożony `list_el` do reprezentowania pojedynczego elementu w liście (dla uproszczenia przyjmujemy w tym przykładzie, że będą to liczby całkowite). Oprócz samej liczby (zmienna typu `int`) potrzebne są dwa wskaźniki, `prev` i `next`. Oprócz tego trzymamy dwie zmienne wskazujące na początek i koniec kolejki, odpowiednio `head` i `tail`.

Początkowo lista jest pusta (`head` wskazuje na `NULL`). Funkcja `list_empty()` sprawdza, czy mamy w kolejce jakiś element; `list_front()` i `list_back()` zwracają odpowiednio pierwszy albo ostatni element listy i usuwają go z listy zwalniając miejsce po nim. Funkcje `list_push_front(x)` i `list_push_back(x)` wstawiają odpowiednio na początku albo końcu listy nowy element x. Czas potrzebny na wyczyszczenie listy (`list_clear()`) jest proporcjonalny do liczby elementów w liście. Nietrudno byłoby napisać funkcję wstawiającą nowy element w środku kolejki, a nie tylko na początku albo końcu.

```
001 struct list_el {
002     int el;
003     list_el *prev, *next;
004 };
005
006 list_el *head = NULL, *tail;
007
008 bool list_empty()
009 {
010     return head == NULL;
011 }
012
013 void list_erase(list_el *ptr)
014 {
015     if (ptr == head || ptr == tail) {
016         if (ptr == head && ptr == tail) {   //usuwamy jedyny element
017             head = tail = NULL;
018         } else if (ptr == head) {           //usuwamy głowę
019             head = head->next;
020             head->prev = NULL;
021         } else {                            //usuwamy ogon
022             tail = tail->prev;
023             tail->next = NULL;
024         }
025     } else {
026         ptr->prev->next = ptr->next;
027         ptr->next->prev = ptr->prev;
028     }
029
030     delete ptr;
031 }
032
033 void list_clear()
034 {
035     while (!list_empty())
036         list_erase(head);
037 }
038
039 void list_push_front(int x)
040 {
041     list_el *ptr = new list_el;
042     ptr->el = x;
043
044     if (list_empty()) {
045         ptr->prev = ptr->next = NULL;
046         head = tail = ptr;
047     } else {
048         ptr->prev = NULL;
049         ptr->next = head;
```

```
050         head->prev = ptr;
051         head = ptr;
052     }
053 }
054
055 void list_push_back(int x)
056 {
057     list_el *ptr = new list_el;
058     ptr->el = x;
059
060     if (list_empty()) {
061         ptr->prev = ptr->next = NULL;
062         head = tail = ptr;
063     } else {
064         ptr->next = NULL;
065         ptr->prev = tail;
066         tail->next = ptr;
067         tail = ptr;
068     }
069 }
070
071 int list_front()
072 {
073     int result = head->el;
074     list_erase(head);
075     return result;
076 }
077
078 int list_back()
079 {
080     int result = tail->el;
081     list_erase(tail);
082     return result;
083 }
```

Listy cykliczne ze strażnikiem

Jak widać powyżej, standardowa implementacja list dwukierunkowych generuje dość sporo przypadków szczególnych, z którymi trzeba uważnie działać. Najsilniej widać to w przypadku wycinania elementów z listy: usuwanie głowy czy ogona wymaga większej ostrożności w działaniu niż usunięcie elementu ze środka listy.

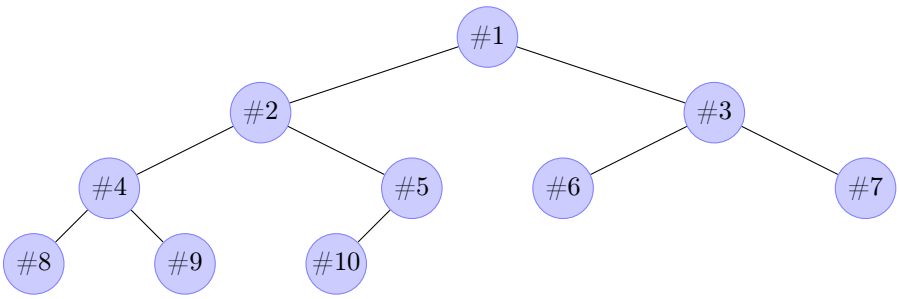
Dokonamy niewielkiej modyfikacji pomysłu na listę. Zamiast wyróżnionego początku i końca zwiniemy listę w cykl i wstawimy sztuczny element (tzw. *strażnika*, ang. *sentinel*). Sam strażnik nigdy nie będzie przechowywał znaczącej dla nas wartości, ale jego obecność i „nieusuwalność" sprawi, że nagle pozbędziemy się wszystkich przypadków szczególnych:

- Prawdziwą głową będzie pierwszy element „na prawo" (`next`) od strażnika, a ogonem element „na lewo" od strażnika (`prev`). A zatem pustą listę reprezentujemy jako listę składającą się wyłącznie ze strażnika. Zachodzi wtedy `sentinel->next == sentinel` i `sentinel->prev == sentinel`.
- Ponieważ żaden ze wskaźników `next` i `prev` w elementach listy nie jest `NULL`, to wstawianie na początku i na końcu nie różni się niczym od wstawienia elementu w środku listy.
- Analogiczny argument dotyczy usuwania elementów z listy.

Przykładowa implementacja #2

Strażnikiem jest pojedynczy element kolejki `sentinel`. Przed pierwszym użyciem należy zainicjować strażnika ustawiając jego wskaźniki na kolejny i poprzedni element listy na samego siebie (funkcja `list_init()`). Operacje wstawienia elementu na początek (`list_push_front()`) i koniec listy (`list_push_back()`) korzystają z pomocniczej funkcji `list_insert(after, x)`, która powoduje wstawienie nowego elementu x zaraz za elementem wskazywanym przez argument `after`. Łatwo się przekonać metodą kartkologiczno-rysunkową, że zastosowanie strażnika eliminuje wszelkie przypadki szczególne zarówno przy wstawianiu jak i usuwaniu elementów.

```
001 struct list_el {
002     int el;
003     list_el *prev, *next;
004 };
005
006 list_el sentinel;
007
008 void list_init()
009 {
010     //elementem poprzednim i następnym względem strażnika
011     //jest początkowo on sam
012     sentinel.prev = &sentinel;
013     sentinel.next = &sentinel;
014 }
015
016 bool list_empty()
017 {
018     return sentinel.next == &sentinel;
019 }
020
021 void list_erase(list_el *ptr)
022 {
023     ptr->prev->next = ptr->next;
024     ptr->next->prev = ptr->prev;
025     delete ptr;
026 }
027
028 void list_clear()
029 {
030     while (!list_empty())
031         list_erase(sentinel.next);
032 }
033
034 void list_insert(list_el *after, int x)
035 {
036     list_el *ptr = new list_el;
037     ptr->el = x;
038     ptr->prev = after;
039     ptr->next = after->next;
040
041     after->next->prev = ptr;
042     after->next = ptr;
043 }
044
045 void list_push_front(int x)
046 {
047     list_insert(&sentinel, x);
048 }
049
050 void list_push_back(int x)
051 {
052     list_insert(sentinel.prev, x);
053 }
054
055 int list_front()
056 {
057     int result = sentinel.next->el;
058     list_erase(sentinel.next);
059     return result;
060 }
```



Rysunek 5.2: Kolejność indeksowania węzłów w kopcu.

	tablica	posortowana tablica	kopiec
usunięcie elementu	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log\ n)$
dodanie elementu	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log\ n)$
znalezienie maksimum	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Tablica 5.1: Asymptotyczny czas działania operacji na poszczególnych implementacjach kolejki priorytetowej.

```
061
062 int list_back()
063 {
064     int result = sentinel.prev->el;
065     list_erase(sentinel.prev);
066     return result;
067 }
```

5.6 Kolejki priorytetowe

Elementy w kolejce charakteryzują się pewną wartością liczbową (*priorytetem*), która ustala rzeczywistą kolejność przetwarzania obiektów. To znaczy, że może dojść do sytuacji, kiedy niedawno dodany element przetwarzany jest wcześniej od innego, przebywającego w kolejce o wiele dłużej.

Najprostsza implementacja takiej kolejki, czyli zwykła jednowymiarowa tablica, w praktyce okazuje się za wolna. Wynika to z oczywistego faktu — aby znaleźć element o maksymalnym priorytecie należy przeszukać całą tablicę. Znaczną poprawę czasu operacji na kolejce można osiągnąć stosując tzw. *kopiec binarny*, czyli odmianę drzewa binarnego (patrz rozdział 7). Porównanie czasu działania poszczególnych elementarnych operacji znajduje się w tablicy 5.1.

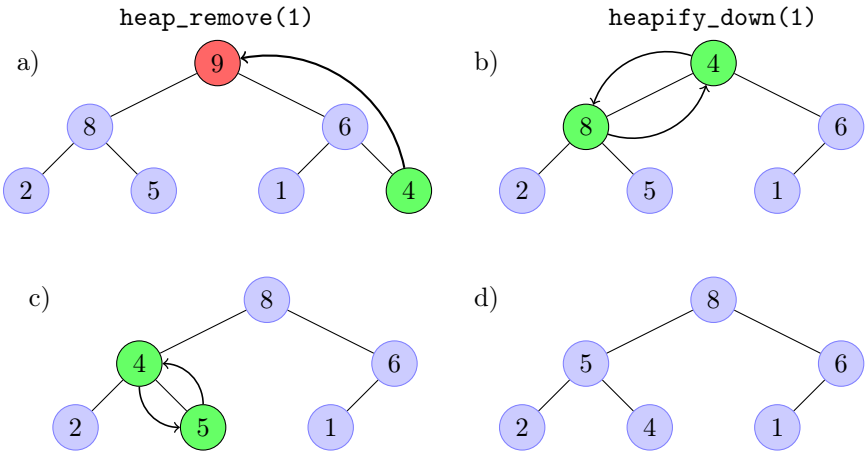
Kopiec jest zawsze zrównoważony, tzn. różnica maksymalnej głębokości (wysokości) dwóch podkopców nie przekracza 1. Kolejne elementy wstawiane są „warstwowo", tak jak na rysunku 5.2. Jak łatwo zauważyć, lewe dziecko węzła o indeksie i ma numer $2i$, zaś prawe $2i+1$ (jeśli uznajemy, że korzeń ma numer 1). I oczywiście w drugą stronę, rodzic węzła i ma numer $\lfloor \frac{i}{2} \rfloor$.

Główną właściwością kopca jest przechowywanie maksymalnego (ze względu na priorytet) elementu w jego korzeniu. Czyli dzieci (o ile istnieją) mają niższą wartość priorytetu. Jednakże każde dziecko wyznacza nowy podkopiec, który także ma własność kopca, czyli lewe dziecko korzenia jest maksymalne ze względu na priorytet w całym lewym podkopcu, prawe dziecko jest maksymalne w całym prawym podkopcu. Z takiej konstrukcji otrzymujemy stały czas wyciągania następnego elementu — jest on od razu dany w korzeniu kopca.

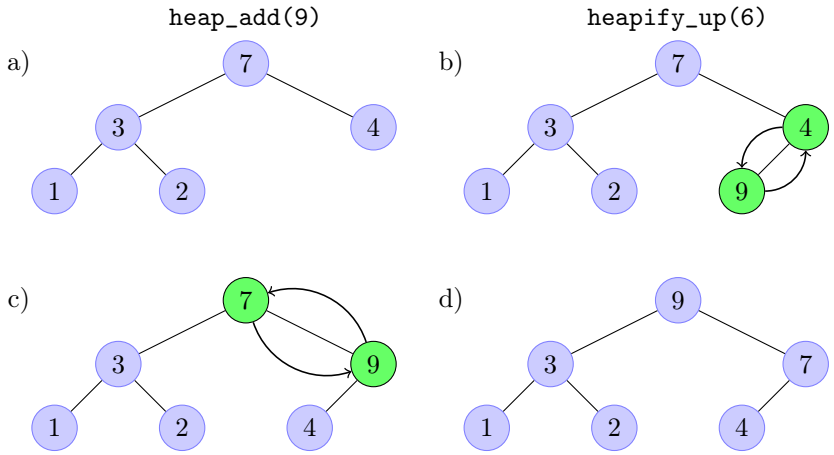
Logarytmiczny czas pozostałych operacji wynika z konieczności przywracania własności kopca wskutek jej zaburzenia powstałego po usunięciu lub wstawieniu elementu. Usunięcie węzła ze środka kopca powoduje powstanie nieakceptowalnej dziury, którą należy czym prędzej załatać. Łatanie z kolei może spowodować zaburzenie własności kopca, np. jeśli w miejsce wyrwy przeniesiony zostanie element z końca kopca o niskim priorytecie (wtedy należy zrzucić go w dół) lub gdy na koniec kopca dorzucony zostanie obiekt o wysokim priorytecie (tego z kolei trzeba wywindować).

Ponieważ liczność kolejnych warstw kopca to potęgi dwójki, wysokość H wynosi $\lfloor \log_2 n \rfloor + 1$ (gdzie n — ilość elementów w kopcu). Przemieszczanie elementu w górę lub w dół wykonuje się co najwyżej $H = \mathcal{O}(\log n)$ razy, z czego otrzymujemy logarytmiczny czas operacji usuwania i wstawiania elementu.

Strukturę można wykorzystać np. do implementacji sortowania. W korzeniu kopca zawsze znajduje się element o maksymalnym priorytecie (czyli o największej lub najmniejszej wartości w zależności od „kierunku" sortowania), który należy zdjąć z kopca i umieścić w oddzielnej pamięci przeznaczonej na posortowane elementy, a następnie przywrócić własność kopca. Ponownie w korzeniu mamy maksymalny element, który zdejmujemy itd. aż do przetworzenia wszystkich elementów w kopcu.



Tablica 5.2: Usuwanie elementu z kopca.



Tablica 5.3: Wstawianie elementu do kopca.

Przykładowa implementacja

Tablica liczb całkowitych `heap` reprezentuje kopiec o rozmiarze `heap_size`. Funkcja `heap_top()` zwraca element w korzeniu kopca, `heap_remove(i)` usuwa element w `heap[i]`, `heap_add(x)` dodaje element x na końcu kopca. Pomocnicze funkcje `heapify_up(i)` oraz `heapify_down(i)` pomagają przywrócić zaburzoną własność kopca poprzez przemieszczanie elementu w indeksie i odpowiednio w górę lub w dół; `heap_swap(i, j)` zamienia miejscami elementy w kopcu o indeksach i i j; `heap_empty()` odpowiada, czy istnieje jakiś element w kopcu. Przykład działania funkcji `heap_remove` i `heap_add` można zobaczyć na rysunkach 5.2 i 5.3. Standardowe czyszczenie w `heap_clear()`. Dla wygody indeksuję kopiec od jedynki, a nie od zera, jak przykładowe implementacje wcześniejszych kolejek.

Sposób użycia

- `heap_add(x)` dodaje liczbę x do kopca
- `heap_top()` zwraca korzeń kopca
- usunięcie elementu o najwyższym priorytecie (czyli korzenia) realizuje wywołanie `heap_remove(1)`

```
001 int heap[MAX];
002 int heap_size = 0;
003
004 void heap_swap(int i, int j)
005 {
006     int temp = heap[i];
007     heap[i] = heap[j];
008     heap[j] = temp;
009 }
```

```
010
011 void heapify_up(int i)
012 {
013     int current = i, parent = i / 2;
014
015     //czy rodzic (o ile istnieje) ma niższy priorytet?
016     //jeżeli tak - zamiana miejsc
017     //i badamy własność kopca wyżej
018     if (parent > 0 && heap[parent] < heap[current]) {
019         heap_swap(current, parent);
020         heapify_up(parent);
021     }
022 }
023
024 void heapify_down(int i)
025 {
026     int parent = i, left = i * 2, right = i * 2 + 1;
027     int best;
028
029     //jeśli lewy potomek istnieje,
030     //to czy ma wyższy priorytet od rodzica?
031     if (left <= heap_size && heap[left] > heap[parent])
032         best = left;
033     else
034         best = parent;
035
036     //jeśli prawy potomek istnieje,
037     //to czy ma wyższy priorytet od rodzica i lewego potomka?
038     if (right <= heap_size && heap[right] > heap[best])
039         best = right;
040
041     //jeśli to nie rodzic ma najwyższy priorytet,
042     //to zamieniamy miejscami z maksymalnym węzłem
043     //i badamy własność kopca poniżej
044     if (best != parent) {
045         heap_swap(best, parent);
046         heapify_down(best);
047     }
048 }
049
050 void heap_remove(int i)
051 {
052     //nadpisuję element usuwany ostatnim i zmniejszam kopiec
053     heap_swap(i, heap_size);
054     --heap_size;
055
056     //własność kopca może być zaburzona w miejscu przepisania
057     //ostatniego elementu na nowy
058     //(jeśli usunięty element nie był na ostatnim poziomie kopca)
059     if (2 * i <= heap_size)
060         heapify_down(i);
061 }
062
063 void heap_add(int x)
064 {
065     heap[++heap_size] = x;
066
067     //własność kopca może być zaburzona
068     //więc ją przywracam w miejscu dodania tego obiektu
069     heapify_up(heap_size);
070 }
```

```
071
072 int heap_top()
073 {
074     return heap[1];
075 }
076
077 bool heap_empty()
078 {
079     return heap_size == 0;
080 }
081
082 void heap_clear()
083 {
084     heap_size = 0;
085 }
```

5.7 Modyfikowalne kolejki priorytetowe

Zwykłe kopce można łatwo wzbogacić o dodatkowe informacje powiązane z konkretnymi wartościami. Np. w każdym kroku algorytmu Dijkstry (patrz rozdział 9.2) potrzebujemy nieprzetworzonego wierzchołka o minimalnej odległości od źródła. Tworzymy więc kopiec, w którym priorytet określamy odwrotnie względem odległości, tzn. najwyższy priorytet mają wierzchołki najmniej oddalone od wierzchołka początkowego. Każdy węzeł w kopcu przechowuje dwa pola — oprócz oddalenia także numer wierzchołka, który znajduje się w tej odległości (brak tej informacji dyskwalifikuje użyteczność kopca — po co odległość, gdy nie wiadomo który wierzchołek należy przetwarzać?).

Wszystko śmiga ładnie do momentu, w którym zmianie (poprawieniu) ulega minimalna odległość od źródła wierzchołka, który już leży w kopcu. Najprostszym ominięciem tego problemu jest beztroskie dorzucenie do kopca dodatkowego węzła, przechowującego informację o tym samym wierzchołku, lecz z nową wartością. Adekwatnej modyfikacji wymaga wtedy część kodu odpowiedzialna za wyciąganie informacji z korzenia kopca oraz dbająca o jego własności: jeśli korzeń przechowuje nieaktualne dane (np. wierzchołek v w korzeniu kopca ma odległość 10, a w zewnętrznej, niezależnej od kopca, tablicy odległości wisi informacja, że `odl[v] == 8`), to bez zastanowienia wywalamy korzeń.

Problem: wierzchołki mogą trafiać do kopca wielokrotnie. Ponieważ do korzenia trafiają elementy najbardziej pożądane ze względu na priorytet, te niepotrzebne będą usuwane bardzo późno, w niektórych przypadkach drastycznie zwiększając rozmiar kopca. Pogarsza nam się czas działania i zwiększamy zużycie pamięci.

Inny pomysł: iterować przez elementy na kopcu, w razie znalezienia informacji o modyfikowanym wierzchołku uaktualnić dane i w razie potrzeby przywrócić własność kopca. Takie rozwiązanie niewiele różni się od zaimplementowania kolejki priorytetowej na bazie tablicy jednowymiarowej z pełnym przeglądem zawartości przy każdej zmianie. Wysoce prawdopodobne przekroczenie dozwolonego czasu działania programu. Konkluzja: tak też *nie należy* robić.

Jeśli udałoby się przyspieszyć etap wyszukiwania położenia w kopcu modyfikowanego wierzchołka, dokonywałoby się aktualizacji w miejscu i ew. przywracało własność kopca. Cel można osiągnąć poprzez utrzymywanie dodatkowych tablic, jednej przechowującej informacje o lokacji wierzchołków w kopcu, drugiej spełniającej dokładnie odwrotną funkcję, tzn. zapamiętującej jakiego wierzchołka dotyczy badany węzeł kopca.

O ile zasadność istnienia pierwszej z tych tablic jest dość oczywista, celowość drugiej może być niewidoczna na pierwszy rzut oka. Rzeczywiście z punktu widzenia użytkownia kopca (*chcę coś dodać, czemuś zmienić priorytet, znaleźć/zabrać element o najwyższym priorytecie*) pamiętanie gdzie jest węzeł opisujący konkretny wierzchołek to przydatna rzecz (bo chcę mu zmienić priorytet), o tyle przechowywanie dla każdego węzła informacji o tym którego wierzchołka dotyczy przydaje się głównie od strony implementacji flaków kopca, w szczególności elementarnej operacji `heap_swap(i, j)`.

Niech `loc[i]` określa położenie wierzchołka o numerze i w kopcu, czyli na przykład `loc[2] == 7` oznacza, że siódmy indeks kopca zawiera informację o odległości wierzchołka 2, a więc `heap[loc[2]] == heap[7]`. Jeśli zamieniamy miejscami węzeł i-ty z j-tym, to musimy zamienić także konkretne dane w tablicy `loc`, odpowiadające wierzchołkom zamienianych miejscami w kopcu. Doprecyzowanie: trzeba znaleźć takie u i v, że `loc[u] == i` oraz `loc[v] == j`, a następnie dokonać zamiany wartości w `loc[u]` i `loc[v]`. Bez użycia dodatkowej tablicy wyszukanie pary liczb (u, v) zajęłoby czas liniowy (konieczność przeszukania całej tablicy `loc`).

Wprowadźmy więc tablicę `ver` spełniającą dokładnie odwrotną funkcję do tablicy `loc` — niech `ver[i]` określa którego wierzchołka dotyczy i-ty indeks kopca. Dla dowolnych liczb x, y takich, że `loc[x] == y` zachodzi `ver[y] == x`, czyli `loc` informuje, że opis wierzchołka x znajduje się w `heap[y]`, a `ver`, że w `heap[y]` leży opis wierzchołka x.

Szukane indeksy w *loc* wyciągamy wprost z *ver*:

- $u =$ `ver[i]`
- $v =$ `ver[j]`

- zamieniamy wartości w `loc[u]` i `loc[v]`
- zamieniamy wartości w `ver[i]` i `ver[j]`
- zamieniamy wartości w `heap[i]` i `heap[j]`
- fanfary i fajerwerki

Implementacja kopca modyfikowalnego od zwykłego różni się głównie dodaniem funkcji sprawdzającej czy pewien element, który ulega zmianie, znajduje się już w kopcu i należy go poprawić, czy też dopiero dodać. Nie licząc sporej rozbudowy funkcji `heap_swap` (opis wyżej) i poprawce w funkcji czyszczącej, różnica w pozostałych funkcjach jest marginalna i dotyczy tylko dbania o wprowadzanie właściwych danych do tablic `loc` i `ver`.

Przykładowa implementacja

Tablica liczb całkowitych `heap` reprezentuje kopiec o rozmiarze `heap_size`. Funkcja `heap_top()` zwraca element w korzeniu kopca, `heap_remove(i)` usuwa element w `heap[i]`, `heap_modify(v, x)` dodaje element o numerze v i wartości x na końcu kopca lub modyfikuje element o numerze v jeśli znajduje się już w kopcu. Pomocnicze funkcje `heapify_up(i)` oraz `heapify_down(i)` pomagają przywrócić zaburzoną własność kopca poprzez przemieszczanie elementu w indeksie i odpowiednio w górę lub w dół; `heap_swap(i, j)` zamienia miejscami elementy w kopcu o indeksach i i j. Funkcja `heap_empty()` odpowiada, czy istnieje jakiś element w kopcu. Pomocnicze tablice `ver` i `loc` pełnią funkcje jak w opisie powyżej.

Indeksacja kopca jak poprzednio zaczyna się od jedynki. Jeśli dla pewnego v zachodzi `loc[v] == 0`, to znaczy, że element v nie leży w kopcu. Z powodu konieczności utrzymywania tej tablicy w spójnym stanie, czas czyszczenia kopca zwiększa się ze stałego do liniowego. Nie wystarczy bowiem wyzerowanie rozmiaru kopca, należy jeszcze wyzerować wartości w tablicy `loc` dla węzłów, o których informacje znajdowały się w kopcu w chwili jego czyszczenia.

Zakładam, że modyfikacje wartości elementów mogą tylko zmniejszać ich wartości (czyli zwiększać priorytet). W przeciwnym wypadku należy zmodyfikować funkcję `heap_modify`, aby uwzględniała także przypadek odwrotny.

Sposób użycia

- `heap_modify(v, x)` zastępuje `heap_add(x)` ze zwykłego kopca
- reszta jak wcześniej

```
001 int heap[MAX], loc[MAX], ver[MAX];
002 int heap_size = 0;
003
004 void heap_swap(int i, int j)
005 {
006     int temp, u, v;
007
008     u = ver[i];
009     v = ver[j];
010
011     temp = loc[u];
012     loc[u] = loc[v];
013     loc[v] = temp;
014
015     temp = ver[i];
016     ver[i] = ver[j];
017     ver[j] = temp;
018
019     temp = heap[i];
020     heap[i] = heap[j];
021     heap[j] = temp;
022 }
023
024 void heapify_up(int i)
025 {
026     int current = i, parent = i / 2;
027
028     if (parent > 0 && heap[parent] < heap[current]) {
```

```
029         heap_swap(current, parent);
030         heapify_up(parent);
031     }
032 }
033
034 void heapify_down(int i)
035 {
036     int parent = i, left = i * 2, right = i * 2 + 1;
037     int best;
038
039     if (left <= heap_size && heap[left] > heap[parent])
040         best = left;
041     else
042         best = parent;
043
044     if (right <= heap_size && heap[right] > heap[best])
045         best = right;
046
047     if (best != parent) {
048         heap_swap(best, parent);
049         heapify_down(best);
050     }
051 }
052
053 void heap_remove(int i)
054 {
055     //przenoszę element ostatni w miejsce i
056     heap_swap(i, heap_size);
057
058     //zaznaczam w loc, że wierzchołek u
059     //nie przebywa już w kopcu
060     int u = ver[heap_size];
061     loc[u] = 0;
062
063     --heap_size;
064
065     if (2 * i <= heap_size)
066         heapify_down(i);
067 }
068
069 void heap_modify(int v, int x)
070 {
071     //czy wierzchołek v jest już w kopcu?
072     if (loc[v] == 0) {
073         //jeśli nie, to dodaję
074         ++heap_size;
075         loc[v] = heap_size;
076         heap[heap_size] = x;
077         ver[heap_size] = v;
078         heapify_up(heap_size);
079     } else {
080         //jeśli tak, to zmieniam wartość
081         heap[loc[v]] = x;
082         heapify_up(loc[v]);
083     }
084 }
085
086 int heap_top()
087 {
088     //dla odmiany interesuje nas który element
089     //leży w korzeniu, a nie jaką ma wartość
```

```
090     return ver[1];
091 }
092
093 bool heap_empty()
094 {
095     return heap_size == 0;
096 }
097
098 void heap_clear()
099 {
100     for (int i = 1; i <= heap_size; ++i)
101         loc[ver[i]] = 0;
102     heap_size = 0;
103 }
```

6 1001 przepisów na drzewo binarne

Ta część tekstu poświęcona jest budowie i zastosowaniom różnych odmian drzew binarnych. Można by pomyśleć, że drzewo to drzewo i niczego ciekawego się nie wymyśli, jednak postaram się oprócz oczywistych oczywistości przedstawić też jakieś mniej rzucające się w oczy przykłady zastosowań. Tym bardziej, że drzewo binarne (w którymś z licznych smaków) jest jedną z najbardziej przydatnych struktur danych przy rozwiązywaniu wielu zadań konkursowych.

Oczywiście niektórych rodzajów drzew opisywanych dalej można wprost używać jako kolejek priorytetowych opisywanych wcześniej (zresztą kopiec binarny jest oczywiście rodzajem drzewa binarnego), jednak przedstawiane dalej struktury mają zazwyczaj dużo szersze zastosowania, dlatego ich opis znajduje się właśnie tutaj.

6.1 Binarne drzewo wyszukiwań (BST)

Przedstawiona w rozdziałach 5.6 i 5.7 struktura kopców binarnych ma bardzo miłe własności, sensownie zrozumiałą zasadę działania, relatywnie nieskomplikowaną implementację i w ogóle super. Bywa jednak, że trzeba czegoś więcej, niż wyznaczenie największego/najmniejszego elementu w pewnym zbiorze. Czasami trzeba określić, czy w pewnym zbiorze istnieje element, który niekoniecznie jest największy lub najmniejszy, a także robić to w sensownym czasie, więc zwykła tablica jednowymiarowa odpada.

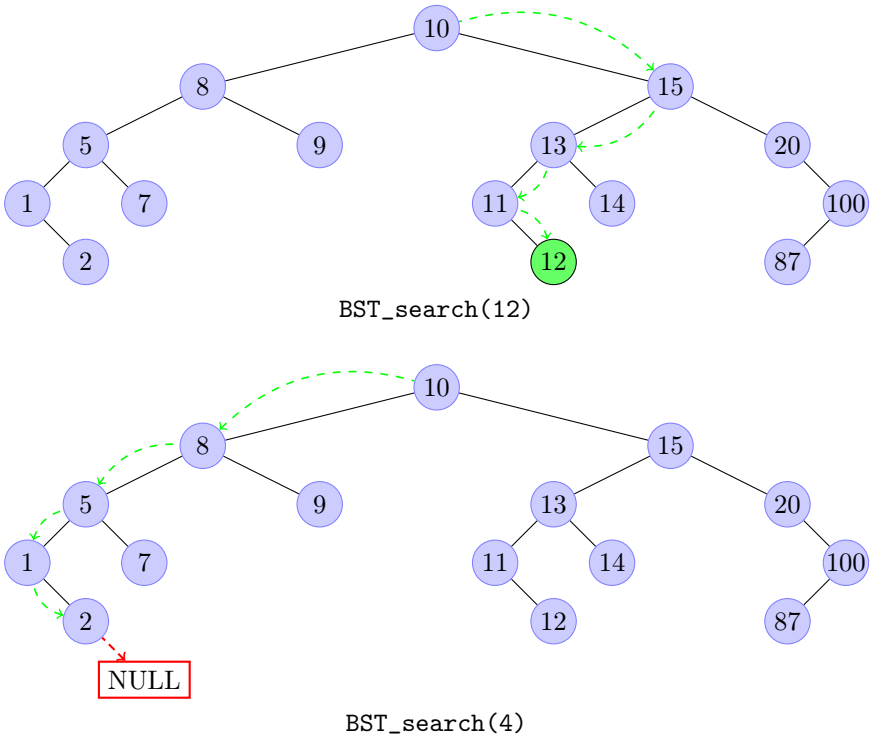
Zmieńmy nieco własności kopca i sposób nazewnictwa. Po pierwsze, węzeł *child* jest *dzieckiem* pewnego innego węzła *root*, jeśli istnieje krawędź $root \to child$ oraz *child* znajduje się o poziom niżej względem *root*. Wtedy *potomkiem* węzła *root* będzie każdy węzeł, do którego można poprowadzić ścieżkę od *root*, która schodzi jedynie w dół drzewa. W szczególności dzieci dowolnego węzła są także jego potomkami.

Po drugie, nie określamy elementów zbioru według *priorytetu*, lecz według *klucza*. Kopiec był takim drzewem, w którym dzieci każdego węzła mieli klucz niższy, niż ten węzeł. W ten sposób w korzeniu całego kopca wisiał węzeł o najwyższym kluczu w całym kopcu. Zdefiniujmy własności *binarnego drzewa wyszukiwań* (BST — Binary Search Tree):

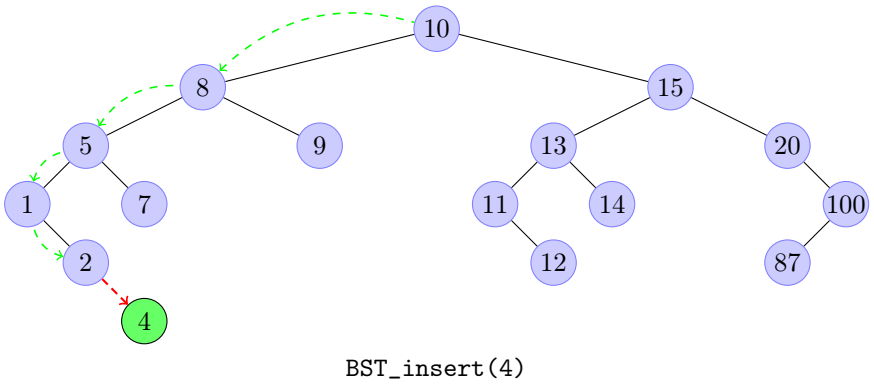
- Niech *root* będzie pewnym węzłem w tym drzewie, wtedy jego lewe dziecko oznaczymy $left$, a prawe $right$.
- Niech T_v oznacza poddrzewo ukorzenione w węźle v.
- Niech $key(v)$ oznacza wartość klucza węzła v.
- Klucz w każdym węźle poddrzewa ukorzenionego w $left$ jest mniejszy lub równy, niż $key(root)$, a klucz w każdym węźle poddrzewa ukorzenionego w $right$ jest większy lub równy, niż $key(root)$:

$$\forall\, u \in T_{left}\ \forall\, v \in T_{right}\ key(u) \leqslant key(root) \leqslant key(v)$$

I tyle. Nie wymagamy, żeby węzły do takiego drzewa były dodawane „poziomami”, jak miało to miejsce w przypadku kopca (rys. 5.2). Dla uproszczenia przyjmuję, że każdy węzeł w drzewie ma inny klucz. Można wtedy wstawić ostre nierówności w ostatniej własności, zmieniając jej postać na $key(u) < key(root) < key(v)$.



Rysunek 6.1: Wyszukiwanie w przykładowym BST.



Rysunek 6.2: Wstawianie do BST.

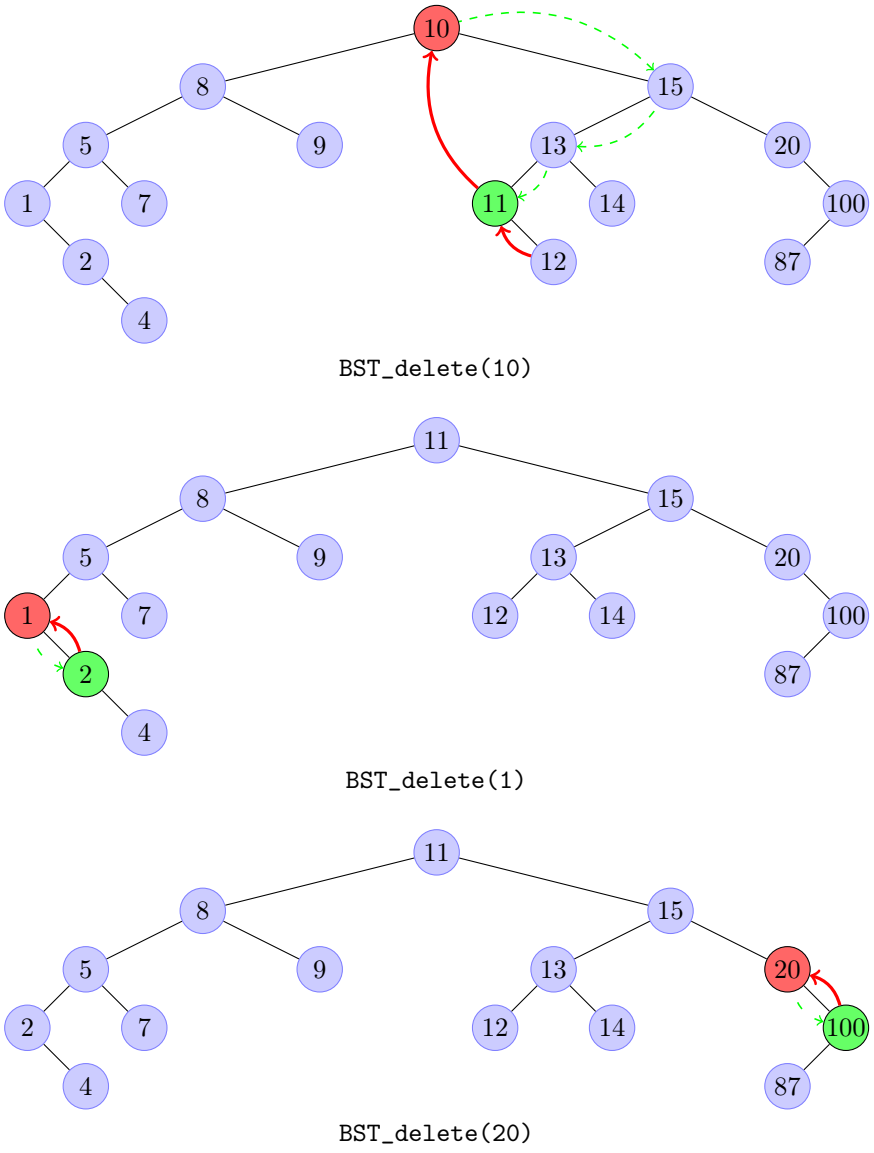
Te własności mają przełożenie na bardzo praktyczne zastosowanie drzew binarnych. Trywialna obserwacja: wszyscy potomkowie dowolnego węzła v na lewo od niego mają mniejszą wartość klucza niż v, a wszyscy potomkowie na prawo mają większą wartość klucza. W ten sposób aby znaleźć najmniejszy element w drzewie binarnym, należy znaleźć element położony na jego lewym skraju. Największy będzie oczywiście na prawym skraju.

Jeśli chcemy sprawdzić, czy istnieje pewien klucz k w drzewie, to rozpoczynamy wyszukiwanie od korzenia $root$. Jeśli $k == key(root)$, to świetnie, bo znaleźliśmy klucz w drzewie. W przeciwnym wypadku albo $k < key(root)$ i musimy kontynuować wyszukiwanie w lewym poddrzewie ($root = left$), albo $k > key(root)$ i wyszukujemy w prawym poddrzewie ($root = right$). Jeśli w pewnym momencie wywalimy się poza drzewo (wskaźnik do lewego lub prawego dziecka będzie `NULL`), to kończymy wyszukiwanie, bo klucza k w drzewie nie ma (rys. 6.2).

Operacja wstawiania nowego elementu do BST przeprowadzana jest za pomocą wyszukiwania. Najpierw sprawdzamy, czy wstawiany klucz k istnieje w drzewie. Jeśli nie, to tworzymy go w miejscu, w którym wychodzilibyśmy poza drzewo w trakcie poszukiwaniu klucza k. Na przykład gdyby w drzewie z rysunku 6.1 wstawić węzeł o kluczu 4, to otrzymalibyśmy drzewo takie, jak na rysunku 6.2.

Usuwanie jest operacją trochę trudniejszą. Na początek oczywiście wyszukujemy położenie węzła do usunięcia, nazwijmy go v. Dalsze działanie uzależnione jest od ilości dzieci węzła v. Jeśli jest liściem, to możemy go po prostu usunąć. Jeśli ma jedno dziecko, to zastępujemy nim węzeł v. Jeśli zaś węzeł v ma dwójkę dzieci, to należy w poddrzewie wyznaczanym przez ten węzeł znaleźć element o największym kluczu mniejszym od $key(v)$ lub o najmniejszym kluczu większym od $key(v)$ i tym znalezionym węzłem zastąpić węzeł v.

Można łatwo pokazać, że ten nowy węzeł (oznaczmy go u) ma co najwyżej jedno dziecko. Na przykład, jeśli chcieli-



Rysunek 6.3: Usuwanie z BST.

byśmy zastąpić v przez u o kluczu jak największym, ale mniejszym od $key(v)$, to musimy szukać u w lewym poddrzewie v (ponieważ z własności BST wynika, że wszystkie mniejsze elementy będą na lewo). W tym poddrzewie szukamy największego elementu, czyli skrajnie prawej ścieżki. Z tego musi wynikać, że znaleziony węzeł u może mieć co najwyżej jedno dziecko, po lewej stronie. Gdyby miał po prawej, to nie byłaby to skrajnie prawa ścieżka, a więc sprzeczność. Analogicznie możemy wyszukiwać najmniejszego większego elementu, przechodząc po skrajnie lewej ścieżce prawego poddrzewa.

Jeśli węzeł u nie miał żadnych potomków (był liściem), to już jest dobrze. W przeciwnym wypadku przeniesienie węzła u na miejsce v powoduje powstanie „dziury" w BST. Żeby ją załatać wystarczy przenieść dziecko węzła u w jego poprzednie miejsce. W skrócie $delete(v)$ powoduje znalezienie węzła u o podanych wcześniej własnościach, zapisaniu go w miejscu węzła v i wywołanie $delete(u)$. Po takiej operacji własność BST dalej jest zachowana. Nowym korzeniem jest węzeł u, którego klucz jest większy od kluczy wszystkich elementów w lewym poddrzewie oraz mniejszy od kluczy wszystkich elementów w prawym poddrzewie. Przykład usuwania można znaleźć na rysunku 6.3. Na rysunku zakładam, że przy usuwaniu wyszukujemy węzeł najmniejszy spośród większych od usuwanego (czyli skrajnie lewa ścieżka w prawym poddrzewie).

Czas działania Każda operacja w BST trwa co najwyżej tyle, jaka jest wysokość drzewa. W najlepszym przypadku możemy otrzymać drzewo binarne, które niejako wypełnione jest tak jak kopiec binarny. Wtedy czas działania dowolnej operacji w drzewie jest $\mathcal{O}(\log n)$. Niestety jeśli dane wejściowe są podane w sposób złośliwy (np. ciąg wstawień kluczy kolejno $1, 2, \ldots, 10^5$), to może się okazać, że budujemy drzewo, które niewiele różni się od zwykłej listy (koszt czasowy pojedynczej operacji $\mathcal{O}(n)$), a nawet zachowuje się gorzej — tracimy czas na zbudowanie drzewa, zamiast po prostu wrzucić dane do tablicy i wyszukiwać liniowo. Czasami można sobie z tym radzić poprzez losowe przemieszanie zbioru

kluczy, z których budujemy drzewo, ale nie zawsze jest to możliwe. W następnym rozdziale przedstawiony zostanie sposób radzenia sobie z tymi problemami, za cenę pewnego skomplikowania struktury danych.

Przykładowa implementacja

`BST_node` to struktura reprezentująca pojedynczy węzeł drzewa. W węźle zapamiętujemy klucz i wskaźniki do lewego i prawego poddrzewa. W przykładowym kodzie kluczem jest pojedyncza liczba całkowita, ale nie ma żadnych przeciwwskazań do wrzucenia bardziej skomplikowanych danych. Funkcje `BST_search()` i `BST_insert()` są dość proste i zasada ich działania powinna być widoczna. Trochę bardziej śmieciowate jest `BST_delete()`, bo trzeba rozpatrywać dwa istotne przypadki:

1. usuwany węzeł v nie ma prawego poddrzewa — w takiej sytuacji przepinamy lewe poddrzewo na jego miejsce, usuwamy rzeczywiście v i po kłopocie,
2. usuwany węzeł v ma prawe poddrzewo — wtedy szukamy w tym poddrzewie skrajnie lewego węzła (to pomocnicza funkcja `BST_delete_helper()`), znaleziony węzeł t wyjmujemy z drzewa i odkładamy na bok (będzie zaraz potrzebny), a jego ewentualne prawe poddrzewo przepinamy na miejsce t; ponieważ t zastąpi usuwany węzeł v, to podczepiamy do t lewe i prawe poddrzewa v przed jego usunięciem.

Potencjalnie dziwaczny sposób implementacji tych funkcji – rekurencyjny, z przekazywaniem korzenia poddrzewa jako argument i zwracaniem wskaźników na węzły jako wynik – ma swoje zalety w postaci uproszczenia kodu (mniej przypadków do rozpatrywania) oraz uniknięcia konieczności występowania w węzłach drzewa wskaźników do rodziców. Pełna siła takiego sposobu implementacji ujawni się w momencie implementowania zrównoważonych binarnych drzew wyszukiwań, zwłaszcza że wtedy będziemy mieli zagwarantowaną głębokość wywołań rekurencyjnych na poziomie $\mathcal{O}(\log n)$, co pozwoli kompletnie zignorować koszt rekurencji.

Sposób użycia

- wstawienie klucza k: `root = BST_insert(k, root)`
- usunięcie klucza k: `root = BST_delete(k, root)`
- sprawdzenie czy klucz k jest w drzewie: `if (BST_search(k, root) != NULL) ...`

```
001 struct BST_node {
002     int key;
003     BST_node *left, *right;
004 
005     BST_node(int _k)
006         : key(_k), left(NULL), right(NULL) {}
007 };
008 
009 //korzeń drzewa, drzewo na początku jest puste
010 BST_node *root = NULL;
011 
012 //pomocniczy wskaźnik, przydatny przy usuwaniu
013 BST_node *del_helper;
014 
015 BST_node * BST_search(int key, BST_node *node)
016 {
017     //zabezpieczenie przed wypadnięciem z drzewa
018     if (node == NULL)
019         return NULL;
020 
021     //znaleźliśmy właściwy węzeł?
022     if (key == node->key)
023         return node;
024 
025     //jeśli nie, to idziemy w lewo albo prawo
026     if (key < node->key)
027         return BST_search(key, node->left);
028     return BST_search(key, node->right);
029 }
```

```
030
031 BST_node * BST_insert(int key, BST_node *node)
032 {
033     //wypadliśmy z drzewa, więc tu trzeba wstawić nowy węzeł.
034     if (node == NULL)
035         return new BST_node(key);
036
037     //w przeciwnym wypadku schodzimy niżej
038     if (key < node->key)
039         node->left = BST_insert(key, node->left);
040     else
041         node->right = BST_insert(key, node->right);
042     return node;
043 }
044
045 //pomocnicza funkcja, znajdująca skrajnie lewą ścieżkę
046 BST_node * BST_delete_helper(BST_node *node);
047
048 BST_node * BST_delete(int key, BST_node *node)
049 {
050     //zabezpieczenie przed wypadnięciem z drzewa
051     if (node == NULL)
052         return NULL;
053
054     //jesteśmy w węźle do usunięcia?
055     if (key == node->key) {
056         BST_node *temp;
057
058         //przypadek prosty - usuwany węzeł nie ma prawego dziecka
059         if (node->right == NULL) {
060             temp = node->left;
061             delete node;
062             return temp;
063         }
064
065         //przypadek trudniejszy - znajdujemy węzeł o najmniejszym
066         //kluczu większym od klucza aktualnego węzła i zapamiętujemy
067         //go w del_helper, przepinamy do del_helper
068         //odpowiednie wskaźniki i stary węzeł usuwamy
069         temp = BST_delete_helper(node->right);
070         del_helper->left = node->left;
071         del_helper->right = temp;
072         delete node;
073
074         return del_helper;
075     }
076
077     if (key < node->key)
078         node->left = BST_delete(key, node->left);
079     else
080         node->right = BST_delete(key, node->right);
081
082     return node;
083 }
084
085 //pomocnicza funkcja, znajdująca skrajnie lewą ścieżkę
086 BST_node * BST_delete_helper(BST_node *node)
087 {
088     //dalej w lewo się nie da, więc jesteśmy w węźle, który
089     //zastąpi węzeł usuwany - zapamiętujemy go w del_helper
090     if (node->left == NULL) {
```

```
091         del_helper = node;
092         return node->right;
093     }
094
095     node->left = BST_delete_helper(node->left);
096     return node;
097 }
```

Wzbogacanie

Węzły binarnego drzewa wyszukiwań można wzbogacać o dodatkowe informacje, które pozwalają na rozszerzenie funkcjonalności tej struktury niewielkim kosztem. Przykładowo można pamiętać w każdym węźle rozmiar poddrzewa, co ułatwia znajdowanie n-tego leksykograficznie klucza w węźle. Aktualizacja takiej dodatkowej informacji jest niesłychanie tania i można jej dokonać w trakcie dowolnej operacji modyfikującej drzewo (wstawianie i usuwanie).

Przykładowa implementacja #2

Struktura `BST_node` wzbogacona została o pole `cnt`, oznaczające liczbę węzłów w poddrzewie rozpinanym przez dany wierzchołek (łącznie z nim samym). Wartość `cnt` jest adekwatnie aktualizowana przy modyfikacjach drzewa. Zaprezentowana też została funkcja `BST_nth()`, zwracająca n-ty leksykograficznie element w drzewie. Wykorzystuje ona prostą obserwację: jeśli musimy znaleźć n-ty leksykograficznie element i jesteśmy w drzewie, którego lewe poddrzewo ma k elementów, to możliwe są trzy sytuacje:

1. $k+1 > n$, wtedy poszukiwany n-ty leksykograficznie element jest jednocześnie n-tym leksykograficznie elementem w lewym poddrzewie,
2. $k+1 = n$, wtedy szukana wartość znajduje się w korzeniu,
3. $k+1 < n$, wtedy poszukiwany n-ty leksykograficznie element jest jednocześnie $(n-k-1)$-tym leksykograficznie elementem w prawym poddrzewie.

```
001 struct BST_node {
002     int key, cnt;
003     BST_node *left, *right;
004
005     BST_node(int _k)
006         : key(_k), cnt(1), left(NULL), right(NULL) {}
007 };
008
009 BST_node *root = NULL;
010 BST_node *del_helper;
011
012 int BST_size(BST_node *node)
013 {
014     if (node == NULL)
015         return 0;
016     return node->cnt;
017 }
018
019 //aktualizacja dodatkowych danych w węźle
020 void BST_update(BST_node *node)
021 {
022     node->cnt = BST_size(node->left) + BST_size(node->right);
023 }
024
025 BST_node * BST_search(int key, BST_node *node)
026 {
027     if (node == NULL)
028         return NULL;
029     if (key == node->key)
030         return node;
031     if (key < node->key)
```

```
032         return BST_search(key, node->left);
033     return BST_search(key, node->right);
034 }
035
036 BST_node * BST_insert(int key, BST_node *node)
037 {
038     if (node == NULL)
039         return new BST_node(key);
040     if (key < node->key)
041         node->left = BST_insert(key, node->left);
042     else
043         node->right = BST_insert(key, node->right);
044     BST_update(node); //aktualizacja
045     return node;
046 }
047
048 BST_node * BST_delete_helper(BST_node *node);
049
050 BST_node * BST_delete(int key, BST_node *node)
051 {
052     if (node == NULL)
053         return NULL;
054     if (key == node->key) {
055         BST_node *temp;
056
057         if (node->right == NULL) {
058             temp = node->left;
059             delete node;
060             return temp;
061         }
062
063         temp = BST_delete_helper(node->right);
064         del_helper->left = node->left;
065         del_helper->right = temp;
066         delete node;
067
068         BST_update(del_helper); //aktualizacja
069         return del_helper;
070     }
071
072     if (key < node->key)
073         node->left = BST_delete(key, node->left);
074     else
075         node->right = BST_delete(key, node->right);
076     BST_update(node); //aktualizacja
077     return node;
078 }
079
080 BST_node * BST_delete_helper(BST_node *node)
081 {
082     if (node->left == NULL) {
083         del_helper = node;
084         return node->right;
085     }
086
087     node->left = BST_delete_helper(node->left);
088     BST_update(node); //aktualizacja
089     return node;
090 }
091
092 //szukanie n-tego leksykograficznie elementu
```

```
093 BST_node * BST_nth(int nth, BST_node *node)
094 {
095     //temp == rozmiar lewego poddrzewa + korzeń
096     int temp = 1 + BST_size(node->left);
097     //jesteśmy na miejscu?
098     if (temp == nth)
099         return node;
100
101     //jeśli nie, to trzeba zejść niżej
102     if (temp > nth)
103         return BST_nth(nth, node->left);
104     else
105         return BST_nth(nth - temp, node->right);
106 }
```

Implementacja niewskaźnikowa

Jeśli zawczasu wiemy, że rozmiar drzewa nie przekroczy pewnej z góry ustalonej granicy, to możemy zamiast korzystać z dynamicznej alokacji pamięci zająć od razu spory kawałek pamięci – tyle, żeby nie zabrakło w odniesieniu do wspomnianej granicy – i dbać o pamięć własnoręcznie. Korzyści czasowe z tego tytułu potrafią być znaczne; własne doświadczenie pokazuje zyski czasowe nawet 20-30%.

Należy się zastanowić przed rozpoczęciem pisania, czy potencjalny zysk wydajnościowy przeważa dodatkową złożoność kodu i idące za tym prawdopodobieństwo popełnienia błędu w implementacji. Tym bardziej, że różnorakie drzewa binarne i ich pochodne są jednym z tych miejsc w praktyce konkursowej, w których korzystanie ze wskaźników oraz rekurencji rzeczywiście upraszcza kod i czyni go bardziej czytelnym[1]. Pokazaną niżej metodę można stosować oczywiście dużo częściej, niż przy byle drzewach binarnych. Załączona implementacja dotyczy drzewa niewzbogaconego o żadne dodatkowe informacje, żeby nie zaciemniać dodatkowo kodu.

Przykładowa implementacja #3

Załóżmy, że pewna stała liczbowa `MEM_MAX` przechowuje maksymalną liczbę węzłów, jakie jednocześnie mogą należeć do drzewa. Tablica `mem_pool` rezerwuje miejsce na właśnie taką liczbę elementów. Będziemy trzymali w dodatkowej tablicy `mem_pool_queue` indeksy wszystkich miejsc w `mem_pool`, które są niezajęte. Łatwo zauważyć, że `mem_pool_queue` to zwykła kolejka cykliczna (rozdział 5.2), przechowująca dostępne „adresy”.

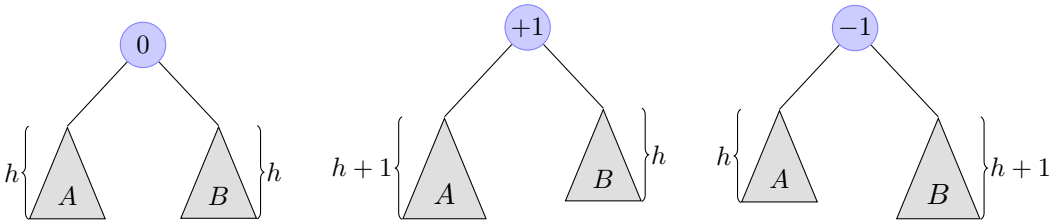
Pierwsze, co należy zrobić, to wywołać funkcję `BST_mem_init()`. Powoduje ona zainicjowanie kolejki wolnych miejsc. Wszędzie w kodzie zamiast wskaźników używane są liczby całkowite. Przechowują one indeks danego węzła w tablicy `mem_pool`. Odpowiednik wskaźnika `NULL` został tutaj ustalony na `-1`.

Funkcja `BST_alloc_node()` używana jest zamiast operatora `new`: wyciąga z kolejki indeks niezajętego pola tablicy `mem_pool`, tworzy tam węzeł drzewa z zadanym kluczem i zwraca „wskaźnik” (liczbę całkowitą) do tego miejsca. Analogicznie `BST_delete_node()` zastępuje `delete`: zwraca do kolejki wolnych adresów indeks usuwanego węzła.

```
001 struct BST_node {
002     int key;
003     int left, right;
004
005     BST_node(int _k)
006         : key(_k), left(-1), right(-1) {}
007 };
008
009 int root = -1;
010 int del_helper;
011
012 int mem_head, mem_tail;
013 int mem_pool_queue[MEM_MAX];
014 BST_node mem_pool[MEM_MAX];
015
016 //inicjowanie puli pamięci
017 void BST_mem_init()
018 {
```

[1] Disclaimer: zdanie czysto subiektywne.

```
019     mem_head = 0;
020     mem_tail = MEM_MAX;
021     for (int i = 0; i < MEM_MAX; ++i)
022         mem_pool_queue[i] = i;
023 }
024
025 //znalezienie wolnego kawałka pamięci i stworzenie
026 //w jego miejscu węzła drzewa z przekazanym do funkcji kluczem
027 int BST_alloc_node(int key)
028 {
029     int idx = mem_pool_queue[mem_head % MEM_MAX];
030     ++mem_head;
031     mem_pool[idx] = BST_node(key);
032     return idx;
033 }
034
035 //podany indeks węzła w puli pamięci jest "zwalniany"
036 //i zostaje zwrócony do puli
037 void BST_delete_node(int node)
038 {
039     mem_pool_queue[mem_tail % MEM_MAX] = node;
040     ++mem_tail;
041 }
042
043 int BST_search(int key, int node)
044 {
045     if (node == -1)
046         return -1;
047
048     if (key == mem_pool[node].key)
049         return node;
050
051     if (key < mem_pool[node].key)
052         return BST_search(key, mem_pool[node].left);
053     return BST_search(key, mem_pool[node].right);
054 }
055
056 int BST_insert(int key, int node)
057 {
058     if (node == -1)
059         return BST_alloc_node(key);
060
061     if (key < mem_pool[node].key)
062         mem_pool[node].left = BST_insert(key, mem_pool[node].left);
063     else
064         mem_pool[node].right = BST_insert(key, mem_pool[node].right);
065     return node;
066 }
067
068 int BST_delete_helper(int node);
069
070 int BST_delete(int key, int node)
071 {
072     if (node == -1)
073         return -1;
074
075     if (key == mem_pool[node].key) {
076         int temp;
077
078         if (mem_pool[node].right == -1) {
079             temp = mem_pool[node].left;
```



Rysunek 6.4: Trzy dopuszczalne stany (pod)drzewa zrównoważonego.

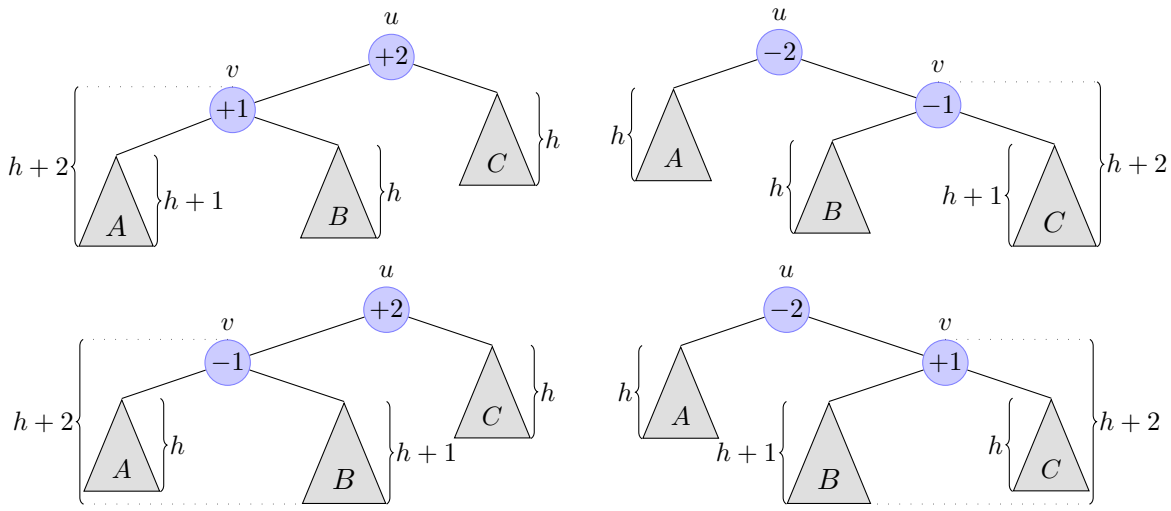
```
080             BST_delete_node(node);
081             return temp;
082         }
083
084         temp = BST_delete_helper(mem_pool[node].right);
085         mem_pool[del_helper].left = mem_pool[node].left;
086         mem_pool[del_helper].right = temp;
087         BST_delete_node(node);
088
089         return del_helper;
090     }
091
092     if (key < mem_pool[node].key)
093         mem_pool[node].left = BST_delete(key, mem_pool[node].left);
094     else
095         mem_pool[node].right = BST_delete(key, mem_pool[node].right);
096
097     return node;
098 }
099
100 int BST_delete_helper(int node)
101 {
102     if (mem_pool[node].left == -1) {
103         del_helper = node;
104         return mem_pool[node].right;
105     }
106
107     mem_pool[node].left = BST_delete_helper(mem_pool[node].left);
108     return node;
109 }
```

6.2 Zrównoważone BST (AVL)

Najdłuższa ścieżka z korzenia drzewa do któregoś z jego liści to wysokość drzewa (oznaczenie: $height(v)$ dla drzewa ukorzenionego w v). Wysokość drzewa dla v będącego pojedynczym wierzchołkiem (także np. liściem w jakimś większym drzewie) przyjmujemy 1, wysokość drzewa pustego to 0. Mówimy, że drzewo binarne jest *zrównoważone*, gdy dla każdego poddrzewa ukorzenionego w v o lewym dziecku $left$ i prawym dziecku $right$ zachodzi $|height(left) - height(right)| \leqslant 1$.

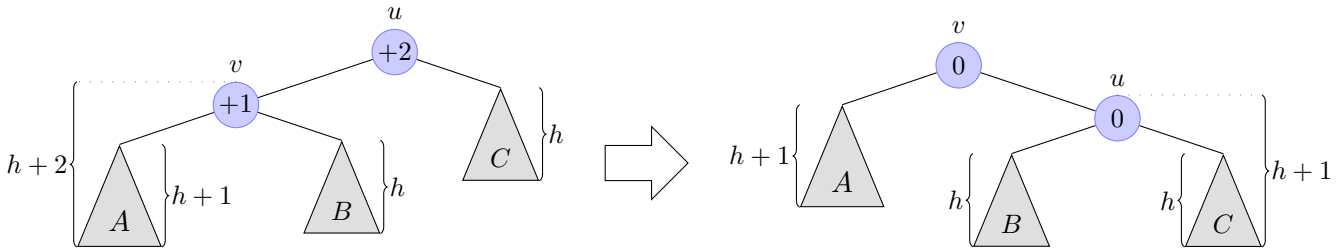
Gdy drzewo jest zrównoważone według powyższej definicji, wtedy górne ograniczenie na jego wysokość w przybliżeniu wynosi $1.44 \log_2(n)$. Jeśli zatem zapewnimy równoważenie drzewa po każdej operacji wstawienia lub usunięcia elementu, to każda operacja wyszukiwania ma gwarancję działania w czasie logarytmicznym. Poniższy algorytm (autorstwa Rosjan G. Adelsona-Velskij i E. Landisa – stąd nazwa *drzewo AVL*) budowania i utrzymywania zrównoważonego drzewa przedstawia operacje wstawiania i usunięcia działające także w czasie $\mathcal{O}(\log n)$, gwarantujące zachowanie zrównoważonego drzewa.

Ponieważ drzewo AVL jest tylko odmianą BST, operacja wyszukiwania pozostaje bez zmian. Zastanowić się należy co może się stać, gdy dokonamy wstawienia lub usunięcia elementu, który spowoduje zaburzenie własności równowagi. Liczby w wezłach na rysunku 6.4 reprezentują „przechylenie" drzewa: dodatnie liczby oznaczają, że lewe poddrzewo jest wyższe od prawego, ujemne przeciwnie, a zero oznacza, że oba poddrzewa są tej samej wysokości. Rozważmy drzewo AVL po pewnej operacji usunięcia lub wstawienia węzła, która zaburza równowagę, wprowadzając któryś ze stanów pokazanych na rysunku 6.5.



Rysunek 6.5: Możliwe zaburzenia własności AVL.

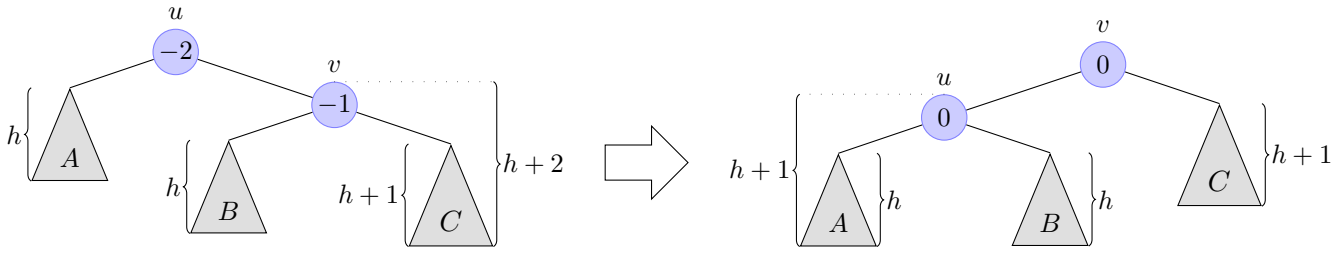
Rozprawmy się z pierwszym przypadkiem (na rysunku 6.5 u góry po lewej). Aby przywrócić drzewo do zrównoważonego stanu dokonamy *obrotu w prawo*, polegającego na zamianie miejscami węzłów u i v z odpowiednim przepięciem poddrzew A, B i C:



Rysunek 6.6: Obrót w prawo.

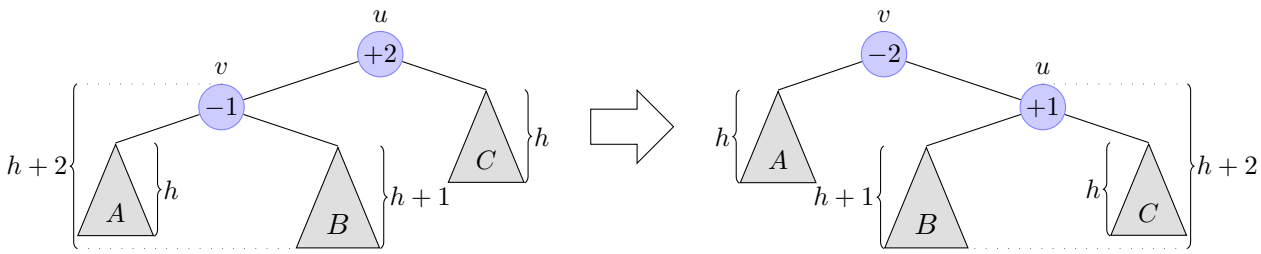
Takie przemieszczenie nie powoduje zaburzenia własności BST. Zauważmy bowiem, że dla dowolnych $a \in A$, $b \in B$ i $c \in C$ mamy $a < v < b < u < c$ i ta relacja zostaje zachowana w poddrzewie po dokonaniu obrotu, co widać także na rysunku.

Analogicznie działamy w drugim przypadku, przeprowadzając *obrót w lewo*:



Rysunek 6.7: Obrót w lewo.

W pozostałych dwóch przypadkach sprawa nieco się komplikuje. Zastosowanie pojedynczego obrotu jest nieskuteczne. Zastosowanie obrotu w prawo do sytuacji trzeciej z rysunku 6.5 (na dole po lewej) powoduje otrzymanie sytuacji czwartej:



Rysunek 6.8: Nieskuteczny obrót w prawo.

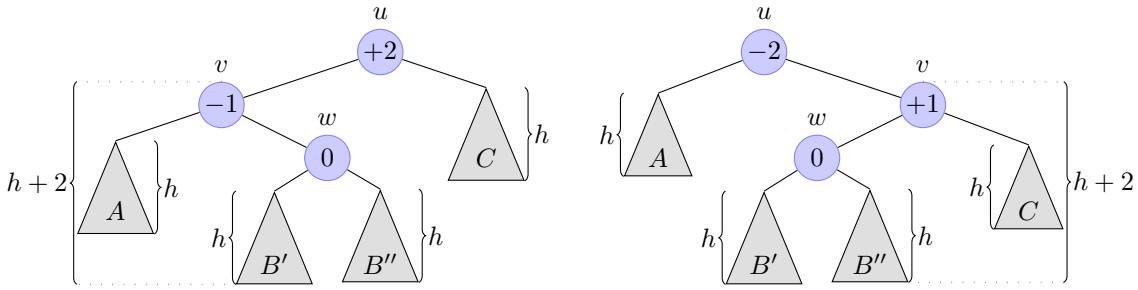
Rozwiązaniem jest zejście o poziom niżej w drzewie (rysunek 6.9). Jeśli rozpatrzymy poddrzewo B, które jest ukorzenione w węźle w, to **co najmniej jedno** poddrzewo spośród lewego i prawego poddrzewa węzła w (oznaczmy je B' i B'') ma wysokość h. We wszystkich rysunkach przyjęte zostało $height(B') = height(B'') = h$ dla czytelności. W tak rozpisanym fragmencie drzewa należy zastosować *podwójną rotację*, czyli tak naprawdę złożenie dwóch pojedynczych rotacji. Dla sytuacji po lewej stronie rysunku 6.9 będzie to najpierw rotacja w lewo przeprowadzona na parze wierzchołków v i w, a następnie rotacja w prawo na parze u i w. Wizualizację tych obrotów można zobaczyć na rysunkach 6.10 i 6.11. Wniosek: można dokonywać wstawienia i usuwania w drzewie AVL, a następnie wracając po ścieżce do korzenia porównywać wysokość lewego i prawego poddrzewa. Jeśli różnica między nimi jest zbyt duża, dokonujemy odpowiednich obrotów. Ponieważ wysokość drzewa jest logarytmiczna, a każdy obrót można wykonać w czasie stałym poprzez odpowiednie zamienienie kilku wskaźników, mamy czas $\mathcal{O}(\log n)$ na wstawianie i usuwanie zachowujące własność AVL.

Widać, że obroty doprowadzają drzewo do postaci zrównoważonej, ale co z odpowiednim uporządkowaniem elementów, tzn. zgodnym z warunkiem BST? Zamiast pieczołowicie przeprowadzać dowód polegający na wykazaniu zachowania wszystkich istotnych nierówności wystarczy powołać się na poprawność pojedynczych obrotów, która została wcześniej wykazana – ponieważ podwójny obrót to dwa pojedyncze, nic po drodze nie może się popsuć w kolejności elementów.

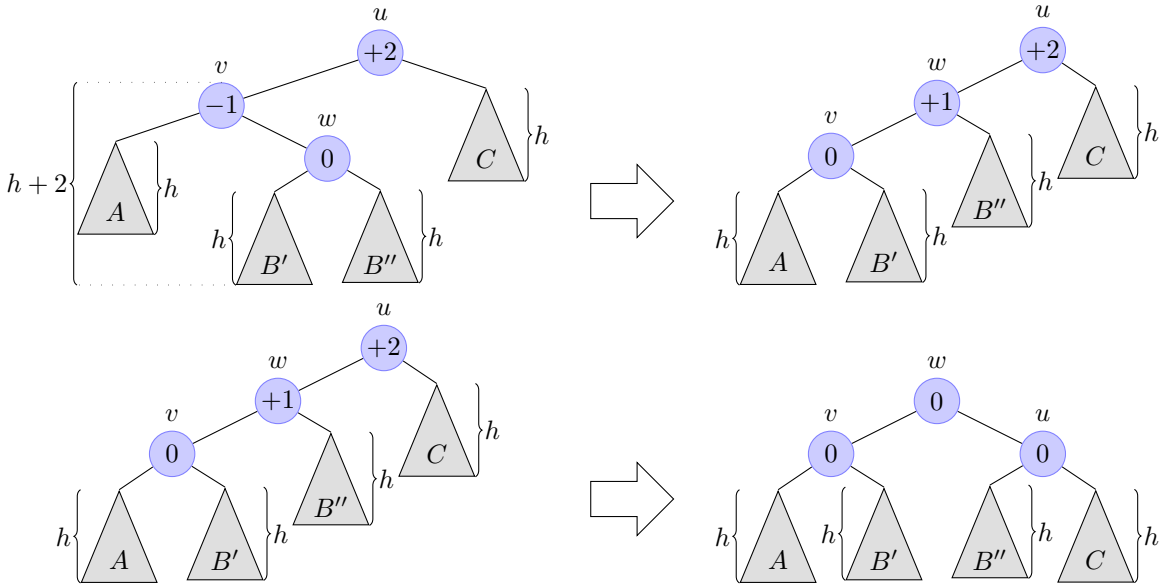
Przykładowa implementacja

W porównaniu do BST, w węzłach drzewa AVL trzymamy dodatkowo jedynie wysokość poddrzewa ukorzenionego w tym węźle. `AVL_search()` nie różni się niczym od `BST_search()`. Jedyna różnica w funkcjach wstawiających i usuwających polega na dodaniu wywołań pomocniczych funkcji, które odświeżają wysokość drzewa (`AVL_recalc_height()`) i sprawdzają, czy właśnie obliczona wartość nie oznacza zaburzenia własności AVL (`AVL_rebalance()`). Zaburzenie oznacza wywołanie odpowiedniego obrotu i przywrócenie drzewa do legalnego stanu. Sposób implementacji jest analogiczny do zastosowanego wcześniej w BST, co umożliwia m.in. eleganckie rozprawieniem się z kwestią podwójnych obrótów jako złożenia dwóch pojedynczych.

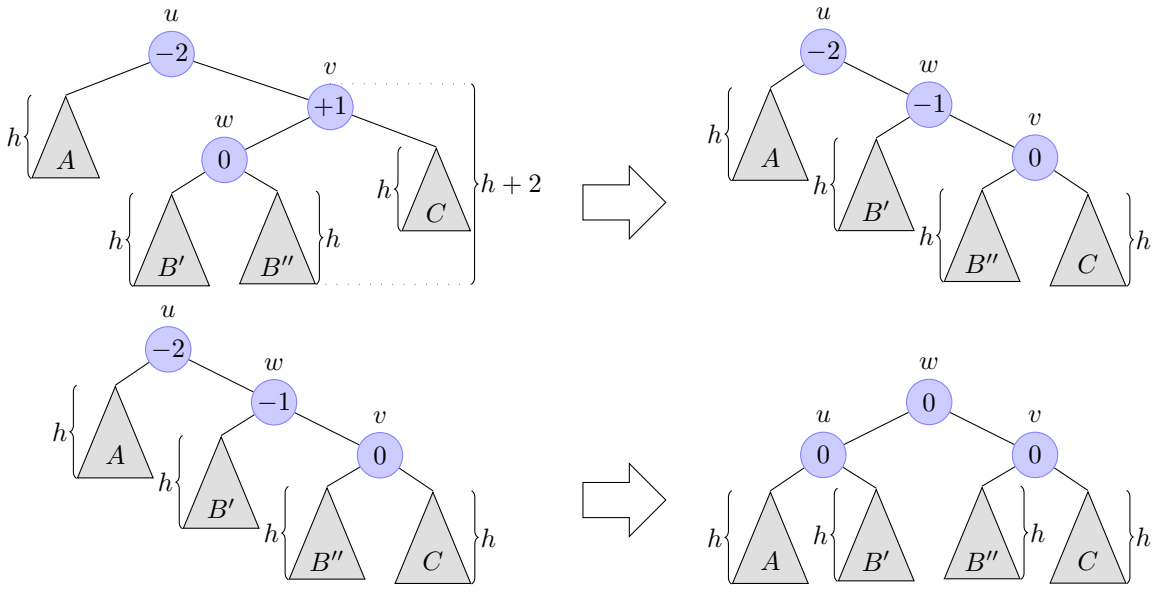
```
001 //pomocnicza funkcja zwracająca maksimum z dwóch argumentów
002 int max(int a, int b)
003 {
004     if (a > b)
005         return a;
006     return b;
007 }
008
009 struct AVL_node {
010     int key, height;
011     AVL_node *left, *right;
012
013     AVL_node(int _k)
014         : key(_k), height(1), left(NULL), right(NULL) {}
015 };
016
017 AVL_node *root = NULL;
018 AVL_node *gl_store;
019
020 //wysokość drzewa ukorzenionego w node
021 int AVL_height(AVL_node *node)
022 {
023     if (node == NULL)
024         return 0;
025     return node->height;
```



Rysunek 6.9: Uszczegółowienie poddrzewa B. Przynajmniej jedno z poddrzew B' i B'' musi mieć wysokość h – tutaj przyjęto, że oba mają właśnie taką.



Rysunek 6.10: Podwójny obrót lewo-prawo.



Rysunek 6.11: Podwójny obrót prawo-lewo.

```
026 }
027
028 //przeliczanie wysokości drzewa zgodnie z definicją
029 void AVL_recalc_height(AVL_node *node)
030 {
031     node->height = 1 + max(AVL_height(node->left), AVL_height(node->right));
032 }
033
034 AVL_node * AVL_rotate_left(AVL_node *node);
035 AVL_node * AVL_rotate_right(AVL_node *node);
036
037 //obrót w lewo
038 AVL_node * AVL_rotate_left(AVL_node *node)
039 {
040     //w razie potrzeby wywoływany jest obrót w prawo w poddrzewie
041     //tzn. wychodzi podwójny obrót prawo-lewo
042     if (AVL_height(node->right->left) > AVL_height(node->right->right))
043         node->right = AVL_rotate_right(node->right);
044
045     AVL_node *temp = node->right;
046     node->right = temp->left;
047     temp->left = node;
048
049     AVL_recalc_height(node);
050     AVL_recalc_height(temp);
051
052     return temp;
053 }
054
055 //obrót w prawo
056 AVL_node * AVL_rotate_right(AVL_node *node)
057 {
058     //w razie potrzeby wywoływany jest obrót w lewo w poddrzewie
059     //tzn. wychodzi podwójny obrót lewo-prawo
060     if (AVL_height(node->left->left) < AVL_height(node->left->right))
061         node->left = AVL_rotate_left(node->left);
062
063     AVL_node *temp = node->left;
064     node->left = temp->right;
065     temp->right = node;
066
067     AVL_recalc_height(node);
068     AVL_recalc_height(temp);
069
070     return temp;
071 }
072
073 //sprawdzenie czy podane poddrzewo nie zaburza warunku AVL,
074 //a jeśli tak, to następuje obrót w odpowiednią stronę;
075 //podwójne obroty implementowane są jako dwa pojedyncze,
076 //które "same" się wywołują (jak widać powyżej)
077 AVL_node * AVL_rebalance(AVL_node *node)
078 {
079     if (AVL_height(node->left) - AVL_height(node->right) > 1)
080         node = AVL_rotate_right(node);
081     if (AVL_height(node->right) - AVL_height(node->left) > 1)
082         node = AVL_rotate_left(node);
083     return node;
084 }
085
086 AVL_node * AVL_search(int key, AVL_node *node)
```

```
087 {
088     if (node == NULL)
089         return NULL;
090     if (key == node->key)
091         return node;
092     if (key < node->key)
093         return AVL_search(key, node->left);
094     return AVL_search(key, node->right);
095 }
096
097 AVL_node * AVL_insert(int key, AVL_node *node)
098 {
099     if (node == NULL)
100         return new AVL_node(key);
101
102     if (key < node->key)
103         node->left = AVL_insert(key, node->left);
104     else
105         node->right = AVL_insert(key, node->right);
106
107     //przywracanie własności AVL
108     AVL_recalc_height(node);
109     node = AVL_rebalance(node);
110
111     return node;
112 }
113
114 AVL_node * AVL_delete_helper(AVL_node *node);
115
116 AVL_node * AVL_delete(int key, AVL_node *node)
117 {
118     if (key == node->key) {
119         AVL_node *temp;
120         if (node->right == NULL) {
121             temp = node->left;
122             delete node;
123             return temp;
124         }
125
126         temp = AVL_delete_helper(node->right);
127         gl_store->left = node->left;
128         gl_store->right = temp;
129         delete node;
130
131         //przywracanie własności AVL
132         AVL_recalc_all(gl_store);
133         gl_store = AVL_rebalance(gl_store);
134
135         return gl_store;
136     }
137
138     if (key < node->key)
139         node->left = AVL_delete(key, node->left);
140     else
141         node->right = AVL_delete(key, node->right);
142
143     //przywracanie własności AVL
144     AVL_recalc_all(node);
145     node = AVL_rebalance(node);
146
147     return node;
```

```
148 }
149
150 AVL_node * AVL_delete_helper(AVL_node *node)
151 {
152     if (node->left == NULL) {
153         gl_store = node;
154         return node->right;
155     }
156
157     node->left = AVL_delete_min(node->left);
158
159     //przywracanie własności AVL
160     AVL_recalc_all(node);
161     node = AVL_rebalance(node);
162
163     return node;
164 }
```

Czy warto?

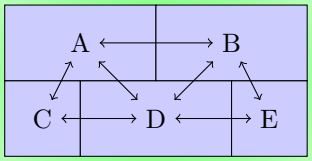
Generalnie nie oczekuje się, że zawodnicy na konkursach będą siedzieć i pieczołowicie klepać AVL, bo zapotrzebowanie na akurat taką strukturę danych wymyślili autorzy zadań i gdzieś koniecznie trzeba tego użyć, żeby zgarnąć sensowne punkty. Zwłaszcza, że w wielu przypadkach można sobie poradzić, korzystając z STL i zawartych tamże struktur `set` i `map`, które również są implementacją drzew binarnych o logarytmicznym czasie działania (nie są to drzewa AVL, tylko tzw. *drzewa czerwono-czarne*).

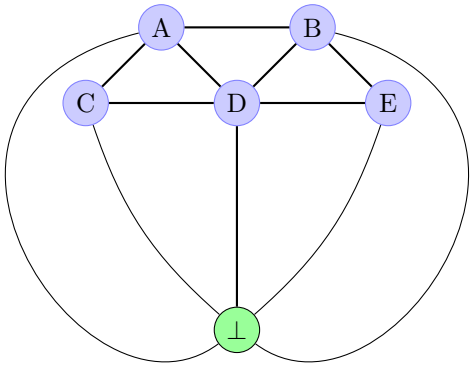
Tym niemniej znając różne „ponadprogramowe" algorytmy i struktury danych można czasami pójść na skróty. Zamiast gimnastykować się nad fikuśnym rozwiązaniem wymyślonym przez autora, który zakłada, że AVL nikt pisać nie będzie (lub nie przewiduje tego syllabus konkursu), można trochę więcej kodu wyprodukować i błyskawicznie rozwiązać zadanie.

Nie należy za to dać się ponosić – jeśli np. potrzebujemy struktury danych, w której pojawiają się i znikają jakieś obiekty, ale zawsze szukamy elementu najmniejszego czy największego wedle jakiegoś kryterium, nie ma sensu babrać się AVL ani nawet BST, skoro można użyć kopca binarnego.

Część III

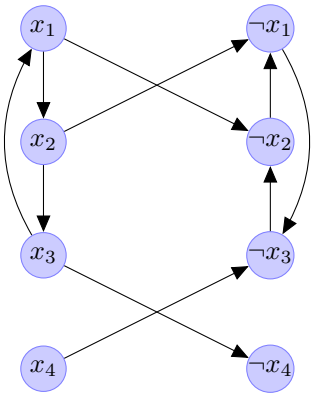
Teoria grafów





$$(\neg x_1 \vee \neg x_2) \wedge (x_3 \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee \neg x_1) \wedge (\neg x_3 \vee \neg x_4)$$

$$\begin{aligned}
\neg x_1 \vee \neg x_2 &\iff \begin{cases} x_1 \Rightarrow \neg x_2 \\ x_2 \Rightarrow \neg x_1 \end{cases} \\
x_3 \vee \neg x_2 &\iff \begin{cases} \neg x_3 \Rightarrow \neg x_2 \\ x_2 \Rightarrow x_3 \end{cases} \\
x_1 \vee \neg x_3 &\iff \begin{cases} \neg x_1 \Rightarrow \neg x_3 \\ x_3 \Rightarrow x_1 \end{cases} \\
x_2 \vee \neg x_1 &\iff \begin{cases} \neg x_2 \Rightarrow \neg x_1 \\ x_1 \Rightarrow x_2 \end{cases} \\
\neg x_3 \vee \neg x_4 &\iff \begin{cases} x_3 \Rightarrow \neg x_4 \\ x_4 \Rightarrow \neg x_3 \end{cases}
\end{aligned}$$



7 Wprowadzenie i implementacja

Grafy bardzo często występują w zadaniach algorytmicznych, dlatego umiejętność sprawnego kodowania ich reprezentacji i algorytmów grafowych jest kluczem do rozwiązywania problemów na konkursach i olimpiadach.

Grafem nazwiemy taką parę $G = (V, E)$, w której V jest zbiorem zadanych obiektów (nazywanych *wierzchołkami grafu*) lub *węzłami*, a E jest relacją dwuargumentową pomiędzy tymi obiektami. Jeśli między dwoma wierzchołkami u i v zachodzi relacja, to powiemy, że są one połączone *krawędzią* w grafie, a zbiór E będziemy nazywać zbiorem krawędzi.

Mniej formalny przykład: weźmy sobie zbiór miast w Polsce, pomiędzy którymi istnieje jakaś sieć dróg. Drogi są dwukierunkowe (czyli jeśli z Gdyni można dojechać do Gdańska, to także z Gdańska można dojechać do Gdyni). Jeśli teraz utożsamimy każde miasto z wierzchołkiem, a każdą drogę pomiędzy dwoma miastami z krawędzią, to otrzymamy graf, będący reprezentacją sieci dróg w Polsce.

Taki graf jest relatywnie mało pomocny — droga z Gdańska do Gdyni jest zdecydowanie krótsza, niż droga z Gdańska do Warszawy, a takiej informacji w aktualnej reprezentacji grafu nie ma. Na razie wiemy tylko, między jakimi miastami istnieją drogi.

Każdej krawędzi możemy przypisać pewne wartości (*wagi*), oznaczające np. długość drogi w kilometrach, uzyskamy wtedy graf *ważony*. Poprzedni graf był *jednostkowy*.

Jeśli z jakiegoś powodu niektóre drogi są jednokierunkowe, to wtedy relacja zachodzi tylko w jedną stronę. Taki graf nazwiemy *skierowanym*, w przeciwieństwie do poprzedniego, *nieskierowanego*.

Konwencje notacyjne:

- oznaczenie liczby wierzchołków i krawędzi odpowiednio $|V|$ i $|E|$,
- oznaczenie krawędzi między parą wierzchołków: $u \to v$, czasami też $u \leftrightarrow v$ albo jako para (u, v),
- oznaczenie ścieżki między parą wierzchołków: $u \rightsquigarrow v$; oczywiście jednokrawędziowa ścieżka też jest ścieżką,
- jeśli graf jest nieskierowany, to liczbę przyłączonych krawędzi do określonego wierzchołka v nazywamy jego *stopniem*, oznaczenie: $deg(v)$,
- w grafach skierowanych rozróżniamy krawędzie wchodzące i wychodzące do/z określonego wierzchołka v, ich liczbę oznaczał będę odpowiednio $indeg(v)$ i $outdeg(v)$.

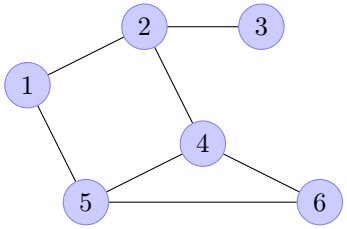
7.1 Macierz sąsiedztwa

Jednym z najprostszych do zaprogramowania (i zrozumienia) sposobów reprezentacji grafu jest użycie dwuwymiarowej tablicy, tzw. *macierzy sąsiedztwa* lub *tablicy sąsiedztwa*.

Załóżmy, że mamy graf nieskierowany o jednostkowych krawędziach. Jeśli istnieje krawędź między wierzchołkami o numerach (u, v), to tablica sąsiedztwa w indeksie `[u][v]` przyjmuje wartość 1. Ponieważ graf jest nieskierowany, to relacja zachodzi symetrycznie, czyli także pole `[v][u]` przyjmuje wartość 1. Dla każdego wierzchołka w, z którym wierzchołek v nie jest połączony krawędzią, tablica sąsiedztwa w indeksie `[w][v]` oraz `[v][w]` przyjmuje wartość 0.

Jak łatwo można zauważyć, modyfikacja tablicy sąsiedztwa do reprezentacji grafów innego typu jest bardzo łatwa. Na przykład, jeśli dany graf jest grafem skierowanym, to dla krawędzi $u \to v$ mamy `[u][v] == 1`, ale `[v][u] == 0`.

Jeśli chcemy reprezentować graf ważony, to zamiast jedynek zapisujemy w tablicy wagi krawędzi.



Rysunek 7.1: Przykładowy graf, wierzchołki etykietowane są kolejnym liczbami całkowitymi.

—	v_1	v_2	v_3	v_4	v_5	v_6
v_1	—	1	0	0	1	0
v_2	1	—	1	1	0	0
v_3	0	1	—	0	0	0
v_4	0	1	0	—	1	1
v_5	1	0	0	1	—	1
v_6	0	0	0	1	1	—

Tablica 7.1: Macierz sąsiedztwa grafu przykładowego (rys. 7.1). Jak widać, macierz jest symetryczna dla grafów nieskierowanych.

Przykładowa implementacja

Poniższa funkcja wczytuje opis grafu nieskierowanego jednostkowego i uzupełnia globalną tablicę dwuwymiarową `graph` informacjami o grafie.

W pierwszej linii wejścia znajduje się liczba wierzchołków grafu n i liczba krawędzi m. Dalej następuje m linii, w każdej para liczb u, v oznaczająca, że istnieje krawędź pomiędzy wierzchołkami o numerach u i v. Przez stałą `MAX` wyrażam maksymalną liczbę wierzchołków w grafie.

- $1 \leqslant n \leqslant MAX$
- $1 \leqslant u, v \leqslant n$

```
001 #include <cstdio>
002
003 using namespace std;
004
005 int graph[MAX][MAX];
006 int n, m;
007
008 void graph_matrix()
009 {
010     int u, v, i;
011
012     scanf("%d %d", &n, &m);
013
014     for (i = 0; i < m; ++i) {
015         scanf("%d %d", &u, &v);
016         graph[u][v] = 1;
017         graph[v][u] = 1;
018     }
019 }
```

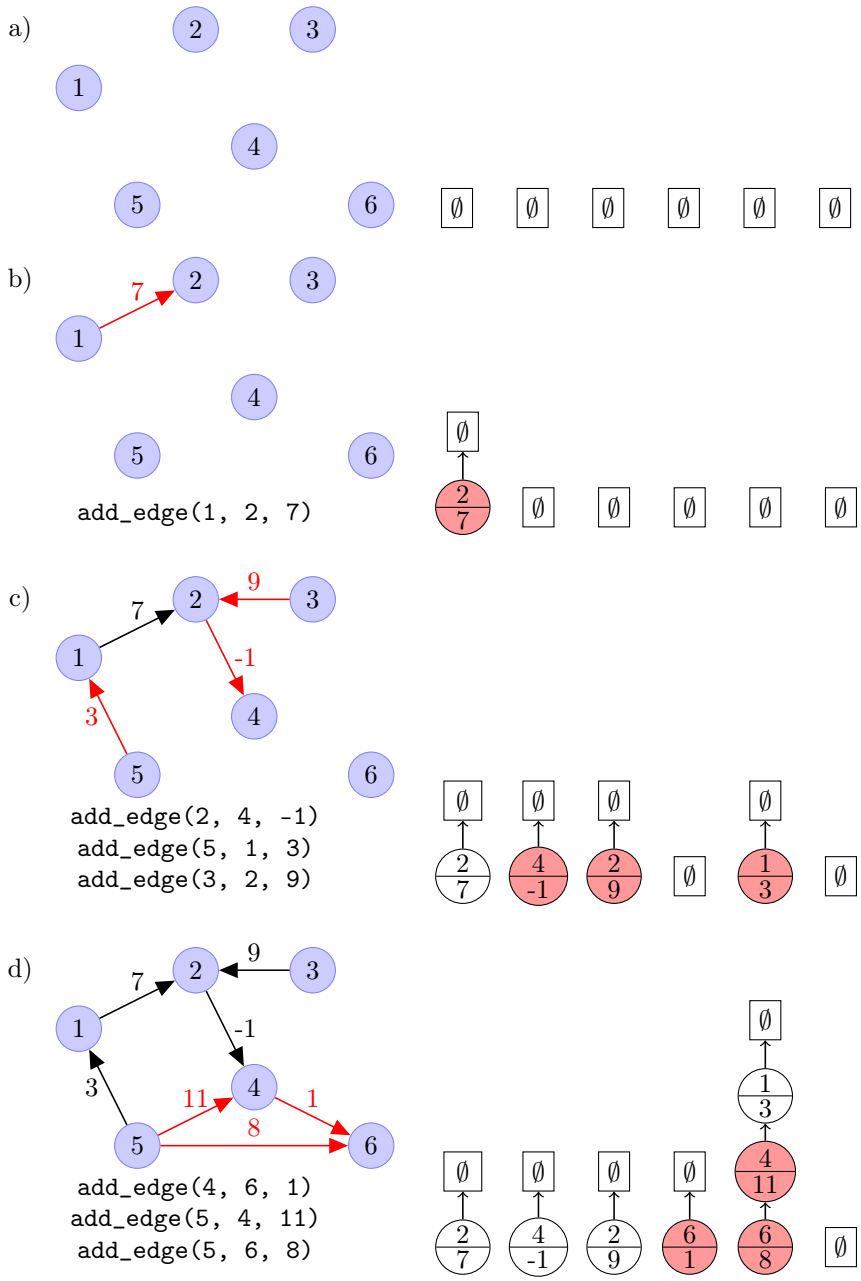
7.2 Listy wskaźnikowe

Wszechstronną metodą spamiętywania budowy grafu w programie jest użycie *list wskaźnikowych*. Każdy wierzchołek ma przypisaną jemu strukturę wskaźnikową, która zawiera:

- opis jednej krawędzi wychodzącej z tego wierzchołka,
- wskaźnik na następną krawędź wychodzącą z tego wierzchołka,
- tylko to, czego potrzebujemy — w przeciwieństwie do macierzy sąsiedztwa, która informuje nas także gdzie *nie ma* krawędzi, a która to informacja w większości przypadków nie jest nam do niczego potrzebna.

Ich przewagą nad macierzą sąsiedztwa jest oszczędność pamięci oraz fakt, że listę krawędzi wychodzących z zadanego wierzchołka mamy podaną na srebrnej tacy — nie trzeba iterować po całym wierszu (lub kolumnie) tablicy (która może okazać się całkiem duża), bo wszystkie krawędzie interesującego nas wierzchołka są w jednej liście.

Zysk szybkościowo-oszczędnościowy korzystania z list wskaźnikowych spada gwałtownie, jeśli mamy bardzo gęsty graf, w szczególności graf *pełny* (każda para wierzchołków jest połączona krawędzią). Zużycie pamięci jest wtedy generowane przez konieczność przechowywania wskaźnika do następnej badanej krawędzi. Na komputerach pracujących w trybie 32-bitowym wskaźnik zajmuje 4 bajty, więc jeśli w przykładowym grafie każda krawędź ma dwa pola z „użytecznymi”



Tablica 7.2: Dodawanie krawędzi do grafu zbudowanego na listach wskaźnikowych.

informacjami (załóżmy, że są to liczby czterobajtowe), to dodanie wskaźnika zwiększa zużycie pamięci dla każdej krawędzi o 50%.

Na listę wskaźnikową można patrzeć jak na tablicę o niesprecyzowanej długości, tzn. możemy ją sobie rozszerzać element po elemencie w razie nagłej potrzeby. Lista na początku jest pusta (wskaźnik ustawiony na `NULL`). Dodanie elementu polega na zapisaniu wskaźnika na pierwszy element na liście do zmiennej tymczasowej, dalej utworzeniu nowej jednoelementowej listy, której z kolei do pola pamiętającego następny element listy przypisujemy uprzednio zapisany wskaźnik. Przechodzenie po takiej liście polega na „wejściu" do pierwszego elementu i sukcesywnym przeglądaniu dalszych, aż natrafimy na koniec listy (`NULL`). Przykład działania list wskaźnikowych można zobaczyć w tabeli 7.2.

W istocie struktura, z której korzystamy jest tak naprawdę wskaźnikową implementacją stosu (rozdział 5.4), bardzo podobną do wskaźnikowej kolejki prostej (rozdział 5.3). Różnica leży oczywiście w braku dodatkowego wskaźnika do ostatniego elementu. Kolejność krawędzi niespecjalnie nas interesuje, więc równie dobrze moglibyśmy wykorzystać kolejkę wskaźnikową czy listę dwukierunkową i też by działało. W sytuacjach dowolności rozwiązania (lub doboru struktury danych) wybieramy najprostsze i najgłupsze co działa, czyli oczywiście stos — nie musimy utrzymywać niepotrzebnych wskaźników.

Przykładowa implementacja

Działamy na grafie skierowanym o krawędziach ważonych. Każdy wierzchołek grafu ma swoją listę krawędzi wychodzących z niego, czyli cały graf jest reprezentowany przez tablicę list wskaźnikowych. Dokładniej — lista krawędzi wychodzących z wierzchołka o numerze i znajduje się w `graph[i]`.

Funkcja `add_edge(u, v, c)` dodaje do tablicy list `graph` krawędź $u \to v$ o wadze c. Funkcja `get_list(u)` przegląda listę krawędzi wychodzących z wierzchołka u.

- $1 \leqslant n \leqslant MAX$
- $1 \leqslant u, v \leqslant n$

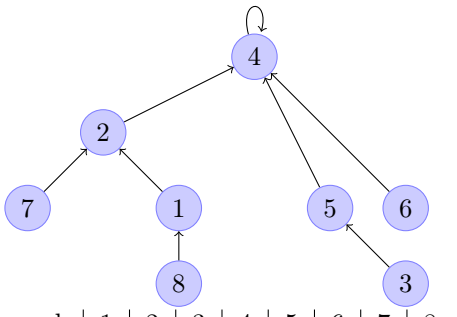
```
001 struct edge {
002     int v, cost;
003     edge *next;
004 };
005
006 edge *graph[MAX]; //tablica list wskaźnikowych
007
008 void add_edge(int u, int v, int c)
009 {
010     edge *temp;
011
012     temp = graph[u];
013     graph[u] = new edge;
014     graph[u]->v = v;
015     graph[u]->cost = c;
016     graph[u]->next = temp;
017 }
018
019 void get_list(int u)
020 {
021     edge *e;
022
023     for (e = graph[u]; e != NULL; e = e->next) {
024         //tu wstaw użyteczny kod
025     }
026 }
```

Bonusowa implementacja za pomocą STL <vector>

Jeśli nie wiesz co ten kod robi, to:

1. Nie używaj.
2. http://www.sgi.com/tech/stl
3. Patrz pkt. 1.

```
001 #include <vector>
002
003 struct edge {
004     int v, cost;
005
006     edge() {}
007     edge(int _v, int _c) : v(_v), cost(_cost) {}
008 };
009
010 vector <edge> graph[MAX]; //tablica list wskaźnikowych
011
012 void add_edge(int u, int v, int c)
013 {
014     graph[u].push_back(edge(v, c));
```



węzeł	1	2	3	4	5	6	7	8
rodzic	2	4	5	4	4	4	2	1

Rysunek 7.2: Drzewo ukorzenione w węźle 4. Zazwyczaj ustawia się rodzica korzenia na niego samego.

```
015 }
016
017 void get_list(int u)
018 {
019     vector <edge>::iterator e;
020
021     for (e = graph[u].begin(); e != graph[u].end(); ++e) {
022         //tu wstaw użyteczny kod
023     }
024 }
```

7.3 Specjalne klasy grafów

Niektóre grafy zbudowane są w pewien szczególny sposób, a jednocześnie pojawiają się względnie często zarówno w zadankach jak i rzeczywistym życiu, aby doczekały się własnej nazwy i klasyfikacji. Dla niektórych z tych grafów rozwiązywanie pewnych problemów staje się dużo prostsze, niż rozwiązywanie ich w przypadku ogólnym. Z tego powodu warto takie grafy umieć rozpoznawać.

Znaki szczególne wypisane w pobliżu każdego specjalnego typu grafów definiują go i zarazem określają jego własności.

Drzewa

Znaki szczególne

- spójny, nieskierowany graf (chyba, że istnieje ustalona hierarchia rodzic → potomek),
- n-wierzchołkowe drzewo ma dokładnie $n - 1$ krawędzi,
- pomiędzy dowolną parą węzłów istnieje dokładnie jedna ścieżka (brak cykli).

Większość drzew konstruowanych jest jako ukorzenione, tzn. pewien wierzchołek zostaje wyróżniony jako *korzeń drzewa*. Wtedy istnieje naturalny porządek rodzicielski — na samym szczycie drzewa znajduje się *korzeń*, którego wierzchołki sąsiadujące nazywa się dziećmi. Każde dziecko jest korzeniem swojego poddrzewa, czyli ma swoje dzieci, i tak dalej aż do liści (wierzchołków o stopniu równym 1).

Przy takim porządku każdy węzeł w drzewie (za wyjątkiem korzenia) ma dokładnie jednego rodzica. Można więc reprezentować drzewo za pomocą jednowymiarowej tablicy, w której dla każdego wierzchołka v zapamiętujemy kto jest jego rodzicem w drzewie. Przykład można zobaczyć na rysunku 7.2. Pomimo tej możliwości, często drzewo reprezentuje się jak zwykły graf, zwykle bowiem pamiętanie tylko rodzica nie wystarczy.

W przyrodzie często spotyka się drzewa o pewnej ustalonej maksymalnej liczbie dzieci na wierzchołek, np. drzewa binarne (maksymalnie dwoje dzieci), drzewa ternarne, czwórkowe itd.

Pseudodrzewa

Znaki szczególne

- pseudodrzewo to drzewo, do którego dodano jedną krawędź, generując w ten sposób cykl.

Pseudodrzewa mają postać albo jednego wielkiego cyklu, albo cyklu z drzewiastymi odnogami. Algorytmy działające dla drzew często można przystosować do działania na pseudodrzewach. Drzewowe algorytmy zwykle opierają się na ustaleniu korzenia i aplikowaniu obliczeń w poddrzewach. Z kolei przy pseudodrzewach często działa strategia rozpatrzenia dwóch przypadków: wybieramy parę wierzchołków u i v w taki sposób, aby ze sobą sąsiadowały i należały do cyklu. Następnie zapuszczamy odpowiedni algorytm dwukrotnie – jeden raz w u jako korzeniu tak powstałego drzewa i drugi raz w v.

Grafy funkcyjne

Znaki szczególne

- graf skierowany, w którym z każdego wierzchołka wychodzi dokładnie jedna krawędź.

Nietrudno zauważyć, że graf po zmianie krawędzi na nieskierowane jest pseudodrzewem. Nazwa wywodzi się z podobieństwa do funkcji jako pewnego odwzorowania $y = f(x)$. W tym przypadku zarówno dziedziną jak i przeciwdziedziną funkcji są wierzchołki grafu, a krawędź $u \to v$ równoznaczna jest $v = f(u)$.

Ponieważ zwykle mamy do czynienia z grafami skończonymi oraz funkcja określona jest dla każdego elementu dziedziny, to gdzieś w takim grafie musi pojawić się cykl. W skrajnym przypadku może to być cykl postaci $x = f(x)$ – wtedy taki graf ma kształt drzewa, którego korzeń ma „pętelkę" do siebie.

Acykliczne grafy skierowane

Znaki szczególne

- skierowany graf bez cykli (duh!).

Często określa się takie grafy skrótem DAG (*Directed Acyclic Graph*). Czasami węzły końcowe DAG-u (takie, których *outdeg* wynosi 0) nazywa się liśćmi, podobnie jak w drzewach. Wiele problemów w tych grafach można rozwiązać w czasie liniowym, zazwyczaj poprzez zastosowanie sortowania topologicznego (p. 8.5).

DAG-i zwykle reprezentują jakieś zależności czy czynności, które trzeba spełniać w określonej kolejności. Krawędź $u \to v$ może oznaczać, że aby zrobić v należy najpierw zrobić u. W świecie rzeczywistym spotyka się coś takiego instalując pakiety w systemach uniksopodobnych — jeśli chcę mieć odtwarzacz do filmów, to warto mieć biblioteki, które umożliwiają rysowanie obrazu na ekranie oraz jakieś kodeki. Biblioteki mają swoje zależności itd. Uroczy przykład podaje też *Wprowadzenie do algorytmów* — skarpetki zakłada się przed butami i tym podobne wariacje na temat ubierania się.

Multigrafy

Znaki szczególne

- pomiędzy parą wierzchołków może występować więcej niż jedna krawędź.

Multigrafy nie pojawiają się zbyt często, bo zwykle można sobie z nimi poradzić w kontekście zadania i w jakiś sposób sprowadzić do standardowego grafu, np. z kilku dostępnych krawędzi i tak wybieramy jedną najbardziej opłacalną, a pozostałe są na doczepkę. W niektórych przypadkach (zazwyczaj w problemach związanych z maksymalnym przepływem) kilka krawędzi występujących pomiędzy parą wierzchołków można skleić w jedną.

Grafy dwudzielne

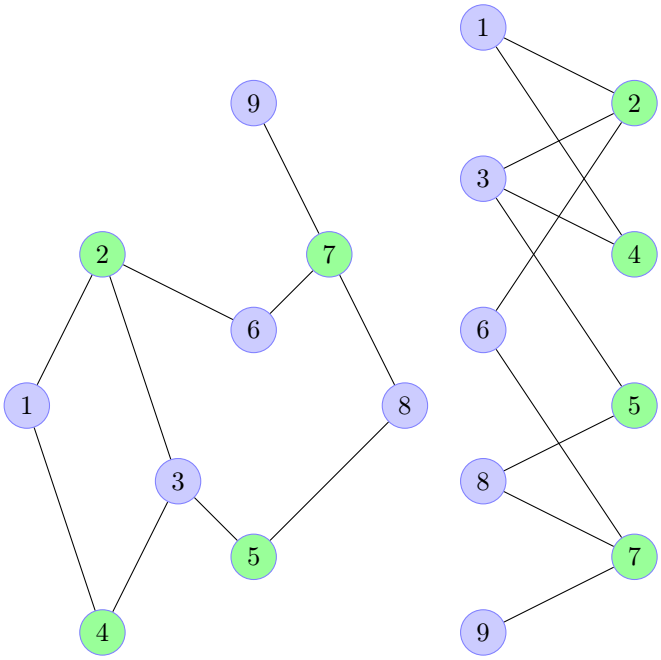
Znaki szczególne

- jeśli V to zbiór wierzchołków grafu, to da się go podzielić na takie podzbiory $V_1, V_2 \subseteq V$, że $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$ (czyli każdy wierzchołek należy do dokładnie jednej z tych dwóch grup) i dla każdej krawędzi (u, v) zachodzi

  $$(u \in V_1 \wedge v \in V_2) \vee (u \in V_2 \wedge v \in V_1)$$

  czyli każda krawędź łączy wierzchołki należące do różnych „grup",
- graf jest dwukolorowalny, tzn. można tak kolorować wierzchołki za pomocą dwóch kolorów, że nie będzie dwóch sąsiadujących wierzchołków o tym samym kolorze,
- nie istnieją cykle o nieparzystej długości.

Grafów dwudzielnych używa się zwykle do modelowania pewnych wzajemnie wykluczających się zależności, np. jedna z grup wierzchołków oznacza jakieś zadania, a druga pracowników, którzy mogą je wykonać. Niestety pracownicy są mało współbieżni i mogą wykonywać tylko jedno zadanie w danym momencie. Ponieważ każde zadanie jest w sam raz dla jednego pracownika, to nie ma także sensu przypisywać kilku pracowników do jednego zadania. Należy zatem tak przypisać zadania, aby jak najwięcej pracowników coś robiło.



Rysunek 7.3: Graf dwudzielny narysowany na dwa różne sposoby. Zwykle rysuje się je tak, jak po prawej.

Kliki (grafy pełne)

Znaki szczególne

- każdy wierzchołek grafu jest połączony krawędzią z każdym innym,
- n-wierzchołkowy graf ma zatem $\frac{n(n-1)}{2}$ krawędzi.

Oznacza się je $\mathbf{K}_n$, gdzie n jest rozmiarem kliki. Istnieje też wariant $\mathbf{K}_{m,n}$ — klika dwudzielna, gdzie jedna grupa wierzchołków ma rozmiar m, druga n i każdy wierzchołek z jednej z grup jest połączony z każdym z drugiej grupy. Trywialne kliki to na przykład:

- $\mathbf{K}_1$ – pojedynczy wierzchołek,
- $\mathbf{K}_2$ – dwa wierzchołki połączone krawędzią,
- $\mathbf{K}_3$ – trójkąt,
- $\mathbf{K}_{2,2}$ – kwadrat.

Grafy planarne

Znaki szczególne

- da się je narysować na płaszczyźnie w taki sposób, że krawędzie nie przecinają się albo bardziej precyzyjnie – krawędzie mają punkty wspólne tylko w wierzchołkach grafu,
- przy narysowaniu w powyższy sposób spełniają tzw. równanie Eulera:

  $$n - m + f = 2,$$

  gdzie n oznacza liczbę wierzchołków grafu, m liczbę krawędzi, a f liczbę *ścian*, czyli obszarów ograniczonych pewnym zestawem krawędzi tworzących cykl prosty (tzn. bez przechodzenia kilkakrotnie przez jeden wierzchołek) i nie zawierających „wewnątrz” innych krawędzi; do ścian wliczamy także nieskończoną zewnętrzną ścianę otaczającą cały graf,
- nie zawierają podgrafu *homeomorficznego* do $\mathbf{K_5}$ ani $\mathbf{K_{3,3}}$. Homeomorfizm w tym wypadku oznacza, że „zwijamy” krawędzie póki się da i patrzymy, czy nie powstała któraś z dwóch wymienionych klik. Zwinięcie polega na zmianie ciągu dwóch krawędzi w jedną, tzn. zmieniamy $v_1 \leftrightarrow v_2 \leftrightarrow v_3$ w $v_1 \leftrightarrow v_3$ i wywalamy w ogóle wierzchołek v_2. Możemy tak zrobić tylko wtedy, gdy wierzchołek v_2 sąsiaduje jedynie z v_1 i v_3 i żadnymi innymi wierzchołkami.

Jeśli w zadaniu jest napisane, że mamy jakąś sieć dróg i z jednej drogi można zjeżdżać w inną tylko w miastach, skrzyżowaniach czy innych wierzchołkach (bo drogi się nie przecinają), ale drogi mogą przebiegać estakadami lub tunelami, to mamy do czynienia z grafem nieplanarnym. Takie owijanie w bawełnę służy często uzasadnieniom historyjek w rodzaju „mamy ulice (czyli krawędzie), ale nie można zjechać z jednej do drugiej (byłoby to bez sensu z punktu widzenia teorii grafów), bo się nie przecinają, a jednocześnie graf nie jest planarny, więc nie da się go narysować tak, żeby krawędzie się nie przecinały na rysunku".

Przykładowy graf dwudzielny (rys. 7.3) jest grafem planarnym, co widać z lewej części rysunku.

8 Przeszukiwanie grafu

Umiejętność badania zawartości i budowy grafu za pomocą przeszukiwań przydaje się w niemal każdym zadaniu grafowym. Czasami jest celem samym w sobie, np. przy badaniu spójności, innym razem stanowi jedynie (istotny!) krok do rozwiązania bardziej złożonego problemu, jak znajdowanie ścieżek powiększających maksymalny przepływ.

8.1 Przeszukiwanie w głąb (DFS)

Algorytm przeszukiwania grafu w głąb (*Depth–First Search*) polega wybraniu sobie pewnego wierzchołka (*źródła*), z którego rozpoczynamy operację. Oznaczamy wierzchołek źródłowy jako już odwiedzony, następnie wyszukujemy dowolny nie odwiedzony jeszcze wierzchołek, do którego prowadzi krawędź ze źródła. Przechodzimy do tego wierzchołka, oznaczamy go jako odwiedzony, szukamy osiągalnego nieodwiedzonego wierzchołka, wchodzimy itd.

W momencie, kiedy nie mamy żadnych osiągalnych wierzchołków lub są one odwiedzone, DFS cofa się do ostatniego wierzchołka, z którego można jeszcze osiągnąć nieodwiedzone wierzchołki i kontynuuje algorytm w ich kierunku. Jeśli nawet w „głównym" źródle nie ma takich wierzchołków, to szukamy wierzchołków jeszcze nie odwiedzonych i umieszczamy tam nowe źródło, odpalamy DFS i tak aż do odwiedzenia całego grafu (chyba, że nie jest to konieczne, bo potrzebna informacja została już wyszukana). Widać zatem, że zgodnie z nazwą algorytm usiłuje możliwie jak najbardziej pogłębiać aktualną ścieżkę przeszukiwania i dopiero przy wpadnięciu w ślepy zaułek wykonywany jest nawrót i poszukiwanie innej ścieżki do jak największego pogłębienia.

Kolejność przechodzenia wierzchołków w grafie może dużo powiedzieć o jego strukturze. Przeszukiwanie w głąb jest istotną częścią algorytmów dzielących graf na spójne składowe lub wyszukujących miejsc, w których usunięcie wierzchołka czy krawędzi rozspójni graf (odpowiednio *punkty artykulacji* i *mosty dwuspójne*).

Przykładowa implementacja

Dany jest graf skierowany, reprezentowany przez listy wskaźnikowe. Funkcja `dfs(v)` oznacza wierzchołek v w tablicy `visited` jako odwiedzony i szuka osiągalnych z v nieodwiedzonych wierzchołków, żeby rozprzestrzeniać się dalej.

```
001 struct edge {
002     int v;
003     edge *next;
004 };
005
006 edge *graph[MAX]; //tablica list wskaźnikowych
007 bool visited[MAX];
008
009 void dfs(int v)
010 {
011     edge *e;
012     visited[v] = true;
013     //tu wstaw użyteczny kod
014
015     for (e = graph[v]; e != NULL; e = e->next)
016     //albo tu
017         if (!visited[e->v])
```

```
018             dfs(e->v);
019
020     //a może i tu
021 }
```

8.2 Przeszukiwanie wszerz (BFS)

Przeszukiwanie wszerz (*Breadth-First Search*) przegląda graf „poziomami", tzn. ustalamy w pewnym wierzchołku źródło wyszukiwania i oznaczamy jako odwiedzony. W pierwszej kolejności odwiedzone zostają wierzchołki położone najbliżej źródła (oddalone tylko o jedną krawędź), dalej wierzchołki oddalone o jedną krawędź od właśnie odwiedzonych itd. Przebieg kolejnych kroków algorytmu BFS przypomina układ poziomic na mapie.

Ponieważ występuje konieczność spamiętywania wierzchołków, z których trzeba teraz wyszukiwać następnych kandydatów do odwiedzenia, potrzebna jest dodatkowa pamięć. Najłatwiej użyć w tym celu zwykłej kolejki typu FIFO.

DFS kontra BFS

Przeszukiwania w głąb należy użyć, kiedy oczekiwany wynik znajduje się głęboko w drzewie przeszukiwań, a dodatkowo może się okazać, że istnieje kilka poprawnych rozwiązań, z których należy wybrać „najlepsze" (według określonych kryteriów). Względem głębokości oczywiście także nie należy przesadzać, coby stosu nie przepełnić.

Przeszukiwanie wszerz jest dużo lepsze, jeśli oczekiwany wynik znajduje się możliwie płytko w drzewie poszukiwań. Przykład: znaleźć drogę opuszczenia labiryntu, która jest najkrótsza. Wykorzystanie DFS-a do rozwiązania tego problemu wymagałoby sprawdzenia każdej możliwej kombinacji ruchów po labiryncie aby upewnić się, że znaleźliśmy wśród nich najkrótszą. BFS zapewnia dużo szybsze rozwiązanie przy dość niewielkim zużyciu dodatkowej pamięci — poruszając się „warstwowo" wiemy, że pierwsza znaleziona trasa wyjścia z labiryntu będzie najkrótszą lub jedną z kilku najkrótszych; w obu wypadkach lepiej być nie może, więc pierwszy znaleziony wynik jest optymalny.

Przykład modelowego zadania na przeszukiwanie w głąb: na szachownicy rozmiaru $N \times N$ należy rozstawić dokładnie N niebijących się hetmanów. Już dla szachownic rozmiaru niewiele większego od standardowego 8×8 drzewo poszukiwań rośnie zbyt gwałtownie, aby w sensownej pamięci spamiętywać wszystkie możliwe stany planszy na kolejce FIFO[1]. Uznając ustawienie nowego hetmana za ruch w głąb drzewa wyszukiwania od razu widać, że poprawne rozstawienia hetmanów znajdują się tylko i wyłącznie na skraju drzewa (głębokość N). Jeśli znajdujemy się w położeniu, gdzie nie można dodać nowego hetmana (ale nie jest to położenie końcowe), to cofamy się o krok w drzewie poszukiwań i próbujemy innej ścieżki. Takie wyczerpujące wyszukiwanie nazywa się wyszukiwaniem z nawrotami (*backtracking*). DFS także będzie odpowiedni, jeśli należy znaleźć *wszystkie* konfiguracje N niebijących się hetmanów.

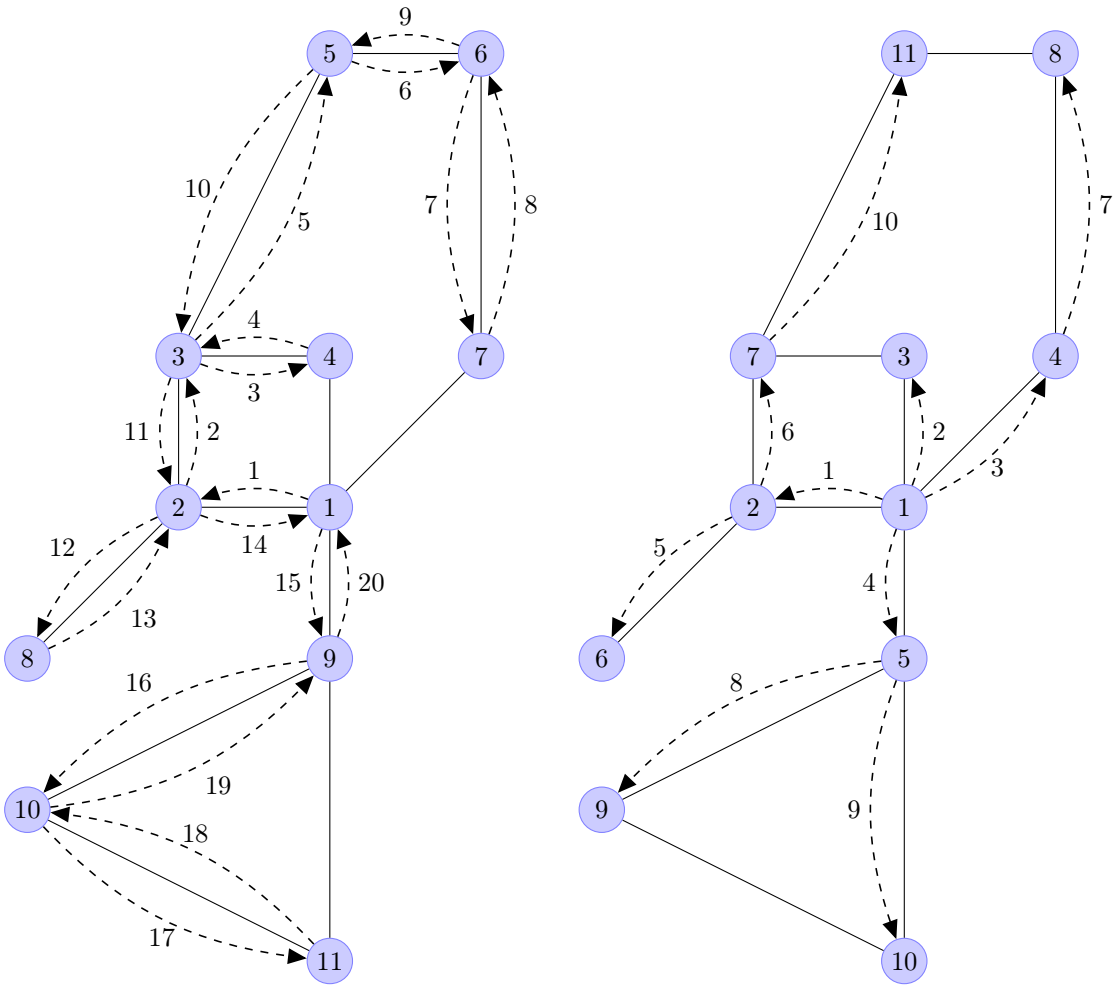
Istnieje oczywiście także cała klasa problemów, dla których użycie dowolnego z tych algorytmów jest jednakowo dobre i wybranie metody jest wtedy osobistą preferencją (np. badanie spójności grafu nieskierowanego). Wizualizację jednego z możliwych przebiegów algorytmów DFS i BFS można zobaczyć na rysunku 8.1.

Przykładowa implementacja

Dany jest graf skierowany, którego reprezentacją są listy wskaźnikowe. Funkcja `bfs(v)` rozpoczyna przeszukiwanie wszerz od wierzchołka v. Korzystam z najprostszej kolejki FIFO z rozdziału 5.1.

```
001 //kolejka
002 int Q[MAX];
003 int head = 0, tail = 0;
004 int queue_front();
005 void queue_push(int);
006 bool queue_empty();
007
008 //graf
009 struct edge {
010     int v;
011     edge *next;
012 };
013
014 edge *graph[MAX]; //tablica list wskaźnikowych
015 bool visited[MAX];
016
017 void bfs(int v)
```

[1] Nie uwzględniając haszowania.



Rysunek 8.1: Na lewo DFS, na prawo BFS. Numery w wierzchołkach oznaczają kolejność ich odwiedzania, zaczynając od numeru 1. Przerywane strzałki oznaczają kierunki działania przeszukiwania, a liczby na nich „czas", w którym dane przejście zostało przetworzone.

```
018 {
019     edge *e;
020     int cur;
021
022     queue_push(v);
023     visited[v] = true;
024
025     while (!queue_empty()) {
026         cur = queue_front();
027         for (e = graph[cur]; e != NULL; e = e->next)
028             if (!visited[e->v]) {
029                 visited[e->v] = true;
030                 queue_push(e->v);
031                 //tu wstaw użyteczny kod
032             }
033     }
034 }
```

8.3 Spójność i silna spójność

Spójność to własność grafu mówiąca o tym, czy dla określonej pary wierzchołków u, v istnieje ścieżka $u \rightsquigarrow v$. W przypadku grafów nieskierowanych badanie spójności jest proste – możemy użyć dowolnego z algorytmów wyszukiwania, wystartować w u i na końcu sprawdzić, czy odwiedziliśmy v. Jeśli tak, to znaczy że oba wierzchołki leżą w tej samej

spójnej, w przeciwnym wypadku leżą w różnych. Oczywiście można w ten sam sposób sprawdzić, czy cały graf jest spójny: jeśli po zapuszczeniu dowolnego z wyszukiwań z wybranego wierzchołka okaże się, że któregoś węzła nie odwiedziliśmy, to wtedy oczywiście graf spójny nie jest.

Czasami potrzebujemy odpowiadać szybko na zapytania postaci „czy dwa określone wierzchołki znajdują się w tej samej spójnej składowej". Wiedząc, że każde przeszukiwanie odpalone w dowolnym wierzchołku pewnej spójnej przejdzie ją w całości, możemy odwiedzane wierzchołki niejako kolorować: przed każdym uruchomieniem wyszukiwania tworzymy nowy, unikalny „kolor" (identyfikowany np. kolejną liczbą całkowitą), a następnie wszystkie odwiedzone w tym przeszukiwaniu wierzchołki kolorujemy. Potem wystarczy tylko dla nadchodzących zapytań porównywać kolory wierzchołków, których zapytania dotyczą.

Sytuacja nieco komplikuje się gdy mamy do czynienia z grafami skierowanymi. Mamy wtedy do czynienia z dwoma rodzajami spójności:

- spójność *słaba*, która jest równoznaczna spójności dla takiego samego grafu, ale w wersji nieskierowanej,
- spójność *silna*, która oznacza, że dla dowolnej pary wierzchołków u i v istnieje zarówno ścieżka $u \rightsquigarrow v$ jak i $v \rightsquigarrow u$ (przy zachowaniu skierowania krawędzi).

Ze spójnością słabą potrafimy już sobie radzić. Okazuje się, że spójność silną można rozstrzygnąć za pomocą algorytmu przeszukiwania w głąb.

Graf skierowany można w jednoznaczny sposób podzielić na podgrafy (*silnie spójne składowe*) tak, żeby każdy taki podgraf był silnie spójny oraz żeby były one maksymalne, tzn. nie dało się do żadnego z nich dołożyć pewnego niepustego zbioru wierzchołków bez naruszania silnej spójności. Czasami silnie spójne składowe „zwija się" (*kondensuje*) do pojedynczych wierzchołków, otrzymując w ten sposób nowy graf – graf silnie spójnych składowych. Nietrudno zauważyć, że tak otrzymany graf jest DAGiem (rys. 8.2).

Weźmy graf G, w którym będziemy szukać silnie spójnych składowych. Ponadto wygenerujmy pomocniczy graf G^T, który jest grafem G z odwrotnie skierowanymi krawędziami (tzw. *graf transponowany*). Kluczowa dla algorytmu znajdowania silnie spójnych jest obserwacja, że grafy G i G^T mają dokładnie takie same silnie spójne składowe.

Algorytm przejdzie dwukrotnie cały graf za pomocą przeszukiwania w głąb: najpierw przy zadanym początkowo skierowaniu krawędzi, a potem przy odwróconym (czyli przejdzie tak naprawdę G^T). W pierwszym przejściu dla każdego wierzchołka wyliczymy jego *czas przetworzenia* przez DFS, czyli moment, w którym ten wierzchołek nie ma nieodwiedzonych jeszcze sąsiadów i procedura przeszukująca wychodzi z niego.

Oczywiście może się zdarzyć tak, że nawet jeśli graf G jest słabo spójny, to jedno przejście DFS nie spowoduje odwiedzenia wszystkich wierzchołków. Na rysunku 8.2 taka sytuacja będzie miała miejsce, jeśli zaczniemy przeszukiwanie od dowolnego węzła znajdującego się poza spójną składową oznaczoną literą A. Nietrudno także zbudować przykładowe grafy, w których nie istnieje taki wierzchołek, z którego rozpoczęcie przeszukiwania gwarantuje nam odwiedzenie całego grafu. Z tego wynika, że po zakończeniu przeszukiwania należy przejrzeć tablicę odwiedzonych wierzchołków aby upewnić się, że istotnie przeszukiwanie przebyło cały graf. Jeśli nie, to uruchamiamy kolejne instancje DFS w nieodwiedzonych wierzchołkach i tak dalej aż do zwiedzenia całego grafu.

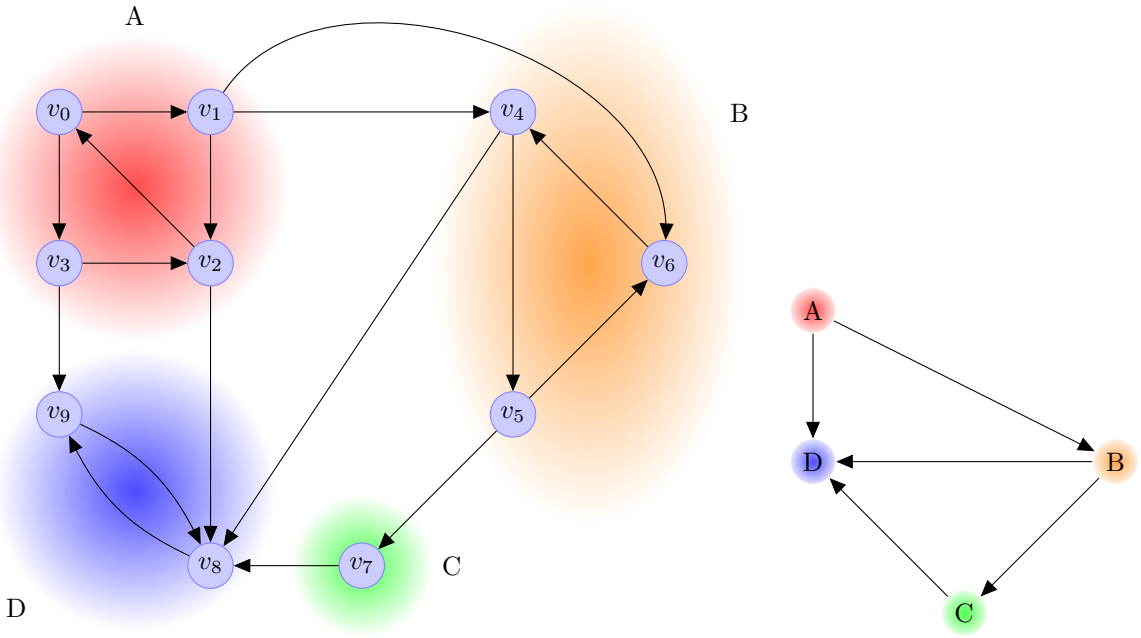
Drugie przejście niewiele różni się od pierwszego. Najistotniejszą zmianą (oprócz działania na grafie G^T) jest ustalenie pewnego porządku przeglądania grafu. W pierwszej fazie mogliśmy zapuszczać DFS skądkolwiek i w jakiejkolwiek kolejności. W drugiej robimy to w kolejności od wierzchołka o największym czasie przetworzenia. Działając w ten sposób jedno zapuszczenie algorytmu DFS w G^T odwiedzi tylko i wyłącznie wierzchołki należące do dokładnie jednej silnie spójnej składowej.

Cały algorytm działa w zasadzie w czasie $\mathcal{O}(n)$ (dwa proste przejścia po grafie) z dokładnością do pewnego szczegółu. Skoro wyliczamy dla wierzchołków ich czasy przetworzenia, a następnie drugie przeszukiwanie uruchamiamy według tej kolejności, to najpewniej chcielibyśmy dokonać jakiegoś sortowania, co psuje złożoność obliczeniową. W praktyce możemy za darmo otrzymać pożądaną kolejność wierzchołków po prostu odkładając je na stosie podczas wychodzenia z nich podczas pierwszej fazy algorytmu. Pomysł ten został uwzględniony w poniższym kodzie.

Przykładowa implementacja

Mamy n-wierzchołkowy graf skierowany reprezentowany przez listy wskaźnikowe (`graph`) oraz ten sam graf z odwróconym skierowaniem krawędzi (`trans`). Pierwsza faza odwiedza cały graf za pomocą zwykłego przeszukiwania w głąb (funkcja `dfs_graph`), odkładając przy tym na stosie wierzchołki w odpowiedniej kolejności. Druga faza korzysta z tego porządku i oznacza silnie spójne składowe za pomocą DFS uruchomionego na grafie transponowanym (funkcja `dfs_trans`). Liczba silnie spójnych składowych zapamiętywana jest w zmiennej `components` i jest wykorzystywana do identyfikowania wierzchołków grafu według przynależności do odpowiednich składowych (tablica `component_tab`). Korzystam ze stosu z rozdziału 5.4.

```
001 //stos
002 int S[MAX], height;
003 int stack_pop();
```



Rysunek 8.2: Graf skierowany z pokolorowanymi silnie spójnymi składowymi, oznaczonymi kolejnymi literami od A do D. Po prawej graf silnie spójnych składowych. Jak widać pojedynczy wierzchołek także może być silnie spójną składową.

```
004 void stack_push(int);
005 bool stack_empty();
006
007 //graf
008 struct edge {
009     int v;
010     edge *next;
011 };
012
013 edge *graph[MAX], *trans[MAX];
014 bool visited[MAX];
015
016 //liczba silnie spójnych
017 int components;
018
019 //do której silnie spójnej należy dany wierzchołek
020 int component_tab[MAX];
021
022 void dfs_graph(int v)
023 {
024     edge *e;
025     visited[v] = true;
026
027     for (e = graph[v]; e != NULL; e = e->next)
028         if (!visited[e->v])
029             dfs_graph(e->v);
030
031     stack_push(v);
032 }
033
034 void dfs_trans(int v)
035 {
036     edge *e;
```

```
037     component_tab[v] = components;
038
039     for (e = trans[v]; e != NULL; e = e->next)
040         if (component_tab[e->v] == 0)
041             dfs_trans(e->v);
042 }
043
044 void calc_components()
045 {
046     int i, v;
047
048     //pierwsze przejście
049     for (i = 1; i <= n; ++i)
050         if (!visited[i])
051             dfs_graph(i);
052
053     //drugie przejście i oznaczenie spójnych
054     while (!stack_empty()) {
055         v = stack_pop();
056
057         //może oznaczyliśmy już ten wierzchołek?
058         if (component_tab[v] == 0) {
059             //jednak nie, czyli mamy nową silnie spójną
060             ++components;
061             dfs_trans(v);
062         }
063     }
064 }
```

8.4 Punkty artykulacji, mosty, dwuspójne składowe

W grafach nieskierowanych można określić kilka dodatkowych pojęć związanych ze spójnością:

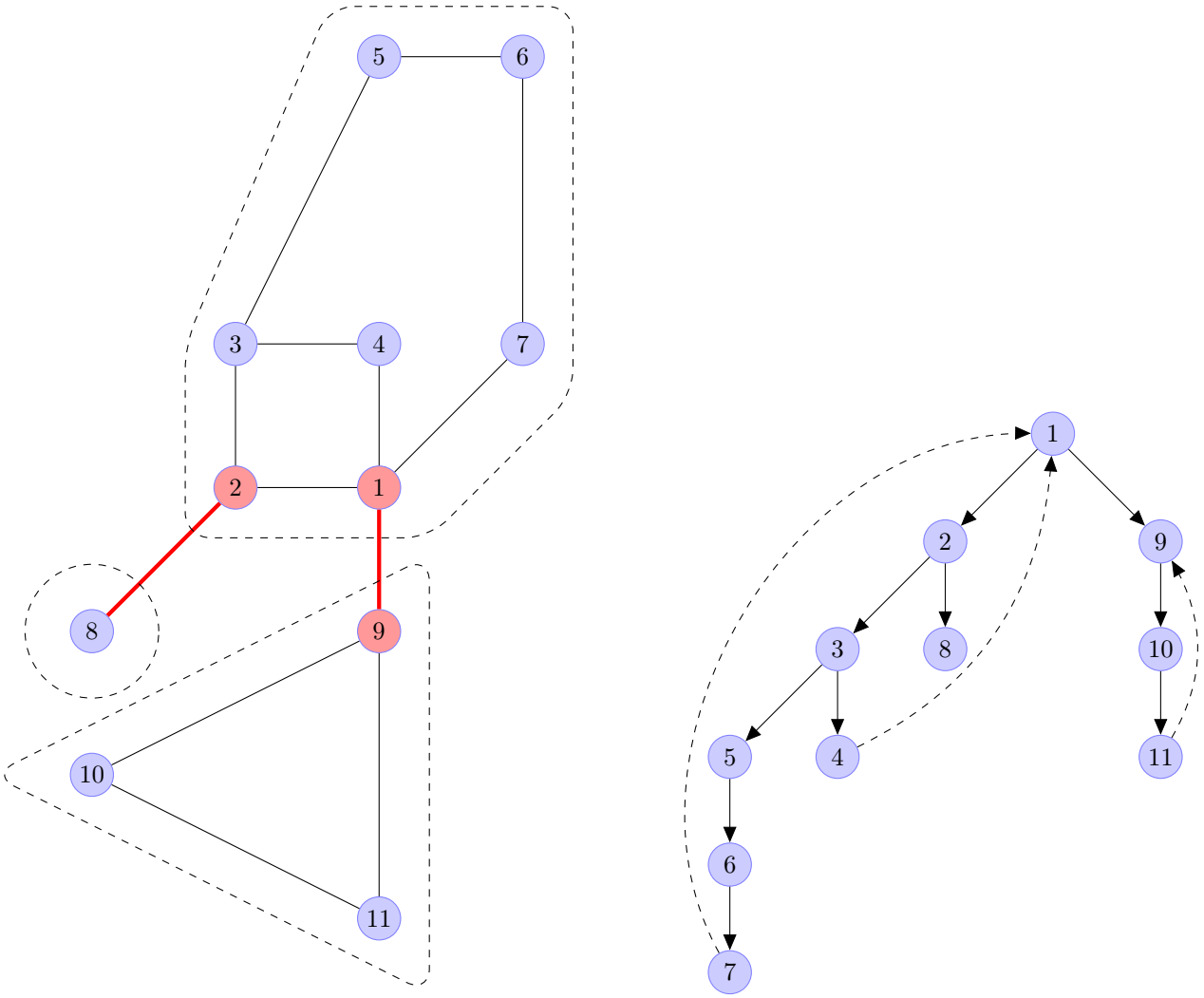
- **Punkt artykulacji** to wierzchołek, którego usunięcie powoduje zwiększenie liczby spójnych składowych grafu.
- Analogicznie **most** to krawędź, której usunięcie zwiększa liczbę spójnych składowych. Każda krawędź, która nie jest mostem leży na jakimś cyklu.
- **Dwuspójna składowa** to maksymalny spójny podgraf, który nie zawiera żadnego mostu. Każda krawędź grafu jest albo mostem, albo należy do jakiejś dwuspójnej składowej. W dwuspójnej składowej istnieją przynajmniej dwie rozłączne ścieżki pomiędzy każdą parą wierzchołków.

Przez „maksymalne" powyższej definicji rozumiemy takie spójne podgrafy, których nie możemy rozszerzyć przez dodanie jakiejś krawędzi bez naruszenia warunku o braku mostów. Nie należy maksymalności mylić z „największością" – każdy byt największy jest jednocześnie maksymalny, ale nie każdy maksymalny jest największy. Analogicznie do dwuspójnych można definiować k-spójne składowe jako takie podgrafy, w których należy usunąć k odpowiednio wybranych krawędzi, aby doprowadzić do jego rozspójnienia.

Znajdowanie punktów artykulacji i mostów jest ściśle związane z przeszukiwaniem grafu w głąb. Jeśli rozrysujemy sposób przejścia algorytmu DFS w grafie G, to otrzymamy pewne drzewo T_G (rys. 8.3). Okazuje się, że drzewo to niesie bardzo wiele informacji o strukturze grafu.

W rozdziale 8.3 kluczowe znaczenie dla algorytmu znajdowania silnie spójnych składowych miało zapamiętanie momentu wyjścia z danego wierzchołka. W tym przypadku będzie nas interesował czas wejścia – oznaczymy go $entry(u)$ dla wierzchołka u. Oprócz tego zdefiniujemy funkcję $low(u)$ jako minimalny numer $entry(v)$ dla takich wierzchołków v, które możemy osiągnąć idąc **w drzewie DFS** ścieżką $u \rightsquigarrow v$. Zgodnie ze skierowaniem krawędzi na rysunku 8.3 możemy iść w T_G jedynie w dół po krawędziach drzewowych i w górę po niedrzewowych. Nakładamy jednak dodatkowe obostrzenie: ścieżka $u \rightsquigarrow v$ może prowadzić jedynie w dół drzewa z dokładnością do ostatniej krawędzi, która może prowadzić w górę drzewa. Takie zdefiniowanie dopuszczalnych ścieżek prowadzi do rekurencyjnej definicji funkcji low, a ta z kolei do wygodnej implementacji:

$$\begin{aligned} low(u) = \min(\{entry(u)\} \cup \\ \{entry(v) \colon u \to v \text{ nie jest krawędzią w } T_G\} \cup \\ \{low(v) \colon u \to v \text{ jest krawędzią w } T_G\}) \end{aligned}$$



Rysunek 8.3: Graf G z rysunku 8.1 z numeracją wierzchołków odpowiadającą przejściu algorytmu DFS. Punkty artykulacji i mosty zaznaczone są kolorem czerwonym. Poszczególne dwuspójne składowe zostały otoczone linią przerywaną. Po prawej stronie drzewo T_G przejścia algorytmu DFS: liniami ciągłymi zaznaczono krawędzie drzewowe, przerywanymi niedrzewowe.

Pierwsza część powyższej definicji jest oczywista: $low(u)$ nigdy nie będzie większe, niż $entry(u)$. Druga część wynika z definicji pozwalającej na użycie jednej krawędzi niedrzewowej: sprawdzamy, czy jej użycie poprawia $low(u)$. Trzecia część jest konsekwencją dowolnego schodzenia w dół T_G oraz drugiej części: możemy wdepnąć do dowolnego węzła v leżącego w poddrzewie DFS ukorzenionym w u i będąc w tym węźle użyć jednej krawędzi niedrzewowej, tym samym „przejmując" $low(v)$.

Intuicyjnie wartość $low(u)$ mówi nam jak wysoko w T_G możemy uciec rozpoczynając podróż z poddrzewa ukorzenionego w u. Jeśli weźmiemy teraz jakiś węzeł u oraz jego potomka v w T_G takiego, że $low(v) \geqslant entry(u)$ to widać, że z v nie można uciec w górę drzewa inaczej, niż przechodząc przez u w oryginalnym grafie G. Z tego wynika, że usunięcie u odetnie v od wszystkich wierzchołków powyżej u, a zatem u jest punktem artykulacji.

Analogiczne kryterium sprawdza istnienie mostów: jeśli mamy w T_G węzeł u i potomka v, to krawędź $u \leftrightarrow v$ jest mostem gdy $low(v) > entry(u)$. Ostra nierówność jest zabezpieczeniem przed multigrafami: jeśli w T_G mamy $u \to v$ oraz $low(v) = entry(u)+1$ to z pewnością u jest punktem artykulacji, a potencjalnie także v – wystarczy, że v nie będzie liściem w T_G. Ponadto krawędź je łącząca jest mostem w G. Natomiast w sytuacji $u \rightrightarrows v$ możemy mieć już co najwyżej $low(v) = entry(u)$. Wtedy ponownie u jest punktem artykulacji i może być nim także v, ale żadna z krawędzi je łączących nie jest mostem. Łatwo zapamiętać która nierówność jest ostra, a która nie zauważając, że zwykle jest więcej punktów artykulacji niż mostów, a nieostra nierówność jest łagodniejsza i łatwiej spełnialna (więc dotyczy punktów artykulacji, a nie mostów).

W powyższym rozumowaniu schowała się jedna sytuacja szczególna, której nie wolno przeoczyć. Weźmy korzeń u drzewa T_G oraz dowolnego jego potomka v. Ponieważ $entry(u) = 1$ (bo u to korzeń), zawsze zachodzi $low(v) \geqslant entry(u)$, co sugeruje, że korzeń zawsze będzie punktem artykulacji. Jest to oczywiście nieprawda, co można zaobserwować odpalając DFS w przykładowym grafie z rysunku 8.3 w dowolnym węźle niebędącym punktem artykulacji.

Jeśli korzeń ma dokładnie jednego potomka v w T_G, to znaczy, że obeszliśmy cały graf G bez konieczności cofania się do korzenia. Jeśli usunęlibyśmy wierzchołek u i odpalili DFS w v, to obeszlibyśmy cały graf $G - \{v\}$. Zatem cały graf będzie spójny nawet, jeśli usuniemy z niego wierzchołek u. Jeśli natomiast u ma więcej niż jednego potomka, to jego usunięcie podzieli graf na tyle spójnych składowych, ile jest potomków. Znaleźliśmy w ten sposób kryterium określające, czy korzeń T_G jest punktem artykulacji: jest nim tylko wtedy, gdy posiada więcej niż jednego potomka w T_G.

Przykładowa implementacja

Mamy n-wierzchołkowy graf nieskierowany reprezentowany przez listy wskaźnikowe. Funkcja `dfs_low` odpalona zostaje dla dowolnego węzła `v` z argumentem `parent` ustawionym na -1 (czyli np. `dfs_low(1, -1)`). Liczba -1 w argumencie reprezentującym rodzica w drzewie DFS oznacza, że ten węzeł jest korzeniem drzewa. Z tego wynika zasadność umieszczenia na końcu funkcji `dfs_low` warunku testującego, czy korzeń drzewa jest punktem artykulacji.

Całość działa oczywiście w czasie liniowym – każdy wierzchołek i każda krawędź grafu rozpatrywana jest dokładnie raz. W czasie przeszukiwania grafu patrzymy, czy sprawdzany wierzchołek już był odwiedzony czy nie i w zależności od tego stosownie aktualizujemy wartość `low`, zgodnie z podaną wcześniej definicją.

Załączony program znajduje jedynie punkty artykulacji. Modyfikacja go do postaci znajdującej mosty albo dwuspójne składowe jest prosta.

- Dla mostów wystarczy zmienić nierówność sprawdzającą czy `v` jest punktem artykulacji na nierówność ostrą. Oczywiście aby zapamiętać dla każdej krawędzi czy jest mostem, musimy mieć albo jakiś sposób identyfikowania jej (tak jak identyfikujemy wierzchołki kolejnymi liczbami całkowitymi), albo dodatkowe pole w strukturze `edge`.
- Dwuspójne składowe będziemy zapamiętywać jako zbiory krawędzi. Możemy je znajdować odkładając na stosie (rozdział 5.4) kolejne krawędzie podczas przechodzenia grafu algorytmem DFS. Jeśli cofając się z przejścia w głąb trafimy na węzeł `v` będący punktem artykulacji, to niejako odcinamy całe poddrzewo DFS jako pojedynczą dwuspójną składową, co odpowiada zbieraniu ze stosu kolejnych krawędzi aż do napotkania węzła `v`.

Uwaga! Poniższy program **nie jest** odporny na multigrafy. Jego potencjalne błędne działanie wynika z warunku w funkcji `dfs_low` sprawdzającego, czy nie rozpatrujemy właśnie krawędzi do rodzica w drzewie DFS. Jeśli istnieje kilka krawędzi `parent` $\rightarrow$ `v`, to warunek ten odrzuci z rozważań wszystkie z nich, zamiast jedynie tej, którą weszliśmy do wierzchołka o numerze zapisanym w zmiennej `v`.

```
001 //graf
002 struct edge {
003     int v;
004     edge *next;
005 };
006
007 edge *graph[MAX];
008
009 //wartości entry i low z opisu algorytmu
010 int entry[MAX], low[MAX];
011 //licznik czasu wejścia
012 int entry_cnt = 0;
013 //czy wierzchołek jest punktem artykulacji
014 bool is_art[MAX];
015
016 int min(int a, int b)
017 {
018     if (a < b)
019         return a;
020     return b;
021 }
022
023 void dfs_low(int v, int parent)
024 {
025     int child_cnt = 0;
026     edge *e;
027
028     low[v] = entry[v] = ++entry_cnt;
029
030     for (e = graph[v]; e != NULL; e = e->next)
```

```
031         if (e->v != parent) {
032             //jeśli wierzchołek na końcu krawędzi był już odwiedzony,
033             //to mamy krawędź niedrzewową
034             if (entry[e->v] != 0) {
035                 low[v] = min(low[v], entry[e->v]);
036             } else { //jeśli nie, to puszczamy DFS głębiej
037                 ++child_cnt;
038                 dfs_low(e->v, v);
039                 low[v] = min(low[v], low[e->v]);
040 
041                 //czy v jest punktem artykulacji?
042                 if (low[e->v] >= entry[v])
043                     is_art[v] = true;
044             }
045         }
046 
047     //sprawdzenie dla korzenia drzewa DFS
048     if (parent == -1 && child_cnt > 1)
049         is_art[v] = true;
050 }
```

8.5 Sortowanie topologiczne

Mając acykliczny graf skierowany (DAG, zobacz też rozdział 7.3) możemy uporządkować jego wierzchołki w takiej kolejności (*porządek topologiczny*), że dla każdej krawędzi $u \to v$ wierzchołek u występuje przed wierzchołkiem v w tym porządku. Dla grafu $G = (V, E)$ można łatwo skonstruować działający w czasie $\mathcal{O}(|V| + |E|)$ algorytm oparty o przeszukiwanie grafu wszerz (rozdział 8.2) i zliczanie krawędzi wchodzących do każdego wierzchołka (*indeg*). Będziemy budowali metodą przyrostową ciąg wierzchołków posortowanych topologicznie:

- Trzymamy kolejkę wierzchołków v takich, że $indeg(v) = 0$.
- W pojedynczej fazie algorytmu wygarniamy jeden wierzchołek u z kolejki i dodajemy go na końcu budowanego posortowanego ciągu. Oprócz tego odcinamy wszystkie krawędzie wychodzące z u. Oczywiście nie odcinamy ich rzeczywiście, tylko udajemy, że to robimy: dla każdej krawędzi postaci $u \to v$ zmniejszamy $indeg(v)$ o jeden.
- Jeśli w ten sposób zmniejszymy $indeg(v)$ do zera, to dorzucamy v do kolejki. Spełniliśmy bowiem wymaganie, aby wszystkie wierzchołki o krawędziach wchodzących do v znalazły się w porządku topologicznym przed v.

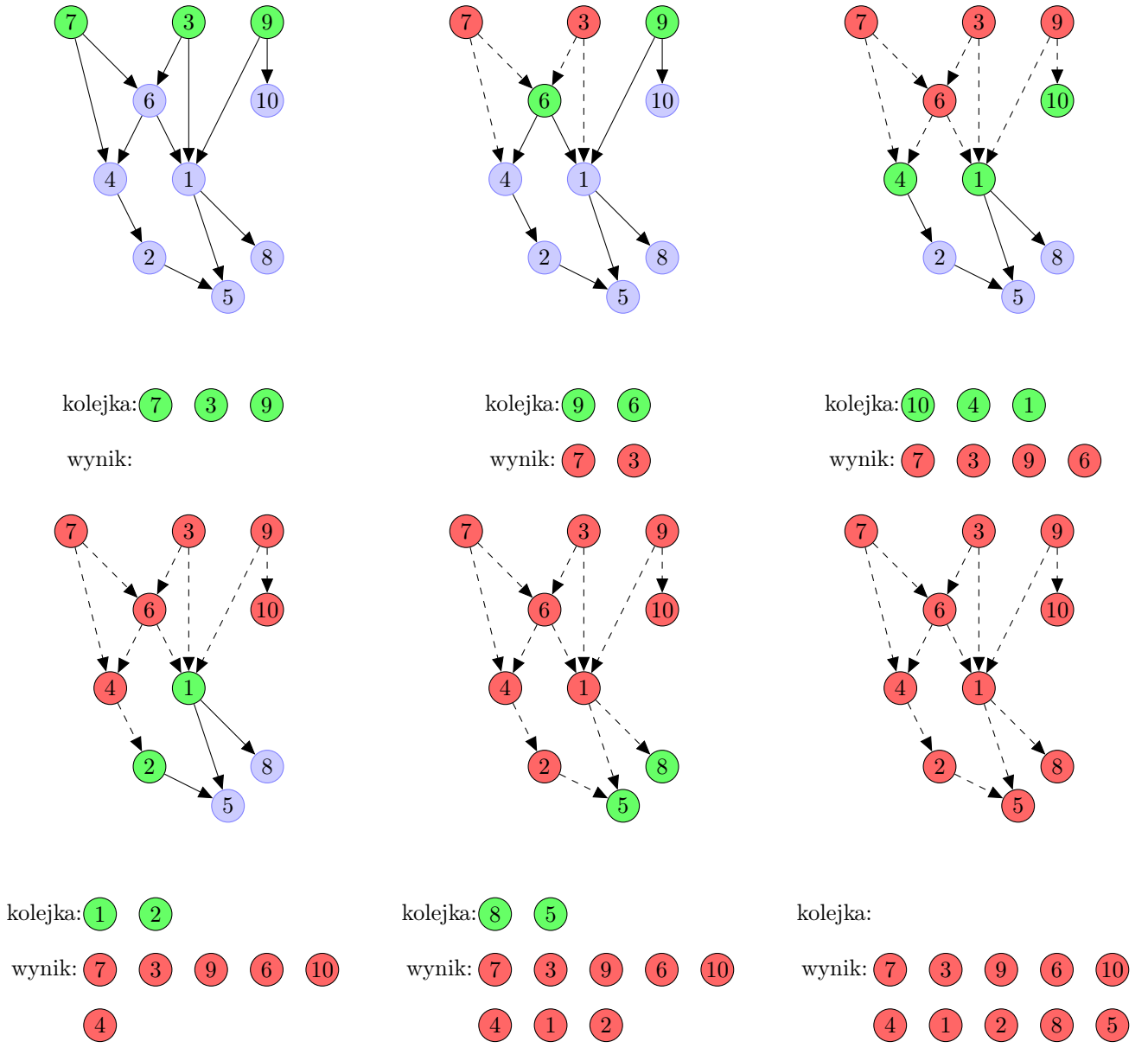
Miłą własnością powyższego algorytmu jest jego poprawne działanie na multigrafach. Zauważmy także, że podana metoda pozwala wykryć, czy graf wejściowy jest DAGiem. Jeśli jest, to każdy wierzchołek znajdzie się w obliczonym ciągu. Jeśli jednak w grafie znajduje się cykl, to nie wrzucimy do kolejki żadnego węzła należącego do tego cyklu, ponieważ nie osiągniemy tamże *indeg* równego zero. Zatem algorytm skończy działanie przed przetworzeniem wszystkich wierzchołków (rys. 8.5).

Porządek topologiczny przydaje się podczas wykonywania obliczeń w grafie, w którym wynik dla wierzchołka v zależy wprost od wyników dla wszystkich wierzchołków wchodzących do v. Weźmy dwa przykłady: policzenie wszystkich ścieżek w DAGu (oczywiście różnych) oraz policzenie najdłuższej ścieżki w DAGu. Łatwo zmodyfikować te algorytmy w taki sposób, żeby zliczać jedynie ścieżki wychodzące z określonego podzbioru wierzchołków (zobacz też rozdział 9.4).

Najpierw najdłuższe ścieżki. Oznaczmy przez $maxpath(v)$ długość najdłuższej ścieżki w grafie, która **kończy się** w węźle v. Intuicyjnie widać, że powinniśmy zacząć ich obliczanie od wierzchołków „najwyżej” położonych w grafie, tzn. takich, do których nie wchodzi żaden inny wierzchołek. Gdybyśmy bowiem znaleźli ścieżkę rozpoczynającą się od jakiegoś v, dla którego $indeg(v) > 0$, to moglibyśmy przedłużyć tę ścieżkę o wierzchołek wchodzący do v.

Przyjmijmy, że do pewnego wierzchołka v da się wejść bezpośrednio (tj. dokładnie jedną krawędzią) ze zbioru wierzchołków $\{u_1, u_2, \ldots, u_k\}$. Jeśli znamy wartość *maxpath* dla tych węzłów, to możemy obliczyć $maxpath(v)$. Wystarczy w tym celu wybrać takie u_i, że $maxpath(u_i)$ jest maksymalne i ustalić $maxpath(v) = maxpath(u_i) + 1$ – odpowiada to wyborowi najdłuższej ścieżki kończącej się w którymś z poprzedników v w grafie i przedłużeniu jej o jedną krawędź tak, aby kończyła się w v. Z tego sposobu można natychmiast wyprowadzić także samą ścieżkę, a nie tylko jej długość. Oczywiście należy jakoś zainicjować *maxpath* dla wierzchołków, które zaczynają z *indeg* równym zero. Ustalamy dla nich *maxpath* równe 0 albo 1, w zależności od tego czy przez długość ścieżki rozumiemy liczbę krawędzi czy liczbę wierzchołków.

Liczba ścieżek będzie liczona niemal identycznie. Oznaczmy przez $pathcount(v)$ liczbę ścieżek kończących się w v i zaczynających się gdziekolwiek. Tak jak wcześniej załóżmy, że mamy zbiór $\{u_1, u_2, \ldots, u_k\}$ poprzedników v w grafie



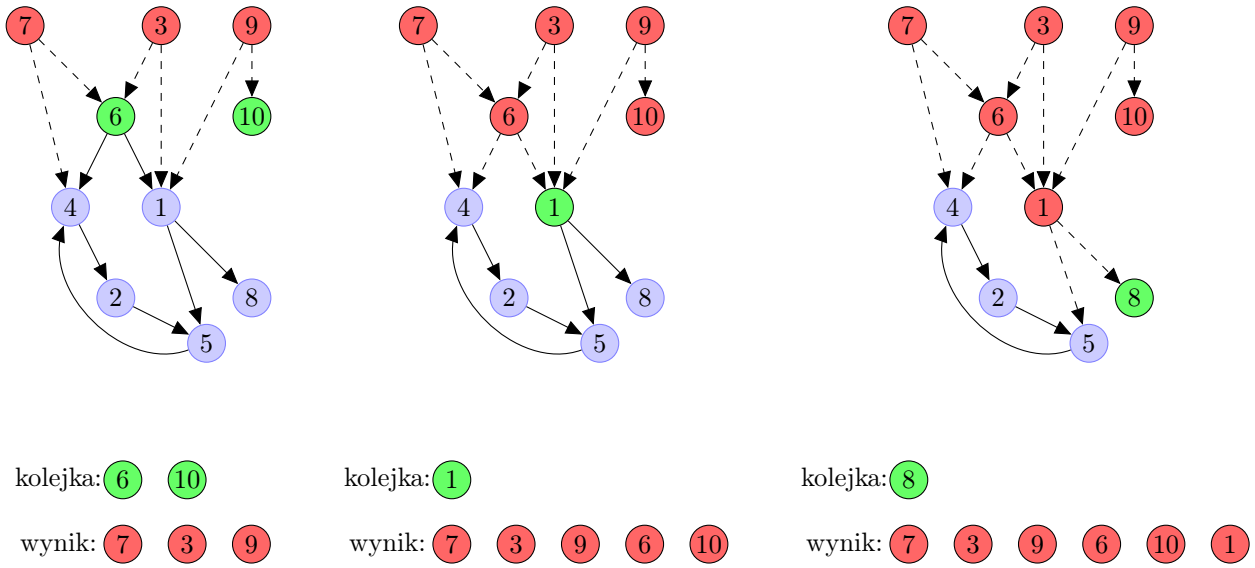
Rysunek 8.4: Kilka wybranych etapów sortowania topologicznego dla przykładowego grafu.

oraz znamy dla nich wartość *pathcount*. Wtedy mamy $pathcount(v) = pathcount(u_1)+\ldots+pathcount(u_k)$. Uzasadnienie tego wzoru jest następujące. Dla pewnego u_i jest dokładnie $pathcount(u_i)$ różnych ścieżek kończących się w nim. Możemy zatem każdą taką ścieżkę przedłużyć o krawędź $u_i \to v$, tym samym zwiększając liczbę różnych ścieżek kończących się w v dokładnie o wartość $pathcount(u_i)$.

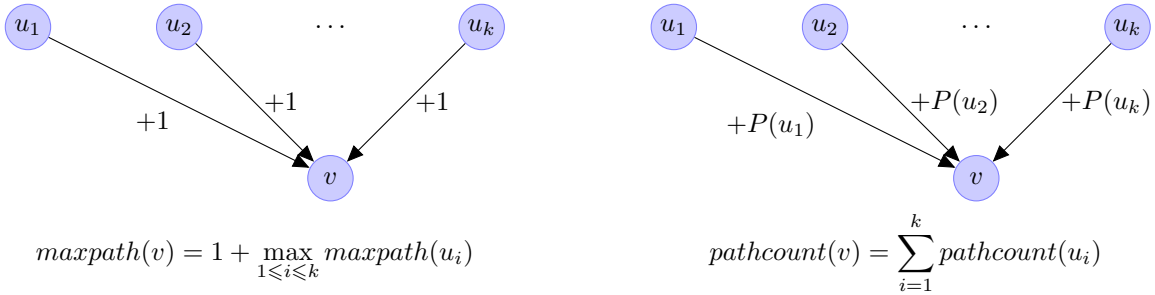
Ponownie należy ustalić wartość *pathcount* dla wierzchołków początkowych. Sensowną wartością jest jedynka – jeśli wzięlibyśmy zero, to wszędzie sumowałyby się jedynie zera, co byłoby bezsensem. Możemy przyjąć, że jedynka w wierzchołkach początkowych oznacza jedyną możliwą ścieżkę dojścia do nich, czyli ścieżkę pustą. Jeśli chcemy zliczyć wszystkie ścieżki w całym grafie, to musimy na końcu posumować wartości *pathcount* dla wszystkich wierzchołków grafu. Jeśli nie chcemy liczyć ścieżek pustych, to od końcowej sumy odejmujemy liczbę wierzchołków początkowych. Jeśli natomiast chcemy policzyć ścieżki puste, to musimy dodać do wyniku wartość równą liczbie wierzchołków niepoczątkowych. Schemat przedstawionego rozumowania można zobaczyć na rysunku 8.6.

Przykładowa implementacja

Mamy n-wierzchołkowy graf oparty o listy wskaźnikowe. Zakładam, że dla każdego wierzchołka policzona jest uprzednio liczba krawędzi wchodzących do niego (tablica `indeg`). Funkcja `toposort` zapisuje do tablicy `result` wierzchołki (numery z zakresu $[1, n]$) w kolejności poprawnej topologicznie. Wyliczanie poprawnego porządku wykonane jest zgodnie z opisanym powyżej zmodyfikowanym algorytmem BFS, dlatego potrzebna jest kolejka prosta (rozdział 5.1).



Rysunek 8.5: Graf z rysunku z 8.4 z dodaną jedną krawędzią powodującą powstanie cyklu i popsucie sortowania topologicznego. W następnym kroku kolejka zostanie opróżniona z wierzchołka 8 i algorytm zakończy działanie. Wierzchołki 2, 4 i 5 nigdy nie trafią do kolejki.



Rysunek 8.6: Wykorzystanie sortowania topologicznego do rozwiązywania niektórych problemów. Dla czytelności rysunku nazwa *pathcount* została skrócona do P.

```
001 //kolejka
002 int Q[MAX];
003 int head = 0, tail = 0;
004 int queue_front();
005 void queue_push(int);
006 bool queue_empty();
007
008 //graf
009 struct edge {
010     int v;
011     edge *next;
012 };
013
014 edge *graph[MAX]; //tablica list wskaźnikowych
015 int n;
016 int indeg[MAX]; //stopnie wejściowe wierzchołków
017 int result[MAX], result_cnt; //miejsce na wynik
018
019 void toposort()
020 {
021     int i, cur;
022     edge *e;
023
024     //inicjalizacja węzłami bez wierzchołków wchodzących
```

```
025     for (i = 1; i <= n; ++i)
026         if (indeg[i] == 0)
027             queue_push(i);
028
029     while (!queue_empty()) {
030         cur = queue_front();
031         result[result_cnt++] = cur; //dopychamy bieżący węzeł do wyniku
032         for (e = graph[cur]; e != NULL; e = e->next) {
033             --indeg[e->v];
034             if (indeg[e->v] == 0) //czy odcięliśmy wszystkie krawędzie?
035                 queue_push(e->v); //jeśli tak, to wrzucamy do kolejki
036         }
037     }
038 }
```

8.6 Ścieżki i cykle Eulera

Ścieżką Eulera nazwiemy taką ścieżkę $u \rightsquigarrow v$ w grafie, która przechodzi każdą krawędzią grafu dokładnie jeden raz. Jeśli dodatkowo wymagamy, aby $u = v$, to wtedy mamy do czynienia z *cyklem Eulera*. O grafach, w których istnieją cykle Eulera mówimy, że są *eulerowskie*. Grafy zawierające jedynie ścieżkę Eulera nazywamy *półeulerowskimi*.

Na początek zajmiemy się kwestią istnienia cykli Eulera w grafach nieskierowanych. Przyjmijmy, że mamy nieskierowany graf $G = (V, E)$, który jest jednocześnie spójny – brak spójności implikuje nieistnienie cyklu ani ścieżki Eulera, bowiem nie potrafimy „pokryć" wszystkich krawędzi jednym ciągłym przejściem. Cykl Eulera istnieje w G wtedy i tylko wtedy, gdy stopień każdego wierzchołka jest liczbą parzystą. Parzystość intuicyjnie implikuje, że zawsze wchodząc do pewnego wierzchołka v poprzez nieodwiedzoną wcześniej krawędź, istnieć będzie jeszcze przynajmniej jedna nieodwiedzona krawędź, którą będziemy mogli opuścić v.

Znajdowanie cyklu Eulera możemy zacząć w dowolnym wierzchołku s. Na początek chcemy znaleźć dowolny cykl, niekoniecznie Eulera, zaczynający się i kończący w s. Ze spójności grafu oraz założenia o parzystości wierzchołków wynika, że generując dowolną scieżkę wychodzącą z s w końcu wrócimy do wierzchołka początkowego.

Mamy teraz jakiś cykl $s \to v_1 \to \ldots \to v_k \to s$. Jeśli odwiedziliśmy wszystkie krawędzie, to kończymy, bo znaleźliśmy cykl Eulera. W przeciwnym przypadku wybieramy takie v_i, z którego wychodzi nieodwiedzona jeszcze krawędź. Jeśli potraktujemy odwiedzone krawędzie jako „usunięte", to nadal stopień każdego wierzchołka będzie liczbą parzystą. Z tego wynika, że uruchamiając z węzła v_i analogiczne do początkowego przejście w poszukiwaniu cyklu, to na pewno go znajdziemy. Powiedzmy, że ma on postać $v_i \to u_1 \to \ldots \to u_\ell \to v_i$. Teraz możemy „wpleść" drugi cykl w pierwszy, łącząc je w jeden duży cykl:

$$s \to v_1 \to \ldots \to v_{i-1} \to \underbrace{v_i \to u_1 \to \ldots \to u_\ell \to v_i}_{\text{mniejszy cykl}} \to v_{i+1} \to \ldots \to s$$

Krok ten powtarzamy dopóki nie wyczerpiemy wszystkich krawędzi grafu. Cały algorytm możemy zaimplementować tak, żeby działał w czasie $\mathcal{O}(|V| + |E|)$. Korzystamy w tym celu z listy dwukierunkowej (r. 5.5), do której wstawiamy początkowo dowolny cykl w grafie. Następnie przechodzimy tę listę od lewej do prawej szukając takiego v_i, z którego wychodzi nieodwiedzona jeszcze krawędź. Wtedy znajdujemy poboczny cykl zaczynający się i kończący w v_i, wplatamy go do listy i kontynuujemy algorytm z v_i (możliwe, że wychodzą z niego jeszcze jakieś nieodwiedzone krawędzie).

Ten sam algorytm można wykorzystać także do znajdowania ścieżek Eulera. Wystarczy zainicjować listę wierzchołków w odrobinę inny sposób: zamiast umieszczać tam dowolny cykl, zaczynamy od wstawienia dowolnej ścieżki $s' \rightsquigarrow s''$, gdzie s' i s'' to początkowy i końcowy wierzchołek ścieżki. Zauważmy, że w ten sposób sprowadziliśmy problem do znajdowaniu cyklu Eulera, bowiem teraz dla każdego wierzchołka liczba nieodwiedzonych krawędzi wychodzących z niego jest parzysta.

Przykładowa implementacja #1

Dany jest n-wierzchołkowy graf reprezentowany za pomocą listy wskaźnikowej. Funkcja `find_euler_cycle(s)` znajduje cykl Eulera począwszy od wierzchołka s. Używamy listy dwukierunkowej z rozdziału 5.5. Wplatanie nowego podcyklu do listy wierzchołków odbywa się niejako na bieżąco, tzn. znajdujemy dowolną nieodwiedzoną krawędź i idziemy na oślep aż do powrotu do wierzchołka początkowego, wstawiając odwiedzane po drodze wierzchołki do listy.

Aby rzeczywiście osiągnąć liniową złożoność algorytmu musimy zastosować pewien trik powodujący, że każdą krawędź „obejrzymy" dokładnie raz, zamiast np. każdorazowego przeglądania całej listy krawędzi wychodzących z określonego wierzchołka w poszukiwaniu jakiejkolwiek jeszcze nieodwiedzonej. Duplikujemy w tym celu tablicę list wskaźnikowych – tworzymy tablicę `next_edge`, którą inicjujemy początkami list wskaźnikowych grafu (`next_edge[v] = graph[v]` dla każdego v). Wtedy `next_edge[v]` będzie wskazywało na pierwszą niewykorzystaną krawędź na liście wskaźnikowej

wierzchołka v, a usunięcie tej krawędzi realizujemy poprzez przesunięcie wskaźnika tej listy na kolejny element. Zużycie wszystkich krawędzi na liście rozpoznajemy poprzez porównanie wskaźnika z wartością NULL.

Uwaga! Ten program nie będzie działał poprawnie dla grafów nieskierowanych, które są implementowane jako grafy podwójnie skierowane. Wynika to z zapamiętania krawędzi dwukierunkowej jako dwóch niezależnych i niepowiązanych elementów list wskaźnikowych. Rozwiązanie tego problemu znajduje się w kolejnym przykładowym programie.

```
001 //lista dwukierunkowa
002 struct list_el {
003     int el;
004     list_el *prev, *next;
005 };
006
007 void list_init();
008 void list_insert(list_el *after, int x);
009 void list_push_front(int x);
010
011 list_el sentinel;
012
013 //graf
014 struct edge {
015     int v;
016     edge *next;
017 };
018
019 edge *graph[MAX]; //tablica list wskaźnikowych
020 edge *next_edge[MAX]; //kolejna niewykorzystana krawędź
021 int n;
022
023 void find_euler_cycle(int s)
024 {
025     list_init();
026     list_push_front(s);
027
028     list_el *current = sentinel.next;
029     for (int i = 1; i <= n; ++i)
030         next_edge[i] = graph[i];
031
032     //działamay dopóki nie wrócimy do początku listy
033     while (current != &sentinel) {
034         //czy została jakaś nieodwiedzona krawędź wychodząca
035         //z bieżącego wierzchołka?
036         while (next_edge[current->el] != NULL) {
037             //będziemy wstawiać kolejno odwiedzane wierzchołki
038             //od lewej do prawej, zaraz za bieżącym wierzchołkiem
039             //w liście
040             list_el *inserter = current->next;
041             int v = current->el;
042
043             do {
044                 int u = next_edge[v]->v;
045                 next_edge[v] = next_edge[v]->next; //"zużywamy" krawędź
046                 v = u;
047                 list_insert(current->prev, v);
048             } while (v != current->el); //przerywamy po zbudowaniu cyklu
049         }
050
051         current = current->next;
052     }
053 }
```

Przykładowa implementacja #2

Standardowy algorytm konstruuje cykl Eulera sklejając kolejne krawędziowo rozłączne podcykle. Można zresztą wykazać, że graf jest eulerowski wtedy i tylko wtedy, gdy jest sumą rozłącznych krawędziowo cykli. Zmienimy trochę podejście i zbudujemy cykl Eulera wierzchołek po wierzchołku.

W pierwotnym programie przeglądaliśmy od lewej do prawej listę wierzchołków w poszukiwaniu elementu, z którego wychodzi niewykorzystana jeszcze krawędź. Zauważmy, że taka kolejność jest podyktowana wyłącznie wygodą implementacyjną – gdybyśmy wybierali wierzchołki w dowolnej kolejności, algorytm nadal byłby poprawny. W szczególności oznacza to, że możemy przechodzić od prawej do lewej, a takie zachowanie łatwo da się symulować za pomocą przeszukiwania grafu w głąb. Cały algorytm sprowadzi się do odpalenia przeszukiwania w dowolnym miejscu i zapisywania jako kolejnych elementów cyklu tych wierzchołków grafu, z których przeszukiwanie wycofuje się.

Miłym efektem ubocznym takiego podejścia jest automatyczne radzenie sobie ze znajdowaniem ścieżek Eulera – wystarczy odpalić funkcję `find_euler_cycle` w wierzchołku mającym być **początkiem** ścieżki Eulera i też będzie dobrze. Przy okazji zadbamy o jednokrotne rozpatrywanie krawędzi występujących podwójnie w grafie. Z każdą krawędzią skojarzymy unikalny identyfikator (liczbę całkowitą od 0 do $|V|-1$) i zużyte krawędzie będziemy odznaczać w globalnej tablicy `edge_used`. Oczywiście musimy zastosować wcześniejszy trik dotyczący jednokrotnego oglądania każdej krawędzi w grafie żeby nie stracić złożoności obliczeniowej.

Prostsze rozwiązanie pozwala też na przejście do prostszej struktury danych – zamiast listy dwukierunkowej wystarczy zwykły stos do zapamiętywania wierzchołków. Na samym końcu działania algorytmu wierzchołek, z którego zaczęliśmy znajdować się będzie na szczycie stosu (ma to znaczenie w przypadku poszukiwania ścieżki Eulera).

```
001 //stos
002 int S[MAX], height = 0;
003 int stack_pop();
004 void stack_push(int x);
005
006 //graf
007 struct edge {
008     int v, id;
009     edge *next;
010 };
011
012 edge *graph[MAX];
013 edge *next_edge[MAX];
014 bool edge_used[MAX_EDGES];
015 int n;
016
017 void dfs(int v)
018 {
019     while (next_edge[v] != NULL) {
020         int edge_id = next_edge[v]->id, u = next_edge[v]->v;;
021         next_edge[v] = next_edge[v]->next;
022         if (!edge_used[edge_id]) {
023             edge_used[edge_id] = true;
024             dfs(u);
025         }
026     }
027     stack_push(v);
028 }
029
030 void find_euler_cycle(int s)
031 {
032     for (int i = 1; i <= n; ++i)
033         next_edge[i] = graph[i];
034     dfs(s);
035 }
```

9 Najkrótsze ścieżki

Problem znalezienia najkrótszej ścieżki między dwoma wierzchołkami w zadanym grafie o krawędziach jednostkowych można rozwiązać za pomocą przeszukiwania wszerz. Jeśli graf jest ważony, to należy użyć bardziej wyrafinowanych metod, jak np. algorytmu Forda–Bellmana (dla grafów o dowolnych wagach) lub algorytmu Dijkstry (dla grafów o wagach nieujemnych). Wyszukanie kosztu najkrótszej ścieżki dla dowolnej pary wierzchołków realizuje się przez wielokrotne (dla każdego wierzchołka) wywoływanie algorytmu Dijkstry lub Forda–Bellmana albo użycie algorytmu Floyda–Warshalla.

Każdy z tych algorytmów działa zachłannie poprawiając *macierz odległości* (albo *kosztu*), którą w implementacji będę nazywał `cost`. Odległość do wierzchołka początkowego ustala się na 0, do wszystkich pozostałych ∞, czyli w praktyce jakąś dużą liczbę, która na pewno zostanie poprawiona przez algorytm, np. 10^9. W algorytmach Forda–Bellmana i Dijkstry macierz odległości jest jednowymiarową tablicą (*wektorem*) o długości $|V|$, która po wykonaniu obliczeń zawiera koszt najtańszej ścieżki od określonego wierzchołka początkowego do wszystkich innych. Jeśli `cost[u]` $= \infty$, to wtedy nie istnieje ścieżka od wierzchołka początkowego do wierzchołka o numerze u. Wykonanie algorytmu Floyda–Warshalla oblicza macierz odległości w postaci kwadratowej tablicy o boku $|V|$, gdzie `cost[u][v]` zawiera koszt najkrótszej ścieżki $u \rightsquigarrow v$ (i podobnie jak wcześniej `cost[u][v]` $= \infty$ oznacza brak ścieżki $u \rightsquigarrow v$).

9.1 Algorytm Forda–Bellmana

Zasada działania tego algorytmu jest prosta jak budowa czołgu T-55[1]. Wykonuje się $|V| - 1$ iteracji po wszystkich krawędziach grafu poprawiając zachłannie macierz odległości, tzn. dla każdej krawędzi $u \to v$ dokonuje się tzw. *relaksacji*, czyli mówiąc mniej szumnie: jeśli koszt dojścia do u zsumowany z kosztem przejścia tą krawędzią jest mniejszy niż aktualnie wyliczony koszt dojscia do v, to poprawiamy. W skrócie: `cost[v]` $= \min($`cost[v], cost[u]` $+ |u \to v|)$.

W pierwszej iteracji budujemy ścieżki o długości 1, w drugiej ścieżki o długości 2 itd. Załóżmy, że graf nie ma cyklu o ujemnej sumie wag na krawędziach. W takim razie dla n–wierzchołkowego grafu najdłuższe ścieżki będą miały $n - 1$ krawędzi, stąd wynika dlaczego taka liczba iteracji. W grafie bez cyklu ujemnego nie opłaca się żadnej krawędzi włączać do ścieżki więcej niż raz. Wniosek: jeśli po wykonaniu jeszcze jednej (n–tej) iteracji okazuje się, że wciąż można poprawić macierz odległości, to wtedy w grafie istnieje cykl o ujemnej sumie kosztów. Dla takiego grafu nie można obliczyć macierzy odległości, ponieważ taki cykl pozwala na poprawianie jej w nieskończoność.

Jest już prawie dobrze — mamy poprawny algorytm, który potrafi obronić się przed zapętleniem w ujemnym cyklu. Dlaczego więc tylko **prawie** dobrze? Ponieważ nie rozpatrywaliśmy jeszcze czasu działania, a ten jest niemały. Każda iteracja przebiega po wszystkich krawędziach, czyli trwa czas proporcjonalny do $|E|$. Liczba iteracji wynosi $|V|$ (licząc razem z ostatnią, sprawdzającą istnienie ujemnego cyklu). Dla odpowiednio dużych grafów ($|V| \approx 5 \cdot 10^3, |E| \approx 2 \cdot 10^4$) procesor dostaje obstrukcji na myśl o liczeniu macierzy odległości. Tym niemniej algorytm Forda–Bellmana bywa jedynym sensownym rozwiązaniem przy niektórych zadaniach.

Przykładowa implementacja #1

Dany jest graf ważony reprezentowany za pomocą list wskaźnikowych. Funkcja `ford_bellman(v)` oblicza macierz odległości `cost` dla wierzchołka początkowego v i zwraca `false` w przypadku wykrycia ujemnego cyklu. Stała `INF` reprezentuje umowną nieskończoność. Wierzchołki grafu indeksowane są w zakresie $[1, n]$.

```
001 const int INF = 1000000000; //10^9
002
003 struct edge {
004     int v, cost;
005     edge *next;
006 };
007
008 edge *graph[MAX]; //tablica list wskaźnikowych
009 int n;
```

[1] Jak ktoś nie rozumie, to ma udawać, że rozumie.

```
010 int cost[MAX];
011
012 int min(int a, int b)
013 {
014     if (a < b)
015         return a;
016     return b;
017 }
018
019 //relaksacja
020 void relax(int v, int new_cost)
021 {
022     cost[v] = min(cost[v], new_cost);
023 }
024
025 bool ford_bellman(int v)
026 {
027     edge *e;
028     int i, j;
029
030     //inicjalizacja macierzy odległości
031     for (i = 1; i <= n; ++i)
032         cost[i] = INF;
033     cost[v] = 0;
034
035     //wykonanie n-1 iteracji po wszystkich krawędziach
036     for (i = 1; i < n; ++i)
037         for (j = 1; j <= n; ++j)
038             for (e = graph[j]; e != NULL; e = e->next)
039                 relax(e->v, cost[j] + e->cost);
040
041     //sprawdzenie istnienia cyklu ujemnego
042     for (j = 1; j <= n; ++j)
043         for (e = graph[j]; e != NULL; e = e->next)
044             if (cost[e->v] > cost[j] + e->cost)
045                 return false;
046
047     return true;
048 }
```

Przykładowa implementacja #2

Powyższy algorytm można nieco przyspieszyć dokonując pewnej trywialnej obserwacji. Każdy obieg zewnętrznej pętli poprawia koszt osiągnięcia pewnego węzła, który albo już wcześniej mogliśmy osiągnąć, albo docieramy do niego po raz pierwszy (czyli `cost[u]` $== \infty$). W każdym obiegu możemy więc wydłużać znalezione ścieżki o jedną krawędź (bo nie dokonamy poprawienia dla $u \to v$ jeśli `cost[u]` $==$ `cost[v]` $== \infty$). Unikając niepotrzebnych sprawdzeń można znacznie przyspieszyć praktyczne działanie tego algorytmu.

Podkreślając powyższą obserwację algorytm Forda–Bellmana zachowuje się podobnie do przeszukiwania wszerz — poprawienie kosztu osiągnięcia wierzchołka powoduje wrzucenie go do kolejki, z elementów której wypuszczamy nowe krawędzie próbując poprawić osiągalne wierzchołki.

Przed rzuceniem się do kodowania trzeba rozważyć istnienie cyklu ujemnego. Wystąpienie takiego zjawiska pozwala na poprawianie kosztów krawędzi w nieskończoność, co znaczy, że kolejka przeszukiwania wszerz nigdy nie zostanie opróżniona i algorytm nie zakończy działania. Z własności algorytmu Forda–Bellmana wynika, że zbudowanie ścieżki składającej się z większej liczby krawędzi niż $|V - 1|$ oznacza znalezienie cyklu ujemnego. Wystarczy więc dla każdego wierzchołka pamiętać ile krawędzi liczy sobie znaleziona dotychczasowa najkrótsza ścieżka i przerywać działanie algorytmu w odpowiednim momencie.

Podany kod korzysta z kolejki FIFO z górnym ograniczeniem na liczbę elementów (rozdział 5.1) oraz dodatkowej tablicy `edge_cnt` dla pamiętania liczby „poprawień” każdego wierzchołka. Dla złośliwie skonstruowanego grafu górne ograniczenie może okazać się zbyt bujne na zmieszczenie całej kolejki w tablicy statycznej i może zmusić do skorzystania z kolejki cyklicznej lub kolejki wskaźnikowej (rozdział 5.3). Dozwolona jest bowiem sytuacja, w której jeden węzeł grafu

trafia do kolejki kilka razy. Można sobie z tym radzić pamiętając w oddzielnej tablicy, czy dany wierzchołek, którego koszt udało się poprawić, znajduje się już w kolejce czy jeszcze nie i dodawać tylko w sytuacjach, gdy w kolejce go nie ma. Wtedy zawsze wystarczy kolejka cykliczna z górnym ograniczeniem elementów równym liczbie wierzchołków grafu.

```
001 const int INF = 1000000000; //10^9
002
003 struct edge {
004     int v, cost;
005     edge *next;
006 };
007
008 edge *graph[MAX]; //tablica list wskaźnikowych
009 int n;
010 int cost[MAX];
011 int edge_cnt[MAX];
012 bool in_queue[MAX]; //czy dany wierzchołek jest w kolejce?
013
014 //kolejka
015 int Q[MAX];
016 int head = 0, tail = 0;
017 int queue_front();
018 void queue_push(int);
019 bool queue_empty();
020
021 bool ford_bellman(int v)
022 {
023     edge *e;
024     int i, j;
025     int cur;
026
027     //inicjalizacja macierzy odległości i liczników długości ścieżki
028     for (i = 1; i <= n; ++i) {
029         cost[i] = INF;
030         edge_cnt[i] = 0;
031     }
032     cost[v] = 0;
033
034     //inicjalizacja kolejki wierzchołkiem początkowym
035     queue_push(v);
036     in_queue[v] = true;
037
038     while (!queue_empty()) {
039         cur = queue_front();
040         in_queue[cur] = false;
041         for (e = graph[cur]; e != NULL; e = e->next)
042             if (cost[e->v] > cost[cur] + e->cost) {
043                 edge_cnt[e->v] = edge_cnt[cur] + 1;
044
045                 //czy mamy cykl ujemny?
046                 if (edge_cnt[e->v] == n)
047                     return false;
048
049                 //jeśli nie, poprawiamy koszt
050                 //i dorzucamy wierzchołek do kolejki
051                 //o ile nie ma go w niej jeszcze
052                 cost[e->v] = cost[cur] + e->cost;
053                 if (!in_queue[e->v]) {
054                     queue_push(e->v);
055                     in_queue[e->v] = true;
056                 }
057             }
```

```
058     }
059
060     return true;
061 }
```

9.2 Algorytm Dijkstry

Poprawne działanie algorytmu Dijkstry zapewnione jest tylko dla grafów o nieujemnych wagach na krawędziach. Faza inicjalizacji ustala wszystkie wierzchołki (poza początkowym) jako nieprzetworzone (nieodwiedzone), a ich koszt na ∞. Krok algorytmu polega na wybraniu spośród nieprzetworzonych wierzchołków jednego o minimalnej wartości, następnie poprzez relaksację dokonuje się poprawy macierzy odległości dla wierzchołków sąsiadujących. Wybrany wierzchołek staje się przetworzony, algorytm wykonuje kolejną iterację, czyli wyszukuje minimalnego nieprzetworzonego itd. Kiedy nie można znaleźć wierzchołka jednocześnie nieprzetworzonego i osiągalnego (czyli jego `cost` $< \infty$), to wtedy wszystkie osiągalne wierzchołki zostały przetworzone i kończymy działanie.

Jeśli trzeba określić najkrótszą drogę od wierzchołka początkowego do jednego konkretnie określonego (będę go nazywał *końcowym*), a nie do wszystkich, to algorytm można zakończyć w momencie oznaczenia wierzchołka końcowego jako przetworzonego, ponieważ jego koszt w macierzy odległości na pewno nie zostanie poprawiony.

W pesymistycznym przypadku algorytm Dijkstry wykonuje $|V|-1$ kroków, w każdym z nich bada krawędzie wychodzące z aktualnie przetwarzanego wierzchołka, a więc każda krawędź badana jest tylko raz, co daje czas proporcjonalny do $|E|$. Trzeba pamiętać o tym, że po każdym kroku należy znaleźć nowy wierzchołek do przetworzenia, czyli wyszukać minimalny nieprzetworzony wierzchołek w macierzy odległości `cost`. Liniowe poszukiwanie minimum daje czas $|V|$ dla każdej iteracji, czyli łącznie $|V|^2$. Ostateczny czas działania wynosi więc $\mathcal{O}(|E|+|V|^2)$.

Uważny czytelnik zapewne zauważy, że przecież czas wyszukiwania minimalnego wierzchołka można znacząco przyspieszyć korzystając z kolejek priorytetowych opartych o kopce binarne (rozdziały 5.6 i 5.7), ale o tym później.

Przykładowa implementacja #1

Dany jest graf ważony reprezentowany za pomocą list wskaźnikowych. Funkcja `dijkstra(v)` oblicza macierz odległości `cost` dla wierzchołka początkowego v. Tablica `visited` zawiera informację o nieprzetworzonych wierzchołkach, a funkcja pomocnicza `next_vertex` zwraca numer kolejnego wierzchołka do przetworzenia. Stała `INF` reprezentuje umowną nieskończoność. Wierzchołki grafu indeksowane są w zakresie $[1, n]$.

```
001 const int INF = 1000000000; //10^9
002
003 struct edge {
004     int v, cost;
005     edge *next;
006 };
007
008 edge *graph[MAX]; //tablica list wskaźnikowych
009 int n;
010 int cost[MAX];
011 bool visited[MAX];
012
013 int min(int a, int b)
014 {
015     if (a < b)
016         return a;
017     return b;
018 }
019
020 void relax(int v, int new_cost)
021 {
022     cost[v] = min(cost[v], new_cost);
023 }
024
025 //zwraca minimalny nieprzetworzony wierzchołek
026 //lub -1 jeśli taki nie istnieje
027 int next_vertex()
028 {
```

```
029     int vertex = 1, i;
030
031     for (i = 2; i <= n; ++i)
032         if (!visited[i] && cost[i] < cost[vertex])
033             vertex = i;
034
035     if (visited[vertex])
036         vertex = -1;
037     return vertex;
038 }
039
040 void dijkstra(int v)
041 {
042     edge *e;
043     int i;
044     int cur;
045
046     //inicjalizacja macierzy odległości
047     //oraz tablicy visited
048     for (i = 1; i <= n; ++i) {
049         cost[i] = INF;
050         visited[i] = false;
051     }
052     cost[v] = 0;
053
054     cur = v;
055
056     //iteruj dopóki istnieją nieprzetworzone i osiągalne
057     //wierzchołki; jeśli mamy określony wierzchołek
058     //końcowy x, to można napisać while (cur != x)
059     while (cur != -1) {
060         visited[cur] = true;
061         for (e = graph[cur]; e != NULL; e = e->next)
062             relax(e->v, cost[cur] + e->cost);
063         cur = next_vertex();
064     }
065 }
```

Przykładowa implementacja #2

Zbiór wierzchołków „kandydatów" do przetworzenia przechowujemy na modyfikowalnym kopcu binarnym z rozdziału 5.7. Funkcja `next_vertex` zostaje wobec tego zastąpiona przez `heap_top`, a relaksacja poza poprawą wartości w macierzy odległości `cost` dokonuje także modyfikacji na kopcu.

Ponieważ modyfikacja może pociągnąć konieczność przywrócenia własności kopca, czas poprawy wyniku pośredniego przy relaksacji wydłuża się z $\mathcal{O}(1)$ do $\mathcal{O}(\log |V|)$. Modyfikację wykonujemy najwyżej raz dla każdej krawędzi, co łącznie daje $\mathcal{O}(|E| \cdot \log |V|)$. Znaczącą poprawę zyskujemy na logarytmicznym wyszukiwaniu minimalnego wierzchołka do przetworzenia: mamy $|V|$ operacji, każda w czasie $\mathcal{O}(\log |V|)$, co łącznie daje $\mathcal{O}(|V| \cdot \log |V|)$ zamiast wcześniejszego $|V|^2$. Ostateczna złożoność w tej implementacji wynosi $\mathcal{O}((|E| + |V|) \cdot \log |V|)$.

Z praktycznego punktu widzenia nie ma sensu wrzucać do kopca od razu wszystkich wierzchołków, lecz tylko te, które są osiągalne, tzn. `cost` $< \infty$. Chociaż nie zyskujemy na złożoności, otrzymujemy w istocie kolejną poprawę czasu działania, ponieważ utrzymujemy na kopcu jedynie ograniczony zbiór rzeczywistych kandydatów — im mniej, tym szybsze działanie kopca. W dodatku takie podejście eliminuje potrzebę istnienia tablicy `visited`.

```
001 const int INF = 1000000000; //10^9
002
003 //struktury i funkcje kopca
004 int heap[MAX], loc[MAX], ver[MAX];
005 int heap_size = 0;
006 int heap_top();
007 void heap_modify(int v, int x);
```

```
008 void heap_remove(int i);
009 bool heap_empty();
010
011 struct edge {
012     int v, cost;
013     edge *next;
014 };
015
016 edge *graph[MAX]; //tablica list wskaźnikowych
017 int n;
018 int cost[MAX];
019
020 void dijkstra(int v)
021 {
022     edge *e;
023     int i;
024     int cur;
025
026     //inicjalizacja macierzy odległości
027     for (i = 1; i <= n; ++i)
028         cost[i] = INF;
029     cost[v] = 0;
030
031     //inicjalizacja kopca
032     heap_modify(v, cost[v]);
033
034     //iteruj dopóki istnieją nieprzetworzone i osiągalne
035     //wierzchołki; jeśli mamy określony wierzchołek
036     //końcowy x, to można dopisać while (cur != x)
037     while (!heap_empty()) {
038         cur = heap_top();
039         heap_remove(1);
040
041         for (e = graph[cur]; e != NULL; e = e->next)
042             if (cost[e->v] > cost[cur] + e->cost) {
043                 cost[e->v] = cost[cur] + e->cost;
044                 heap_modify(e->v, cost[e->v]);
045             }
046     }
047 }
```

Ford–Bellman $\rightsquigarrow$ Dijkstra

Co się stanie, jeśli do grafu o wagach ujemnych zastosujemy algorytm Dijkstry oparty o kopiec binarny? Jeśli graf zawiera cykl ujemny, to uzyskamy pętlę nieskończoną i program nie zakończy swojego działania. Można się przed tym zabezpieczyć stosując zliczanie liczby krawędzi na ścieżce podobnie, jak w implementacji algorytmu Forda–Bellmana na kolejce FIFO. Mając kolejkę priorytetową (kopiec) zamiast kolejki prostej czas wstawienia elementu zwiększa się z $\mathcal{O}(1)$ do $\mathcal{O}(\log n)$. Co wobec tego można zyskać?

Po pierwsze, mamy zagwarantowane zużycie pamięci na poziomie $\mathcal{O}(n)$, ponieważ w przeciwieństwie do kolejki prostej, każda poprawa kosztu dojścia do wierzchołka jest zmieniana w miejscu. Po drugie, wybieranie za każdym razem najtańszego wierzchołka do przetworzenia w znaczącej większości praktycznych przypadków spowoduje bardzo szybkie wyczerpanie najkrótszych ścieżek lub znalezienie cyklu ujemnego. Innymi słowy, algorytm Forda–Bellmana potencjalnie zakończy swoje działanie po zdecydowanie mniejszej liczbie faz, niż zaimplementowany na bazie kolejki prostej.

9.3 Algorytm Floyda–Warshalla

Ponieważ obliczamy najkrótsze ścieżki pomiędzy wszystkimi parami wierzchołków, macierz odległości zmieniamy z liniowej na kwadratową. Początkowo ustalamy wszystkie jej pola na ∞, po czym poprawiamy nieskończoności w indeksach, które odpowiadają istniejącym w grafie krawędziom, tzn. jeśli istnieje krawędź $u \to v$ o wadze w, to zapisujemy

`cost[u][v] = w`. W większości przypadków można bezpiecznie przyjąć, że dla każdego wierzchołka v przypisujemy `cost[v][v] = 0`. Taka konstrukcja jest niemal tym samym, co macierz sąsiedztwa z rozdziału 7.1.

Dalsze poprawianie macierzy odległości odbywa się poprzez kolejne relaksacje: dla każdej ścieżki $u \rightsquigarrow v$ sprawdzamy czy nie tańsze okazuje się pokonanie drogi $u \rightsquigarrow w$, a potem $w \rightsquigarrow v$ dla pewnego wierzchołka w. Jeśli tak, to poprawiamy minimalny koszt `cost[u][v] = cost[u][w] + cost[w][v]`.

Sens algorytmu jest następujący. Wykonujemy $|V|$ iteracji, polegających na znalezieniu najkrótszych ścieżek pomiędzy każdą parą wierzchołków. W k-tej iteracji obliczamy najkrótsze ścieżki, korzystając z wierzchołków pośrednich o numerach od 1 do k. Tzn. jeśli obliczamy najkrótszą ścieżkę z u do v, to pośrednio możemy przechodzić tylko przez wierzchołki o numerach nie większych od k. W kolejnej iteracji dodajemy kolejny wierzchołek do dozwolonego zbioru, co łatwo pozwala zaktualizować najkrótsze ścieżki: skoro dodajemy nowy węzeł k, to najkrótsza ścieżka od u do v albo pozostaje bez zmian, albo pojawiła się nowa, lepsza (krótsza) ścieżka, która korzysta z wierzchołka k. Sprawdzamy zatem, czy ścieżka $u \rightsquigarrow k \rightsquigarrow v$ jest tańsza od wcześniej obliczonej $u \rightsquigarrow v$ nieprzechodzącej przez k (bo ten wierzchołek nie był jeszcze „dozwolony").

Dla każdego wierzchołka pośredniego w próbujemy poprawić koszt każdej ścieżki pomiędzy dowolną parą wierzchołków, więc złożoność obliczeniowa wynosi $\mathcal{O}(|V|^3)$. Pamiętanie kwadratowej macierzy odległości wymusza złożoność pamięciową $\mathcal{O}(|V|^2)$. Z tego powodu algorytm można stosować tylko dla dość małych grafów (kilkaset wierzchołków, patrz też 7.1). Potencjalnie odstraszająca złożoność obliczeniowa jest nieco rekompensowana przez bardzo mały koszt przeprowadzenia pojedynczej operacji.

Przykładowa implementacja

Funkcja `floyd_warshall` oblicza kwadratową macierz odległości `cost` dla dowolnego grafu. Algorytm uznaje, że macierz jest zainicjalizowana adekwatnymi wagami i nie przeprowadza inicjalizacji komórek na ∞. Wierzchołki grafu indeksowane są w zakresie $[1, n]$.

```
001 int cost[MAX][MAX];
002 int n;
003
004 int min(int a, int b)
005 {
006     if (a < b)
007         return a;
008     return b;
009 }
010
011 void relax(int u, int v, int w)
012 {
013     cost[u][v] = min(cost[u][v], cost[u][w] + cost[w][v]);
014 }
015
016 void floyd_warshall()
017 {
018     int i, j, k;
019
020     for (k = 1; k <= n; ++k)
021         for (i = 1; i <= n; ++i)
022             for (j = 1; j <= n; ++j)
023                 relax(i, j, k);
024 }
```

9.4 Najkrótsze ścieżki w DAGach

Znajdowanie najkrótszych ścieżek w acyklicznych grafach skierowanych ma dużo wspólnego z algorytmem sortowania topologicznego. Korzystając z przedstawionego w rozdziale 8.5 schematu obliczania rozwiązań różnych zagadnień w takich grafach możemy analogicznie obliczyć najkrótsze ścieżki. W zasadzie rozwiązywaliśmy tam już niemal identyczny problem – zamiast najkrótszych ścieżek z określonego miejsca szukaliśmy najdłuższej ścieżki w całym grafie, przy czym tam był to graf jednostkowy, a tutaj poradzimy sobie z ważonym (żadna różnica). Tym niemniej można bez problemu zastosować takie samo rozwiązanie.

Oznaczmy $cost_s(v)$ długość najkrótszej ścieżki $s \rightsquigarrow v$. Jeśli nie istnieje żadna ścieżka z s do v, to przyjmujemy $cost_s(v) = \infty$. Przyjmijmy, że dla pewnego węzła v mamy w grafie zbiór poprzedników $\{u_1, u_2, \ldots, u_k\}$ i znamy dla nich $cost_v$. Wtedy możemy wziąć takie u_i, że $cost_s(u_i) + |u_i \to v|$ jest minimalne.

Korzystając z sortowania topologicznego możemy znaleźć w czasie liniowym odpowiedni porządek przeglądania wierzchołków. Potem, gdy obliczamy najkrótsze ścieżki, każdą krawędź przeglądamy dokładnie raz (relaksacja), co nie zwiększa złożoności czasowej algorytmu.

Przykładowa implementacja

Mamy n-wierzchołkowy graf oparty o listy wskaźnikowe oraz policzone uprzednio stopnie wchodzące dla każdego węzła (tablica `indeg`). Funkcja `shortest_paths_dag` oblicza najkrótsze ścieżki z wierzchołka s przekazanego w argumencie do wszystkich pozostałych i zapisuje te wartości do tablicy `cost`. Jeśli nie istnieje ścieżka $s \rightsquigarrow v$, to `cost[v]` $= \infty$. Wierzchołki grafu przeglądane są w porządku topologicznym. Warto zauważyć, że zainicjowanie macierzy odległości dla wszystkich węzłów poza s wartością ∞ automatycznie radzi sobie z wierzchołkami „powyżej" s w grafie. Dopiero gdy porządek topologiczny osiągnie s (mamy `cost[`s`]` $=$ `0`), relaksacja zacznie poprawiać wartości w macierzy odległości.

```
001 //kolejka
002 int Q[MAX];
003 int head = 0, tail = 0;
004 int queue_front();
005 void queue_push(int);
006 bool queue_empty();
007
008 const int INF = 1000000000; //10^9
009
010 struct edge {
011         int v, cost;
012         edge *next;
013 };
014
015 edge *graph[MAX]; //tablica list wskaźnikowych
016 int n;
017 int indeg[MAX], cost[MAX];
018
019 int min(int a, int b)
020 {
021     if (a < b)
022         return a;
023     return b;
024 }
025
026 void shortest_paths_dag(int s)
027 {
028     int i, cur;
029     for (i = 1; i <= n; ++i) {
030         cost[i] = INF;
031         if (indeg[i] == 0)
032             queue_push(i);
033     }
034     cost[s] = 0;
035
036     while (!queue_empty()) {
037         cur = queue_front();
038         for (e = graph[cur]; e != NULL; e = e->next) {
039             cost[e->v] = min(cost[e->v], cost[cur] + e->cost); //relaksacja
040             --indeg[e->v];
041             if (indeg[e->v] == 0)
042                 queue_push(e->v);
043         }
044     }
```

```
045 }
```

9.5 Konstrukcja najkrótszych ścieżek

Czasami zachodzi potrzeba poznania dokładnego układu najkrótszej ścieżki celem np. usunięcia jej z grafu. W tym celu wystarczy zmodyfikować nieco krok relaksacji: jeśli macierz odległości zostaje poprawiona, wtedy podmieniamy wpis w pomocniczej tablicy `from`, służącej do zapamiętywania z którego wierzchołka poprawiliśmy ścieżkę. Czyli jeśli okaże się, że przeprowadzamy relaksację z wierzchołka 2 i poprawiliśmy macierz odległości dla wierzchołka 5, to wtedy zapisujemy `from[5] = 2`. Odtworzenie ścieżki łatwo wtedy przeprowadzić od wierzchołka końcowego.

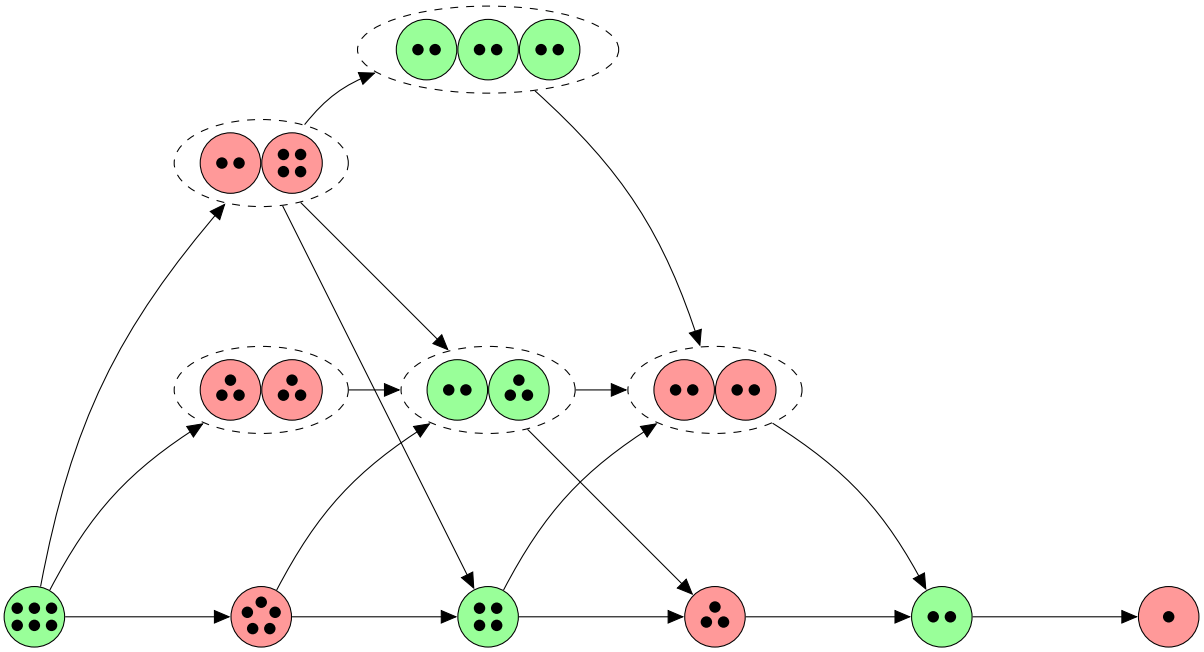
Przykładowa implementacja

Funkcja `construct_path(v)` zapisuje do stosu `S` (patrz rozdział 5.4) ścieżkę od wierzchołka końcowego v do wierzchołka początkowego. Założeniem algorytmu jest zainicjalizowanie tablicy `from` zerami, co umożliwia wykrycie dotarcia do początku. Ścieżka zapisywana jest w kolejności odwrotnej do rzeczywistego kierunku przejścia, dlatego użyty został stos zamiast kolejki — zdejmowanie elementów w sposób właściwy dla kolejki LIFO przywróci właściwy porządek przechodzonych wierzchołków.

```
001 //stos
002 int S[MAX], height = 0;
003 void stack_push(int x);
004
005 int from[MAX];
006
007 void construct_path(int v)
008 {
009     int cur = v;
010
011     while (cur != 0) {
012         stack_push(cur);
013         cur = from[cur];
014     }
015 }
```

Część IV

Kombinatoryczna teoria gier



Ten rozdział jest nieco bardziej teoretyczny od wcześniejszych. Raczej mało będzie tutaj przykładowych kodów źródłowych, ale przedstawiane algorytmy nie powinny sprawić problemów implementacyjnych. Rozpatrywane tutaj gry są relatywnie prostymi grami *bezstronnymi*. W takich zadaniach na ogół należy sprawdzić, czy dla określonego gracza istnieje *strategia wygrywająca*, czyli taki zestaw posunięć, dzięki którym zawsze można wygrać, niezależnie od ruchów przeciwnika.

Zdecydowania większość rozpatrywanych gier określa istnienie dwóch graczy, którzy wykonują ruchy naprzemiennie. Wynika to m.in. z faktu, że wyznaczenie strategii wygrywającej dla gry o większej liczbie graczy często jest niemożliwe.

Aby swobodnie poruszać się w temacie, należy jasno określić kilka podstawowych pojęć. *Pozycja przegrywająca* to taki stan gry, w którym gracz, który ma teraz ruch albo od razu przegrywa (np. szach-mat), albo nie może zrobić niczego, aby zapobiec swojej przegranej w dalszej (nawet bardzo odległej) części gry. Dzięki takiej definicji *pozycję wygrywającą* można określić indukcyjnie względem pozycji przegrywającej, tzn. dana pozycja jest wygrywająca, jeśli możemy z niej zrobić ruch na przynajmniej jedną pozycję przegrywającą. Wtedy pozycją przegrywającą jest taki stan gry, z którego wszystkie dozwolone ruchy prowadzą na pozycje wygrywające.

Taka definicja wynika intuicyjnie z przebiegu samej gry — gracz A, który gra optymalnie próbuje spychać przeciwnika do pozycji przegrywającej. Ponieważ pozycja przegrywająca (jeśli nie kończy jeszcze gry) z powyższej definicji zawsze przemieszcza stan gry do pozycji wygrywającej, więc po ruchu gracza B ponownie gracz A może wykonać ruch, który zmieni stan na przegrywający. Dlatego właśnie pozycja jest wygrywająca wtedy i tylko wtedy, gdy można wykonać przynajmniej jeden ruch sprowadzający na pozycję przegrywającą.

10 Gry bezstronne

W dużym uproszczeniu gry bezstronne to takie gry, w których każdy z graczy może wykonywać ten sam zestaw ruchów i po wykonaniu posunięcia przez określonego gracza nie można rozpoznać kto dokonał zmiany stanu gry. Go, szachy ani warcaby nie są grami bezstronnymi, ponieważ jeden gracz kontroluje tylko czarne kamienie (figury), drugi tylko białe, więc po każdym ruchu widać który z graczy zadziałał.

Aby sprawdzić które pozycje są wygrywające wystarczy skorzystać z definicji i przeszukać stany gry „od końca". Dla każdej pozycji przegrywającej (początkowo wiemy, że przegrywający jest tylko sam koniec gry) badamy, z jakich innych pozycji możemy osiągnąć ten stan w jednym ruchu i właśnie te stany oznaczamy jako wygrywające (ponieważ z nich możemy zrzucić przeciwnika do przegrywającej, jak w definicji).

10.1 Przykład — gra Fibonacciego

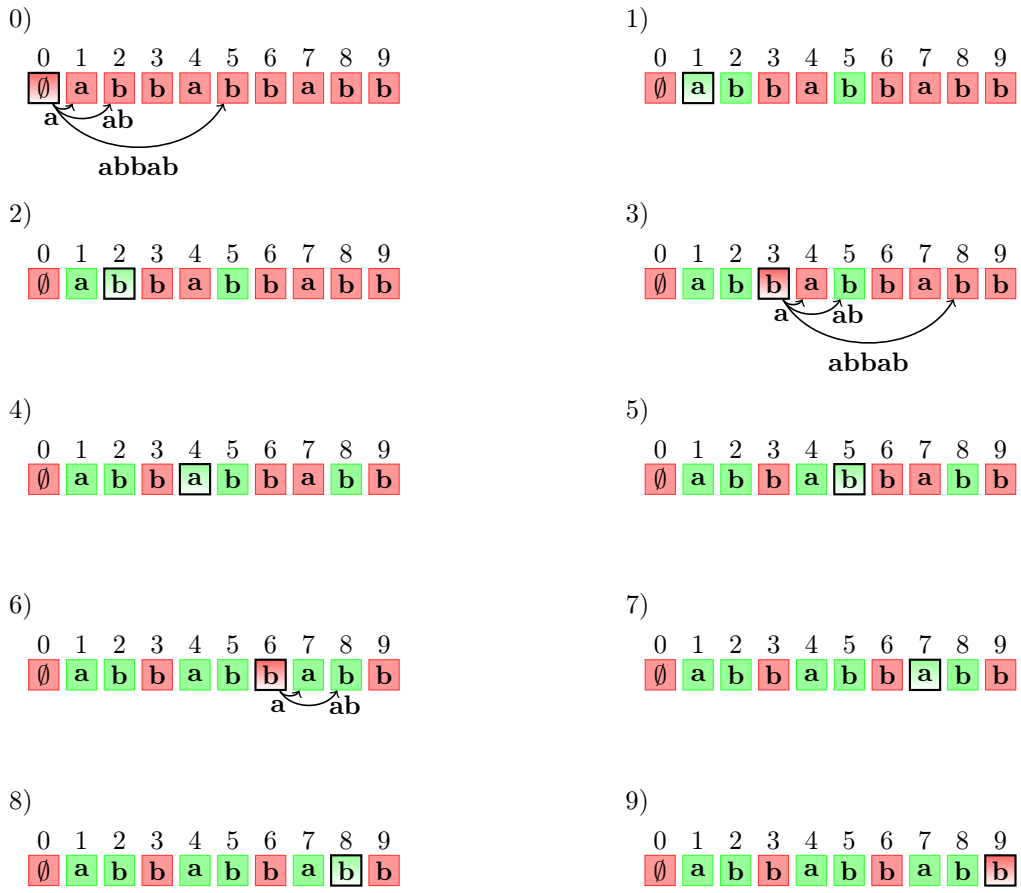
Dany jest ciąg znaków 'a' i 'b' o określonej długości n. Dwóch graczy na przemian odcina po jednym słowie Fibonacciego z prawej strony ciągu. Przegrywa gracz, który nie może wykonać ruchu. Czy gracz rozpoczynający grę ma strategię wygrywającą dla określonego na wejściu ciągu?

Słowa Fibonacciego określamy podobnie do liczb Fibonacciego:

- $S_1 = a$
- $S_2 = b$
- $S_3 = S_1 + S_2 = ab$
- $S_4 = S_2 + S_3 = bab$
- ...

Czyli np. dla danego ciągu *abababab* możemy odciąć:

- S_2 otrzymując *abababa*
- S_3 otrzymując *ababab*
- S_4 otrzymując *ababa*



Tablica 10.1: Sprawdzanie pozycji wygrywających w grze Fibonnaciego. Jak widać gracz rozpoczynający nie ma strategii wygrywającej.

Jak łatwo zauważyć jedyną końcową pozycją przegrywającą jest pusty ciąg. Pozycji końcowych mogłoby być więcej, gdyby niektóre ze słów Fibonacciego nie były dostępne, np. brak S_1 uniemożliwiłby ruch w ciągu składającym się tylko ze znaków `a`.

Początkowo wszystkie niekońcowe stany gry oznaczamy jako przegrywające. Takich stanów mamy dokładnie tyle, ile wynosi długość zadanego ciągu. Wynika to wprost z warunków gry — odcinać można słowa tylko z prawej strony ciągu, więc stan o numerze i będzie oznaczał podciąg składający się z pierwszych i znaków ciągu wejściowego; wtedy stan 0 oznacza podciąg pusty, a stan n to ciąg wejściowy przed dokonaniem jakichkolwiek ruchów.

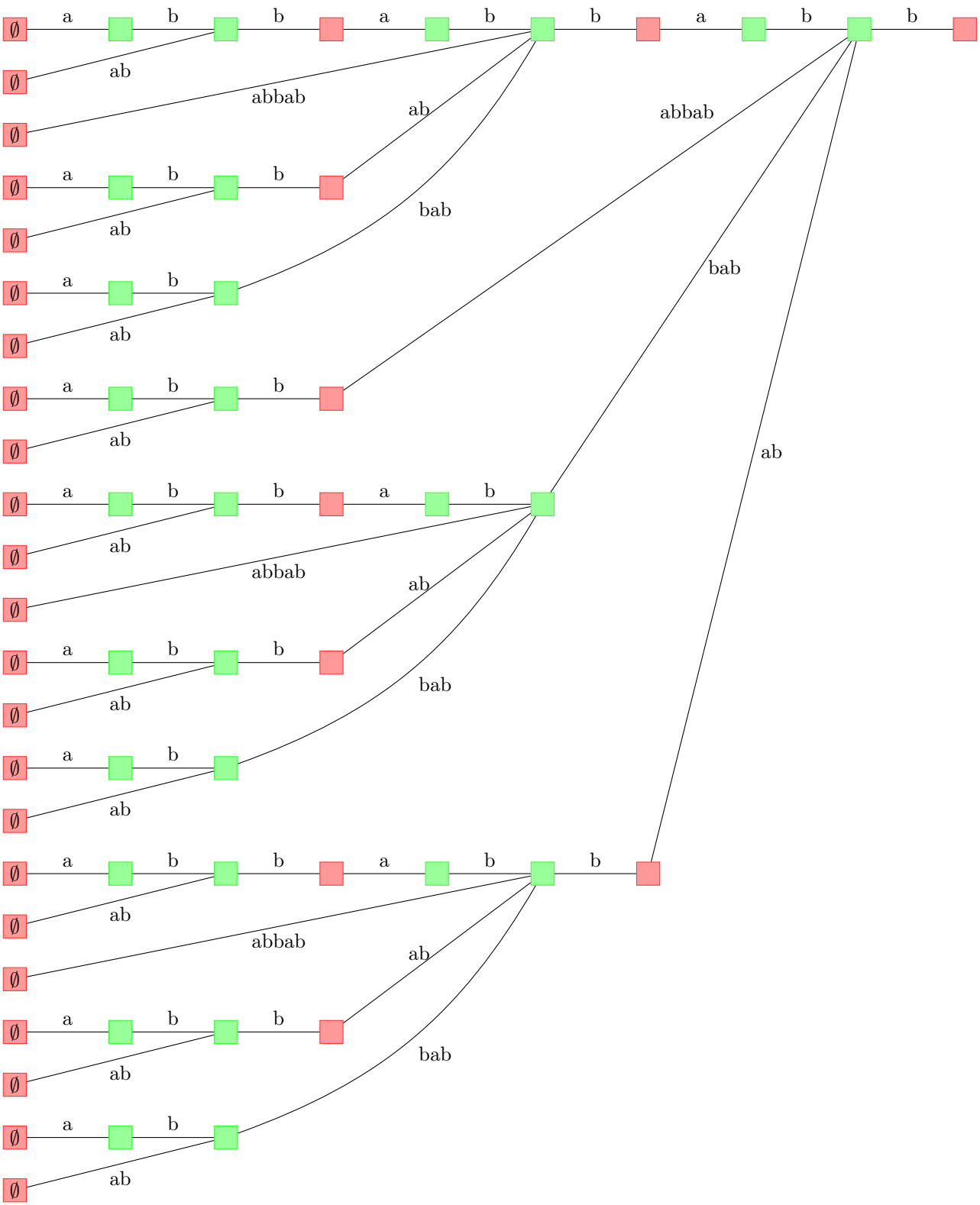
Aby stwierdzić, czy gracz rozpoczynający zawsze może wygrać, należy dowiedzieć się czy stan n jest pozycją wygrywającą. Algorytm rozwiązujący ten problem przebiega wszystkie stany gry od 0 do $n-1$ wykonując następujące operacje:

1. jeśli aktualny stan jest wygrywający, nie rób niczego;
2. w przeciwnym wypadku oznacz wszystkie stany, z których można osiągnąć stan aktualny w jednym ruchu jako wygrywające.

Przebieg algorytmu dla przykładowego ciągu znajduje się w tab. 10.1. Pełne drzewo gry (wszystkie możliwe ruchy) pokazane jest na rys. 10.1. Warto zauważyć, że pozycja raz oznaczona jako wygrana nie zmieni nigdy swojego stanu, więc można przerwać działanie algorytmu w momencie wyliczenia wygrywającego stanu końcowego.

10.2 Wiele gier jednocześnie

Na stole znajduje się pewna liczba kamieni pogrupowanych w stosy. Stosy mogą się od siebie bardzo różnić licznością. Legalny ruch polega na wybraniu niepustego stosu i zabraniu z niego dowolnej (ale koniecznie niezerowej) liczby kamieni. Przegrywa ten gracz, który nie może wykonać legalnego ruchu — na żadnym ze stosów nie ma kamieni do zabrania.



Rysunek 10.1: Pełne drzewo przykładowej gry Fibonnaciego.

and	0	1
0	0	0
1	0	1

or	0	1
0	0	1
1	1	1

xor	0	1
0	0	1
1	1	0

Tablica 10.2: Schemat działania alternatywy wykluczającej, porównanie z innymi operacjami logicznymi.

Niektórzy zapewne poznają powyższy opis — jest to gra Nim. Na jej podstawie przedstawię ogólny schemat sprawdzania istnienia strategii wygrywającej dla wielu gier jednocześnie. Dlaczego dla wielu? Ponieważ każdy stos kamieni rozpatrujemy jako oddzielną grę, których dopiero połączenie na pewnych ściśle określonych zasadach da pełną informację o istnieniu (lub nie) strategii wygrywającej dla całej gry.

Zacznijmy od trywialnych przypadków. Jeśli w grze istnieje dokładnie jeden stos o dowolnej liczności n, to gracz rozpoczynający ma strategię wygrywającą — po prostu zabiera wszystkie kamienie. Jeśli gra składa się z dokładnie dwóch stosów o takich samych licznościach, to wtedy gracz rozpoczynający nie ma strategii wygrywającej. Strategia gracza drugiego polega wtedy na kopiowaniu ruchów przeciwnika. Prędzej czy później jeden ze stosów wyzeruje się, wtedy gracz drugi także wyzeruje pozostały stos i gracz pierwszy przegrywa.

Dokładnie odwrotny przebieg gry zachodzi dla dwóch stosów o różnych licznościach. Pierwszy gracz zabiera tyle kamieni z liczniejszego stosu, aby wyrównać ich stan i mamy dokładnie taką sytuację jak wyżej.

Pozornie niczego nam to nie daje, bo „zadaniowa" gra odbywa się często na setkach tysięcy stosów jednocześnie. Na szczęście dwaj panowie, Sprague i Grundy, opracowali (niezależnie od siebie) pewną metodę „wyliczania" gier bezstronnych.

Każdą grę bezstronną (także taką pojedynczą, jak np. stos kamieni w grze Nim) można przedstawić za pomocą pewnej liczby naturalnej, nazywanej dalej *nimber*. Nimber dwóch gier bezstronnych równa się operacji bitowej *xor* (oznaczanej dalej symbolem $\oplus$) na ich nimberach, tzn. niech N_1, N_2 będą nimberami pewnych podgier. Jeśli chcemy policzyć nimber całej gry, to obliczamy $N_1 \oplus N_2$. Nimbery pozycji przegrywających (w tym stanów końcowych) są równe 0.

Małe przypomnienie z elementarnej logiki: $\oplus$, czyli alternatywa wykluczająca (albo–albo) pomiędzy dwoma wartościami logicznymi zwraca prawdę tylko wtedy, gdy dokładnie jedna z nich jest prawdziwa (patrz tabela 10.2). W naszym przypadku nie chodzi o operację zwracającą wartość logiczną (prawda–fałsz), ale o operację bitową działającą na liczbach całkowitych w systemie dwójkowym. Przykład działania w tab. 10.3.

Kilka właściwości alternatywy wykluczającej:

- $a \oplus b = b \oplus a$
- $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- $a \oplus a = 0$
- $a \oplus 0 = a$

Z powyższych równań wprost wynika, że jeśli chcemy obliczyć nimber całej gry poprzez xorowanie nimberów jej składowych, to kolejność wykonywania działań jest całkowicie nieistotna.

Na razie wiemy jak składać mniejsze gry w większe, ale jeszcze nie potrafimy obliczać nimberów dla mniejszych gier. Zacznijmy od definicji pewnej przydatnej funkcji:

$$\mathrm{mex}(X) = \min\{x \colon x \notin X\}$$

gdzie X jest pewnym skończonym zbiorem liczb naturalnych. Mniej formalnie: funkcja mex (*minimal excludant*) użyta na pewnym zbiorze zwróci najmniejszy element **nie występujący** w nim. Zastosowanie do zbioru pustego zwraca w wyniku zero. Jak zwykle kilka przykładów:

- $X = \{0, 1, 3\}$ $\mathrm{mex}(X) = 2$
- $X = \{1, 3, 4\}$ $\mathrm{mex}(X) = 0$
- $X = \{0, 1, 2, 3, 4\}$ $\mathrm{mex}(X) = 5$

Jeszcze jedna kluczowa definicja: weźmy sobie pewien stan gry G, a *stany osiągalne* ze stanu G to zbiór stanów gier $\{G'_1, G'_2, \ldots, G'_n\}$, do których można trafić wykonując **dokładnie jeden** legalny ruch w grze. Przy założeniu, że pozycja końcowa w grze ma nimber równy 0, to nimber dowolnego niekońcowego stanu gry wyraża się wzorem

$$\mathit{nimber}(G) = \mathrm{mex}(\{\mathit{nimber}(G'_1), \mathit{nimber}(G'_2), \ldots, \mathit{nimber}(G'_n)\})$$

Wróćmy do gry Nim, rozpatrując nimbery pojedynczej sterty kamieni. Istnieje tylko jedna pozycja końcowa G_0 (stos pusty), której nimber ustawiamy na 0. Weźmy sobie stos kamieni o pewnej liczności $n \geq 1$. Ponieważ można wziąć dowolną liczbę kamieni z jednego stosu, to możemy osiągnąć każdy stan „wcześniejszy". Czyli:

	1	1	0	0
xor		1	0	1
	1	0	0	1

Tablica 10.3: $12 \oplus 5 = 9$

- $nimber(G_0) = 0$
- $nimber(G_1) = \text{mex}(\{nimber(G_0)\}) = \text{mex}(\{0\}) = 1$
- $nimber(G_2) = \text{mex}(\{nimber(G_0), nimber(G_1)\}) = \text{mex}(\{0, 1\}) = 2$
- $nimber(G_3) = \ldots = \text{mex}(\{0, 1, 2\}) = 3$
- ...
- $nimber(G_n) = \ldots = \text{mex}(\{0, 1, 2, \ldots, n-1\}) = n$

Jak widać nimber każdego stosu równa się jego liczności, więc nimber całej gry to *xor* po licznościach stosów. Pozostaje jeszcze poznać kiedy istnieje strategia wygrywająca, ale tak naprawdę już to wiemy:

Twierdzenie. *W grze bezstronnej G istnieje strategia wygrywająca $\iff nimber(G) \neq 0$.*

Procedura obliczania nimberów za pomocą funkcji mex oraz składanie mniejszych gier w większe poprzez $\oplus$ są tak zdefiniowane, aby **każdy** ruch ze stanu przegrywającego ($nimber = 0$) powodował przejście do stanu wygrywającego ($nimber > 0$) oraz żeby z dowolnego stanu wygrywającego dało się wykonać ruch do stanu przegrywającego (czyli wyzerować nimber). Jest to uogólnienie zasady o stanach wygrywających i przegrywających, podanej na początku rozdziału.

Użycie operacji *xor* można uzasadnić w ten sposób: jeśli jesteśmy w stanie o zerowym nimberze, to jakiegokolwiek ruchu byśmy nie wykonali, zawsze zmienimy przynajmniej jeden bit w sumie wszystkich gier, czyli przemieścimy się z nimbera zerowego do niezerowego. Zarazem wtedy możemy zrobić ruch w taki sposób, aby sprowadzić przeciwnika z powrotem do nimbera zerowego. Dociekliwi Czytelnicy bez problemu znajdą dokładny dowód tego stwierdzenia w literaturze.

Spróbujmy teraz odrobinę zmienić zasady. Zmodyfikujmy grę Nim nakładając pewne ograniczenia na dozwoloną liczbę zabieranych w jednym ruchu kamieni. Teraz zróbmy tak, że w jednym ruchu można wziąć co najwyżej k kamieni. Dla przykładu weźmy $k = 3$. Rozkład nimberów dla poszczególnych stanów gry przebiega następująco:

- $nimber(G_0) = 0$
- $nimber(G_1) = \text{mex}(\{nimber(G_0)\}) = \text{mex}(\{0\}) = 1$
- $nimber(G_2) = \text{mex}(\{nimber(G_0), nimber(G_1)\}) = \text{mex}(\{0, 1\}) = 2$
- $nimber(G_3) = \ldots = \text{mex}(\{0, 1, 2\}) = 3$
- $nimber(G_4) = \ldots = \text{mex}(\{1, 2, 3\}) = 0$
- $nimber(G_5) = \ldots = \text{mex}(\{0, 2, 3\}) = 1$
- $nimber(G_6) = \ldots = \text{mex}(\{0, 1, 3\}) = 2$
- $nimber(G_7) = \ldots = \text{mex}(\{0, 1, 2\}) = 3$
- $nimber(G_8) = \ldots = \text{mex}(\{1, 2, 3\}) = 0$
- ...

Ograniczenie spowodowało, że dla każdej gry G_n, $n \geqslant 3$ jej zbiór stanów osiągalnych zamyka się w trzech innych: $\{G_{n-3}, G_{n-2}, G_{n-1}\}$. Efekt: widoczna powyżej cykliczna powtarzalność nimberów i w konsekwencji banalny sposób obliczania ich wartości:

$$nimber(G_n) = n \bmod 4$$

Lub bardziej ogólnie, dla dowolnego k

$$nimber(G_n) = n \bmod (k+1)$$

To jeszcze też było proste. Powiedzmy, że teraz możemy brać tylko liczby pierwsze: 2 kamienie, 3 kamienie, 5 kamieni itd. Najbardziej widoczną konsekwencją jest rozszerzenie zbioru stanów przegrywających — nie można wziąć dokładnie jednego kamienia, więc stan G_1 także staje się końcowym. Czyli:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$nimber(G_n)$	0	0	1	1	2	2	3	3	4	0	0	1	1	2	2

Tablica 10.4: Gra Nim na liczbach pierwszych.

- $nimber(G_0) = nimber(G_1) = 0$
- $nimber(G_2) = \text{mex}(\{nimber(G_0)\}) = \text{mex}(\{0\}) = 1$
- $nimber(G_3) = \text{mex}(\{nimber(G_0), nimber(G_1)\}) = \text{mex}(\{0\}) = 1$
- $nimber(G_4) = \text{mex}(\{nimber(G_1), nimber(G_2)\}) = \text{mex}(\{0, 1\}) = 2$
- ...

To już nie jest tak proste. Z powodu bardzo specyficznego dozwolonego zbioru ruchów nie jest znana regularność (chociaż na pierwszy rzut oka może się wydawać coś innego, patrz tab. 10.4) pozwalająca na liczenie nimberów stosów kamieni w czasie stałym, tak jak można było w powyższych przypadkach. Skonstruowana w ten sposób gra Nim wymusza „siłowe" obliczanie nimberów, bez żadnych szczególnie sprytnych kruczków pozwalających na pójście skrótem.

10.3 Rozbijanie gier na mniejsze

Rozpatrzmy wariant gry Nim, w której legalnym ruchem jest rozbicie stosu kamieni na dwa niepuste stosy[1]. Z tego oczywiście wynika, że stan G_1 jest przegrywający i G_2 wygrywający. G_3 jest przegrywający, bo rozbija się na wygrywający stos G_2 oraz stos G_1, z którym niczego nie da się zrobić. Można jeszcze na palcach rozważyć kilka gier, np. G_4 jest wygrywające, bo możemy zrobić rozbicie na dwie gry G_2. Dwie takie same gry przypominają sytuację o dwóch jednakowo licznych stosach kamieni z początku rozdziału – jeden z graczy kopiuje ruchy drugiego aż do końca gry.

Takie rozważania jednak bardzo szybko staną się trudne do kontynuowania ze względu na szybko rosnący rozmiar drzewa gry. W dodatku co się będzie działo, jeśli skomplikujemy odrobinę zasady gry, na przykład zażądamy, aby legalnym ruchem było rozbicie wybranego stosu kamieni na pięć różnolicznych stosów? Potrzebujemy sensownie prostego mechanizmu wyliczania nimberów dla takich gier.

W naszym przypadku jeden ruch polega na usunięciu wybranej gry i wstawieniu na jej miejscu dwóch mniejszych gier. One zupełnie niezależnie od siebie mogą rozbijać się na kolejne, gwałtownie zwiększając liczbę indywidualnych gier, z którymi mamy do czynienia. Aby nie mnożyć bytów ponad potrzebę, chcielibyśmy policzyć nimbery dla stosów kamieni w jakiś sensownie unikający zbyt wielu „rozbić" sposób.

Tak naprawdę mamy już wszystkie narzędzia, aby poradzić sobie z wyliczaniem nimberów dla rozpadających się gier. Załóżmy, że mamy grę G, którą rozbijamy w jednym ruchu na zbiór gier $\{G'_1, G'_2, \ldots, G'_k\}$. Z poprzedniego rozdziału wiadomo, że nimber „dużej" gry składającej się z wielu małych jest równy xorowi nimberów małych gier. Możemy zatem uznać, że nowopowstały zbiór gier jest jedną dużą grą H i obliczyć jej nimber:

$$nimber(H) = nimber(G'_1) \oplus nimber(G'_2) \oplus \ldots \oplus nimber(G'_k)$$

Zatem o ile znamy już nimbery tych mniejszych gier, możemy obliczyć nimber wyjściowej gry G (w standardowej drodze użycia funkcji mex do zbioru nimberów stanów gier osiągalnych z G).

Na rysunku 10.2 dokładnie pokazano proces obliczania nimberów dla stosów kamieni o coraz większych licznościach. Na podstawie tych kilku przykładów może się wydawać, że nimber w tej grze jest zawsze 1 dla stosów o parzystej liczbie kamieni i zawsze 0 dla stosów o nieparzystej kamieni, można się jednak łatwo przekonać o nieprawdziwości tej tezy, obliczając nimber dla gry o 9 kamieniach.

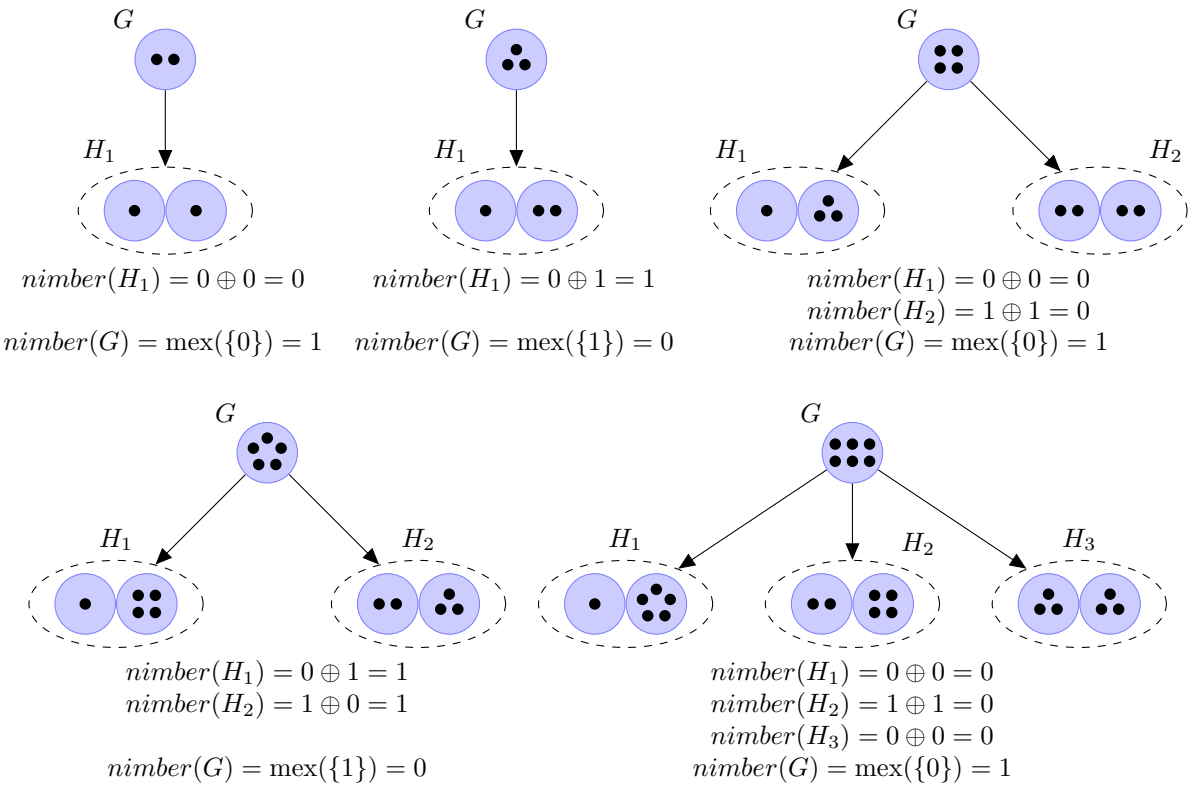
Zadanie związane bezpośrednio z ideą rozbijania gier pojawiło się na I etapie VII Olimpiady Informatycznej i nazywało się *Paski*.

10.4 Schodkowy Nim

Tym razem stosy kamieni ustawione są na schodkach ponumerowanych od 1 do pewnej liczby n. Niektóre schodki mogą być puste. Ruch polega na wybraniu schodka k z niezerową liczbą kamieni oraz przeniesieniu wybranej ich liczby na schodek $k-1$. Kamienie zabrane ze schodka pierwszego wypadają z gry. Standardowo gracz, który nie może zrobić ruchu, przegrywa. Żeby nie skatować do reszty własnej klawiatury i cierpliwości Czytelnika, od teraz „schodek i-ty" będę w skrócie oznaczał S_i.

Na pierwszy rzut oka nie bardzo wiadomo jak analizować taką grę. Widać oczywiście, że gra, w której wszystkie kamienie znajdują się na S_1 jest wygraną gracza rozpoczynającego (gracza A), który po prostu usunie w jednym ruchu

[1] Jest to gra bardzo podobna do *gry Grundy'ego*, w której istnieje dodatkowe wymaganie, aby nowe stosy były różnoliczne.



Rysunek 10.2: Rozbijanie stosów kamieni.

wszystkie kamienie z gry. Natomiast jeśli wszystkie kamienie są w S_2, to A zawsze przegra – każdy jego ruch będzie polegał na przeniesieniu pewnej liczby kamieni z S_2 do S_1, co z kolei będzie kontrowane przez gracza drugiego (gracza B) przeniesieniem wszystkich kamieni (czyli tak naprawdę tej samej liczby, ile przemieścił pierwszy gracz) z S_1 do S_0.

Co ciekawe, analogiczne rozumowanie można przeprowadzić dla dowolnego S_i: jeśli i jest nieparzyste, to A zawsze wygrywa. Optymalna strategia gracza A będzie polegała na przeniesieniu wszystkich kamieni o stopień niżej. Jeśli B w swoim ruchu przełoży wszystkie kamienie z S_{i-1} do S_{i-2}, to mamy tak naprawdę sytuację wyjściową, przemieszczoną dwa stopnie niżej. Jeśli natomiast B przełoży jedynie część kamieni, to my od razu ten nowopowstały stos przemieszczamy znowu w całości na stopień poniżej. Zauważmy pewną ciekawą prawidłowość: każdy ruch gracza A doprowadza do stanu, w którym wszystkie kamienie znajdują się na stopniach o parzystym numerze. Wtedy B musi przynajmniej jeden kamień przemieścić do stopnia o numerze nieparzystym i koło się zamyka.

Powyższa reguła radzenia sobie z pojedynczym stosem wystarczy na dokonanie gwałtownego skoku z gry o jednym stosie do gier o wielu stosach. Stosy umieszczone na schodkach o numerach parzystych są całkowicie nieistotne – każdy ruch przemieszczający pewną liczbę kamieni z S_{2i} do S_{2i-1} możemy natychmiast skontrować przemieszczając tyle samo kamieni z S_{2i-1} do S_{2i-2}. Możemy zatem sprowadzić schodkową wersję Nim do zwykłego Nim, biorąc jedynie stosy kamieni znajdujące się na stopniach o nieparzystych numerach i zapominając o wszystkich pozostałych.

Zasada schodkowego Nim pojawia się czasami w mało spodziewanych miejscach. Na przykład w zadaniu *Kamyki* z I etapu XVI Olimpiady Informatycznej dośc wyraźnie widać co się święci, ale w bardzo ciekawym zadaniu *Gra* z I etapu XI Olimpiady Informatycznej już zdecydowanie mniej.

10.5 Więcej niż jeden ruch

Rozważmy teraz wariację Nim, w której możemy zabrać dowolną liczbę kamieni z **co najwyżej** k różnych stosów (ale zawsze musimy z przynajmniej jednego). Jest to odmiana wymyślona i przeanalizowana przez Eliakima Moore'a, przez co nazywana jest często *Nimem Moore'a*. Alternatywnie można przedstawić reguły jako „każdy gracz może wykonać co najwyżej k pojedynczych ruchów" – takie przedstawienie problemu może być przydatne w przypadku mieszania ze sobą różnych gier.

Będziemy potrzebowali pewnego uogólnienia operacji $\oplus$. Do tej pory uznawaliśmy xor za operację logiczną albo-albo, którą stosowaliśmy do bitów liczb binarnych, otrzymując w ten sposób inne liczby. Można jednak dokonać definicji w efekcie równoważnej: $\oplus$ jest operacją *dodawania bez przeniesienia w bazie dwójkowej*. Tzn. jeśli xorujemy kilka liczb, to zapisujemy je jako binarne ciągi i dodajemy szkolną metodą „w słupku" z jedyną różnicą w postaci pominięcia przeniesienia „przekręconych" wartości bitowych (tab. 10.5). Określmy zatem działanie $\oplus_n$ jako operację bitowego *dodawania bez przeniesienia w bazie* n – wedle tej notacji do tej pory używaliśmy operacji $\oplus_2$.

	1	1		1	
		1	0	0	1
			1	0	1
+		1	1	0	1
	1	1	0	1	1

	1	0	0	1
		1	0	1
⊕	1	1	0	1
	0	0	0	1

Tablica 10.5: $9 + 5 + 13 = 27$ (przeniesienie zaznaczone na czerwono u góry), $9 \oplus 5 \oplus 13 = 1$

Chcemy nieformalnie pokazać, że w grze, w której możemy zabrać dowolną liczbę kamieni z co najwyżej k stosów (w każdym stosie niezależnie od pozostałych) stan $G = \{G_1, G_2, \ldots, G_n\}$ jest przegrywający wtedy i tylko wtedy, gdy $nimber(G_1) \oplus_k nimber(G_2) \oplus_k \ldots \oplus_k nimber(G_n) = 0$. Dość łatwo zauważyć, że będąc w niekończowym stanie o nimberze równym 0 dowolny ruch przemieszcza nas do stanu o nimberze niezerowym. Wynika to z obliczania nimberów za pomocą $\oplus_k$ oraz ograniczenia na liczbę ruchów, które można wykonać: możemy zmienić każdy bit co najwyżej $k-1$ razy, co nie wystarczy, aby ten bit przemieścić ze stanu równego 0 do stanu niezerowego i z powrotem.

Pozbywając się wszelkich hamulców moralnych dotyczących braku formalizmu w powyższym uzasadnieniu, można zamachać rękoma, że korzystając z analogicznego argumentu zawsze ze stanu gry o niezerowym nimberze można przejść do stanu o zerowym. Widać bowiem, że wartość każdego „bitu" o podstawie k różna od zera może zostać sprowadzona do wartości zerowej drogą co najwyżej $k-1$ modyfikacji tego „bitu".

10.6 Gry z remisami

Do tej pory zajmowaliśmy się wyłącznie grami, które z każdym ruchem zbliżały się do nieuchronnego końca. Stany gry i przejścia między nimi miały postać drzewa albo acyklicznego grafu skierowanego, stąd łatwo było obliczyć nimber każdego stanu (np. w kolejności topologicznej). Rozpatrzmy teraz grę, w której graf przejść pomiędzy jej stanami może zawierać cykle. Dokładniej określimy grę następująco: mamy graf skierowany $G = (V, E)$, w którym znajduje się przynajmniej jeden wierzchołek końcowy, tzn. taki, z którego nie wychodzą żadne krawędzie. W niektórych węzłach grafu znajdują się pionki. Dwóch graczy na przemian wybiera jeden pionek i przesuwa go o jedną krawędź. Przegrywa gracz, który nie może wykonać żadnego ruchu.

O ile wcześniej gra zawsze dobiegała końca, tutaj wcale tak być nie musi. Przypuśćmy, że w grze mamy dokładnie jeden pionek, który znajduje się na pewnym cyklu. W dodatku cykl ten jest tak umiejscowiony względem reszty grafu i wierzchołków końcowych, że wyjście pionkiem poza cykl powoduje pewną przegraną. Wtedy żadnemu graczowi nie opłaca się doprowadzać do końca gry – trwa ona w nieskończoność i mówimy, że wynikiem jest remis.

Spróbujmy policzyć nimber dla każdego wierzchołka grafu. Być może, ze względu na cykle, nie uda nam się tego dokonać. Jeśli jednak uda się to przynajmniej dla wierzchołków, w których znajdują się pionki, to potrafimy także obliczyć nimber całej gry, a zatem umiemy odpowiedzieć na pytanie dotyczące wyniku rozgrywki przy założeniu optymalnej strategii obu graczy. Przechodząc w odwróconym porządku topologicznym policzymy bez problemu nimbery dla wierzchołków „łatwych", tzn. nie występujących na żadnym cyklu. Dla pozostałych wierzchołków (nazwiemy je *nieoznaczonymi*) przyjmijmy umownie, że ich nimber jest równy nieskończoności. Zastosujemy tzw. *regułę Smitha*:

1. Wybieramy węzeł v taki, że $nimber(v) = \infty$. Weźmy zbiór następników v i podzielmy go na dwa podzbiory: do zbioru $N(v)$ wrzucimy takie węzły u że $nimber(u) \neq \infty$, natomiast do $N_\infty(v)$ wrzucimy takie u, że $nimber(u) = \infty$ (czyli wszystkie pozostałe).

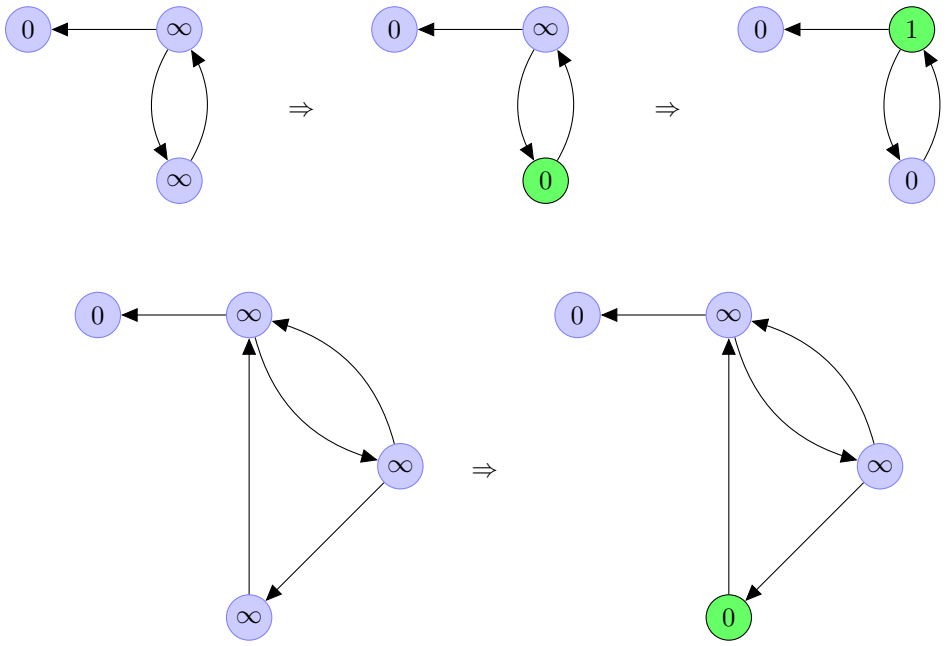
2. Obliczamy wartość $M = \text{mex}(N(v))$. Jeśli $N(v)$ jest pusty, to oczywiście $M = 0$.

3. Jeśli **dla każdego** wierzchołka $u \in N_\infty(v)$ istnieje krawędź $u \to w$ do takiego węzła w, że $nimber(w) = M$, to wtedy możemy bezpiecznie ustalić $nimber(v) = M$. Wynika to z faktu, że jeśli wszystkie wierzchołki należące do $N_\infty(v)$ mają sąsiada o nimberze równym M, to żaden z tych wierzchołków nie może mieć nimbera M, a zatem pierwszym nimberem nienależącym do zbioru następników v będzie właśnie M. Czyli z definicji funkcji mex wynika $nimber(v) = M$.

Przykład zastosowania powyższego rozumowania można zobaczyć na rysynku 10.3. W pierwszym grafie udaje się dwukrotnie zastosować regułę Smitha do obliczenia nimberów wszystkich wierzchołków leżących na cyklu. W drugim grafie można zastosować regułę Smitha tylko raz. Pozostałe wierzchołki nieoznaczone mają po jednym oznaczonym następniku z nimberem równym 0 (czyli wartość mex dla nich wynosi 1) i zarazem nie posiadają nieoznaczonego następnika, który miałby sąsiada o nimberze 1.

Mając obliczone wartości *nimber* dla możliwie największego zbioru wierzchołków grafu, możemy przystąpić do rozwiązania początkowego problemu: czy dla danego rozstawienia pionków w grafie gracz rozpoczynający ma strategię wygrywającą? Musimy rozpatrzyć kilka przypadków:

1. Wszystkie pionki znajdują się w wierzchołkach, dla których dało się policzyć ich nimbery. Sytuacja ta nie różni się niczym od gry bez cykli – obliczamy standardowo xor nimberów i sprawdzamy czy wynik jest niezerowy.



Rysunek 10.3: Wizualizacja reguły Smitha.

2. Dokładnie jeden pionek znajduje się na polu nieoznaczonym v. Oznaczmy nimbery wierzchołków, na których stoją pozostałe pionki przez n_1, n_2, ..., n_k. Niech $N = n_1 \oplus n_2 \oplus \ldots \oplus n_k$. Jeśli istnieje krawędź z v do takiego wierzchołka u, że $N \oplus nimber(u) = 0$, to mamy strategię wygrywającą – ta krawędź oznacza ruch, który należy wykonać.

3. Dokładnie jeden pionek znajduje się na polu nieoznaczonym, lecz bez wygrywającego ruchu. Wtedy mamy remis, bowiem graczowi rozpoczynającemu nie opłaca się opuszczać cyklu. Nawet jeśli pozostałe pionki (o ile istnieją) zostaną w toku gry przemieszczone do pozycji końcowej, to zostanie ten jeden pionek, którym żadnemu z graczy nie będzie opłacało się przerwać cyklu. Gra zatem trwać będzie w nieskończoność.

4. Więcej niż jeden pionek znajduje się na polu nieoznaczonym. Wtedy też mamy remis, z powodów analogicznych do wymienionych powyżej.