OLSZTYŃSKA WYŻSZA SZKOŁA
INFORMATYKI i ZARZĄDZANIA
im. Prof. Tadeusza Kotarbińskiego

KRZYSZTOF GIARO

ZŁOŻONOŚĆ OBLICZENIOWA ALGORYTMÓW W ZADANIACH

OLSZTYN, GDAŃSK 2011

Spis treści

Wstęp

Skrypt ten przeznaczony jest dla studentów informatyki jako pomoc dydaktyczna z przedmiotu „Podstawy teorii obliczeń". Może też stanowić literaturę uzupełniającą dla osób pragnących samodzielnie zaznajomić się z problemami praktycznej analizy programów pod kątem ich złożoności.

W chwili obecnej na rynku wydawniczym uderza brak jakiejkolwiek literatury polskojęzycznej mogącej posłużyć za wprowadzenie do analizy złożonościowej. Nieliczne prace, jak np. [7], przeznaczone są bowiem dla czytelników zaawansowanych i nie powinny być używane w ramach kursu podstawowego. Inne (patrz [4]) prezentują bardziej matematyczne niż praktyczne spojrzenie na poruszane zagadnienia, posługując się aparatem teorii automatów i klas złożoności języków formalnych. Ten sposób opisu jest równie istotny z teoretycznego punktu widzenia, co trudny do przełożenia na grunt praktycznego badania programów implementowanych na współczesnych komputerach. Chyba jedyny wyjątek od obu wymienionych nurtów stanowi książka [5] prof. Marka Kubale, która wraz z [6] od lat zapewnia literaturową bazę wykładu ze złożoności obliczeniowej dla studentów informatyki Wydziału ETI Politechniki Gdańskiej. Obie prace definiują podstawowe pojęcia i opisują metody, jakimi posługujemy się przy określaniu złożoności algorytmów, przeprowadzają analizę procedur rekurencyjnych, zawierają wprowadzenie do teorii NP–zupełności, omawiają algorytmy suboptymalne dla problemów NP–trudnych. Jednak celem pełnego przyswojenia materiału przez studentów, poza przekazaniem wiedzy teoretycznej, konieczne było opracowanie zestawu praktycznych ćwiczeń.

Niestety, podczas zajęć ze studentami nader często można było obserwować daleko idącą niesamodzielność myślenia analitycznego i niezrozumienie tematu. Zapewne stanowi to odzwierciedlenie ogólnego schematu edukacyjnego preferującego pamięciowe opanowanie gotowych procedur postępowania nad głębszą merytoryczną refleksją. W przypadku przedmiotu wymagającego samodzielnego formułowania hipotez oraz prób ich dowodzenia, czy też tworzenia bardziej abstrakcyjnych konstrukcji myślowych (takich jak wielomianowe redukcje z problemów NP–zupełnych) to wyrobione latami podejście całkowicie się nie sprawdza. W rezultacie w trakcie zajęć często okazywało się, że obok zadań do samodzielnego rozwiązania niezbędne były dodatkowe przykłady i komentarze metodologiczne. W końcu ilość materiału dydaktycznego przerosła przewidziane przez plan studiów ramy godzinowe – został więc on przelany na papier w formie niniejszego skryptu.

Praca ta nie zawiera definicji podstawowych, ani nie przeprowadza systematycznego kursu przedmiotu. Zakłada się, że czytelnik dysponuje książkami [5] i [6], lub przynamniej bierze udział w wykładzie. Jej treść stanowi ponad sto podzielonych na bloki tematyczne zadań – wszystkie zostały opatrzone szczegółowo omówionymi rozwiązaniami. Źródłem ćwiczeń były prywatne zbiory osób prowadzących od lat zajęcia z przedmiotu, zadania zawarte w wymienianych, wcześniej opublikowanych podręcznikach, wreszcie przykłady z kolokwiów – te, które okazywały się szczególnie niewdzięczne. Przy doborze kierowano się kryterium umożliwienia stopniowego wprowadzania w tematykę, od ćwiczeń elementarnych do bardziej skomplikowanych oraz prezentacji jak najszerszego spektrum skutecznych metod działania. Obok rozwiązań pojawiają się oznaczone ramkami „teoretyczne" komentarze eksponujące najistotniejsze elementy przedstawionych rozumowań i analizujące najczęściej popełniane błędy oraz odpowiedzi na powtarzające się pytania. Uważam, że szczególnie godne uwagi są ostatnie trzy rozdziały podręcznika, zawierające elementy wprowadzenia w teorię NP–zupełności oraz rozwiązywanie problemów NP–trudnych – a to z uwagi na brak jakiejkolwiek polskojęzycznej literatury traktującej o wymienionych zagadnieniach na poziomie podstawowym.

W tym miejscu chciałbym wyrazić podziękowanie panu prof. Markowi Kubale, który skłonił mnie do podjęcia pracy nad tą publikacją, za udostępnienie zasobów własnego wieloletniego dorobku dydaktycznego i zgodę na jego wykorzystanie, cenne sugestie i ogólny patronat nad poprawnością naukową.

Gdańsk, luty 2011 r.

Krzysztof Giaro
giaro@eti.pg.gda.pl

1. Podstawy matematyczne

W tym rozdziale przypomnimy podstawowe narzędzia matematyczne używane podczas analizy algorytmów. Są nimi: metoda dowodzenia indukcyjnego oraz symbole asymptotycznych oszacowań tempa wzrostu funkcji.

Zadanie 1.1.
Udowodnij przez indukcję, że jeśli *n* jest liczbą naturalną, to:

1) $\sum_{i=2}^{n}\binom{i}{2}=\binom{n+1}{3}$, dla $n\geq 2$

2) $\sum_{i=0}^{n} i^3 = (n(n+1)/2)^2$

Rozwiązanie.

1) Najpierw sprawdzamy, że dla $n=2$ po lewej stronie uzyskujemy $\binom{2}{2}=1$ i podobnie po prawej mamy $\binom{2+1}{3}=1$, czyli w tym przypadku zachodzi równość.

Dalej rozpisując lewą stronę naszego równania dla $n+1$ i zakładając jego poprawność dla n otrzymujemy prawą stronę dla $n+1$:

$$\sum_{i=2}^{n+1}\binom{i}{2}=\sum_{i=2}^{n}\binom{i}{2}+\binom{n+1}{2}=^{*}\binom{n+1}{3}+\binom{n+1}{2}=$$
$$=(n+1)n(n-1)/3!+(n+1)n/2=(n+2)(n+1)n/3!=\binom{n+2}{3}$$

Równość oznaczona gwiazdką jest konsekwencją założenia indukcyjnego.

2) Podobnie jak poprzednio zaczynamy od sprawdzenia równości w punkcie początkowym, tym razem jest to $n=0$ – po obu stronach otrzymujemy 0. A zatem dla $n+1$ lewa strona przybiera postać:

$$\sum_{i=0}^{n+1} i^3 = \sum_{i=0}^{n} i^3 + (n+1)^3 = ^{*} (n(n+1)/2)^2 + (n+1)^3 =$$
$$=(n+1)^2(n^2+4(n+1))/2^2=((n+1)(n+2)/2)^2$$

Tu zaznaczona równość przedstawia założenie indukcyjne. A zatem uzyskaliśmy tezę naszego twierdzenia dla $n+1$.

Zachodzą wzory:

$$\sum_{i=0}^{n} i = n(n+1)/2, \qquad \sum_{i=0}^{n} i^2 = n(n+\frac{1}{2})(n+1)/3, \qquad \sum_{i=0}^{n} i^3 = (n(n+1)/2)^2,$$

ogólniej dla dowolnej liczby naturalnej k mamy:

$\sum_{i=0}^{n} i^k = \frac{1}{k+1} n^{k+1} + w_k(n)$, gdzie $w_k(n)$ jest pewnym wielomianem stopnia k zmiennej n.

Zadanie 1.2.
Uszereguj funkcje według tempa wzrostu (tzn. zgodnie z relacją $o(...)$)
$2^{\sqrt{\ln n}}$, $2n$, $\log_2 n$, $\log_2\log_2 n$, $\sqrt{n}$, $n/\log_2 n$, $\sqrt{n}\log_2 n$, $(1/3)^n$, $(3/2)^n$, 17, $(n/2)^{\ln n}$, n^2

Rozwiązanie.
Wszystkie funkcje dają się ułożyć w ciąg w zgodzie z relacją $o(...)$, a właściwa kolejność to:

$$(1/3)^n,\ 17,\ \log_2\log_2 n,\ \log_2 n,\ 2^{\sqrt{\ln n}},\ \sqrt{n},\ \sqrt{n}\log_2 n,\ n/\log_2 n,\ 2n,\ n^2,\ (n/2)^{\ln n},\ (3/2)^n$$

Od razu widać, jak uszeregować funkcje:

$$(1/3)^n,\ 17,\ \log_2 n,\ \sqrt{n},\ \sqrt{n}\log_2 n,\ n,\ n/\log_2 n,\ 2n,\ n^2,\ (3/2)^n$$

mamy tu bowiem funkcję dążącą do 0 (a więc typu $o(1)$), dalej funkcje stałą (czyli $\Theta(1)$) kolejne 6 funkcji rosnących do nieskończoności postaci $n^\alpha \log^\beta n$ – ustawiamy je według rosnącego wykładnika α, a gdy te są równe – według rosnącego β, oraz na końcu funkcję wykładniczą, która rośnie szybciej niż dowolny wielomian.

Funkcja $\log_2\log_2 n$ rośnie do nieskończoności, czyli jest $\omega(1)$, ale jest też $o(\log_2 n)$, gdyż

$$\lim_{n\to\infty} \log_2\log_2 n/\log_2 n = \lim_{t\to\infty} \log_2 t/t = 0.$$

Zauważmy teraz, że z ciągłości logarytmu dla dowolnych funkcji dodatnich f i g mamy:

$$f=o(g) \Leftrightarrow f/g\to 0 \Leftrightarrow \log_2 f - \log_2 g \to -\infty$$

Zatem możemy rozpoznawać relację $o(\cdot)$ badając różnicę logarytmów funkcji.
Funkcja $(n/2)^{\ln n}$ ma logarytm postaci $\Theta(\log_2^2 n)$, czyli rośnie szybciej, niż funkcje wielomianowe (o logarytmie typu $\Theta(\log_2 n)$) i wolniej od wykładniczych (o logarytmie liniowym). Jest ona jedyną funkcją superwielomianową w ciągu.

Podobnie $2^{\sqrt{\ln n}}$ ma logarytm typu $\Theta(\sqrt{\log_2 n})$ czyli rośnie wolniej niż wielomiany, ale szybciej, niż funkcja $\log_2 n$.

Zadanie 1.3.
Oszacuj wyrażenie:

$$\left(\frac{2}{3}\right)^n + \sum_{i=1}^{n} \sin^2 i + n^2 + \ln\left[\sum_{i=1}^{n} \binom{n}{i}\right]$$

Rozwiązanie.

Przy szacowaniu z góry interesują nas oszacowania jak najdokładniejsze, a więc postaci $O(f)$, gdzie f jest funkcją rosnącą jak najwolniej. Pierwszy składnik dąży do zera, czyli jest $o(1)$, drugi można ograniczyć z góry przez n, gdyż wszystkie wyrażenia pod sumą nie przekraczają 1. Ostatni składnik pod logarytmem przedstawia liczbę wszystkich niepustych podzbiorów zbioru n–elementowego, czyli 2^n–1, co po zlogarytmowaniu jest $\Theta(n)$. Najszybciej rośnie więc składnik trzeci, a zatem on określa tempo wzrostu całego wyrażenia – $\Theta(n^2)$.

Zadanie 1.4.
Oszacuj wyrażenie:

$$\binom{n}{2} + \sum_{i=1}^{n} \log i + \frac{1}{2} n^2 \sin n$$

Rozwiązanie.
Pierwszy składnik wynosi $n(n–1)/2$, czyli jest $\Theta(n^2)$, drugi można oszacować z góry przez $n\log n$, czyli rośnie wolniej niż pierwszy. Jednak ostatni składnik ma moduł typu $O(n^2)$, ale przyjmuje on zarówno wartości dodatnie, jak i ujemne. Chociaż całe wyrażenie jest dodatnie (począwszy od pewnego n), to z uwagi na pierwszy i trzeci składnik możemy oszacować je przez $O(n^2)$, ale nie mamy już podstaw twierdzić, że jest to $\Theta(n^2)$.

Warto przypomnieć tożsamości:

$a = b^{\log_b a}$	$\log_c(ab) = \log_c a + \log_c b$
$\log_b a = 1/\log_a b$	$\log_c(a/b) = \log_c a - \log_c b$
$\log_b a = \log_c a / \log_c b$	$a^{\log_b n} = n^{\log_b a}$

Zadanie 1.5.
Uszereguj funkcje według tempa wzrostu.

1) $2^{\sqrt{n}}, e^{\log_{10} n^3}, n^{3,01}, 2^{n^2}$

Rozwiązanie.
Funkcja $e^{\log_{10} n^3} = (n^3)^{\log_{10} e} = n^{3\log_{10} e}$ jest typu wielomianu o wykładniku mniejszym od 3, ostatnia jest superwykładnicza, zaś pierwsza ma logarytm typu $\Theta(\sqrt{n})$, czyli jest superwielomianowa. Właściwa kolejność, to:

$$e^{\log_{10} n^3}, n^{3,01}, 2^{\sqrt{n}}, 2^{n^2}.$$

2) $n^{1.6}, 1 + \log^3 n, \sqrt{n!}, n^{\ln n}, 2^n$

Rozwiązanie.
Funkcja $n^{\ln n}$ ma logarytm typu $\Theta(\log^2 n)$, więc rośnie ona szybciej niż wielomiany i wolniej niż funkcje wykładnicze – jest superwielomianowa. Funkcję $\sqrt{n!}$ można bezpośrednio oszacować z dołu np. przez:

$$\sqrt{n!} = \sqrt{1 \cdot \ldots \cdot n} \geq \sqrt{\lceil n/2 \rceil \cdot \ldots \cdot n} \geq \left(\frac{n}{2}\right)^{\lfloor n/2 \rfloor/2} \geq \left(\frac{n}{2}\right)^{n/6}$$

lub skorzystać ze znanej (choć trudniejszej do udowodnienia) formuły Strilinga:

$$n! = \Theta(n^{n+0.5} e^{-n}),$$

czyli

$$\sqrt{n!} = \Theta(n^{n/2+0.25} e^{-n/2})$$

co jest nadal superwykładnicze (ma logarytm typu $\Theta(n \log_2 n)$). Pozostaje zwykła funkcja polilogarytmiczna, wielomianowa i wykładnicza, a zatem właściwa kolejność to:

$$1 + \log^3 n, n^{1.6}, n^{\ln n}, 2^n, \sqrt{n!}.$$

3) $n^3 \log_2 n, (\log_2 \log_2 n)^2, 2^n \sqrt{n}, (3n+4)^9$.

Rozwiązanie.
Pierwsza funkcja oraz ostatnia to zwykłe funkcje typu wielomianowego stopni odpowiednio 3 i 9. Ponadto mamy $(\log_2 \log_2 n)^2 < \log_2{}^2 n$, a to ostatnie jest funkcją polilogarytmiczną, czyli właściwa kolejność to:

$$(\log_2 \log_2 n)^2, n^3 \log_2 n, (3n+4)^9, 2^n \sqrt{n}.$$

Zadanie 1.6.
Niech f_i i g_i dla i=1,2 będą takimi funkcjami dodatnimi, że $f_1=O(g_1)$ i $f_2=O(g_2)$. Które z poniższych stwierdzeń muszą być prawdziwe przy takich założeniach?
1) $f_1+f_2=O(g_1+g_2)$
2) $|f_1-f_2|=O(|g_1-g_2|)$
3) $f_1 f_2=O(g_1 g_2)$
4) $f_1/f_2=O(g_1/g_2)$

Rozwiązanie.
1) Jest to prawda, gdyż z założenia muszą istnieć liczby dodatnie n_i oraz c_i, i=1,2, że dla $n>n_i$ mamy $f_i(n) \leq c_i g_i(n)$, co po dodaniu stronami dla $n>max(n_1,n_2)$ daje:

$$f_1(n)+f_2(n) \leq max(c_1,c_2)\ (g_1(n)+g_2(n))$$

2) Oczywiście nie jest to prawda, kontrprzykładem może być np. zestaw funkcji $f_1(n)=n^2$, $f_2(n)=n^2+n$, $g_1(n)=n^3$, $g_2(n)=n^3+1$. Wówczas byłoby $n=O(1)$.
3) Jest to prawda, dowód przebiega podobnie jak w przypadku 1), choć tym razem mnożymy stronami obie nierówności, co dla $n>max(n_1,n_2)$ daje:

$$f_1(n) f_2(n) \leq c_1 c_2\ g_1(n) g_2(n)$$

4) Nie jest to prawda, wystarczy przyjąć np. $f_1(n)=n^2$, $f_2(n)=n$, $g_1(n)= g_2(n)=n^3$. Wówczas byłoby $n=O(1)$.

Zadanie 1.7.
Niech $C_1,...,C_k$ będą liczbami dodatnimi a $f_1,...,f_k$ dowolnymi funkcjami dodatnimi. Wykaż, że

$$C_1f_1+...+C_kf_k = O(f_1+...+f_k).$$

Rozwiązanie.
Wystarczy zauważyć, że dla wszystkich argumentów zachodzi nierówność:

$$C_1f_1+...+C_kf_k \le max(C_1,...,C_k)(f_1+...+f_k).$$

Zadanie 1.8.
Niech f_1, f_2 będą funkcjami dodatnimi oraz f_2 rośnie wolniej, niż f_1 (tj. $f_2=o(f_1)$). Wykaż, że:

$$f_1+f_2 = \Theta(f_1) \text{ oraz } f_1-f_2 = \Theta(f_1).$$

Rozwiązanie.
Oczywiście $f_1-f_2 \le f_1 \le f_1+f_2$, zatem wystarczy pokazać, że $f_1-f_2=\Omega(f_1)$ i $f_1+f_2=O(f_1)$. Ale z założenia $\lim_{n\to\infty} f_2/f_1=0$, czyli począwszy od pewnej wartości argumentu zachodzi $f_2<f_1/2$ i wówczas mamy $f_1-f_2>f_1/2$ oraz $f_1+f_2<3f_1/2$.

> Z tego faktu (choć nie formułowanego wprost) korzystaliśmy już np. w rozwiązaniu zadania 1.3.

Zadanie 1.9.
Niech f i g będą nieujemnymi funkcjami określonymi na R^+ o nieujemnych pochodnych. Czy $f=\Theta(g)$ implikuje oszacowanie $f' = \Theta(g')$.

Rozwiązanie.
Nie jest to prawda. Za kontrprzykład mogą służyć funkcje $f(x)=x$ oraz $g(x)=x+\sin(x)$. Oczywiście $x+\sin(x) \le x+1 \le 2x$ oraz $x/2 \le x-1 \le x+\sin(x)$ dla dostatecznie dużych x, czyli $f=\Theta(g)$. Ich pochodne odpowiednio równe 1 i $1+\cos(x)$ nie spełniają $f'=O(g')$ (a tym bardziej Θ), jako że $1+\cos(x)$ przyjmuje wartości zerowe dla dowolnie dużych liczb rzeczywistych.

2. Równania rekurencyjne

Podczas analizy złożoności algorytmów nader często przydatność swą okazują równania i układy równań rekurencyjnych. Stosują się zarówno do badania procedur używających wywołań rekurencyjnych, jak również do analizy sekwencyjnych kodów nie zawierających takich odwołań. Niekiedy dopiero przetłumaczenie procedury na równoważną postać rekurencyjną pozwala na pełne zaobserwowanie jej zachowania. W rozdziale tym w sposób pełny omówimy metodę rozwiązywania dwóch często pojawiających się typów równań – „dziel i rządź" oraz „jeden krok wstecz". Opiszemy też sposoby pozwalające na oszacowanie rozwiązania równania rekurencyjnego nie wymagające określania tego rozwiązywania w sposób dokładny.

Rozważamy funkcję określoną na pewnym podzbiorze zbioru liczb naturalnych za pomocą rekursji:

$$(1)\qquad T(n)=\begin{cases} c, & \text{gdy} \quad n=1 \\ aT(n/b)+d(n), & \text{gdy} \quad n=b^k, k\geq 1. \end{cases}$$

Jest to równanie rekurencyjne typu „dziel i rządź". Dziedziną T są liczby będące potęgami b, tj. $n=b^k$, zaś rozwiązanie ma postać:

$$(2)\qquad T(n)=ca^k+\sum_{j=0}^{k-1}a^j d(b^{k-j}).$$

Korzystając z tego wzoru w przypadku, gdy d jest funkcją iloczynową (czyli $d(xy)=d(x)d(y)$) otrzymujemy oszacowania:

$$(3)\qquad T(n)=\begin{cases} \Theta(n^{\log_b d(b)}), & \text{gdy} \quad a<d(b) \\ \Theta(n^{\log_b a}\log n), & \text{gdy} \quad a=d(b) \\ \Theta(n^{\log_b a}), & \text{gdy} \quad a>d(b). \end{cases}$$

Można wykazać, że jedynymi monotonicznymi funkcjami iloczynowymi określonymi na zbiorze liczb naturalnych są: funkcja stała równa zero oraz funkcje postaci n^{α}.

Warto zapamiętać wzory obowiązujące dla $q\neq 1$ oraz całkowitego, nieujemnego n.

$$\sum_{i=0}^{n}q^i=\frac{1-q^{n+1}}{1-q},$$

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

Zadanie 2.1.
Rozwiąż równanie:

$$T(n)=\begin{cases}1, & \text{gdy} \quad n=1\\ 8T(n/2)+n^3, & \text{gdy} \quad n=2^k.\end{cases}$$

Rozwiązanie.
Jest to przypadek równania (1) dla c=1, a=8 i b=2 oraz $d(n)=n^3$, czyli funkcja jest określona dla potęg dwójki $n=2^k$ i mamy:

$$T(n)=8^k+\sum_{j=0}^{k-1}8^j(2^{k-j})^3=8^k+\sum_{j=0}^{k-1}8^k=8^k(1+k)=n^3(1+\log_2 n).$$

Zadanie 2.2.
Rozwiąż równanie:

$$T(n)=\begin{cases}1, & \text{gdy} \quad n=1\\ 3T(n/2)+2n\sqrt{n}, & \text{gdy} \quad n=2^k.\end{cases}$$

Rozwiązanie.
Tym razem mamy c=1, a=3, b=2 oraz $d(n)=2n^{3/2}$, czyli funkcja jest określona dla $n=2^k$ i:

$$T(n)=3^k+\sum_{j=0}^{k-1}3^j2(2^{k-j})^{3/2}=3^k+2\sum_{j=0}^{k-1}(\sqrt{8})^k(\frac{3}{\sqrt{8}})^j=3^k+2\sqrt{8^k}\,\frac{\left(\frac{3}{\sqrt{8}}\right)^k-1}{\frac{3}{\sqrt{8}}-1}=$$

$$=3^k+\frac{2}{\frac{3}{\sqrt{8}}-1}(3^k-\sqrt{8}^k).$$

Aby wyrazić powyższą formułę jako funkcję n zauważamy, że $a^{\log_2 n}=n^{\log_2 a}$, czyli ostatecznie:

$$T(n)=n^{\log_2 3}+\frac{2}{\frac{3}{\sqrt{8}}-1}(n^{\log_2 3}-n^{3/2}).$$

Gdybyśmy chcieli jedynie oszacować $T(n)$, na mocy powyższego byłoby $T(n)=\Theta(n^{\log_2 3})$, gdyż $3/2<\log_2 3$. Warto zauważyć, że funkcja $d(n)=2n^{3/2}$ nie jest iloczynowa, a więc nie można bezpośrednio skorzystać z oszacowania (3).

Rozważamy teraz równanie rekurencyjne typu „jeden krok w tył”:

$$T(n)=\begin{cases}c, & \text{gdy} \quad n=0\\ aT(n-1)+d(n), & \text{gdy} \quad n>0.\end{cases}$$

Jest ono określone dla wszystkich całkowitych $n\geq 0$, a rozwiązanie ma postać:

(4) $$T(n)=ca^n+\sum_{j=1}^{n}a^{n-j}d(j).$$

Zadanie 2.3.
Rozwiąż równanie:

$$T(n)=\begin{cases}0, & \text{gdy} \quad n=0\\ 3T(n-1)-2, & \text{gdy} \quad n>0.\end{cases}$$

Rozwiązanie.
Mamy c=0, a=3, $d(n)$= – 2 jest funkcją stałą, czyli:

$$T(n)=\sum_{j=1}^{n}3^{n-j}(-2)=-2(3^{n-1}+3^{n-2}+\Lambda\ +3^{0})=-2\sum_{j=0}^{n-1}3^{j}=-2\frac{3^{n}-1}{3-1}=1-3^{n}.$$

Zadanie 2.4.
Rozwiąż rekursję:

$$T(n)=\begin{cases}1, & \text{gdy} \quad n=0\\ 2T(n-1)+(-1)^{n}, & \text{gdy} \quad n>0.\end{cases}$$

Rozwiązanie.
Mamy teraz c=1, a=2, $d(n)=(-1)^n$, czyli

$$T(n)=2^{n}+\sum_{j=1}^{n}2^{n-j}(-1)^{j}=2^{n}+\sum_{j=1}^{n}2^{n-j}(-1)^{(n-j)+n+2(j-n)}=2^{n}+(-1)^{n}\sum_{j=1}^{n}(-2)^{n-j}=$$

$$=2^{n}+(-1)^{n}\sum_{j=0}^{n-1}(-2)^{j}=2^{n}+(-1)^{n}\frac{(-2)^{n}-1}{-2-1}=\frac{2}{3}2^{n}+\frac{1}{3}(-1)^{n}.$$

Zadanie 2.5*.
Rozwiąż rekursję:

$$T(n)=\begin{cases}1, & \text{gdy} \quad n=0\\ \sqrt{2T(n-1)^{2}+1}, & \text{gdy} \quad n>0.\end{cases}$$

Rozwiązanie.
Nie jest to równanie żadnego ze standardowych typów, ale dokonując podstawienia $U(n)=T(n)^2$ powyższe równanie można przepisać w postaci rekursji na $U(n)$:

$$U(n)=\begin{cases}1, & \text{gdy} \quad n=0\\ 2U(n-1)+1, & \text{gdy} \quad n>0.\end{cases}$$

Jest to równanie opisujące znany problem „wież z Hanoi”, jego rozwiązaniem jest $U(n)=2^{n+1}-1$, co można otrzymać ze wzoru (4), lub wykazać przez bezpośrednią indukcję. Ostatecznie $T(n)=(2^{n+1}-1)^{1/2}$.

Zadanie 2.6.
Rozwiąż rekursję:

$$T(n)=\begin{cases}1, & \text{gdy} \quad n=1\\ nT(n-1)+n!, & \text{gdy} \quad n>1.\end{cases}$$

Rozwiązanie.
Tym razem najprościej jest rozwinąć kilka pierwszych równości definiujących ciąg:

$T(n) = n!+nT(n–1) = n!+n((n–1)T(n–2)+(n–1)!) = 2n!+n(n–1)T(n–2) =$
$= 2n!+n(n–1)((n–2)T(n–3)+(n–2)!) = 3n!+n(n–1)(n–2)T(n–3) =$
.....
$= kn! + n(n–1)...(n–k+1)T(n–k) =$
....
$= (n–1)n!+n!T(1) = nn!$

Równania typu (1) określone są jedynie na pewnym podzbiorze zbioru liczb naturalnych (potęgi b) i wówczas potrafimy je rozwiązać w sposób dokładny. W praktyce spotykamy się zwykle z ich modyfikacjami, określonymi na całym zbiorze liczb naturalnych i pragniemy zamiast rozwiązań uzyskać jak najdokładniejsze oszacowanie rozwiązania.

Zadanie 2.7.
Oszacuj rozwiązanie równania:

$$T(n) = \begin{cases} 1, & \text{gdy} \quad n = 1 \\ 3T(\lfloor n/2 \rfloor) + 2n\sqrt{n}, & \text{gdy} \quad n > 1. \end{cases}$$

Rozwiązanie.
To równanie jest określone na całym zbiorze liczb naturalnych, zaś dla potęg liczby 2 ma ono postać:

$$T(n) = \begin{cases} 1, & \text{gdy} \quad n = 1 \\ 3T(n/2) + 2n\sqrt{n}, & \text{gdy} \quad n = 2^k. \end{cases}$$

Jego rozwiązanie jest, jak już wiemy $\Theta(n^{\log_2 3})$ – ale tylko dla potęg 2. Czy na całym zbiorze liczb naturalnych zajdzie oszacowanie $\Theta(n^{\log_2 3})$? Wykażemy, że tak jest. Konsekwencją powyższego oszacowania dla potęg liczby 2 jest istnienie stałych c i d, takich że począwszy od pewnego miejsca zachodzi:

$$c(2^k)^{\log_2 3} \le T(2^k) \le d(2^k)^{\log_2 3}$$

Sprawdźmy indukcyjnie, że $T(n)$ jest funkcją niemalejącą. Wystarczy wykazać, że $\forall_n T(n) \le T(n+1)$. Nierówność $T(1) < T(2)$ – sprawdzamy bezpośrednio, zaś dalej ma zastosowanie formuła rekurencyjna, czyli:

$$T(n) = 3T(\lfloor n/2 \rfloor) + 2n\sqrt{n} \le 3T(\lfloor (n+1)/2 \rfloor) + 2(n+1)\sqrt{n+1} = T(n+1)$$

– środkowa nierówność wynika z założenia indukcyjnego.
Skoro $T(n)$ jest niemalejąca, to weźmy dowolne n i liczbę naturalną k, taką by było:

$$n/2 \le 2^k \le n \le 2^{k+1} \le 2n$$

(każdą liczbę naturalną można umieścić pomiędzy dwiema kolejnymi potęgami dwójki).
Jeżeli n będzie dostatecznie duże, to mamy:

$$c(n/2)^{\log_2 3} \le c(2^k)^{\log_2 3} \le T(2^k) \le T(n) \le T(2^{k+1}) \le d(2^{k+1})^{\log_2 3} \le d(2n)^{\log_2 3}$$

A zatem otrzymujemy $T(n) = \Theta(n^{\log_2 3})$ w całej dziedzinie.

Zadanie 2.8*.
Oszacuj rozwiązanie równania:

$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log_2 n$, $T(1) = 0$.

Rozwiązanie.

Podobnie, jak w poprzednim punkcie przez natychmiastową indukcję widzimy, że $T(n)$ jest funkcją niemalejącą. Ponadto dla liczb postaci $n = 2^{2^k}$ (a więc wtedy $k=\log_2 \log_2 n$) równanie przybiera postać:

$$T(2^{2^k}) = 2T(2^{2^{k-1}}) + 2^k.$$

Po zastosowaniu podstawienia $U(k) = T(2^{2^k})$ otrzymujemy równanie na $U(k)$:

$U(k) = 2U(k-1)+2^k$, $U(0)=T(2)$.

Stosując wzór (4) widzimy, że $U(k) = T(2)2^k + \sum_{j=1}^{k} 2^{k-j}2^j = (k+T(2))2^k = \Theta(k2^k)$, czyli $T(n) = \Theta((\log_2 n)(\log_2 \log_2 n))$ dla $n = 2^{2^k}$. Dla tych n istnieją stałe c i d, takie że począwszy od pewnego miejsca zachodzi:

$$c(\log_2 n)(\log_2 \log_2 n) \le T(n) \le d(\log_2 n)(\log_2 \log_2 n).$$

Wykażemy, że jest to poprawne oszacowanie na całym zbiorze liczb naturalnych. Mianowicie, dla dostatecznie dużych n można znaleźć taką liczbę naturalną k, że: $\sqrt{n} \le 2^{2^k} \le n \le 2^{2^{k+1}} \le n^2$ i wówczas otrzymamy:

$$\frac{c}{2}[(\log_2 n)((\log_2 \log_2 n)-1)] \le c(\log_2 \sqrt{n})(\log_2 \log_2(\sqrt{n})) \le c(\log_2 2^{2^k})(\log_2 \log_2(2^{2^k}))$$
$$\le T(2^{2^k}) \le T(n) \le T(2^{2^{k+1}}) \le$$
$$d(\log_2 2^{2^{k+1}})(\log_2 \log_2(2^{2^{k+1}})) \le d(\log_2 n^2)(\log_2 \log_2(n^2)) \le 2d[(\log_2 n)((\log_2 \log_2 n)+1)]$$

Zatem ostatecznie $T(n)= \Theta((\log_2 n)(\log_2 \log_2 n))$.

Podsumujmy powyższe przykłady. Jeżeli rozwiązanie równania jest funkcją niemalejącą, to często jego oszacowanie jedynie na pewnym podciągu zbioru liczb naturalnych (np. na takim, na którym można je łatwo rozwiązać) pozwala na sformułowanie globalnego oszacowania asymptotycznego.

Należy zwrócić uwagę na fakt, że wcześniejsze sprawdzenie monotoniczności rozwiązania stanowi istotny warunek poprawności procedury szacowania. Dla przykładu rozważmy równanie:

$$T(1) = 1,$$
$$T(n) = T(\lfloor n/2 \rfloor) + (n - 2\lfloor n/2 \rfloor), \text{ dla } n>1.$$

Łatwo zauważyć, że $T(n)$ jest liczbą jedynek w zapisie bitowym liczby n (składnik w nawiasie definicji rekurencyjnej jest resztą z dzielenia n przez 2). Zatem $T(n)=O(\log_2 n)$, choć oczywiście nie zachodzi $T(n)=\Theta(\log_2 n)$. Jednak po ograniczeniu się do potęg dwójki równanie redukuje się do postaci $T(n)=T(\lfloor n/2 \rfloor)$, czyli $T(n)=1$. Niefrasobliwe zastosowanie opisanej procedury sugerowałoby, że $T(n)=O(1)$. Jest to oczywiście nieprawda – zachodzi bowiem $T(2^k-1)=k$.

Sama monotoniczność może i tak być niewystarczająca do przeprowadzenia oszacowania powyższą metodą. Rozważmy teraz równanie:

$$T(1) = 2,$$
$$T(n) = T(\lceil n/2 \rceil)^2, \text{ dla } n>1.$$

Łatwo zauważyć, że $T(2^0)=2^1$ i dla liczb postaci $n=2^k$ widzimy, że zachodzi równość $T(n)=2^n$, bo wtedy przez natychmiastową indukcję $T(n)=T(n/2)^2=(2^{n/2})^2=2^n$. Sugerując się rozwiązaniem na potęgach dwójki moglibyśmy pokusić się o oszacowanie $T(n) = O(2^n)$. Jest to znów nieprawda. Przyjrzyjmy się bowiem argumentom postaci $n=2^k+1$. Zauważmy, najpierw że $\lceil (2^k+1)/2 \rceil = 2^{k-1}+1$. Teraz przy $k=0$ jest $n=2$ i $T(2)=4$, a dalej mamy $T(n)=4^{n-1}$ gdyż tutaj przez indukcję $T(n)=T(\lceil n/2 \rceil)^2=T((n+1)/2)^2=(4^{(n/2+1/2-1)})^2=4^{(n-1)}$. Ciągu 4^{n-1} w żaden sposób nie można ograniczyć przez $O(2^n)$, a takie właśnie wartości przyjmuje funkcja T w nieskończenie wielu punktach. Błąd wynika z próby oszacowania T jego rozwiązaniem na potęgach dwójki, pomimo że rozwiązanie to było funkcją rosnącą „zbyt szybko" (np. niewielomianowo).

Zadanie 2.9.
Dane są funkcje f i g określone na zbiorze liczb naturalnych, wyliczane za pomocą poniższego algorytmu:

```
function f(n:integer):integer;
begin
        if n = 1 then return 1
        else return f(n–1)+g(n);
end;

function g(n:integer):integer;
begin
        if n = 1 then return 1
        else return f(n–1)+g(⌊n/2⌋);
end;
```

Określ funkcję f jako element jednej z następujących kategorii: liniowa, polilogarytmiczna, wielomianowa, superwielomianowa, wykładnicza, superwykładnicza.

Rozwiązanie.
Łatwo zauważyć, że powyższa procedura jest poprawna – nie zapętla się pozwalając na obliczenie wartości $f(n)$ dla dowolnego naturalnego n. Rzeczywiście, analizując wywołania rekurencyjne do głębokości 2 dla argumentu $n>1$ widzimy, że obie funkcje odwołują się do swych wartości jedynie w punktach mniejszych od n. Podobnie przez natychmiastową indukcję widzimy, że obie funkcje są niemalejące, tj. dla dowolnego n zachodzi $f(n)\le f(n+1)$ i $g(n)\le g(n+1)$, obie też są dodatnie. Zacznijmy od oszacowania f z dołu. Widzimy, że dla $n>1$ mamy:

$$f(n) = f(n-1)+g(n) = f(n-1)+f(n-1)+g(\lfloor n/2 \rfloor) \ge 2f(n-1)$$

i dalej podobnie:

$$f(n)\ge 2f(n-1)\ge 2^2 f(n-2)\ge 2^3 f(n-3)\ge \ldots \ge 2^{n-1} f(1),$$

więc $f(n)=\Omega(2^n)$, czyli f rośnie nie wolniej, niż funkcja wykładnicza 2^n. Zatem f jest wykładnicza lub superwykładnicza. Aby wykazać, iż f jest wykładnicza musimy oszacować f z góry za pomocą pewnej funkcji wykładniczej. W tym celu zauważmy, że

$$f(n)=f(n-1)+g(n)\ge g(n)$$

oraz $g(n)$ jest niemalejąca, czyli

$$f(n) = f(n-1)+g(n) = f(n-1)+f(n-1)+g(\lfloor n/2 \rfloor) \le f(n-1)+f(n-1)+g(n-1) \le 3f(n-1)$$

i kontynuując jak wyżej dostaniemy:

$$f(n) \le 3f(n-1) \le 3^2 f(n-2) \le 3^3 f(n-3) \le \ldots \le 3^{n-1} f(1), \text{ zatem } f(n)=O(3^n).$$

Udało nam się więc umieścić f pod względem tempa wzrostu pomiędzy dwiema funkcjami wykładniczymi, co pozwala stwierdzić, że jest to funkcja typu wykładniczego. Nie jest tutaj potrzebne dokładnie oszacowanie f w sensie symbolu Θ.

3. Złożoność obliczeniowa prostych procedur

Zamieszczone tu zadania prezentują metody obliczania złożoności prostych procedur iteracyjnych, nie zawierających wywołań podprogramów. Omawiane są zasady, jakimi należy kierować się przy wyznaczaniu zbioru instrukcji podstawowych oraz techniki szacowania liczby kroków dla programów zawierających pętle. Krótko zaprezentowano też metodę dowodzenia poprawności procedury poprzez indukcję względem niezmienników pętli.

Zadanie 3.1.
Podaj algorytm obliczający wartość sumy liczb wpisanych w „dolnej połówce" tablicy,

1	2	3	...	n–1	n
1	2	3	...	n–1	n
...					...
...					...
1	2	3	...	n–1	n
1	2	3	...	n–1	n

czyli zwracający wartość 1+(1+2)+(1+2+3)+...+(1+2+...+n–1)+(1+2+...+n),
wykonujący:
1) $\Theta(n^2)$ dodawań,
2) $\Theta(n)$ dodawań,
3) $O(1)$ dodawań.

Rozwiązanie.
1) Zastosujemy metodę zliczania bezpośrednio z formuły w zadaniu:

```
function suma(n:integer):integer;
var
        i,j,s:integer;
begin
        s := 0;
        for i := 1 to n do
                for j := 1 to i do s := s+j;
        return s;
end;
```

Wówczas liczba wykonanych dodawań wyniesie $\sum_{i=1}^{n} i = n(n+1)/2 = \Theta(n^2)$.

2) Ponieważ dodawane elementy w każdym wierszu tablicy stanowią sumę kolejnych wyrazów ciągu arytmetycznego – możemy skorzystać z gotowego wzoru na tą sumę:

```
function suma(n:integer):integer;
var
        i,s:integer;
begin
        s := 0;
```

```
        for i := 1 to n do s := s+i*(i+1)/2;
        return s;
end;
```

Wówczas liczba dodawań wyniesie: $\sum_{i=1}^{n} 2 = 2n = \Theta(n)$.

3) Bezpośrednio wyprowadzamy wzór na wartość wyrażenia w funkcji n:

$$\sum_{i=1}^{n}\sum_{j=1}^{i} j = \sum_{i=1}^{n} i(i+1)/2 = \frac{1}{2}\sum_{i=1}^{n}(i^2+i) = \frac{1}{2}\left[\frac{1}{3}n(n+1/2)(n+1)+\frac{1}{2}n(n+1)\right] = n(n+1)(n+2)/6.$$

Mamy zatem algorytm obliczający naszą sumę, wykonujący 2 dodawania:

```
function suma(n:integer):integer;
var
        s:integer;
begin
        s := n*(n+1)*(n+2)/6;
        return s;
end;
```

Zadanie 3.2.
Policz operacje arytmetyczne wykonywane podczas mnożenia macierzy o wymiarach $n \times m$ i $m \times s$ za pomocą algorytmu klasycznego (czyli z definicji).

Rozwiązanie.
Procedura wykonująca mnożenie ma postać:

```
function pomnoz(A: array [1..n,1..m] of real, B: array [1..m,1..s] of real)
                                :array [1..n,1..s] of real;
var
        i,j,k:integer;
        C: array [1..n,1..s] of real;
begin
        for i := 1 to n do
                for j := 1 to s do begin
                        C[i,j] := 0;
                        for k := 1 to m do C[i,j] := C[i,j] +A[i,k]*B[k,j];
                end;
        return C;
end;
```

Operacje arytmetyczne znajdują się w najbardziej zagnieżdżonej pętli, mamy zaś trzy pętle o zakresach zmienności odpowiednio od 1 do n, s oraz m. Dlatego liczba zarówno dodawań jak i mnożeń jest iloczynem nms.

> Liczbę operacji wykonywanych przez program wyrażoną w funkcji pewnych parametrów danych będziemy też nazywać *czasem działania, liczbą kroków* lub *czasem wykonania* programu.

Zadanie 3.3.
Oszacuj czas działania poniższej procedury:

```
procedure zagadka(n: integer);
begin
    for i := 1 to ⌊√n⌋ do begin
        j := 1;
        while j < n^2 do j := j + j;
    end
end;
```

Rozwiązanie.
Generalnym założeniem, jakie tutaj przyjmujemy jest wykonywanie wszystkich operacji arytmetycznych na liczbach (dodawanie, mnożenie, porównanie, ...) w czasie $O(1)$, a więc w jednej instrukcji, której długość wykonania nie zależy od rozmiaru dodawanych/odejmowanych liczb.

> Analizę programu rozpoczynamy od znalezienia *instrukcji podstawowej*, czyli wykonywanej najczęściej.

Spójrzmy raz jeszcze na program:

```
procedure zagadka(n: integer);
begin
1.  for i := 1 to ⌊√n⌋ do begin
2.          j := 1;
3.          while j < n^2 do
4.                  j := j + j;
5.  end
end;
```

Dobrym kandydatem na instrukcję podstawową jest dodawanie w linii 4, które znajduje się w najbardziej zagłębionej pętli. Policzmy ilość jej wykonań w zależności od danych programu, czyli zmiennej *n*. Najpierw rozważmy sytuację pomiędzy kolejnym wejściem do linii 2 i dojściem do 5, czyli przy ustalonych wartościach zmiennych *n* i *i*. Liczba obiegów pętli 3 nie zależy od *i*. W każdym kolejnym obiegu *j* podwaja się począwszy od wartości 1, przyjmuje więc wartości będące kolejnymi potęgami 2. Proces trwa tak długo, aż *j* stanie się najmniejszą potęgą 2 większą lub równą n^2. Jest to oczywiście $2^{\lceil \log_2 n^2 \rceil}$, czyli liczba obiegów pętli 3 w pojedynczym przebiegu od linii 2 do 5 wynosi $\lceil \log_2 n^2 \rceil = \lceil 2\log_2 n \rceil$. Łączna liczba wykonań instrukcji z linii 4 w programie to $\lfloor \sqrt{n} \rfloor \lceil 2\log_2 n \rceil$ – przemnożyliśmy powyższe przez ilość obiegów pętli 1. Widzimy, że nasz wybór instrukcji najczęściej wykonywanej był trafny – operacje z linii 1 i 2 wykonują się rzadziej, bo tylko $\lfloor \sqrt{n} \rfloor$ razy. Zatem zasadniczy przyczynek do czasu pracy procedury wnosi liczba wykonań naszej instrukcji podstawowej. Ostatecznie procedura zakończy pracę w czasie $\Theta(\sqrt{n} \log_2 n)$.

Jak pamiętamy, zachodzi wzór $\log_b n = \log_c n / \log_c b$. Zmiana podstawy logarytmu z b na c wiąże się więc z koniecznością przemnożenia wyrażenia przez $1/\log_c b$. Jednak mnożenie przez stałą nie zmienia tempa wzrostu funkcji (w sensie symboli oszacowań asymptotycznych), dlatego w sytuacjach, gdy logarytm (lub jakaś jego potęga) występuje jako czynnik w wyrażeniu szacującym np. badany czas działania – możemy bez utraty informacji opuścić nieistotną w tym miejscu podstawę logarytmu. Rozwiązanie poprzedniego zadania możemy więc w skrócie zapisać jako $\Theta(\sqrt{n} \log n)$. Oczywiście uwaga ta jest słuszna w odniesieniu do czynników np. o charakterze polilogarytmicznym domnażanych do całości szacowanego wyrażenia, nie zaś do każdego wystąpienia logarytmów w formule (np. w argumencie innej funkcji wykładniczej). Przykładowo, z dwóch funkcji superwielomianowych $n^{\log_{10} n}$ i $n^{\log_2 n}$ pierwsza rośnie wolniej, niż druga (podstawa logarytmu ma tu znaczenie).

Zadanie 3.4.
Oszacuj czas działania poniższej procedury:

```
procedure zagadka(n: integer);
begin
    for i := 1 to n^2 do begin
        k := 1; l := 1;
        while l < n do begin k := k + 2; l := l + k; end
    end
end;
```

Rozwiązanie.
Procedura jest zbliżona do tej z zadania poprzedniego – mamy tu dwie zagnieżdżone pętle, przy czym liczba wykonań wewnętrznej nie zależy od zmiennej sterującej pętlą zewnętrzną. Analiza będzie podobna do wykonanej wyżej. Rozważamy fragmenty kodu rozbitego na linie:

```
procedure zagadka(n: integer);
begin
1.  for i := 1 to n^2 do begin
2.          k := 1; l := 1;
3.          while l < n do
4.              begin k := k + 2; l := l + k; end
5.  end
end;
```

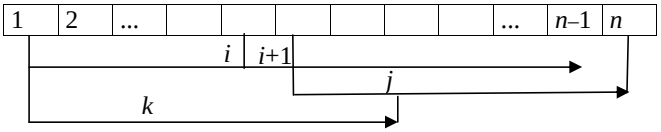
Zauważamy, że najczęściej wykonuje się instrukcja złożona, zapisana w linii 4. Ją też wybieramy jako instrukcję podstawową i zliczmy liczbę jej wywołań. Rozważmy najpierw fragment działania kodu pomiędzy liniami 2 i 5, czyli przy ustalonej wartości zmiennych i i n. Instrukcja elementarna wykona się tyle razy, ile obiegów wykona pętla **while** z linii 3, czyli dopóki nie zajdzie $l \geq n$. Zauważamy przy tym, że po x–tym przejściu linii 4 w pętli 3 zmienna k wynosi $2x+1$ (k przyjmuje wartości kolejnych liczb nieparzystych), zaś l będzie równe $1+3+5+...+(2x+1)=x(x+1)+x+1=(x+1)^2$. Zatem ilość wykonań pętli 3 będzie równa najmniejszemu takiemu x, dla którego $(x+1)^2 \geq n$, czyli $x = \lceil \sqrt{n} \rceil - 1$. Łączna liczba wykonań instrukcji podstawowej to $n^2(\lceil \sqrt{n} \rceil - 1)$, i żadna inna instrukcja w programie nie wykonuje się częściej, czyli czas działania procedury wynosi $\Theta(n^{5/2})$.

Zadanie 3.5.
Oblicz czas działania poniższej procedury:

```
procedure zagadka(n: integer);
begin
    for i := 1 to n – 1 do
        for j := i +1 to n do
            for k := 1 to j do;
 end;
```

Rozwiązanie.

Zależności pomiędzy poszczególnymi zmiennymi sterującymi pętli oraz ich zmiany w trakcie działania programu przedstawia poniższy schemat:



Spójrzmy teraz na kod:

```
procedure zagadka(n: integer);
begin
1.  for i := 1 to n – 1 do
2.      for j := i +1 to n do
3.          for k := 1 to j do;
end;
```

Instrukcją elementarną jest tutaj "średnik" kończący linię 3, a dokładniej operacje uaktualnienia zmiennej sterującej pętlą **for** z linii 3 i sprawdzenia warunku kontynuacji tej pętli, wykonujące się wraz z każdym jej obiegiem, nawet jeżeli sama pętla formalnie jest pusta. Pętla ta jest najgłębiej zagnieżdżona w programie, więc jej operacje wykonują się najczęściej. Obliczmy ilość wykonań "średnika".

- Od wejścia do opuszczenia linii 3, a więc przy ustalonych wartościach *n*, *i* oraz *j* instrukcje pętli 3 wykonują się *j* razy,
- Od wejścia do linii 2 do wyjścia z 3 (pojedynczy obieg pętli 1), a więc przy ustalonych wartościach zmiennych *n* oraz *i* instrukcja elementarna wykona się:

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

Wreszcie łączna liczba wykonań instrukcji elementarnej w programie, to:

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{j} 1 = \sum_{i=1}^{n-1}\left[\frac{1}{2}n^2 - \frac{1}{2}i^2 + \frac{1}{2}(n-i)\right] = \frac{1}{2}\sum_{i=1}^{n-1} n^2 - \frac{1}{2}\sum_{i=1}^{n-1} i^2 + \frac{1}{2}\sum_{i=1}^{n-1} i =$$

$$= \frac{1}{2}n^2(n-1) - \frac{1}{2}\cdot\frac{1}{3}(n-1)(n-\frac{1}{2})n + \frac{1}{2}\cdot\frac{1}{2}(n-1)n = \frac{1}{3}n^3 - \frac{1}{3}n.$$

Inne instrukcje w programie wykonują się mniejszą liczbę razy: liczbę wykonań operacji z linii 1 można oszacować z góry przez *n* (ilość obiegów tej pętli), zaś te z linii 2 wykonują się

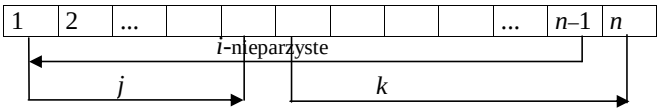
co najwyżej tyle, ile razy wykona się pętla 1 razy maksymalna ilość wykonań pętli 2 – można to oszacować z góry przez n^2. A zatem nasz wybór instrukcji elementarnej był poprawny i czas działania wynosi $\Theta(n^3)$.

Jak widać, dokładne określenie liczby wykonań instrukcji elementarnej może być rachunkowo uciążliwe, niekiedy jest wręcz niewykonalne. Spróbujmy teraz uzyskać oszacowanie $\Theta(n^3)$ na liczbę wykonań instrukcji elementarnej w linii 3, nie obliczając w sposób dokładny tej wartości. Oszacowanie górne można uzyskać łatwo: po każdym wejściu do pętli 1, 2 lub 3 wykonuje się ona nie więcej niż n razy (co najwyżej tyle wynosi jej górna granica), zatem instrukcja najbardziej zagnieżdżona wykona się nie więcej niż n^3 razy. Mamy więc ograniczenie górne na czas pracy postaci $O(n^3)$. Pozostaje nam znaleźć ograniczenie dolne typu $\Omega(n^3)$. Rozważmy więc liczbę wykonań instrukcji elementarnej w tym fragmencie wykonania programu, kiedy zmienna i zawiera się pomiędzy $n/3$ a $2n/3$. Takich wartości zmiennej i jest co najmniej $n/3–1$. Dla każdej z nich zmienna j przybiera wszystkie wartości całkowite z przedziału $[2n/3+1,n]$ – jest ich co najmniej $n/3–2$. Wreszcie dla ustalonych i i j jak wyżej zmienna k przybiera wszystkie wartości całkowite z przedziału $[1,2n/3]$. Zatem w tym tylko fragmencie wykonania instrukcja elementarna wykonuje się co najmniej $(n/3–2)^2(2n/3–2) = \Theta(n^3)$ razy, co kończy dowód.

Zadanie 3.6.
Oszacuj czas działania poniższej procedury:

```
procedure zagadka(n: integer);
begin
    for i := n –1 downto 1 do
        if i mod 2 = 1 then begin
            for j := 1 to i do;
            for k := i +1 to n do
                x := x + 1;
        end;
end;
```

Rozwiązanie.
Obiegi pętli w programie wyglądają następująco:



Tym razem w kodzie występują dwie instrukcje wykonywane szczególnie często. W miarę obiegów pętli zmiennej i, gdy i jest duże, czyli bliskie n – najczęściej wykonuje się "średnik" pętli zmiennej j. Gdy i jest małe – częściej wykonuje się dodawanie w pętli zmiennej k. Zatem obie te operacje potraktujemy jak elementarne i oszacujemy łączną liczbę ich wykonań. Jest oczywiste, że pętla z j wykona się nie więcej niż n^2 razy – górne ograniczenie jej zmiennej sterującej to co najwyżej n, sama jest zaś zamknięta w pętli sterowanej i wykonującej się również co najwyżej n razy. To samo dotyczy liczby wykonań pętli z k, zatem czas działania można oszacować przez $O(n^2)$. Wykażemy teraz oszacowanie $\Omega(n^2)$. Obie instrukcje podstawowe wykonają się w każdym obiegu pętli sterowanej i dla nieparzystego i z przedziału $[1,...,n–1]$ – jest co najmniej $n/2–1$ takich liczb. Dla każdej z nich pierwsza

instrukcja elementarna wykona się i razy, a druga $n–i$. Zatem łączna liczba wykonań instrukcji elementarnych jest nie mniejsza od $(n/2–1)n$. Czas działania procedury to $\Theta(n^2)$.

Wybór zbioru instrukcji podstawowych jest dość swobodny. Nic nie stoi na przeszkodzie, by np. w powyższym przykładzie jako instrukcje podstawowe zliczać dodatkowo sprawdzanie warunku **if** ze zmienną i poza powyższymi operacjami. Zwiększy to zliczaną liczbę operacji o składnik $n–1$, ale nie zmieni faktu, że całe wyrażenie będzie $\Theta(n^2)$ – i takie też będzie oszacowanie czasu wykonania. Dlatego zbiór instrukcji podstawowych powinien być taki, by liczba wykonań wszystkich instrukcji spoza tego zbioru rosła asymptotycznie wolniej w funkcji przekazywanych do programu danych (symbol $o(...)$) od liczby wykonań operacji podstawowych. Dzięki temu dla dużych danych ich czas wykonania będzie pomijalnie mały, nie zmieni asymptotycznego oszacowania.

Złożoność obliczeniowa algorytmu to pojęcie ściśle związane z jego czasem działania. Złożoność stanowi bowiem maksymalny czas pracy kodu po wszystkich możliwych danych o rozmiarze N, wyrażany właśnie jako funkcja tego rozmiaru N. Sam czas działania często określa się (lub szacuje) używając również innych parametrów, niż tylko ich rozmiar danych. W przykładach powyższych jedynymi danymi przekazywanymi do procedur były pojedyncze liczby i ich właśnie używaliśmy do oszacowania czasu. Zazwyczaj mamy do czynienia z bardziej skomplikowanymi sytuacjami np. czas potrzebny do wyszukania według algorytmu Dijkstry najkrótszej drogi w grafie n–wierzchołkowym i m–krawędziowym wynosi $O(n^2)$, a wyszukanie największego skojarzenia wymaga czasu $O(m\sqrt{n})$. Wówczas możemy nie podając wzoru stwierdzić, że algorytm Dijkstry wymaga czasu "kwadratowego względem n", podobnie jak procedura *zagadka* z ostatniego zadania. Kiedy jednak chcemy użyć sformułowania „złożoność obliczeniowa algorytmu wynosi ...", lub słownie określić jego klasę złożoności (np. liniowa, kwadratowa, superwykładnicza), wówczas powinniśmy wyrazić czas działania w funkcji rozmiaru danych – i tu różnego rodzaju struktury danych posiadają rozmaite rozmiary. Sensownym jest np. przyjęcie, iż tablicę n–elementową jednakowych rekordów potraktujemy jak strukturę rozmiaru $\Theta(n)$. Wówczas sortowanie bąbelkowe wymagające czasu $O(n^2)$ ma "kwadratową złożoność obliczeniową", jak również po prostu "jest kwadratowe". Dla grafów najczęstszymi strukturami danych są: macierz sąsiedztwa o rozmiarze $\Theta(n^2)$ oraz lista sąsiadów o rozmiarze $\Theta(n+m)$. Algorytm Dijkstry w przypadku podania doń danych w tej pierwszej strukturze będzie "liniowy", gdyż liczba operacji $O(n^2)$ jest proporcjonalna do rozmiaru macierzy sąsiedztwa, zaś będzie on "kwadratowy" dla listy sąsiadów, bo tejże liczby operacji nie można oszacować rozmiarem danych $n+m$ w potędze niższej, niż druga (dla rzadkich grafów, np. dla drzew mamy $m=n–1$). Dla liczb zakłada się, że rozmiarem danych potrzebnych do zapisania n jest ilość bitów jej kodu binarnego, czyli $\lfloor \log_2 n \rfloor+1$, co z dokładnością do 1 jest równe $N=\log_2 n$. A zatem procedura o czasie działania $O(n)$ pobierająca na wejściu jedynie liczbę będzie „wykładnicza", bo $n \approx 2^N$. Podobnie wszystkie procedury zagadka z tej serii zadań miały wielomianowy względem n czas działania, a zatem wykładniczą złożoność obliczeniową. Należy pamiętać, że zasada powyższa nie wprowadza żadnych zmian w szacowaniu liczby operacji wykonywanych przez procedurę, opisuje ona jedynie pewną umowę terminologiczną co do sposobu słownego określania wcześniej wyliczonych formuł przy opisie złożoności algorytmu.

Zadanie 3.7.
Oblicz i odpowiednio nazwij złożoność obliczeniową poniższej funkcji sprawdzającej, czy dana liczba naturalna jest pierwsza. Zakładamy, że operacje arytmetyczne wykonują się w czasie $O(1)$.

```
function pierwsza(n:integer):boolean;
var i:integer;
begin
        if n = 1 then return false;
        for i := 2 to ⌊√n⌋ do
                if n mod i = 0 then return false;
        return true;
end;
```

Rozwiązanie.
Oczywiście operacją podstawową jest instrukcja **if** w pętli **for** i czas działania wynosi $O(\sqrt{n})$. Jednak dla rozmiaru danych N, rozumianego jako liczba bitów potrzebnych do zapisania danej n mamy $n \approx 2^N$, czyli $\sqrt{n} \approx (\sqrt{2})^N$. Złożoność procedury jest wykładnicza.

Przy określaniu złożoności obliczeniowej kluczowe znaczenie ma rozmiar danych wejściowych procedury. Gdyby funkcja o tym samym kodzie pobierała na wejściu np. tablicę n-elementową tj. miała sygnaturę

function *pierwsza*(*n*:**integer, tab:array[**1…*n***] of integer**):**boolean;**

wówczas, mimo że jej czas działania pozostawałby niezmieniony ($O(\sqrt{n})$), rozmiar danych N byłby liniowy względem $n \approx N$, zatem złożoność obliczeniową $\sqrt{n} = \Theta(\sqrt{N})$ musielibyśmy tym razem określić jako subliniową.

W obrębie tego skryptu przyjmujemy założenie upraszczające, mówiące o wykonywalności operacji arytmetycznych na liczbach całkowitych w jednym kroku. Ma to sens w przypadku, gdy liczby te nie przekraczają zakresów danych całkowitych wbudowanych w architekturę współczesnych komputerów, gdzie operacje takie są wykonywane w pojedynczych instrukcjach maszynowych. Wtedy formuły określające czas pracy, wyliczone przy powyższym założeniu, zgadzają się z faktycznymi czasami wykonywania programów. Należy jednak pamiętać, że w sytuacji ogólnej, gdy długość bitowa argumentów operacji nie pozwala na umieszczenie ich w rejestrach maszyny – skazani jesteśmy na samodzielne implementowanie operacji arytmetycznych np. za pomocą algorytmów zbliżonych do klasycznego dodawania/mnożenia pisemnego. Wówczas dodawanie i odejmowanie stają się procedurami o złożoności liniowej, zaś mnożenie i dzielenie całkowite są kwadratowe. Operacje arytmetyczne na długich liczbach (rzędu kilkuset i więcej bitów) spotyka się w kryptografii, kiedy to dane do zakodowania traktujemy jako pojedynczą liczbę. Najpopularniejszy obecnie system RSA kodowania z kluczem jawnym opiera się właśnie na arytmetyce wielkich liczb. Do tematu tego powrócimy w kolejnych rozdziałach.

Zadanie 3.8.
Co wylicza poniższa funkcja określona na liczbach naturalnych? Podaj jej złożoność.

```
function X(n:integer):integer;
var i,j:integer;
begin
        i := 0; j := 1;
        while j < n do begin
                j := 2*j; i := i+1;
        end;
        return i;
end;
```

Rozwiązanie.
Przez natychmiastową indukcję względem obiegów pętli widzimy, że j są kolejnymi potęgami dwójki zaś i to wykładniki tych potęg. Pętla kończy działanie za pierwszym razem, gdy $j=2^i \geq n$ i wówczas zwracane jest i. Zatem ostatecznie funkcja wylicza $i=\lceil \log_2 n \rceil$, co z dokładnością do 1 jest równe liczbie bitów potrzebnych do zapisu n. Za operację elementarną możemy uznać wnętrze pętli **while**, a wtedy liczba jej obiegów jest dokładnie równa zwracanej przez funkcję wartości. Czas działania wynosi więc $\Theta(\log n)$, czyli złożoność jest liniowa.

Zadanie 3.9.
Niech f: $N \rightarrow R$ będzie funkcją malejącą i zmieniającą znak. Następujący fragment programu

```
i := 1;
while f(i) ≥ 0 do i := i + 1;
n := i – 1;
```

oblicza największą liczbę naturalną n, dla której $f(n) \geq 0$, lecz jego czas działania wynosi $\Theta(n)$. Napisz szybszą procedurę.

Rozwiązanie.
Wiadomo, że procedura szukania binarnego:

```
while b – a > 1 do begin
        c := (a+b) div 2;
        if f(c) ≥ 0 then a := c else b := c;
end;
n := a;
```

uruchomiona dla a i b spełniających $f(a)\geq 0$, $f(b)<0$ zwróci poszukiwaną liczbę n po co najwyżej $\lceil \log_2(b-a) \rceil$ obiegach pętli. Można by więc uruchomić ją dla $a=1$, o ile wcześniej znajdziemy jakieś $b \geq n$. Do tego może służyć procedura:

```
b := 1;
while f(b) ≥ 0 do b := 2*b;
```

Zmienna b przybiera wartości równe kolejnym potęgom dwójki (podwaja się z każdym obiegiem pętli) aż do przekroczenia progu n. A zatem po wykonaniu tego kodu mamy $n \leq b \leq 2n$ i pętla powyższa zostanie wykonana $\lceil \log_2 n \rceil$ razy. Wtedy możemy wrócić do szukania

binarnego, które na mocy powyższego zajmie nam czas $O(\log_2(2n-1))$. Oto pełny kod algorytmu:

```
a := 1; b := 1;
while f(b) ≥ 0 do b := 2*b;

while b–a > 1 do begin
        c := (a+b) div 2;
        if f(c) ≥ 0 then a := c else b := c;
end;
n := a;
```

Zatem łączny czas działania wyniesie $\Theta(\log n)$ i jest lepszy niż $\Theta(n)$.

Zadanie 3.10.
Ustal, co sprawdza poniższa funkcja, a następnie zaprojektuj algorytm wykonujący to samo przy mniejszej złożoności obliczeniowej.

```
function X(tab:array [1..n] of integer):boolean;
var i,j:integer;
begin
        for i := 1 to n–1 do
                for j :=  i+1 to n do
                        if tab[i] = tab[j] then return false;
        return true;
end;
```

Rozwiązanie.
Dla każdej pary różnych pól tablicy sprawdzamy, czy umieszczone w nich liczby są równe i jeśli tak jest – zwracamy wartość false. W przeciwnym razie funkcja zwraca wartość true. Można więc powiedzieć, że zadaniem procedury jest sprawdzenie, czy podany w tablicy ciąg liczb jest różnowartościowy, tj. czy wszystkie jego elementy są różne.

Elementarną operacją jest oczywiście instrukcja **if** umieszczona najgłębiej w obu pętlach. Liczba jej wywołań to n^2, co określa czas działania procedury jako $\Theta(n^2)$, czyli złożoność jest kwadratowa.

Jednym ze sposobów na procedury jest wcześniejsze posortowanie podanej tablicy którąś z szybkich metod sortowania, tj. o czasie działania $\Theta(n\log n)$ (pewną taką procedurę skonstruujemy w kolejnych rozdziałach), dzięki czemu ewentualne komórki zawierające jednakowe liczby znajdą się obok siebie. Później wystarczy sprawdzać komórki sąsiadujące, do czego potrzebny jest tylko jeden przebieg czytania. Oto dokładny kod:

```
function X2(tab:array [1..n] of integer):boolean;
var i:integer;
begin
        sortuj(tab);     // szybko
        for i := 2 to n do
                if tab[i] = tab[i–1] then return false;
        return true;
end;
```

Pętla po procedurze sortującej wymaga czasu $O(n)$, więc zasadniczy przyczynek do złożoności pochodzi od procedury szybkiego sortowania. Łączny czas pracy to $\Theta(n \log n)$.

Zadanie 3.11.
Zaprojektuj funkcję znajdującą w danej tablicy liczb naturalnych element najczęściej powtarzający się. Wymaga się złożoności mniejszej niż kwadratowa.

Rozwiązanie.
Oprzemy się na pomyśle z zadania poprzedniego. Najpierw posortujemy tablicę liczb. Następnie wiedząc, że elementy równe znajdują się obok siebie odczytamy kolejne komórki zliczając powtórzenia. Czas działania takiej procedury wynosi $\Theta(n \log n)$. A oto pełny kod:

```
function ZnajdzNajczestszy(tab:array [1..n] of integer):integer;
var i,pos,licz,maxlicz:integer;
begin
        sortuj(tab);    // szybko
        licz := 1;
        maxlicz := 1;
        pos := 1;
        for i := 2 to n do
                if tab[i] = tab[i–1] then begin
                        licz := licz+1;
                        maxlicz := max(licz,maxlicz);
                        if licz = maxlicz then pos := i;
                end
                else licz := 1;
        return tab[pos];
end;
```

Podczas pracy zmienna *i* indeksuje kolejno przeglądane komórki tablicy. Zmienna *pos* zawiera indeks komórki z wartością najczęściej występującą wśród dotychczas obejrzanego początkowego odcinka tablicy, zaś ilość wystąpień aktualnie oglądanej wartości na tym odcinku zawiera *licz*. Maksymalną wartość *licz* podczas całego działania programu zawiera *maxlicz*.

Zadanie 3.12.
Udowodnij, że funkcja:

```
function Euklides(i,j:integer):integer;
var
        k:integer;
begin
        if j > i then begin
                k := i; i := j; j := k;
        end;
        while j > 0 do begin
                k := i mod j;
                i := j;
                j := k;
        end;
        return i;
end;
```

wywołana dla dowolnych liczb naturalnych i i j zwróci największy wspólny dzielnik NWD(i,j), a jej złożoność obliczeniowa jest liniowa. Zakładamy, że operacje arytmetyczne na liczbach całkowitych wykonują się w czasie $O(1)$.

Rozwiązanie.
Najważniejszą częścią procedury jest pętla **while**. Przed wejściem do niej podane zmienne i i j spełniają $i \geq j$ (jeżeli pierwotnie było inaczej, wartości w zmiennych zostały zamienione w instrukcji **if**). Jeżeli ponadto $i=j$, wówczas procedura kończy się po jednym obiegu pętli zwracając i, co jest zgodne z naszymi oczekiwaniami. Pozostaje rozważyć przypadek $i>j$. W każdym kolejnym obiegu pętli dla danych liczb nieujemnych $i>j$ obliczamy resztę z dzielenia i przez j, wpisujemy ją do zmiennej $k<j$ i para $j>k$ zastępuje parę i, j, po czym zaczyna się nowy obieg pętli. Ostatnim obiegiem jest ten, w którym mniejsza z liczb (właśnie obliczona reszta) staje się równa 0 – wtedy procedura zwraca większą z nich. Zatem w każdym obiegu liczba $min(i,j)$ staje się mniejsza, a ponieważ nie istnieje malejący nieskończony ciąg liczb naturalnych, kiedyś musi pojawić się 0. Algorytm ma więc własność stopu – dla dowolnych poprawnych danych zakończy pracę w skończonym czasie.

Poprawność procedury udowodnimy metodą niezmienników pętli. Wykażemy najpierw, że za każdym rozpoczęciem nowego obiegu **while** zbiór wszystkich wspólnych dzielników liczb i i j pozostaje bez zmian (jest niezmiennikiem tej pętli). W tym celu wystarczy wykazać, że jeśli $k=i \bmod j$, to zbiory wspólnych dzielników i i j oraz j i k są równe. Z definicji dzielenia z resztą musi istnieć liczba całkowita a, taka że: $i=aj+k$, czyli:

- jeżeli $x|i$ oraz $x|j$, to z równości $i–aj=k$ wynika że $x|k$,
- jeżeli $x|j$ oraz $x|k$, to z równości $i=aj+k$ wynika że $x|i$.

Zatem faktycznie zbiór wspólnych dzielników i i j stanowi niezmiennik pętli. Rozumowanie to pozostaje słuszne dla wyniku ostatniego obiegu, kiedy to zmienna $i>0$ i $j=0$. Procedura zwraca wtedy i, czyli największy wspólny dzielnik ostatniej pary $\{i,0\}$. Ponieważ jednak zbiór wspólnych dzielników wartości zmiennych i i j nie zmieniał się w trakcie całego działania programu, w szczególności nie zmieniał się też największy z nich, zatem zwracana liczba musi być największym wspólnym dzielnikiem pary $\{i,j\}$ dla jej początkowych wartości – co kończy dowód poprawności programu.

Pozostaje oszacować z góry złożoność metody. Za operację elementarną można przyjąć całe wnętrze pętli **while** – wtedy czas pracy będzie po prostu liczbą jej obiegów. Zauważmy, że:

- jeśli i i j w zapisie binarnym mają równe liczby bitów, wówczas $k = i \bmod j$ musi mieć mniej bitów (bo $k \leq i–j$, a już ta ostatnia liczba ma mniej bitów),
- jeśli i ma więcej bitów niż j, to $k<j$ ma co najwyżej tyle bitów co j.

Zatem w obu przypadkach suma liczby bitów potrzebnych do zapisania i i j w każdym obiegu pętli zmniejsza się co najmniej o 1. Liczba obiegów pętli jest więc nie większa od początkowej liczby bitów potrzebnych do zapisania i oraz j, co stanowi rozmiar danych, i dowodzi liniowej złożoności algorytmu.

4. Procedury rekurencyjne

W tym rozdziale przedstawimy metody dowodzenia indukcyjnego oraz równań rekurencyjnych jako narzędzi pozwalających na weryfikację poprawności i szacowanie złożoności obliczeniowej procedur wykorzystujących rekursję. Zazwyczaj przy kodowaniu algorytmów staramy się unikać rekursji, jako nieefektywnej techniki programowania. Okazuje się jednak, że w niektórych sytuacjach kody oparte na procedurach rekurencyjnych okazują się być szybsze i prostsze od wersji iteracyjnych.

Zadanie 4.1.
Podaj czas działania procedury rekurencyjnej określonej dla naturalnych n:

```
procedure X(n:integer);
begin
        for i := 1 to n do
                for j := 1 to n do begin
                        writeln(i);
                        writeln(n);
                end;
        if n > 1 then
                for i := 1 to 8 do X(⌈n/2⌉);
end;
```

Rozwiązanie.
Czas wykonania procedury X dla argumentu n jest równy czasowi związanemu z obsługą obu pętli, warunku **if** oraz ośmiokrotnego wykonania procedury X dla argumentu $\lceil n/2 \rceil$. Oznaczając ten czas przez $T(n)$ widzimy, że poza wywołaniami $X(\lceil n/2 \rceil)$ reszta operacji zawarta w kodzie wykonywanym dla $X(n)$ ma czas wykonania typu $\Theta(n^2)$ (za operację elementarną przyjmujemy parę instrukcji *writeln*(...)), możemy więc zaproponować równanie rekurencyjne określające czas działania procedury postaci:

$$\text{(i) } T(n)=8T(\lceil n/2 \rceil)+n^2.$$

To zaś dla liczb postaci $n=2^k$ sprowadza się do rekursji typu „dziel i rządź”:

$$\text{(ii) } T(n)=8T(n/2)+n^2$$

z parametrami $a=8$, $b=2$, $d(n)=n^2$. Bez względu na wartość stałej $c=T(0)$ – czas wykonania procedury dla $n=0$ – rozwiązanie równania (ii) jest na mocy wzoru (3) typu $\Theta(n^{\log_b a}) = \Theta(n^3)$, gdyż $d(b)<a$. Używając znanych argumentów (monotoniczność) przekonujemy się, że oszacowanie to jest słuszne także dla rozwiązania (i), czyli procedura X wykonuje się w czasie $\Theta(n^3)$.

Zadanie 4.2.
Podaj czas pracy procedury rekurencyjnej określonej dla naturalnych n:

```
procedure X(n:integer);
begin
    for i := 1 to n do
        for j := 1 to n do begin
            writeln(i);
            writeln(n);
        end;
    if n > 1 then
        for i := 1 to 4 do X(⌈n/2⌉);
end;
```

Rozwiązanie.
Różnica w porównaniu z zadaniem poprzednim polega na 4–krotnym a nie 8–krotnym wywołaniu $X(\lceil n/2 \rceil)$ w trakcie wykonania $X(n)$. A zatem równanie rekurencyjne przybierze postać:

$$T(n)=4T(\lceil n/2 \rceil)+n^2,$$

co po ograniczeniu się do potęg dwójki da równanie typu „dziel i rządź" z parametrami a=4, b=2 i $d(n)=n^2$. Jest to przypadek $d(b)=a$, czyli tym razem $T(n)$, a więc procedura wykona się w czasie $\Theta(n^2 \log n)$.

Przyjrzyjmy się uważniej sposobowi uzyskania równania rekurencyjnego na czas wykonania procedury $X(n)$:

$$T(n)=4T(\lceil n/2 \rceil)+n^2.$$

W równaniu tym uwzględniamy jedynie liczbę wykonań operacji podstawowej postaci:

```
begin
    writeln(i);
    writeln(n);
end;
```

Ktoś inny przeprowadzając podobną analizę mógłby np. przyjąć za operację elementarną pojedynczą instrukcję *writeln*(...) oraz dodatkowo uwzględnić w obliczeniach czas inkrementacji zmiennej we wszystkich pętlach, wreszcie stwierdzić, że samo wywołanie $X(0)$ musi zajmować przez chwilę procesor, nawet jeśli sama procedura nic nie robi.
Wtedy dostaniemy równanie np. takiej postaci:

$$\text{(iii)} \qquad T(n)=4T(\lceil n/2 \rceil)+3n^2+n+4,\ T(0)=1.$$

Jak to możliwe, że na złożoność tej samej procedury uzyskujemy różne równania?
Otóż jest to efekt analogiczny do omawianego w przypadku procedur nierekurencyjnych. Bez względu na pewną swobodę w wyborze operacji elementarnych i określenia czasu ich wykonania (w efekcie czego możemy uzyskać różne formuły na czas działania), będą to zawsze formuły o tym samym tempie wzrostu (w sensie symbolu Θ). Jest to trochę podobne do

zmiany jednostki czasu, jaką mierzymy wykonanie kodu – bez względu na to, czy będą to sekundy, czy minuty, wyniki choć różniące się 60–krotnie, nadal będą rosły w tempie $\Theta(n^2 \log n)$. A zatem rozwiązanie równania (iii) także powinno być $\Theta(n^2 \log n)$. Sprawdźmy, czy istotnie tak jest. Jeżeli rozważymy to równanie:

$$T(n)=4T(\lceil n/2 \rceil)+3n^2+n+4,\ T(0)=1$$

oraz równanie z pierwszej metody:

$$G(n)=4G(\lceil n/2 \rceil)+n^2,\ G(0)=1,$$

wówczas przez natychmiastową indukcję widzimy, że $G(n) \le T(n)$, czyli $T(n)=\Omega(G(n))=\Omega(n^2 \log n)$. Podobnie biorąc równanie:

$$P(n)=4P(\lceil n/2 \rceil)+8n^2,\ P(0)=1, \qquad \text{(iv)}$$

widzimy, że $T(n) \le P(n)$, ale $P(n)/8$ spełnia czyste równanie typu dziel i rządź: $(P/8)(n)=4(P/8)(\lceil n/2 \rceil)+n^2$ więc $P(n)=\Theta(P(n)/8)=\Theta(n^2 \log n)$.

A zatem faktycznie $T(n)=\Theta(n^2 \log n)$, choć równanie miało nieco inną postać.

Powróćmy na chwilę do równania (iv) na funkcję *P*. Najczęściej popełnianym błędem przy analizie złożoności procedur rekurencyjnych jest bezpośrednie stosowanie wzoru (3). W tym przypadku mielibyśmy $a=4$, $b=2$ i $d(n)=8n^2$, zatem $d(b)=32>2=a$ i oszacowanie przybiera postać $\Theta(n^5)$ – zupełnie inną, niż poprzednio. Błąd polega na zastosowaniu (3) dla funkcji $d(n)=8n^2$, która nie jest funkcją iloczynową.

Zadanie 4.3.
Podaj czas pracy funkcji określonej dla liczb naturalnych *n*:

```
function X(n:integer):integer;
begin
    if n = 1 then return 1
    else return X(X(n–1))+1;
end;
```

Rozwiązanie.
Przy tak podanym kodzie nie mamy gwarancji nawet co do samej poprawności procedury, bowiem nie wiadomo, czy w wykonaniu $X(n)$ funkcja $X(n–1)$ nie zwróci jakiejś wartości większej od *n*, dla której ponownie zostanie wywołane *X*, itd. Zatem policzmy kilka pierwszych wartości według powyższego algorytmu:

$X(1)=1,$
$X(2)=X(X(1))+1=X(1)+1=2,$
$X(3)=X(X(2))+1=X(2)+1=3$
...

A zatem można przypuszczać, że procedura ta oblicza funkcję identycznościową $X(n)=n$.

Faktycznie, można to wykazać przez prostą indukcję. Jej pierwszy krok został już wykonany, kolejny ma zaś postać:

$$X(n)=X(X(n-1))+1=X(n-1)+1=n-1+1=n.$$

Przystąpimy teraz do znalezienia złożoności tej procedury. Jej czas wykonania $T(n)$ dla parametru n zawiera wywołanie kodu dla $X(n-1)$, wywołanie ponowne procedury $X(...)$ dla wyniku poprzedniej operacji oraz dodanie 1 (pojedyncza operacja). Mamy zatem rekursję:

$$T(n)=T(n-1)+T(X(n-1))+1=2T(n-1)+1.$$

Jest to równanie typu „jeden krok w tył", typowe dla problemu wież z Hanoi. Jego rozwiązanie, a zatem i czas pracy całej procedury to $2^n-1 = \Theta(2^n)$. Złożoność obliczeniowa (oczywiście wyrażana w funkcji rozmiaru danych) będzie więc superwykładnicza.

Zadanie 4.4.
Sprawdź, co wylicza poniższa funkcja określona na liczbach całkowitych nieujemnych, a następnie podaj jej czas działania.

```
function X(n:integer):integer;
begin
    if n = 1 or n = 0 then return 1
    else return X(n–1) – X(n–2);
end;
```

Rozwiązanie.

Wyliczmy kilka pierwszych wartości funkcji. Mamy zatem:

n	0	1	2	3	4	5	6	7	8	9	10
$X(n)$	1	1	0	–1	–1	0	1	1	0	–1	–1

Dla argumentu n=8 sytuacja jest analogiczna jak dla n=2 i widzimy, że wartości funkcji będą powtarzać się z okresem 6. Możemy napisać, że:

$$X(n)=\begin{cases}-1, & \text{gdy } n \bmod 6 \in \{3,4\}\\ 0, & \text{gdy } n \bmod 6 \in \{2,5\}\\ 1, & \text{gdy } n \bmod 6 \in \{0,1\}.\end{cases}$$

Natomiast równanie opisujące czas działania funkcji ma postać:

$T(0)=T(1)=1,$
$T(n)=T(n-1)+T(n-2)+1$ dla $n>1,$

gdzie za operację podstawową uznano wykonanie instrukcji **return**. Łatwo zauważyć, że gdyby pominąć ostatni składnik drugiej formuły, mielibyśmy równanie „przesuniętego o jeden wyraz" ciągu Fibonnaciego:

$U(0)=U(1)=1$,
$U(n)=U(n–1)+U(n–2)$ dla $n>1$,

Wzór ogólny na kolejne wyrazy takiego ciągu jest znany, rozwiązaniem jest:

$$U(n)=\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right],$$

Okazuje się, że pominięcie jedynki nie zmienia asymptotycznego tempa wzrostu rozwiązania rekursji, mamy bowiem przez natychmiastową indukcję: $U(n)\le T(n)$ oraz $T(n)<2U(n)$ (jako że dla $n>1$ przez indukcję zachodzi $T(n)=T(n–1)+T(n–2)+1<2U(n–1)+2U(n–2)=2U(n)$), stąd $T(n)=\Theta(U(n))$. Ponieważ $|(1+\sqrt{5})/2|>1$ i $|(1-\sqrt{5})/2|<1$, więc ostatecznie oszacowanie czasu ma postać:

$$T(n)=\Theta(((1+\sqrt{5})/2)^n).$$

Zadanie 4.5*.
Udowodnij, że następujący kod ma własność stopu:

```
function F(n:integer):integer;
begin
    if n mod 2 = 0 then return n/2
    else return F(F(3n+1));
end;
```

tzn. że funkcja powyższa uruchomiona dla dowolnej liczby naturalnej n w skończonym czasie zwróci wynik.

Rozwiązanie.
Załóżmy przez sprzeczność, że istnieje liczba a_0, będąca argumentem inicjalnie przekazywanym do funkcji F, dla której funkcja ta zapętla się w nieskończoność. Oczywiście a_0 musi być nieparzysta (inaczej F zwraca $a_0/2$ kończąc działanie), czyli $3a_0+1$ jest parzyste i wywołanie $F(a_0)$ sprowadza się do wywołania $F(a_0)=F(F(3a_0+1))=F(3a_0/2+1/2)$. Oznaczmy liczbę naturalną $a_1=3a_0/2+1/2$ i powtórzmy dla niej powyższe rozumowanie. Nasze a_1 nie może być parzyste, gdyż inaczej $F(a_1)$ zwróciłoby wynik po wykonaniu jednej instrukcji, a zatem $a_2=3a_1/2+1/2$ jest całkowite i wywołanie $F(a_0)$ sprowadza się do uruchomienia $F(a_2)$. Kontynuując rozumowanie możemy zdefiniować rekurencyjny ciąg liczb postaci:

$a_{k+1}=3a_k/2+1/2$,

którego rozwiązaniem (co można łatwo sprawdzić przez indukcję) jest ciąg:

$a_k=(3/2)^k(a_0+1)–1$.

Przy tym, skoro $F(a_0)$ zapętla się w nieskończoność, to wywołanie takie na pewnym etapie obliczeń sprowadza się do wywołania $F(a_k)$, a zatem wszystkie a_k muszą być całkowite. To ostatnie jest oczywiście niemożliwe dla k przekraczających wykładnik najwyższej potęgi liczby 2 dzielącej jeszcze a_0+1 (patrz ostatnia równość). Doszliśmy zatem do sprzeczności, a więc funkcja F kończy swe działanie po pewnym czasie dla każdego naturalnego argumentu.

Zadanie 4.6.
Napisz procedurę o złożoności wielomianowej wyliczającą dla danych dwóch liczb naturalnych x i y potęgę x^y. Nie są dostępne żadne operacje zmiennoprzecinkowe, można posługiwać się jedynie czterema działaniami podstawowymi na liczbach całkowitych, o których zakładamy, że wykonują się w czasie $O(1)$.

Rozwiązanie.
Rozmiarem danych jest w tym przypadku suma liczby bitów potrzebnych do zapisu x i y, z dokładnością do 2 jest to $\log_2 x+\log_2 y=\log_2 xy$. Zatem metoda opierająca się bezpośrednio na definicji potęgi:

```
function PotegaKlasyczna(x,y:integer):integer;
var
        w:integer;
begin
        w := 1;
        for i := 1 to y do w := w*x;
        return w;
end;
```

wykonująca y mnożeń jest liniowa względem y, czyli wykładnicza względem rozmiaru danych. Musimy więc stworzyć inny algorytm. Zachodzą jednak wzory: $a^{2k}=(a^k)^2$ oraz $a^{2k+1}=(a^k)^2 a$. Na ich podstawie tworzymy procedurę rekurencyjną:

```
function potega(x,y:integer):integer;
var
        pom:integer;
begin
        if y = 0 then return 1
        else  begin
                pom := potega(x,y div 2);
                if y mod 2 = 0 then return pom*pom
                else return pom*pom*x;
        end;
end;
```

Czas obliczeń nie zależy bezpośrednio od x, możemy więc rozpatrywać go jako funkcję y. Funkcja *potega*, poza jednorazowym wywołaniem podprocedury, przeprowadza operacje o czasie dającym się oszacować z góry przez stałą niezależną od danych, czyli równanie rekurencyjne jest postaci:

(i) $T(y)=T(\lfloor y/2 \rfloor)+1$.

Oszacowanie jego rozwiązania poprzez równanie typu dziel i rządź określone na potęgach 2 postaci:

$T(y)=T(y/2)+1$

daje $a=1$, $b=2$, $d(n)=1$, czyli $T(y)=\Theta(\log y)$ – co jest liniowe względem rozmiaru danych (za który uznajemy sumę długości zapisów bitowych liczb x i y). Ten sam wynik możemy również uzyskać bezpośrednio, zauważając że (i) jest równaniem na liczbę bitów potrzebnych do zakodowania y (zapis $\lfloor y/2 \rfloor$ otrzymujemy z kodu bitowego y poprzez skreślenie ostatniej cyfry).

Warto zauważyć, że w ogólności potęgowanie liczb naturalnych x^y nie może być wykonane w czasie wielomianowym, bowiem sam wynik ma liczbę bitów rzędu $\log_2 x^y = y\log_2 x$, co jest wykładnicze względem rozmiaru danych, zaś wypisanie pojedynczego bitu wyniku jest również operacją, którą należy uwzględniać przy szacowaniu złożoności. Do sprzeczności z wynikiem poprzedniego zadania doprowadziło nas założenie o wykonywalności operacji arytmetycznych na dowolnie długich liczbach w stałym czasie. Jest to uproszczenie poprawne tylko w ograniczonym zakresie dla współczesnych komputerów, nie można go jednak utrzymać w ogólności. Przykład powyższy pokazuje, że należy zachowywać daleko idącą ostrożność przy podejmowaniu założeń upraszczających.

Zadanie 4.7*.
Napisz procedurę o złożoności wielomianowej sprawdzającą, czy podana liczba naturalna n jest prostą potęgą innej liczby naturalnej, tj. czy zachodzi $n=a^b$ dla pewnych całkowitych a i b większych do 1. Nie są dostępne żadne operacje zmiennoprzecinkowe, można posługiwać się jedynie czterema działaniami podstawowymi na liczbach całkowitych, o których zakładamy, iż wykonują się w czasie $O(1)$.

Rozwiązanie.
Zauważmy, że jeśli odpowiedź jest twierdząca, to $n=a^b \geq 2^b$, czyli ewentualnych możliwych wartości b należy szukać w przedziale 2,..., $\lfloor \log_2 n \rfloor$. Ta ostatnia liczba jest o 1 mniejsza od długości zapisu bitowego n, może też być w czasie $O(\log n)$ obliczona przy wykorzystaniu operacji arytmetycznych (tematem jednego z wcześniejszych zadań była bardzo podobna funkcja wyliczająca $\lceil \log_2 n \rceil$). Ilość potencjalnych kandydatów na b jest więc mniejsza od rozmiaru danych (tj. długości zapisu bitowego n), który oznaczymy przez N. Wystarczy więc, że będziemy umieli w czasie wielomianowym sprawdzić, czy $n=a^b$ dla pewnego a przy danym n i b. Tu zaś odpowiedniego a można szukać używając metody szukania binarnego w przedziale 1,...,n oraz funkcji potęgującej z poprzedniego zadania. Ostatecznie mamy kod:

```
function PotegaProsta(n:integer):boolean;
var
        s_pocz,x_sr,x_kon,wyn,b:integer;
begin
        for b := 2 to ⌊log2(n)⌋ do begin
                x_pocz := 1;
                x_kon := n;
                while x_kon–x_pocz > 1 do begin
                        x_sr := (x_pocz+x_kon) div 2;
                        wyn :=  potega(x_sr,b);
                        if wyn > n then x_kon := x_sr
                        else
                                if wyn < n then x_pocz := x_sr
                                else return true;
                end;
        end;
        return false;
end;
```

Podczas szacowania złożoności należy zauważyć, że pętla **for** wykona się mniej, niż N razy, zaś dla każdego b pętla **while** wykona co najwyżej $\lceil \log_2(n-1) \rceil \le N$ obiegów. Najbardziej czasochłonną operacją w tej pętli jest wyliczanie potęgi, które na mocy poprzedniego zadania zajmie czas nie dłuższy od $O(\log N)$, jako że zawsze $b \le N$ i ostatecznie czas pracy procedury wyniesie $O(N^2 \log N) = O((\log^2 n)(\log\log n))$. Złożoność jest więc większa, niż kwadratowa, ale mniejsza niż sześcienna.

Odejdźmy na chwilę od założenia, że operacje arytmetyczne wykonywane są w pojedynczych instrukcjach i pomyślmy jak należałoby wykonywać działania na bardzo długich liczbach (np. kilkusetcyfrowych), które dostępne są w postaci odpowiednio długich tablic zawierających ich kolejne bity. Dodając do siebie x_1 i x_2 o długościach bitowych odpowiednio n_1 i n_2 wyliczamy kolejne bity wyniku (licząc od najmłodszego) poprzez dodawanie kolejnych bitów składników i ew. przeniesienia – widać że ilość operacji jest $O(max(n_1,n_2))$, czyli liniowa. Podobnie można zaimplementować odejmowanie, korzystając np. z kodu uzupełnienia do dwóch. Tradycyjna metoda mnożenia „pisemnego" polega na dodawaniu do siebie co najwyżej n_2 kopii liczby x_1 przesuniętej bitowo – kopie te odpowiadają ustawionym bitom zapisu liczby x_2 i mają długość nie przekraczającą n_1+n_2. Dodając te składniki wcześniejszym algorytmem wykonujemy $O(max(n_1,n_2)^2)$ operacji. Tak zaimplementowane mnożenie długich liczb jest więc kwadratowe. To samo dotyczy dzielenia liczb całkowitych (czyli dzielenia z resztą).

Zadanie 4.8.
Udowodnij poprawność oraz oszacuj złożoność obliczeniową procedury mnożącej dwie długie liczby naturalne. Zakłada się, że oba argumenty są przekazywane w postaci ciągów bitów jednakowej długości, będącej potęgą liczby 2 (w razie potrzeby uzupełniamy zapis bitowy zerami z przodu).

```
procedure razy(x,y:superlong): superlong;
var
        n:integer;
        a,b,c,xx,yy,cx,cy:superlong;
begin
// wszystkie dodawania i odejmowania długich liczb wykonujemy liniowo,
// wszystkie mnożenia liczb przez potęgi dwójki wykonujemy za pomocą przesunięć
// bitowych
        n := ilosc_bitow(x);
        if n = 1 then return x*y
        else begin
                podziel ciągi bitów x i y na starsze i młodsze połowy (długości n/2 bitów),
                wówczas x = 2^(n/2)x1+x2 oraz y = 2^(n/2)y1+y2;
                xx+cx :=  x1+x2;        // xx ma n/2 bitów, a cx jest równe 2^(n/2) lub 0
                yy+cy :=  y1+y2;        // yy ma n/2 bitów, a cy jest równe 2^(n/2) lub 0
                a :=  cx*cy + cx*yy + cy*xx + razy(xx,yy);          // czyli a = (x1+ x2)*(y1+ y2)
                b := razy(x1, y1);
                c := razy(x2, y2);
                return b*2^n+(a–b–c)*2^(n/2)+c;
        end
end;
```

Rozwiązanie.
Poprawność procedury można łatwo wykazać przez indukcję względem długości bitowej argumentów. Dla $n = 1$ jest to oczywiste, zaś dla większych n funkcja zwraca wartość:

$$b*2^n+(a–b–c)*2^{n/2}+c = 2^n x_1y_1+((x_1+x_2)(y_1+y_2) – x_1y_1–x_2y_2)2^{n/2}+x_2y_2 =$$
$$= 2^n x_1y_1+(x_1y_2+x_2y_1)\ 2^{n/2}+x_2y_2 = (2^{n/2}x_1+x_2)(2^{n/2}y_1+y_2) = xy,$$

co kończy dowód poprawności.

Oszacujemy teraz złożoność obliczeniową procedury w funkcji liczby bitów argumentu n. Wszystkie operacje wykonywane w ciele funkcji, poza wywołaniami rekurencyjnymi, zgodnie ze specyfikacją są liniowe. Zachodzą też trzy wywołania rekurencyjne dla dwa razy mniejszego argumentu. Mamy zatem:

$$T(n) = 3T(n/2) + n,$$

co zgodnie ze znanymi wzorami, po podstawieniu $a = 3$, $b = 2$ i $d(n) = n$ daje rozwiązanie $T(n) = \Theta(n^{\log_2 3})$.

Oczywiście procedura powyższa pozwala na pomnożenie dowolnych liczb o dowolnych długościach bitowych n_1 i n_2 – po uzupełnieniu zapisów bitowych zerami z przodu (do długości równej najmniejszej potędze dwójki nie mniejszej od $max(n_1,n_2)$) łączna długość bitowa wzrośnie nie więcej, niż czterokrotnie i uzyskany czas pracy $\Theta((n_1+n_2)^{\log_2 3})$ pozostaje słuszny. A zatem rekurencyjna metoda mnożenia jest szybsza niż tradycyjne mnożenie pisemne o złożoności kwadratowej. Znane są zaawansowane, jeszcze szybsze algorytmy mnożenia długich liczb, o „prawie" liniowej złożoności np. $\Theta((n_1+n_2)\log(n_1+n_2)\log\log(n_1+n_2))$ (por. [1]).

Zadanie 4.9.*
Wykaż, że czas działania poniższej funkcji rekurencyjnej X określonej dla liczb naturalnych jest wykładniczy oraz że funkcję tą można wyliczyć inną procedurą o złożoności obliczeniowej $O(1)$.

```
function X(n:integer):integer
const primes:array[1...7] of integer = {2,3,5,7,11,13,17};
begin
    if n<=7 then return primes[n]
    else return (23*X(n–1)*X(n–2)+3*⌊X(n–7)/(X(n–4)+7)⌋+79*X(n–5)*X(n–3)+
            +173*X(n–6)) mod 2011;
end;
```

Rozwiązanie.
Równanie rekurencyjne na czas działania (w którym za instrukcję podstawową przyjęto wywołanie **return**) ma postać:

$$T(1)=T(2)=T(3)=T(4)=T(5)=T(6)=T(7)=1,$$
$$T(n)=T(n–1)+T(n–2)+T(n–3)+T(n–4)+T(n–5)+T(n–6)+T(n–7)+1 \text{ dla } n>7,$$

stąd mamy $1 \le T(n-1) \le T(n)$, a zatem dla $n>7$ zachodzi $T(n) \le 8T(n-1)$ i $T(n) \ge 7T(n-7)$. Pierwsza z nierówności daje nam oszacowanie górne

$$T(n) \le 8T(n-1) \le 8^2T(n-2) \le 8^3T(n-3) \le \ldots \le 8^{n-1}T(1) = O(8^n),$$

zaś druga dolne:

$$T(n) \ge 7T(n-7) \ge 7^2T(n-14) \ge 7^3T(n-21) \ge \ldots \ge 7^{\lceil n/7 \rceil - 1} \ge 7^{n/7}/7 = \Omega(7^{n/7}).$$

Zatem tempo wzrostu T plasuje się pomiędzy dwoma funkcjami wykładniczymi, jest to więc funkcja typu wykładniczego.

Odnośnie drugiej części zadania – już sama formuła rekurencyjna powinna zniechęcać do poszukiwania wzoru ogólnego na $X(n)$. Zamiast tego zauważmy, że wartościami X są reszty z dzielenia przez 2011, są to więc liczby całkowite z przedziału 0,…,2010. Ponadto $X(n)$ dla $n>7$ jest jednoznacznie wyznaczone przez siedem poprzednich wartości tej funkcji. Ciągów długości 7 o elementach z wymienionego przedziału jest skończenie wiele (tj. 2011^7), a zatem istnieją argumenty $b>a>7$, takie że ciąg $X(b-7), X(b-6),\ldots, X(b-1)$ jest identyczny z wcześniejszym $X(a-7), X(a-6),\ldots, X(a-1)$. Wtedy jednak (zgodnie z formułą rekurencyjną) sekwencja wartości $X(a), X(a+1), \ldots, X(b)$ musi powtarzać się dalej w nieskończoność z okresem $b-a$. Możliwe jest więc stablicowanie odpowiednio długiego fragmentu początkowego wartości X i posłużenie się tą tablicą do wyznaczania $X(n)$ dla dowolnego n. Kod procedury o czasie działania $O(1)$ będzie miał postać:

```
function X2(n:integer):integer
const  a:integer=…;
       b:integer=…;
       tab:array[1...b] of integer = {…};   // odpowiednie wartości
begin
       if n<=b then return tab[n] else return tab[n–(b–a)*⌈(n–b)/(b–a)⌉];
end;
```

Przedstawione rozwiązanie nie daje nam żadnych wskazówek odnośnie tego, jakie liczby należy przypisać stałym a, b oraz elementom tablicy. Jest to więc prosty przykład niekonstruktywnego dowodu istnienia pewnego algorytmu o pożądanej własności (tj. złożoności $O(1)$), bez pokazywania konkretnego przykładu jego kodu. Jest też wysoce prawdopodobne, że procedura, której istnienie wykazaliśmy byłaby niewykonalna w praktyce (liczba b może być tak wielka, że stworzenie tablicy *tab* w pamięci żadnego prawdziwego komputera nie jest możliwe). Formalnie jednak, zgodnie z przyjętym modelem obliczeń jest to algorytm o złożoności $O(1)$, w którym „trudności implementacyjne" ukryto w stałej schowanej pod symbolem $O(\ldots)$.

5. Macierze i grafy

Jednym z najczęściej pojawiających się obiektów badań w problemach optymalizacji dyskretnej, badaniach operacyjnych itp. są grafy. Głównym zadaniem tego rozdziału jest prezentacja podstawowych technik pracy z grafami oraz algorytmów grafowych. Najbardziej naturalną metodą przedstawienia grafu jako struktury danych jest jego macierz sąsiedztwa lub różne jej odmiany. Okazuje się, że często możliwe jest wykorzystanie specyficznych operacji na macierzach sąsiedztwa (abstrahując od interpretacji wierzchołkowo–krawędziowej) w celu szybszego określenia właściwości badanych grafów.

Zadanie 5.1.
Dany jest n–wierzchołkowy graf G w postaci macierzy sąsiedztwa:

G: **array**[1..n,1..n] **of boolean**

Napisz procedurę wykonującą się w czasie $O(n^3)$, sprawdzającą, czy G zawiera podgraf w postaci trójkąta C_3.

Rozwiązanie.
Najprościej jest wprowadzić 3 zmienne przebiegające zbiór wszystkich trójek numerów różnych wierzchołków mogących stanowić wierzchołki trójkąta:

```
function trojkat(G: array [1..n,1..n] of boolean):boolean;
var
    i,j,k:integer;
begin
    for i := 1 to n do
        for j := 1 to n do
        if i ≠ j then
            for k := 1 to n do
            if k ≠ i and k ≠ j then
                if G[i,j] and G[i,k] and G[j,k] then return true;
    return false;
end;
```

choć w tym przypadku z uwagi na symetrię możliwe jest przyspieszenie algorytmu (dla dużych n około 6 razy – pomyśl dlaczego?) poprzez założenie, że i,j,k są numerami wierzchołków trójkąta w rosnącej kolejności:

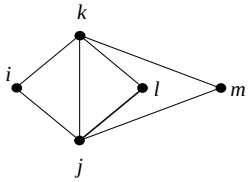
```
function trojkat(G: array[1..n,1..n] of boolean):boolean;
var
    i,j,k:integer;
begin
    for i := 1 to n–2 do
        for j :=i+1 to n–1 do
            for k := j+1 to n do
                if G[i,j] and G[i,k] and G[j,k] then return true;
    return false;
end;
```

Zadanie 5.2.
Dany jest graf *n*–wierzchołkowy *G* w postaci macierzy sąsiedztwa:

G:**array**[1..*n*,1..*n*] **of boolean**

Napisz procedurę sprawdzającą, czy *G* zawiera podgraf $K_{1,1,1}$.

Rozwiązanie.
Znów przeczesujemy graf w poszukiwaniu piątki wierzchołków tworzących nasz podgraf. Dla każdej takiej piątki sprawdzamy, czy między nimi występują krawędzie jak na rysunku. Poniżej widzimy, w jaki sposób wierzchołkom szukanego grafu zostały przypisane zmienne sterujące pętlami.




```
function podgraf(G: array[1..n,1..n] of boolean):boolean;
var
    i,j,k,l,m:integer;
begin
    for i := 1 to n do
        for j := 1 to n do
        if i ≠ j then
            for k := 1 to n do
            if i ≠ k and j ≠ k then
                for l := 1 to n do
                if i ≠ l and j ≠ l and k ≠ l then
                    for m := 1 to n do
                    if i ≠ m and j ≠ m and k ≠ m and l ≠ m then
                        if G[i,j] and G[i,k] and G[j,k] and
                        G[k,m] and G[j,m] and
                        G[j,l] and G[k,l] then return true;
    return false;
end;
```

Czas pracy ze względu na pięć zagnieżdżonych pętli **for** wynosi $O(n^5)$.

> W analogiczny sposób dla dowolnego ustalonego grafu *H* o *k* wierzchołkach można łatwo napisać algorytm sprawdzający, czy w podanym na wejściu grafie *n*–wierzchołkowym *G* istnieje podgraf izomorficzny z *H*. Przy tym czas działania procedury wyniesie $O(n^k)$. To samo dotyczy problemu poszukiwania podgrafów indukowanych.

Zadanie 5.3.
Znajdź liczbę k i pewien graf k–wierzchołkowy H, dla którego problem istnienia podgrafu izomorficznego z H w danym grafie n–wierzchołkowym G będzie można rozwiązać w czasie krótszym, niż $O(n^k)$ (czyli $o(n^k)$).

Rozwiązanie.
Jednym z możliwych rozwiązań są gwiazdy, np. niech $H=K_{1,10}$. Ma ona k=11 wierzchołków, gdy tymczasem istnienie w danym grafie takiej gwiazdy jest równoważne istnieniu wierzchołka stopnia 10 lub więcej. Stopnie wierzchołków grafu można policzyć w czasie $O(n^2)$. Mamy zatem poniższą procedurę wykonującą się w czasie $O(n^2)$ i rozwiązującą problem istnienia podgrafu izomorficznego z 11–wierzchołkowym $K_{1,10}$:

```
function gwiazda(G: array [1..n,1..n] of boolean):boolean;
var
        i,j,deg:integer;
begin
        for i := 1 to n do begin
                deg := 0;
                for j := 1 to n do
                        if G[i,j] then deg++;
                if deg ≥ 10 then return true;
        end;
        return false;
end;
```

W kolejnych zadaniach poznamy dalsze przykłady takich grafów.

A teraz kilka zadań obrazujących przydatność operacji mnożenia macierzy sąsiedztwa:

Zadanie 5.4.
Dany jest graf n–wierzchołkowy w postaci macierzy sąsiedztwa. Podobnie, jak w pierwszym zadaniu, napisz procedurę sprawdzającą, czy G zawiera podgraf w postaci trójkąta C_3, tym razem o czasie działania $O(n^{\log_2 7}) = O(n^{2.81\ldots})$.

Rozwiązanie.
Podana złożoność sugeruje zastosowanie algorytmu Strassena na mnożenie macierzy. Warto przypomnieć, że jeżeli potraktujemy macierz sąsiedztwa grafu G jak macierz liczbową (z zerami i jedynkami), wówczas jej k–ta potęga G^k jest macierzą, która w polu $G^k[i,j]$ zawiera liczbę wszystkich marszrut o długości k prowadzących z wierzchołka i do wierzchołka j (prosty dowód indukcyjny). Widać też, że dany wierzchołek grafu zawiera się w pewnym cyklu C_3 wtedy i tylko wtedy, gdy istnieje wychodząca z niego i powracająca doń marszruta długości 3. Zatem wystarczy sprawdzić, czy na przekątnej macierzy G^3 istnieje jakaś liczba większa od 0.

```
function trojkat(G: array [1..n,1..n] of integer):boolean;
var
        i:integer;
        G2,G3:array[1..n,1..n] of integer;
begin
        G2 := G*G;              // wg Strassena
        G3 := G2*G;             // wg Strassena
        for i := 1 to n do
                if G3[i,i] > 0 then return true;
        return false;
end;
```

Zasadniczy przyczynek do złożoności ma mnożenie macierzy – pozostałe operacje są liniowe względem n. Gdyby zastosować klasyczne mnożenie macierzy – czas pracy wyniósłby znów $O(n^3)$.

Zadanie 5.5.
Dany jest graf n–wierzchołkowy w postaci macierzy sąsiedztwa. Napisz procedurę sprawdzającą, czy G zawiera podgraf w postaci cyklu C_4, wykonującą się w czasie $O(n^{\log_2 7})$.

Rozwiązanie.
Tym razem zauważamy, że istnienie cyklu C_4 w grafie jest równoważne istnieniu pary różnych wierzchołków, pomiędzy którymi są co najmniej dwie różne marszruty długości 2. Ich liczbę znajdujemy podnosząc macierz sąsiedztwa do kwadratu.

```
function C4(G: array [1..n,1..n] of integer):boolean;
var
        i,j:integer;
        G2:array[1..n,1..n] of integer;
begin
        G2 := G*G;              // wg Strassena
        for i := 1 to n–1 do
                for j := i+1 to n do
                        if G2[i,j] > 1 then return true;
        return false;
end;
```

Znów główny przyczynek do złożoności procedury ma mnożenie macierzy – pozostałe operacje są bowiem tylko $O(n^2)$. Ostatecznie czas działania jest $O(n^{\log_2 7})$.

Warto zauważyć, że tym razem zastosowanie podejścia analogicznego do poprzedniego zadania nie jest możliwe. Przykładowo, jeśli G będzie macierzą sąsiedztwa dwuwierzchołkowego grafu krawędzi K_2, wówczas $G^4[1,1]=1$, gdyż K_2 zawiera marszrutę długości 4 z wierzchołka 1 do 1 (przechodzi ona cztery razy jedyną krawędź grafu, naprzemiennie zmieniając kierunek). Oczywiście K_2 nie zawiera żadnego cyklu C_4.

Pamiętajmy, że marszruta w grafie jest pojęciem ogólniejszym od ścieżki (lub cyklu) – w tej drugiej odwiedzane wierzchołki nie mogą się powtarzać. Dlatego możemy w czasie wielomianowym wyznaczyć liczbę marszrut długości $\leq n$ między dowolną parą wierzchołków grafu n-wierzchołkowego (podnosząc macierz sąsiedztwa do potęgi l), choć najprawdopodobniej nie istnieje efektywny algorytm określający liczbę takich ścieżek długości l. Wynika to z NP-zupełności problemu ścieżki Hamiltona (por. rozdziały 7 i 8), którą definiuje się po prostu jako ścieżkę długości $n–1$ w grafie n-wierzchołkowym. Jednak jeżeli ustalimy wartość l jako dowolną stałą np. przyjmując $l=100$, wówczas zmodyfikowane zagadnienie: „wyznacz liczbę ścieżek długości 100 łączących wyróżnioną parę wierzchołków danego grafu" (a więc daną wejściową jest już tylko graf, a nie graf i liczba l) można – przynajmniej teoretycznie – rozwiązać algorytmem wielomianowym. Postępując podobnie, jak w zadaniu 5.2 uzyskamy nie nastrajający optymizmem, lecz formalnie wielomianowy czas działania $O(n^{101})$.

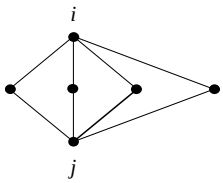
Zmodyfikujmy poprzednie zadanie, żądając od programu zwracania obszerniejszych informacji niż binarne.

Zadanie 5.6.
Przy założeniach z poprzedniego zadania napisz funkcję podającą liczbę różnych cykli C_4 w danym grafie G. Wymagany czas pracy to $O(n^{\log_2 7})$.

Rozwiązanie.
Dla każdej pary różnych wierzchołków liczymy nieuporządkowane pary dróg długości 2 łączących te wierzchołki (każde takie dwie drogi określają jeden cykl C_4). Suma tych liczb po wszystkich parach wierzchołków zliczy nam następujące obiekty w grafie G: podgraf w postaci cyklu C_4 z dwoma wyróżnionymi uporządkowanymi wierzchołkami przeciwległymi (te o numerach i i j). W cyklu C_4 można wyróżnić takie dwa wierzchołki na 4 sposoby, czyli każdy cykl został zliczony właśnie tyle razy.

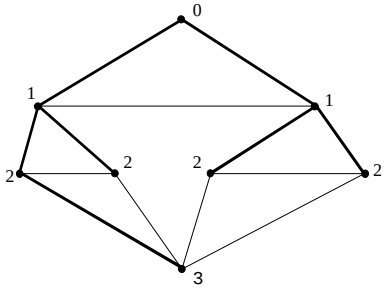
```
function LiczC4(G: array [1..n,1..n] of integer):integer;
var
    i,j,s:integer;
    G2:array[1..n,1..n] of integer;
begin
    G2 := G*G;          // wg Strassena
    s:=0;
    for i := 1 to n do
        for j := 1 to n do
            if i ≠ j then s := s+G2[i,j]*(G2[i,j] –1)/2;
    return s/4;         // każdy cykl został zliczony 4–krotnie
end;
```

Przykładowo na rysunku przedstawiono graf, w którym wierzchołki i oraz j połączone są czterema marszrutami długości 2:



Graf ten zawiera więc 4(4–1)/2=6 różnych cykli C_4.

W tym miejscu warto przypomnieć kilka podstawowych faktów dotyczących algorytmów grafowych. Najczęściej stosowanymi strukturami danych reprezentującymi grafy są macierz sąsiedztwa i lista sąsiadów. Ich zajętości pamięciowe wynoszą odpowiednio $\Theta(n^2)$ i $\Theta(n+m)$, gdzie n jest liczbą wierzchołków, a m liczbą krawędzi. Oba sposoby zakodowania grafu można algorytmicznie tłumaczyć na siebie w obie strony – niezbędny do tego czas wynosi $\Theta(n^2)$.
Jednym z najprostszych algorytmów grafowych jest procedura Dijkstry pozwalająca na znalezienie najkrótszych dróg w grafie pomiędzy danym jego wierzchołkiem a wszystkimi innymi wierzchołkami. Zakłada się tutaj, iż krawędziom przypisane są liczby nieujemne określające ich długości. Czas działania algorytmu wynosi $\Theta(n^2)$ dla obu typów struktur.
Pewną modyfikację algorytmu Dijkstry otrzymujemy zakładając, że wszystkie krawędzie mają długość 1. Wówczas najkrótsze drogi odpowiadają najkrótszym ścieżkom w grafie wychodzącym z danego wierzchołka do wszystkich innych wierzchołków znajdujących się w tej samej składowej spójności. Rozpoczynając od naszego wierzchołka algorytm w pierwszej pętli znajduje wszystkich jego sąsiadów, następnie innych sąsiadów tych sąsiadów, itd. Wierzchołki zostają poustawiane na kolejnych „piętrach", których numeracja odpowiada liczbie krawędzi dzielących je od wierzchołka początkowego (wierzchołek początkowy tworzy piętro zerowe). W rezultacie powstaje drzewo spinające całą składową spójności zawierającą wierzchołek początkowy. Taki sposób wędrówki po grafie nazywa się metodą *przeszukiwania wszerz*. Dla grafu podanego w postaci listy sąsiadów można go zaimplementować w czasie $\Theta(n+m)$, czyli liniowym.



Można także powtórzyć całą procedurę dla pewnego wierzchołka, który nie znalazł się w pierwszym drzewie spinającym (jeśli takowy istnieje), później znów, itd. uzyskując rozbicie danego grafu na składowe spójności (wierzchołki z różnych składowych zostają poetykietowane numerami swoich składowych przypisywanych w kolejności ich znajdowania). Taka procedura również trwa $\Theta(n+m)$ dla listy sąsiadów, a więc jest liniowa. W szczególności możemy zweryfikować, czy dany graf jest spójny.

Zadanie 5.7.
Zaprojektuj procedurę o złożoności liniowej, która mając daną listę sąsiadów grafu G sprawdzi, czy jest to graf dwudzielny.

Rozwiązanie.
Warto przypomnieć, że graf dwudzielny jest to graf, którego zbiór wierzchołków V można przedstawić jako sumę dwóch zbiorów rozłącznych V_1 i V_2 (partycje), takich że każda krawędź łączy wierzchołek z V_1 z wierzchołkiem z V_2. Najpierw dokonujemy rozbicia grafu na składowe spójności, w każdej składowej tworząc drzewo spinające z wierzchołkami poetykietowanymi ich odległościami od pewnego wierzchołka wyróżnionego w tej składowej. Na mocy powyższego można to zrobić w czasie $O(n+m)$, czyli proporcjonalnym do rozmiaru listy sąsiadów. Zauważmy dalej, że dane drzewo spinające pewną składową spójności jednoznacznie określa ewentualne rozbicie wierzchołków tej składowej na partycje dwudzielności. Faktycznie, wierzchołki leżące na piętrach parzystych muszą znaleźć się w tej samej partycji, co wierzchołek początkowy, a te z pięter nieparzystych – w przeciwnej partycji. Do weryfikacji dwudzielności potrzeba jeszcze sprawdzić, czy jakaś krawędź spoza drzewa spinającego nie burzy dwudzielności łącząc dwa wierzchołki z pięter parzystych lub dwa z nieparzystych (tworząc w ten sposób cykl nieparzysty w grafie). A zatem po przeprowadzeniu rozbicia grafu na składowe i poetykietowaniu wierzchołków wystarczy odczytać wszystkie krawędzie grafu dla każdej z nich sprawdzając, czy nie łączy ona wierzchołków o etykietach tej samej parzystości. To oczywiście znów wymaga jedynie $O(n+m)$ operacji.

Jak wiadomo, istnieją dwa podstawowe modele kolorowania grafów: wierzchołkowe i krawędziowe. W obu przypadkach chodzi o przypisanie kolorowanym elementom grafu barw w taki sposób, by elementy sąsiadujące uzyskiwały różne barwy. Wprawdzie nie znaleziono jak dotąd algorytmów wielomianowych kolorujących dowolne grafy (wierzchołkowo lub krawędziowo) optymalnie, tj. przy użyciu minimalnej możliwej liczby barw (w kolejnych rozdziałach przyjrzymy się bliżej tym zagadnieniom), to jednak istnieją twierdzenia szacujące odpowiednie liczby z góry:

Twierdzenia Brooksa: Każdy graf o maksymalnym stopniu Δ nie będący cyklem nieparzystym ani grafem pełnym można pokolorować wierzchołkowo, używając co najwyżej Δ barw.

Oczywiście grafy pełne i cykle nieparzyste koloruje się przy użyciu $\Delta+1$ barw.

Twierdzenie o czterech barwach: Każdy graf planarny można pokolorować wierzchołkowo używając co najwyżej 4 kolorów.

Twierdzenie Vizinga: Każdy graf o maksymalnym stopniu Δ można pokolorować krawędziowo przy użyciu co najwyżej $\Delta+1$ barw.

Łatwo zauważyć, że każdy graf wymaga co najmniej Δ barw przy kolorowaniu krawędziowym.

Ważne jest, że dowody powyższych trzech twierdzeń są konstruktywne – przedstawiają one procedury wielomianowe wykonujące odpowiednie pokolorowania przy użyciu liczby barw nie przekraczającej wielkości z treści twierdzeń (algorytmy dla twierdzeń Brooksa i „o czterech barwach" potrzebują $O(n^2)$ czasu, zaś algorytm Vizinga potrzebuje go $O(\Delta m)$). Niestety, uzyskiwane tą drogą kolorowania, choć spełniają wymienione oszacowania – nie muszą być optymalne.

Zadanie 5.8.
Zaprojektuj procedurę kolorującą wierzchołkowo w sposób optymalny grafy o maksymalnym stopniu nie przekraczającym 3. Wymaga się czasu pracy $O(n^2)$.

Rozwiązanie.
Najpierw rozbijamy graf na składowe spójności oraz liczymy maksymalny stopień Δ dla każdej z nich. Następnie kolorujemy każdą optymalnie. Jeśli $\Delta=0$, to jest to wierzchołek stopnia 0 i dostaje on kolor 1. Dla $\Delta=2$ mamy do czynienia ze ścieżką lub cyklem, którego kolejne wierzchołki można kolorować naprzemiennie barwami 1 i 2. W przypadku składowej w postaci cyklu nieparzystego należy użyć koloru 3 dla ostatniego wierzchołka. Tak czy inaczej, kolorowania rozważone do tej chwili można uzyskać w czasie $O(n+m)$. Pozostaje przypadek $\Delta=3$. Sprawdzamy, czy składowa jest grafem pełnym K_4 i jeśli tak – kolorujemy ją czterema barwami (czas $O(1)$ na składową, czyli $O(n+m)$ dla całego grafu). Jest to jedyny przypadek zmuszający nas do użycia aż tylu kolorów. Inne składowe testujemy pod kątem dwudzielności i jeśli wynik jest TAK – kolorujemy odpowiednio barwami 1 i 2. W przeciwnym wypadku używamy dla nich algorytmu z dowodu twierdzenia Brooksa, który użyje dokładnie trzech barw.

Zadanie 5.9*.
Dany jest digraf n–wierzchołkowy D (graf o krawędziach skierowanych bez pętli) w postaci macierzy sąsiedztwa:

```
D: array[1..n,1..n] of boolean;
```

Napisz procedurę wykonującą się w czasie $O(n)$, sprawdzającą, czy digraf ten zawiera *ujście*, tj. wierzchołek, z którego nie wychodzi żadna krawędź, a wchodzą krawędzie ze wszystkich innych wierzchołków.

Rozwiązanie.
Tym razem graf jest skierowany, a zatem macierz D nie musi być symetryczna. Najprostszą metodą byłoby policzenie wszystkich stopni wejściowych i wyjściowych wierzchołków, a następnie sprawdzenie, czy dla któregoś z nich pierwsza wielkość wynosi n–1, a druga 0.

```
function ujscie(D: array [1..n,1..n] of boolean):boolean;
var
    i,j,deg_in,deg_out:integer;
begin
    for i := 1 to n do begin
        deg_in := 0;
        deg_out := 0;
        for j := 1 to n do begin
            if D[i,j] then deg_out++;
            if D[j,i] then deg_in++;
        end;
        if deg_in = n–1 and deg_out = 0 return true;
    end;
    return false;
end;
```

Niestety, czas w oczywisty sposób wynosi $O(n^2)$. Zastosujemy więc inne podejście. Zauważmy, że dla dowolnych dwóch różnych wierzchołków *i*,*j* istnienie krawędzi z *i* do *j* oznacza, że *i* nie może być ujściem (gdyż wiedzie z niego krawędź), zaś brak krawędzi implikuje, że *j* nie może być ujściem. Spróbujmy więc zorganizować przeszukiwanie macierzy *D* tak, by każdy dostęp do niej wykluczał możliwość „bycia ujściem" dla pewnego wierzchołka. W ten sposób pozostanie nam tylko jeden wierzchołek mogący stanowić ujście, co da się zweryfikować w czasie $O(n)$. Mamy zatem:

```
function ujscie2(D: array [1..n,1..n] of boolean):boolean;
var
        i,a,b:integer;
begin
        a := 1;
        b := 2;
        for i := 1 to n–2 do
                if D[a,b] then a := max(a,b)+1
                else b := max(a,b)+1;
        if D[a,b] then a := b;
        for i := 1 to n do
                if  i ≠ a and not(D[i,a] and not(D[a,i]))
                        then return false;
        return true;
end;
```

Oczywistym jest, że czas działania procedury wynosi $\Theta(n)$, uzasadnijmy zatem jej poprawność. Liczby *a* i *b* w kolejnych obiegach pierwszej pętli **for** oznaczają numery dwóch różnych wierzchołków, takie że wierzchołki 1,...,*max*(*a*,*b*) inne niż te dwa nie mogą być ujściami. Przy tym za każdym obiegiem liczba *max*(*a*,*b*) zwiększa się o 1. Dlatego po *n*–2 przejściach otrzymujemy w *a* i *b* numery jedynych dwóch wierzchołków, które mogą być ujściami. Kolejna instrukcja **if** eliminuje jeden z nich – od tej pory jedynie *a* może stanowić ujście, co sprawdzamy w ostatniej pętli procedury.

Aby jednoznacznie określić postać digrafu, czyli uzyskać pełną informację o posiadanych przez niego krawędziach należy odczytać $n(n–1)$ pól jego macierzy sąsiedztwa. Tymczasem powyższa procedura dokonuje nie więcej, niż $3n–3$ dostępów do tablicy. Zatem znalezienie ujścia w digrafie (lub wykluczenie jego istnienia) nie wymaga nawet znajomości wszystkich jego łuków.

Problem istnienia ujścia w digrafie można rozwiązać wykonując jeszcze mniej, bo tylko $3n–\lfloor \log_2 n \rfloor –3$ dostępów do tablicy sąsiedztwa. Własność digrafu nazwiemy *nietrywialną*, gdy zarówno zbiór grafów, dla których jest ona prawdziwa, jak i jego dopełnienie są nieskończone. Dla przykładu „dany digraf ma mniej, niż 100 wierzchołków" jest własnością trywialną. Udowodniono, że rozstrzygnięcie dowolnej nietrywialnej własności dla digrafu *n*–wierzchołkowego wymaga nie mniej, niż $2n–4$ dostępów do tablicy. Jeszcze bardziej „wymagające" są tzw. *własności monotoniczne*, czyli takie, dla których zachodzenie dla digrafu *n*–wierzchołkowego implikuje jej zachodzenie dla wszystkich jego *n*–wierzchołkowych naddigrafów. I tak np. własnością monotoniczną jest istnienie cyklu Hamiltona, zaś niemonotoniczne jest istnienie ujścia. Wykazano, że weryfikacja dowolnej nietrywialnej własności monotonicznej wymaga co najmniej $n^2/16$ odczytów z tablicy sąsiedztwa.

Dla danego grafu skierowanego *D* (z pętlami, lub bez) jego domknięciem przechodnim nazywamy naddigraf *D** o tym samym zbiorze wierzchołków, w którym z wierzchołka *u* do *v* prowadzi łuk wtedy i tylko wtedy, gdy w *D* istnieje marszruta (co najmniej jednokrawędziowa) z *u* do *v*.

Zadanie 5.10.
Dany jest digraf *n*–wierzchołkowy *D* w postaci macierzy sąsiedztwa:

D: **array**[1..*n*,1..*n*] **of integer**;

Określ, jaką minimalną wartość powinna przyjmować funkcja *f*(*k*), aby kod poniższy zwracał domknięcie przechodnie podanego grafu. Wyznacz czas pracy tak zbudowanej procedury.

```
function domkniecie(D: array [1..n,1..n] of integer): array [1..n,1..n] of integer;
var
        i,j:integer;
        A,B:array [1..n,1..n] of integer;
begin
        A:=D;
        for k  := 1 to f(n) do begin
                B := A*A; // wg Strassena
                A := A+B; // dodajemy wartości w odpowiadających sobie polach tabel
        end;
        for i := 1 to n do
                for  j:= 1 to n do
                        if A[i,j] ≠ 0 then A[i,j] := 1;
        return A;
end;
```

Rozwiązanie.
Tym razem digraf przekazywany jest w postaci tablicy liczb typu **integer**, w której każda wartość niezerowa oznacza istnienie odpowiedniego łuku. Marszruta zaś tym różni się od prostej drogi, że można w niej kilkakrotnie odwiedzać te same wierzchołki.

Trzeba więc zauważyć, że macierz *A* po *k*–tym przejściu przez pierwszą pętle **for** zawiera w polu *A*[*i*,*j*] liczbę niezerową wtedy i tylko wtedy, gdy istnieje w *D* marszruta z *i* do *j* długości $\leq 2^k$. Tezę tą wykażemy przez indukcję względem jej obiegów. Faktycznie, jest to prawda przed dojściem do pętli (możemy wtedy przyjąć *k*=0, bowiem istnienie marszruty długości 1 jest równoważne istnieniu pojedynczej krawędzi). Dalej, z definicji iloczynu macierzy w polu $A_k[i,j]$ macierzy A_k będącej wartością *A* po *k*–tym przejściu pętli może pojawić się liczba niezerowa tylko w dwóch przypadkach:

- gdy $A_{k-1}[i,j]$ było niezerowe, lub
- gdy istniał wierzchołek *x*, taki że $A_{k-1}[i,x]$ oraz $A_{k-1}[x,j]$ były niezerowe.

Z drugiej strony istnieje marszruta długości $l \leq 2^k$ z *i* do *j* wtedy i tylko wtedy, gdy:

- $l \leq 2^{k-1}$, lub
- istnieje wierzchołek *x*, do którego można dojść marszrutą długości nie przekraczającej 2^{k-1} wychodząc z *i* oraz z którego można dojść taką marszrutą do *j*.

Na mocy założenia indukcyjnego odpowiednie przypadki są sobie równoważne, co dowodzi tezy.

A zatem aby *A* odpowiadała domknięciu przechodniemu *D*, musi ona uwzględniać wszystkie możliwe marszruty w *D*. Jednak marszrutę o długości większej od *n* zawsze można skrócić (któryś z wierzchołków występujący w środku marszruty musi ulec powtórzeniu), dlatego wystarczy ograniczyć się do najmniejszego *k*, takiego że $2^k \geq n$. Z drugiej strony chcąc uzyskać domknięcie przechodnie grafu nie możemy ignorować marszrut długości *n* – przykład *n*–wierzchołkowego cyklu skierowanego pokazuje, że dopiero przejście *n* krawędzi pozwala powrócić do wierzchołka początkowego. Dlatego najmniejszym $f(n)$, dla którego program będzie działał poprawnie jest $f(n)=\lceil \log_2 n \rceil$. Procedura wymaga więc $O(n^{\log_2 7} \log n)$ czasu.

Zadanie 5.11*.
Dany jest digraf *n*–wierzchołkowy *D* w postaci macierzy sąsiedztwa:

D: **array**[1..*n*,1..*n*] **of integer**;

Sprawdź, co wylicza poniższa funkcja rekurencyjna oraz oszacuj jej czas działania.

```
function X(D: array [1..n,1..n] of integer; p,k,l:integer):boolean;
var
    i:integer;
begin
    if l = 1 then return D[p,k]
    else for i := 1 to n do
        if X(D,p,i,⌊l/2⌋) and X(D,i,k,⌈l/2⌉) then return true;
    return false;
end;
```

Rozwiązanie.
Funkcja sprawdza, czy od wierzchołka *p* do *k* można przejść marszrutą długości *l*. Fakt ten zweryfikujemy przez indukcję względem *l*. Dla *l*=1 zachodzi pojedyncze sprawdzenie istnienia łuku od *p* do *k*. Dla większego *l* sprawdzamy istnienie wierzchołka *i*, dla którego $X(D,p,i,\lfloor l/2 \rfloor)$ oraz $X(D,i,k,\lceil l/2 \rceil)$ zwrócą true, co zgodnie z założeniem indukcyjnym jest równoważne istnieniu marszruty z *p* do *i* o długości $\lfloor l/2 \rfloor$ oraz marszruty z *i* do *k* długości $\lceil l/2 \rceil$. Ale $l = \lfloor l/2 \rfloor + \lceil l/2 \rceil$ i połączenie obu marszrut da szukane przejście z *p* do *k* o długości *l*. Odwrotnie, jeżeli z *p* do *k* można przejść marszrutą długości *l*, to wystarczy przyjąć za *i* wierzchołek mijany jako $\lfloor l/2 \rfloor$–ty na tej marszrucie. Równoważność powyższa kończy dowód poprawności.

Przystąpimy teraz do oszacowania czasu działania programu. Jest to funkcja dwóch zmiennych *n* i *l*, przy czym rekursja przebiega względem *l*:

$T(n,1) = 1,$
$T(n,l) = n(T(n,\lfloor l/2 \rfloor)+T(n,\lceil l/2 \rceil)+g)$ dla $l>1,$

gdzie *g* jest stałą. Dla *l* będących potęgami dwójki równanie upraszcza się do postaci:

$T(n,1) = 1,$
$T(n,l) = 2nT(n,l/2) + gn$ dla $l>1$

Rozwiązując rekursję po zmiennej l traktujemy n jako niezależny parametr:

$(T/gn)(n,1)=1/gn$
$(T/gn)(n,l) = 2n(T/gn)(n,l/2) + 1$ dla $l>1$

Dzielenie wykonano po to, aby na końcu drugiej równości uzyskać funkcję iloczynową. Mamy więc równanie typu „dziel i rządź" ze stałymi $a = 2n$ i $b = 2$ oraz $c=1/gn$, $d(l)=1$. Stosując znaną formułę na dokładną postać rozwiązania przy $l = 2^m$ otrzymujemy:

$$\frac{T(n,l)}{gn} = \frac{(2n)^m}{gn} + \sum_{j=0}^{m-1}(2n)^j = \frac{(2n)^m}{gn} + \frac{(2n)^m - 1}{2n-1} = O((2n)^m / n),$$

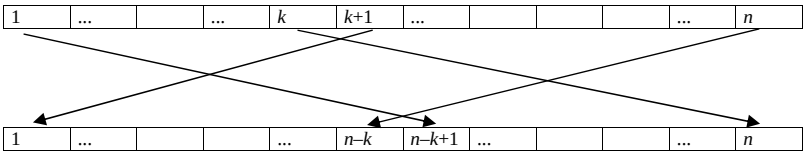
czyli rozwiązanie rekursji jest postaci: $T(n,l) = O((2n)^{\log_2 l}) = O(l^{1+\log_2 n})$. Wyszła nam funkcja superwielomianowa, czyli wystarczająco szybko rosnąca, by uzyskane oszacowanie wartościami na potęgach dwójki mogło być obarczone błędem. Dlatego powinniśmy ostatecznie napisać (dlaczego właśnie tyle?):

$$T(n,l) = O(2^{1+\log_2 n} l^{1+\log_2 n}) = O(nl^{1+\log_2 n}).$$

O programie mówimy, że *działa on w miejscu*, jeżeli ilość dodatkowej pamięci, jaką używa on w trakcie swego działania (poza pamięcią przekazanych doń na wejściu danych) można ograniczyć z góry przez stałą niezależną od rozmiaru tych danych. Inaczej mówiąc, program wymaga $O(1)$ pamięci dodatkowej.

Nieco inczej określamy *złożoność pamięciową algorytmu* – jest to maksymalne zużycie pamięci (zarówno tej, w której przekazano dane do programu, jak i pamięci dodatkowej) przez algorytm, wyrażone w funkcji rozmiaru wprowadzonych danych (podobnie, jak dla złożoności czasowej). Złożoność pamięciową często podaje się w postaci oszacowania asymptotycznego.

Zadanie 5.12*.
Opracuj procedurę dokonującą przesunięcia cyklicznego danej tablicy długości n o k elementów. Wymaga się złożoności obliczeniowej $O(n)$ i działania w miejscu. Zakładamy, że $0<k<n$.



Rozwiązanie.
Rozpoczniemy od kodów niepoprawnych. Najprostsza procedura mogłaby polegać na zapamiętaniu początkowego fragmentu tablicy:

```
procedure shift(var tab: array[1..n] of T; k:integer);
var
        i:integer;
begin
        zaalokuj pamięć dla pom: array [1..k] of T;          // tablica pomocnicza
        for i := 1 to k do
                pom[i] := tab[i];
        for i := k+1 to n do
                tab[i–k] := tab[i];
        for i := 1 to k do
                tab[n–k+i] := pom[i];
        zwolnij pom;
end;
```

Wprawdzie liczba operacji jest proporcjonalna do n, ale zużycie pamięci dodatkowej $O(k)$ jest za duże. Można też wykorzystać powyższą procedurę do przesunięcia o 1 (zużywając $O(1)$ pamięci), powtarzając to k razy, wtedy jednak czas działania wyniesie aż $O(nk)$. Zastosujemy inną metodę. Użyjemy pojedynczej zmiennej *pom* typu **T** zapamiętującej zawartość pola tablicy o numerze i (od którego rozpoczniemy przestawianie). Niech $j \neq i$ będzie numerem pola, którego zawartość po przesunięciu cyklicznym zostaje przemieszczona do i. Przepisujemy tę wartość do *tab*[i] i szukamy pola, które ma zostać przepisane na pozycję j, itd. Liczba pól tabeli jest skończona, a zatem po pewnym czasie trafimy na pierwsze takie pole y, które ma zostać przepisane do jakiegoś x, takie że y było już rozważane. Chwila zastanowienia prowadzi do wniosku, że $y=i$, jako że każde pole ma jednoznacznie wyznaczone pole docelowe. Krótko mówiąc, możemy zużywając $O(1)$ pamięci poustawiać na właściwych miejscach zawartość wszystkich pól należących do tego samego cyklu permutacji przesunięcia cyklicznego, co pole o numerze i:

```
procedure SubShift(var tab: array[1..n] of T; k,i:integer);
var
        i,j1,j2:integer;
        pom:T;

        function next(a:integer):integer;
        // znajdź numer pola, które po przesunięciu trafi do tab[a]
        var b:integer;
        begin
                b := a+k;
                if b > n then b := b–n;
                return b;
        end;

begin
        pom := tab[i];
        j1 := i;
        j2 := next(j1);
        while j2 ≠ i do begin
                tab[j1] := tab[j2];
                j1 := j2;
                j2 := next(j1);
        end;
        tab[j1] := pom;
end;
```

Zmienne $j1$ i $j2$ przechowują w każdym obiegu pętli dwa kolejne numery pól, między którymi ma nastąpić przepisanie wartości. Czas działania tej podprocedury jest równy liczbie ustawianych na właściwych miejscach pól. Jednak nie wszystkie pola tablicy muszą znajdować się w cyklu zawierającym pole i, niektóre mogą nie zostać przemieszczone, podczas gdy spełnienie warunku $j2=i$ zakończy pętlę **while**. Przykładowo, przesuwając tablicę długości $n=6$ o $k=3$ po rozpoczęciu od $i=2$ przestawimy jedynie pola 2 i 5. Musimy więc znaleźć sposób na odszukanie numerów pól znajdujących się w różnych cyklach permutacji i wykonanie naszej podprocedury dla każdego cyklu dokładnie jeden raz. Wtedy każde pole zostanie przemieszczone raz i złożoność obliczeniowa wyniesie $O(n)$ przy zachowaniu warunku działania w miejscu. W tym celu zauważmy, że dla kolejnych numerów pól w cyklu zachodzi:

$$j2=j1+k \text{ lub } j2=j1+k-n,$$

czyli $j2 \equiv j1 \bmod \text{NWD}(n,k)$, tzn. numery pól leżących w tym samym cyklu (czyli przemieszczanych w jednym wywołaniu procedury *sub_shift*) muszą przystawać do siebie mod NWD(n,k). Zatem uruchamiając procedurę *sub_shift* dla kolejnych $i=1,...,\text{NWD}(n,k)$ będziemy za każdym razem przemieszczać inny zbiór pól. Czy jednak tym razem przesuniemy wszystkie pola tablicy? Policzmy ilość l pól przemieszczanych w jednym cyklu, począwszy od pola o indeksie i. Na mocy powyższych równości mamy:

$$j2 \equiv j1+k \bmod n,$$

itd. po l krokach wracamy do i czyli:

$$i+lk \equiv i \bmod n.$$

Stąd $lk \equiv 0 \bmod n$, czyli lk musi być wielokrotnością n. Najmniejsze naturalne l spełniające ten warunek daje oczywiście $lk=\text{NWW}(n,k)$, czyli ilość elementów przestawianych w jednym wywołaniu procedury *SubShift* wynosi $l=\text{NWW}(n,k)/k$.

Możemy teraz odpowiedzieć twierdząco na postawione wcześniej pytanie. Ilość pól przemieszczonych dla NWD(n,k) wywołań *SubShift* wyniesie:

$$l\text{NWD}(n,k) = \text{NWD}(n,k)\text{NWW}(n,k)/k = n$$

(korzystamy z tożsamości $\text{NWD}(n,k)\text{NWW}(n,k)=nk$ łatwej do wykazania po rozważeniu rozkładów na czynniki pierwsze liczb n i k), a zatem wszystkie pola *tab* zostaną przemieszczone. Ostatecznie kod procedury ma postać:

```
procedure shift(var tab: array[1..n] of T; k:integer);
var
        i:integer;
begin
        for i := 1 to Euklides(n,k) do
                SubShift(tab,k,i);
end;
```

Funkcja *Euklides* została opisana w zadaniu 3.12.

Zadanie 5.13*.
Dana jest macierz kwadratowa liczb rzeczywistych *M* o rozmiarze *n*. Podaj wielomianową procedurę obliczającą wyznacznik *det M*. Zakładamy, że dostępne są operacje arytmetyczne na liczbach zmiennoprzecinkowych wykonywane w pojedynczym kroku, oraz pomijamy istnienie błędów w takich obliczeniach.

Rozwiązanie.
Dla macierzy $M=[a_{ij}]_{i,j=1...n}$ zgodnie z definicją wyznacznik wynosi:

$$\det(M) = \sum_{\pi \in S_n} (-1)^{\operatorname{sgn}\pi} a_{1\pi(1)} \cdot \ldots \cdot a_{n\pi(n)},$$

gdzie sumowanie odbywa się po wszystkich permutacjach π zbioru $\{1,...,n\}$. Jest ich jednak $n!$ i gdyby oprzeć się bezpośrednio na powyższym wzorze, algorytm miałby złożoność superwykładniczą. Zamiast tego skorzystamy z pewnych własności wyznacznika:

1. wyznacznik macierzy zmienia znak przy zamianie dwóch wierszy lub dwóch kolumn,
2. wyznacznik nie zmienia się, jeśli do pewnego wiersza dodamy kombinację liniową innych wierszy (to samo dotyczy kolumn),
3. zachodzi wzór: $\det(M) = \sum_{i=1}^{n} (-1)^{i+j} a_{ij} \det(M_{ij})$, gdzie $j \in \{1,...n\}$ a M_{ij} jest macierzą kwadratową wymiaru $n-1$ powstałą z M przez usunięcie i–tego wiersza i j–tej kolumny (można dokonać analogicznego rozwinięcia względem wiersza),
4. jeżeli w macierzy występuje wiersz lub kolumna zawierająca same zera, wówczas wyznacznik jest zerem (wniosek z powyższego),
5. wyznacznik macierzy rozmiaru 1 jest równy jej jedynemu elementowi.

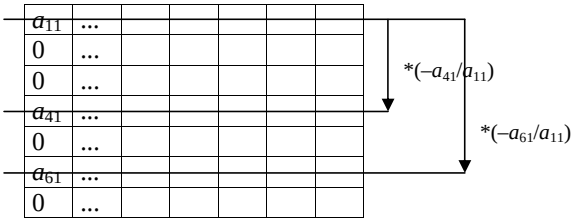
Algorytm może wyglądać następująco. W przypadku macierzy rozmiaru 1 zwracamy wartość jej jedynego elementu. Mając zaś daną macierz M rozmiaru $n>1$ przeglądamy jej pierwszą kolumnę w poszukiwaniu elementu niezerowego a_{i1} (jeżeli takiego nie ma – wyznacznik jest 0). Jeżeli $i\neq 1$ – zamieniamy wiersze i–ty i pierwszy w M zapamiętując, że wynikowy wyznacznik trzeba będzie pomnożyć przez –1. Dzięki temu możemy przyjąć, że $a_{11}\neq 0$. Dalej dla każdego innego j, takiego że $a_{j1}\neq 0$ dodajemy do j–tego wiersza wiersz pierwszy przemnożony przez liczbę $-a_{j1}/a_{11}$. Operacje te nie zmieniają wyznacznika macierzy, a sprawiają, że cała pierwsza kolumna macierzy, z wyjątkiem narożnego elementu a_{11} będzie zawierać zera. Dla tak przetworzonej macierzy M zachodzi więc $det(M)=a_{11}det(M_{11})$ na mocy wzoru 3, ale M_{11} ma wymiar $n-1$ i możemy dlań przez rekursję zastosować tą samą procedurę. Oto schemat działania pierwszego przebiegu procedury dla pewnej macierzy A:

A:

0	...					
0	...					
a_{31}	...					
a_{41}	...					
0	...					
a_{61}	...					
0	...					

$det(A)= -det(B)$

B:



$det(B)=det(C)$

C:

a_{11}	...					
0	...					
0	...					
0	...					
0	...					
0	...					
0	...					

$det(C)=a_{11}det(C_{1,1})$

A oto pełny kod funkcji:

```
function wyznacznik(M:array[1..n,1..n] of real):real;
var     i,j,sgn:integer;
        pom:real;
begin
        if n = 1 then return M[1,1];            // macierz 1×1
        i := 1;
        while i ≤ n and M[i,1] = 0 do
                i++;
        if i = n+1 then return 0;       // pierwsza kolumna zawiera tylko zera
        if i ≠ 1 then                   // zamiana wierszy
                for j := 1 to n do begin
                        pom := M[i,j];
                        M[i,j] := M[1,j];
                        M[1,j] := pom;
                end;
        if i ≠ 1 then sgn := –1 else sgn := 1;
        for j := 2 to n do
                if M[j,1] ≠ 0 then begin
                        pom := M[j,1]/M[1,1];
                        for i := 1 to n do M[j,i] := M[j,i] – M[1,i]*pom;
                end;
        return sgn*M[1,1]*wyznacznik(M11);
end;
```

Oszacujmy czas pracy procedury w funkcji n. Operacje związane z obróbką macierzy przed wywołaniem podprocedury rekurencyjnej zajmują $O(n^2)$. Zatem możemy zapisać równanie na czas:

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

którego rozwiązaniem oczywiście jest: $T(n)=\sum_{i=0}^{n} i^2 = n(n+\frac{1}{2})(n+1)/3$, a zatem procedura wykona się w czasie $O(n^3)$.

6. Sortowanie

Porządkowanie elementów danego ciągu stanowi jedną z najczęściej wykonywanych operacji w obecnie konstruowanych algorytmach. Rozdział ten poświęcimy analizie złożoności czasowej i pamięciowej algorytmów sortujących, tak metodami iteracyjnymi jak i rekurencyjnymi. Przedstawimy też pewne oszacowania dolne na złożoność „najszybszego możliwego" uniwersalnego programu sortującego.

Rozważamy algorytmy sortujące ciąg obiektów pewnego typu ***T*** (mogą to być liczby, napisy, rekordy bazy danych, ...), na którym zdefiniowano pewien porządek liniowy (elementy typu ***T*** można porównywać operatorami $<$, $\leq$, $\geq$, $>$, porównanie jest operacją o czasie trwania $O(1)$), lub pseudoporządek (w sensie braku antysymetrii – sytuacja taka może mieć miejsce w bazie danych rekordów porządkowanych według pola nie będącego kluczem). Zakładamy, że sortowanie elementów porządkuje je w kolejności niemalejącej.

Zadanie 6.1.
Podaj złożoność czasową i pamięciową algorytmu sortowania metodą maksimum:

```
procedure SortMax(var tab: array[1..n] of T);
var
        i,j,k:integer;
        pom:T;
begin
        for i := n downto 2 do begin
                k := 1;
                for j := 2 to i do
                        if tab[k] < tab[j] then k := j;
                if k ≠ i then begin
                        pom := tab[i];
                        tab[i] := tab[k];
                        tab[k] := pom;
                end;
        end;
end;
```

Rozwiązanie.
Operacją elementarną jest porównanie elementów tablicy w drugiej pętli. Jest ono wykonywane $\sum_{i=2}^{n}\sum_{j=2}^{i}1=\sum_{i=2}^{n}(i-1)=n(n-1)/2=\Theta(n^2)$ i tyle też wynosi złożoność obliczeniowa.

W naszym przypadku poza dostarczaną tablicą *tab* program korzysta jedynie ze zmiennych lokalnych *i*, *j* i *k* oraz *pom* – ich rozmiar jest stały, niezależny od wielkości danych (czyli tablicy *tab*), a zatem zużycie pamięci dodatkowej jest $O(1)$. Program ten działa w miejscu, choć jego złożoność pamięciowa wynosi $O(n)$, czyli jest liniowa (pamiętamy, że szacując złożoność pamięciową wliczamy również pamięć, w której przekazano dane).

Zadanie 6.2*.
Dana jest procedura *X*(**var** *tab*: **array**[1..*n*] **of T**) pobierająca na wejściu tablicę elementów i pracująca według następującego schematu: iteracyjnie znajdowana jest para sąsiednich elementów spełniających warunek *tab*[*i*] > *tab*[*i*+1], po czym ich zawartość zostaje zamieniona. Proces ten powtarzamy tak długo, aż w tabeli nie będzie występować żadna taka para, czyli wektor zostanie posortowany. Takie określenie procedury nie jest jednoznaczne, bowiem w danej chwili może istnieć kilka par pól spełniających powyższy warunek – a specyfikacja nie rozstrzyga, którą z nich wybrać do zamiany. Wykaż, że dla danej wejściowej tablicy *tab* liczba wykonanych podczas sortowania zamian jest niezależna od strategii wyboru miejsca zamiany elementów sąsiednich. Jaka jest maksymalna liczba zamian konieczna do wykonania sortowania tablicy długości *n*?

Rozwiązanie.
Rozważmy zbiór wszystkich nieuporządkowanych par indeksów *k* i *l* łamiących „zasadę posortowania" jako funkcję aktualnego stanu tablicy:

$$R(tab: \textbf{array}[1..n]\ \textbf{of T}) = \{\{k,l\}: k < l \wedge tab[k] > tab[l]\}.$$

Oczywiście $R=\varnothing$ wtedy i tylko wtedy, gdy tablica jest posortowana. Zauważmy, że zamiana wykonana na polu o numerze *i*, dla którego zachodziło *tab*[*i*] > *tab*[*i*+1] zmniejsza ilość elementów zbioru *R* dokładnie o 1 bowiem:

- pary {*k*,*l*} nie mające wspólnych elementów z {*i*,*i*+1} nie zmieniają swojej przynależności (lub nieprzynależności) do *R*,
- podobnie ma się rzecz z parami {*i*,*l*} dla *l* > *i*+1 oraz {*k*,*i*+1} dla *k* < *i*,
- para {*i*,*i*+1} jako jedyna „wypada" ze zbioru *R* po zamianie.

A zatem ilość zamian sąsiednich elementów musi być równa początkowej ilości elementów zbioru *R*(*tab*) i nie zależy od kolejności wyboru zamienianych pól.

Wielkość początkową zbioru *R* można oszacować z góry przez liczbę wszystkich par indeksów, czyli *n*(*n*–1)/2 i taką też postać ma *R* w przypadku podania sekwencji uporządkowanej odwrotnie. Liczba *n*(*n*–1)/2 stanowi więc maksymalną konieczną do wykonania liczbę zamian.

Procedurą sortującą w sposób opisany w powyższym zadaniu jest tradycyjne sortowanie bąbelkowe o złożoności $O(n^2)$:

```
procedure BubbleSort(var tab: array[1..n] of T);
var     i,j:integer;
        pom:T;
begin
        for i := n downto 2 do begin
                for j := 1 to i–1 do
                        if tab[j] > tab[j+1] then begin
                                pom := tab[j];
                                tab[j] := tab[j+1];
                                tab[j+1] := pom;
                end;
        end;
end;
```

Zadanie 6.3.
Jaki jest efekt działania poniższej procedury. Oszacuj jej złożoność obliczeniową.

```
procedure X(var tab: array[1..n] of T);
var
        i:integer;
        pom:T;
begin
        i := 1;
        while i < n do
                if tab[i] > tab[i+1] then begin
                        pom := tab[i];
                        tab[i] := tab[i+1];
                        tab[i+1] := pom;
                        i := 1;
                end
                else i++;
end;
```

Rozwiązanie.
Słownie działanie algorytmu można opisać następująco: "Począwszy od pierwszego przeglądaj kolejne pola danej tabeli, sprawdzając czy zawartość aktualnie widzianego pola jest większa od następnego. Jeżeli tak – zamień te dwa pola i rozpocznij przeglądanie od początku". A zatem mamy do czynienia z procedurą sortującą, pracującą zgodnie ze schematem z poprzedniego zadania.

Rozpoczniemy oszacowaniem górnym złożoności obliczeniowej. Maksymalna liczba zamian dwóch sąsiednich elementów tablicy spełniających warunek $tab[i] > tab[i+1]$, na mocy poprzedniego zadania nie przekroczy $n(n-1)/2$. Dalej, zgodnie z naszym słownym opisem algorytmu widzimy, że ilości dostępów do tablicy wykonanych przez program do pierwszej zamiany, pomiędzy dwoma kolejnymi zamianami oraz po ostatniej z nich są nie większe od n. Ten fakt i poprzedni dają nam górne oszacowanie złożoności procedury $O(n^3)$.

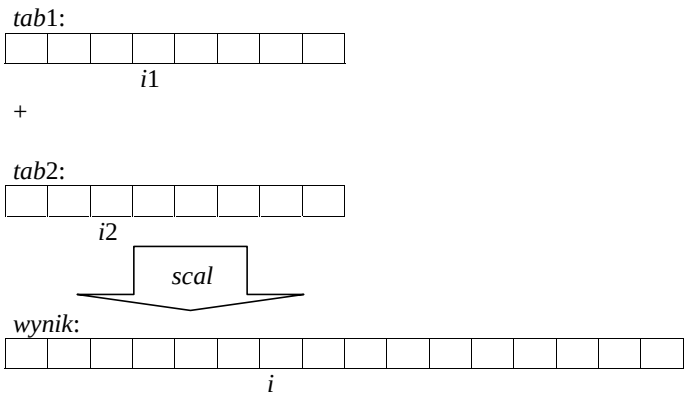
Dolne oszacowanie złożoności uzyskamy rozważając "pesymistyczny przypadek" ciągu odwrotnie posortowanego. Zauważmy, że przed każdą kolejną zamianą pewnych dwóch sąsiednich pól o numerach i oraz $i+1$ wykonuje się co najmniej i instrukcji **if**. Każdy z elementów znajdujących się początkowo w polach o numerach większych od $2n/3$ (jest ich $\lceil n/3 \rceil$) musi zostać przeniesiony do jednego z pól o numerach nie przekraczających $\lceil n/3 \rceil$, a więc wykonać co najmniej $n-2\lceil n/3 \rceil$ operacji "przemieszczenia" z pola o indeksie większym od $\lceil n/3 \rceil$ w kierunku malejących indeksów. Każda taka operacja musi zostać poprzedzona co najmniej $\lceil n/3 \rceil$ dostępami do tablicy. Znaleźliśmy więc co najmniej $\lceil n/3 \rceil^2(n-2\lceil n/3 \rceil) = \Omega(n^3)$ koniecznych do wykonania operacji, co razem z poprzednim wynikiem daje oszacowanie złożoności $\Theta(n^3)$. Procedura jest więc prosta, ale i czasochłonna.

Zadanie 6.4.
Napisz procedurę o liniowej złożoności czasowej, która dwa podane ciągi obiektów typu **T**, wcześniej (przed wywołaniem procedury) posortowane niemalejąco, scala w jeden ciąg posortowany.

Rozwiązanie.
Przepisanie kolejno do ciągu wyjściowego najpierw elementów jednego, a później drugiego ciągu oraz posortowanie całości jakimś prostym algorytmem standardowym miałoby złożoność kwadratową. Zastosujemy więc inne podejście:

```
procedure scal(tab1:array[1..n1] of T,tab2:array[1..n2] of T,var wynik:array[1..n1+n2] of T);
var
        i1,i2,i:integer;
begin
        i1 := 1;
        i2 := 1;
        i := 1;
        while i1 ≤ n1 and i2 ≤ n2 do
                if tab1[i1] ≤ tab2[i2] then begin
                        wynik[i] := tab1[i1];
                        i1++;
                        i++;
                end
                else begin
                        wynik[i] := tab2[i2];
                        i2++;
                        i++;
                end;
        while i1 ≤ n1 do begin
                wynik[i] := tab1[i1];
                i1++;
                i++;
        end;
        while i2 ≤ n2 do begin
                wynik[i] := tab2[i2];
                i2++;
                i++;
        end;
end;
```

Obie tablice wejściowe *tab*1 i *tab*2 są indeksowane własnymi zmiennymi *i*1 i *i*2, zaś tablica wynikowa jest indeksowana przez *i*. Każde dwa aktualne (czyli wskazywane przez indeksy w tablicach) elementy są porównywane i mniejszy z nich jest zapisywany do kolejnego pola tablicy wynikowej. Zwiększamy też o 1 indeks tablicy wejściowej, z której nastąpiło przepisanie tego elementu.



Powtarzamy powyższe aż do wyczerpania jednej z tablic i przepisujemy resztę drugiej tablicy na koniec tablicy *wynik*.

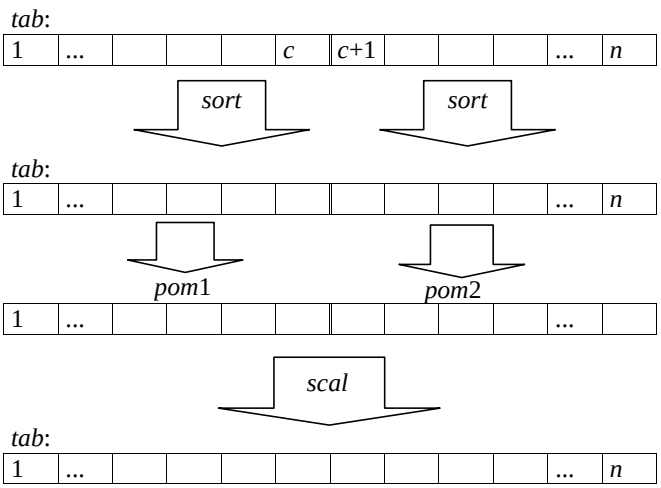
Operacją elementarną może być każda z operacji przepisania kolejnego pola jednej z tablic wejściowych do tablicy *wynik* (są one postaci **begin .... end**;). Liczba ich wykonań jest dokładnie równa długości tablicy wynikowej, czyli złożoność $\Theta(n1+n2)$ jest liniowa. Zużycie dodatkowej pamięci wynosi $O(1)$, gdyż dodatkowym zasobem pamięci w procedurze jest jedynie pamięć zmiennych *i*1, *i*2 i *i*.

Zadanie 6.5.
Podaj złożoność czasową i pamięciową procedury sortującej wykorzystującej poprzednie zadanie:

```
procedure sort(var tab:array[1..n] of T; a,b:integer);
// sortowanie fragmentu tablicy pomiędzy tab[a] i tab[b]
var
        c:integer;
begin
        if b > a then begin
                c := (a+b) div 2;
                sort(tab,a,c);
                sort(tab,c+1,b);
                zaalokuj pamięć pom1:array [1..c–a+1] of  T; pom2:array [1..b–c] of T;
                kopiuj tab[a...c]→pom1;
                kopiuj tab[c+1...b]→pom2;
                scal(pom1,pom2,tab[a…b]);
                zwolnij pom1; zwolnij pom2;
        end;
end;
```

Rozwiązanie.
Procedurę należy wywołać z parametrami $a=1$ i $b=n$. Stosuje ona rekursję. Najpierw sortowaną część wektora pomiędzy polami *tab*[*a*] i *tab*[*b*] dzielimy na dwie połówki indeksem *c*. Rekurencyjnie wywołujemy sortowanie obu połówek. Następnie alokujemy dwie tablice pomocnicze *pom*1 i *pom*2, do których kopiujemy obie posortowane połówki i korzystając z procedury *scal* scalamy ich zawartość w jeden posortowany ciąg zapisywany

z powrotem do *tab*. W tym momencie zaalokowana pamięć na tablice pomocnicze może zostać zwolniona. Schematycznie przebieg procedury można przedstawić następująco:



Niech $T(n)$ będzie czasem trwania sortowania wektora o długości n. Procedura nasza, poza wywołaniem samej siebie dwukrotnie na ciągach długości dwa razy mniejszej (z dokładnością do 1), wykonuje operacje o czasie stałym (obliczenie c, alokacja i dealokacja pamięci pomocniczej) oraz, co ważniejsze, o czasie liniowym $O(n)$ (kopiowanie elementów tablicy do tablic pomocniczych, procedura scalania). Tak więc możemy napisać równanie rekurencyjne na czas pracy:

$$T(n)=2T(\lceil n/2 \rceil)+n,$$

które można oszacować korzystając z równania typu „dziel i rządź” określonego na potęgach dwójki:

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

Mamy więc $a=2$, $b=2$, $d(n)=n$, co daje $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$. Procedura ta jest więc szybsza niż poprzednia.

Celem określenia zużycia pamięci zauważmy, że procedura poza zmiennymi lokalnymi *c*, *pom*1 i *pom*2 dodatkowo alokuje dwa wektory pomocnicze o łącznym rozmiarze równym rozmiarowi sortowanego kawałka tablicy. A zatem złożoność pamięciowa algorytmu jest liniowa $\Theta(n)$ i program ten nie działa w miejscu. Można powiedzieć, że za przyspieszenie algorytmu zapłaciliśmy wzrostem zużycia pamięci.

Warto zauważyć, że wywołane rekurencyjnie podprocedury również zajmują dodatkową pamięć o rozmiarze proporcjonalnym do rozmiaru ich danych, jest ona jednak zwalniana po ich zakończeniu, czyli przed alokacją pamięci przez procedurę wywołującą. Gdyby zmienić kolejność dwóch linii w powyższym kodzie na następującą:

zaalokuj *pom*1:**array** [1..*c*–*a*+1] **of T**; *pom*2:**array** [1..*b*–*c*] **of T**;
sort(*tab*,*a*,*c*); *sort*(*tab*,*c*+1,*b*);

---

wówczas zużycie pamięci przez podprocedurę wywołaną (i jej podprocedury) nakładałoby się na alokację wykonaną przez procedurę wywołującą. Ilość potrzebnej pamięci dodatkowej wzrosłaby ok. 2 razy – jednak nadal byłoby to $\Theta(n)$. Gdyby w ogóle zrezygnować z linii:

zwolnij *pom*1; zwolnij *pom*2;

zwalniając zasoby dopiero po posortowaniu całej tablicy, wtedy złożoność pamięciowa byłaby równa czasowej $\Theta(n \log n)$.

Algorytmy sortujące w czasie $O(n \log n)$ są określane mianem *szybkiego sortowania*. Powyższe rekurencyjne sortowanie przez połowienie jest więc szybkie, ma jednak tę wadę, że wymaga pamięci podręcznej o liniowym rozmiarze. Klasyczne *sortowanie kopcowe* również jest szybkie, ale działa w miejscu. Natomiast popularny *quicksort* z punktu widzenia złożoności pesymistycznej jest gorszy – może bowiem wykonywać rzędu $\Omega(n^2)$ kroków.

Zajmiemy się teraz analizą dowolnej uniwersalnej procedury (szablonu) sortującej tablicę postaci takiej, jak w poprzednich zadaniach (**procedure** *sortuj*(**var** *tab*: **array**[1..*n*] **of T**)). Procedura nie może czerpać żadnych informacji o obiektach typu **T**, poza porównaniem ich ze sobą. Można alokować zmienne typu **T** oraz kopiować ich wartości między sobą w całości, są to jednak jedyne obok porównań operacje dopuszczalne dla tych obiektów.

Zadanie 6.6.
Udowodnij, że uniwersalna procedura sortująca, zaimplementowana zgodnie z powyższymi regułami musi mieć złożoność obliczeniową typu $\Omega(n)$.

Rozwiązanie.
Jeżeli podamy na wejście procedury ciąg długości *n* posortowany odwrotnie:

n	*n*–1	...	...	...	...	2	1

wówczas każdy (być może z wyjątkiem środka tablicy) element przed zakończeniem działania programu będzie musiał zostać zapisany inną wartością. Zatem samych operacji zapisu musi być $\Omega(n)$.

A teraz trudniejsze:

Zadanie 6.7*.
Udowodnij, że taka uniwersalna procedura sortująca musi wymagać $\Omega(n \log n)$ czasu.

A zatem nasz rekurencyjny *sort*(...) jest przynajmniej w sensie asymptotycznym najszybszy z możliwych.

Rozwiązanie.
Tym razem zamiast zapisów do tablicy będziemy zliczać porównania elementów typu **T**. Wykonanie się programu sortującego na danej tablicy można rozpatrywać jako ciąg kolejnych stanów programu (zmieniających się wraz z wykonywaniem instrukcji) od sytuacji początkowej do końca. Możemy jednak rozważyć łącznie wszystkie takie sekwencyjne przebiegi algorytmu na $n!$ różnych danych wejściowych – wszystkich permutacjach ciągu 1,2,...,n skierowanego do posortowania. Poszczególne wykonania można powiązać w drzewo decyzyjne – jego korzeniem jest sytuacja początkowa, kiedy cała tabela została skierowana do posortowania. Sekwencja instrukcji wykonywanych przez program jest zdefiniowana jednoznacznie pomiędzy porównaniami zmiennych typu **T** – których wynik zależy już od konkretnego ciągu podanego na wejście na początku. Przy różnych danych wejściowych wyniki porównań na wszystkich etapach mogą być różne – co może kierować program na inne tory. A zatem drzewo nasze jest drzewem binarnym (tzn. rozwidlenia w wierzchołkach są stopnia 2), w którym wierzchołki rozwidlające przebiegi programu odpowiadają różnym wynikom instrukcji porównania. Liśćmi w drzewie decyzyjnym będą sytuacje końcowe, kiedy program kończy pracę i zwraca posortowaną tablicę. Musi ich być co najmniej $n!$, gdyż dla każdej z $n!$ sekwencji wejściowych program sortujący musi mieć inne wykonanie począwszy od sytuacji początkowej do końcowej (bo w inny sposób jest permutowana tablica). Głębokość drzewa binarnego (w naszym przypadku jest to maksymalna liczba rozwidleń na drodze od korzenia do liścia) o co najmniej x liściach jest nie mniejsza od $\lceil \log_2 x \rceil$ – co oznacza, że dla pewnej tablicy wejściowej program musi wykonać co najmniej $\lceil \log_2(n!) \rceil$ porównań obiektów typu **T**. Teraz wystarczy już tylko zauważyć, że $\log_2(n!)=\Theta(n\log n)$ – można tu wykorzystać oszacowanie silni z zadania 1.5: $(n/2)^{n/3}<n!<n^n$.

> Warunek narzucony na uniwersalną procedurę *sort*, nie pozwalający jej czerpać żadnych innych informacji o sortowanych obiektach, poza ich porównaniami, jest dość restrykcyjny.

Zadanie 6.8.
Zaprojektuj algorytm sortujący alfabetycznie tablicę znaków ze zbioru {a,b,c,d} o złożoności $\Theta(n)$.

Rozwiązanie.
Wystarczy policzyć wystąpienia poszczególnych liter i wypisać je w kolejności alfabetycznej i w odpowiednich ilościach:

```
procedure sort(var tab:array[1..n] of  char);
var
        i,j,la,lb,lc,ld:integer;

begin
        la := 0; lb := 0; lc := 0; ld := 0;
        for i := 1 to n do
                case tab[i] of
                        'a' : la++;
                        'b' : lb++;
                        'c' : lc++;
                        'd' : ld++;
```

```
            end;
        j := 1;
        for i := 1 to la do tab[j++] := 'a';
        for i := 1 to lb do tab[j++] := 'b';
        for i := 1 to ic do tab[j++] := 'c';
        for i := 1 to id do tab[j++] := 'd';
end;
```

Powyższy algorytm można uogólnić. Istnieje liniowa procedura sortująca alfabetycznie słowa dowolnej długości złożone z liter nad ustalonym skończonym alfabetem.

7. Klasy problemów decyzyjnych

Zaprezentowane w tym rozdziale informacje stanowią zarys wprowadzenia do formalnej teorii złożoności obliczeniowej. Szkicujemy dowód twierdzenia, mówiącego o istnieniu rozstrzygalnych problemów decyzyjnych, wymagających „dowolnie dużego" czasu obliczeń, definiujemy algorytmy niedeterministyczne oraz podstawowe klasy złożoności dla problemów decyzyjnych: klasę P zagadnień dających się rozwiązać w sposób efektywny oraz prawdopodobnie szerszą od niej klasę NP. Określamy też podstawowe własności redukcji wielomianowej, a także problemy NP–zupełne jako „maksymalne w klasie NP" względem tej redukcji.

Należy wspomnieć, że w pełni poprawny opis teorii złożoności nie jest możliwy bez wcześniejszego wprowadzenia uniwersalnego modelu obliczeń, a więc jakiejś formalnej definicji opisującej dowolny algorytm oraz przebieg jego działania. Tradycyjnie w rozważaniach teoretycznych rolę tę pełni maszyna Turinga, która jednak w swym działaniu dość istotnie odbiega od codziennej praktyki programowania współczesnych komputerów. W naszych rozważaniach poprzestaniemy więc na bardziej intuicyjnym rozumieniu algorytmu jako kodu programu wyrażonego w pewnym języku wysokiego poziomu. Dla zapewnienia ścisłości przeprowadzanych tu rozumowań należałoby więc zastanowić się nad definicją abstrakcyjnej maszyny bez ograniczeń pamięciowych, na której można uruchamiać dowolne poprawne kody napisane np. w języku PASCAL. Przy tym sposób działania maszyny dla każdej postaci danych i kodu powinien być jednoznacznie określony. Jest to zadanie, które pozostawiamy czytelnikowi – celem tego rozdziału jest bowiem zaprezentowanie podstawowych idei i szkiców rozumowań towarzyszących ważnym pojęciom teoretycznym, a nie ścisłe wnikanie w formalizmy rozmaitych modeli obliczeniowych.

Problemy decyzyjne są to problemy, dla których odpowiedzią przy każdym zestawie danych jest: TAK lub NIE. Dla przykładu: "czy dana liczba jest pierwsza?", "czy dany graf ma cykl Hamiltona?", "czy dana formuła jest tautologią?" – są problemami decyzyjnymi, zaś: "znajdź dzielniki danej liczby", "znajdź najdłuższy cykl w danym grafie" – takimi nie są.

Wśród problemów decyzyjnych istnieją takie, które można rozwiązać algorytmicznie i to algorytmem o złożoności wielomianowej (względem rozmiaru danych) – problemy te tworzą tzw. *klasę P*.

Ponadto dla problemów decyzyjnych (i tylko dla nich) zdefiniowano rozwiązujące je *algorytmy niedeterministyczne*. W normalnym (deterministycznym) algorytmie od chwili przekazania danych wszystkie kolejne kroki programu są jednoznacznie wyznaczone. W algorytmie niedeterministycznym (jedna z możliwych definicji) poza normalnymi (deterministycznymi) instrukcjami może występować funkcja *rand* – zwracająca losowo wartość 0 lub 1. Program może więc różnie zachowywać się, w zależności od wartości zwracanych przez kolejno wywoływane funkcje *rand* – zatem przy tym samym zestawie danych wejściowych algorytm niedeterministyczny może mieć wiele różnych przebiegów, a w rezultacie zwracać różne odpowiedzi. Ponieważ dopuszczalne są tylko odpowiedzi TAK/NIE, przeto przyjęto następującą interpretację zachowania algorytmu niedeterministycznego dla pewnego zestawu danych:

- jeżeli istnieje przebieg (przy pewnej sekwencji wartości zwracanych przez *rand*), dla którego algorytm zwraca TAK, wówczas odpowiedź algorytmu przy tym zestawie danych jest TAK,
- jeżeli przy każdym przebiegu dla tego zestawu danych algorytm zwróci NIE, wtedy przyjmujemy, że jego odpowiedź jest NIE.

Widzimy więc pewną asymetrię w interpretacji odpowiedzi – jeżeli choć jeden przebieg algorytmu niedeterministycznego zwraca TAK, to choćby wszystkie inne przebiegi zwracały odpowiedź NIE – interpretujemy zachowanie się algorytmu na TAK. Możemy myśleć o algorytmach niedeterministycznych, jakby nie zwracały one wartości TAK/NIE tylko TAK/NIEWIEM, przy czym dla każdego zestawu danych istnieje przebieg kończący się na TAK wtedy i tylko wtedy, gdy są to dane, dla których odpowiedź na nasz problem brzmi TAK (przy takiej interpretacji algorytmy te nigdy nie kłamią). Oczywiście aby algorytm niedeterministyczny był poprawnym algorytmem, musi on mieć własność stopu (dla żadnych poprawnych danych algorytm nie ma nieskończonego przebiegu przy żadnej sekwencji wartości *rand*), zaś jego złożoność obliczeniową traktujemy jako maksymalną liczbę operacji wykonanych na danych określonego rozmiaru rozpatrując tu maksimum po wszystkich możliwych przebiegach. Klasę problemów decyzyjnych (i tylko takich) rozwiązywalnych za pomocą wielomianowych algorytmów niedeterministycznych nazywamy *klasą NP*. Oczywiście $P \subseteq NP$, gdyż algorytm wielomianowy (deterministyczny) można uznać za szczególny przypadek algorytmu niedeterministycznego, w którym wywołania *rand* nie występują (lub występują, ale są ignorowane przez algorytm).

Zadanie 7.1.
Udowodnij, że problem 3–KOLOROWALNOŚCI GRAFU należy do NP poprzez opracowanie niedeterministycznego algorytmu o czasie działania $\Theta(n^2)$, który dla danej macierzy sąsiedztwa grafu G o n wierzchołkach sprawdzi, czy $\chi(G) \leq 3$.

Rozwiązanie.

```
function TrzyKolory(G: array[1..n,1..n] of boolean):boolean;
var
        i,j:integer;
        kolory:array[1..n] of integer;
begin
        for i := 1 to n do kolory[i] := rand+rand;
        for i := 1 to n–1 do
                for j := i+1 to n do
                        if G[i,j] and kolory[i] = kolory[j] then return false;
        return true;
end;
```

Oczywiście czas pracy jest $\Theta(n^2)$. W pierwszej pętli losowo przydzielamy wierzchołkom kolory 0, 1 lub 2. Następnie sprawdzamy, czy jest to poprawne pokolorowanie, tj. czy nie ma krawędzi łączącej wierzchołki o tych samych barwach. Łatwo zauważyć, że istnieje przebieg naszego algorytmu, przy którym zwrócone zostanie true wtedy i tylko wtedy gdy $\chi(G) \leq 3$ – jedynie wtedy bowiem instrukcje *rand* mają "możliwość" odgadnąć poprawne pokolorowanie trzema barwami.

Zadanie 7.2.
Udowodnij, że problem LICZBA ZŁOŻONA (sprawdzamy, czy dana liczba naturalna jest złożona) należy do NP.

Rozwiązanie.

```
function zlozona(n: integer):boolean;
var
        dzielnik,tmp:integer;
begin
        if n = 1 then return false;
        dzielnik := rand;
        tmp := n;
        while tmp > 1 do begin
                dzielnik := 2*dzielnik+rand;
                tmp := tmp div 2;
        end;
        if dzielnik > 1 and dzielnik < n and n mod dzielnik = 0 then return true;
        return false;
end;
```

Procedura polega na odgadnięciu liczby *dzielnik* o długości bitowej nie przekraczającej długości danej *n* (jej kolejne bity pobieramy z wywołań *rand*). Następnie sprawdzamy, czy wylosowana liczba jest faktycznie nietrywialnym podzielnikiem *n*. Jest oczywiste, że dla każdej liczby złożonej istnieje przebieg, dla którego algorytm zwróci true, zaś nigdy tak się nie zdarzy dla *n* pierwszego. Zasadniczy przyczynek do złożoności procedury ma pętla **while**, a liczba jej obiegów jest o 1 mniejsza od liczby bitów danej *n*. Czas pracy wynosi więc $O(\log n)$, czyli złożoność jest liniowa.

W 2002 roku wykazano, że problem ten w istocie należy do klasy P – skonstruowano algorytm o złożoności wielomianowej sprawdzający, czy podana liczba naturalna jest złożona, czy pierwsza. Nie jest jednak znana żadna wielomianowa procedura pozwalająca wyznaczyć dla danej liczby złożonej *n* jakikolwiek jej nietrywialny (tj. różny od 1 i *n*) dzielnik.

Pomiędzy problemami decyzyjnymi (nie algorytmami!) wprowadza się relację *redukcji wielomianowej*, lub *α–redukcji* (mogą to być dowolne problemy decyzyjne, nie koniecznie z NP). Jeżeli A i B są problemami decyzyjnymi, to istnieje wielomianowa redukcja z A do B wtedy i tylko wtedy, gdy istnieje procedura o wielomianowej złożoności (deterministyczna) pobierająca na wejście dowolne dane *x* dla problemu A i zwracająca dane *y* dla B, taka że odpowiedź na problem A z danymi *x* jest taka sama, jak odpowiedź na problem B z danymi *y*. Mówimy wtedy, że A jest wielomianowo redukowalny do B (zapis $A \alpha B$). W takiej sytuacji, gdybyśmy mieli algorytm rozwiązujący problem B, moglibyśmy rozwiązać redukujący się do niego problem A następująco: dane *x* dla A przekształcamy procedurą określającą redukcję do danych *y* dla B i uruchamiamy algorytm rozwiązujący B dla *y*, interpretując jego odpowiedź jak rozwiązanie A dla *x*.

Zadanie 7.3.
Wykaż, że α–redukcja jest przechodnia, tj. jeżeli problemy decyzyjne A, B i C są w relacjach $A \alpha B$ i $B \alpha C$, to $A \alpha C$.

Rozwiązanie.
Weźmy wielomianową procedurę *a*2*b* przekształcającą dane dla problemu A w dane dla B i określającą redukcję AαB, podobnie niech procedura *b*2*c* określa redukcję BαC. Redukcję AαC określamy następująco:

pobieramy z wejścia dane *x* dla A, wykonujemy na nich *a*2*b* otrzymując dane *y* dla B i na nich wykonujemy *b*2*c* uzyskując dane *z* dla C.

Jest oczywiste, że odpowiedź na problem A z danymi *x* będzie równoważna odpowiedzi dla C z danymi *z*. Pozostaje wykazać, że procedura ta jest wielomianowa. Niech *wa*2*b* i *wb*2*c* będą rosnącymi wielomianami szacującymi z góry ilość operacji procedur *a*2*b* i *b*2*c* w funkcji rozmiaru ich danych. Pierwsza część redukcji będzie trwała oczywiście nie dłużej niż *wa*2*b*(|*x*|), a druga *wb*2*c*(|*y*|). Ale |*y*|≤*wa*2*b*(|*x*|), gdyż pierwsza procedura redukująca nie może wypisać więcej bitów zapisując |*y*|, niż sama wykonuje operacji (zapis bitu jest też operacją). Ostatecznie czas trwania naszej redukcji można oszacować z góry przez wielomian długości danych: *wa*2*b*(|*x*|)+*wb*2*c*(*wa*2*b*(|*x*|)).

Zadanie 7.4.
Wykaż, że jeśli AαB i B$\in$P, to A$\in$P.

Rozwiązanie.
Mając dane *x* dla A bierzemy procedurę *a*2*b* określającą redukcję AαB (złożoność oszacowana z góry wielomianem *wa*2*b*) oraz deterministyczną procedurę wielomianową *b* (złożoność oszacowana wielomianem rosnącym *wb*) rozwiązującą problem B. Wykonujemy *a*2*b* na *x* otrzymując dane *y* dla *B*, które podajemy na wejście procedury *b*. Wynik interpretujemy jako odpowiedź dla A. Ilość operacji da się oszacować przez *wa*2*b*(|*x*|)+*wb*(*wa*2*b*(|*x*|)) – czyli procedura jest wielomianowa.

Zadanie 7.5.
Wykaż, że jeśli AαB i B$\in$NP, to A$\in$NP.

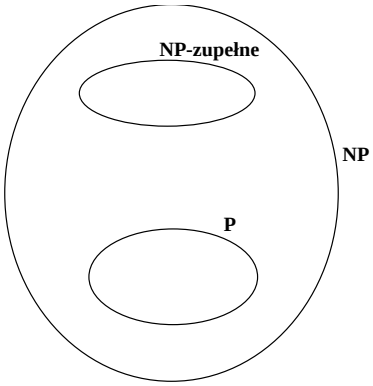
Rozwiązanie.
Postępujemy analogicznie, jak poprzednio, tyle że zamiast wielomianowej procedury deterministycznej *b*, bierzemy wielomianową niedeterministyczną.

Zadanie 7.6.
Udowodnij, że każdy problem A$\in$NP jest rozstrzygalny, co więcej, dla każdego problemu decyzyjnego z NP istnieje rozwiązujący go algorytm (deterministyczny!) o złożoności $o(2^{2^N})$, gdzie *N* jest rozmiarem danych.

Rozwiązanie.
Niech *w* będzie wielomianem rosnącym szacującym z góry liczbę operacji niedeterministycznego algorytmu rozwiązującego problem A. Aby sprawdzić, czy dla danych *x* algorytm ten odpowie TAK, należy rozważyć wszystkie jego możliwe przebiegi na tych danych, które zależą od wartości zwacanych przez kolejno wywoływane instrukcje *rand*. Ale liczba wywołań *rand* nie przekracza *w*(|*x*|), gdyż to ostatnie szacuje z góry ilość operacji w każdym przebiegu algorytmu niedeterministycznego na danych *x*, zaś wywołanie *rand* jest też operacją. Zatem jeżeli wygenerujemy wszystkie możliwe ciągi zero–jedynkowe o długości *w*(|*x*|) i dla każdego z nich wywołamy jeden raz nasz algorytm niedeterministyczny

uruchomiony na danych *x*, w którym kolejne *rand* będą zwracały jako wartości kolejne elementy danego ciągu zero–jedynkowego, wówczas mamy pewność, że zaobserwowaliśmy wszystkie możliwe przebiegi algorytmu niedeterministycznego na słowie *x*. Oszacujmy złożoność tej procedury. Wygenerowanie wszystkich ciągów zero–jedynkowych o długości $w(|x|)$ można przeprowadzić w czasie $w(|x|)2^{w(|x|)}$ i nie więcej zajmie wykonanie dla każdego z nich algorytmu niedeterministycznego o złożoności *w*. Zatem złożoność jest $w(N)2^{w(N)} = 2^{\log_2 w(N)}2^{w(N)} < 2^{2w(N)} = o(2^{2^N})$, gdyż $2w(N)=o(2^N)$ dla każdego wielomianu *w*.

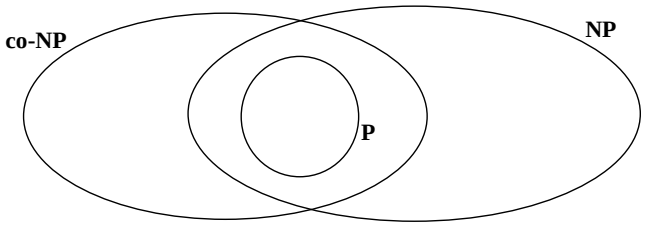
Problem decyzyjny X jest *NP–zupełny* wtedy i tylko wtedy, gdy każdy inny problem decyzyjny z NP daje się do niego wielomianowo zredukować, tj. dla każdego problemu decyzyjnego $A \in NP$ zachodzi $A \alpha X$. Zatem problemy NP–zupełne są w pewnym sensie "najtrudniejsze" w klasie NP. W szczególności na mocy powyższej definicji wszystkie problemy NP–zupełne są α–redukowalne między sobą jeden do drugiego. Zazwyczaj aby udowodnić, że problem $Y \in NP$ jest NP–zupełny – znajdujemy inny problem X, o którym już wiemy, że jest NP–zupełny i dowodzimy istnienie redukcji $X \alpha Y$. Wówczas każdy problem A z NP spełnia $A \alpha X$ (bo X jest NP–zupełny), więc z przechodniości α–redukcji (patrz wcześniejsze zadania) zajdzie też $A \alpha Y$, stąd Y musi być NP–zupełny. Bezpośrednio udowodniono NP–zupełność jedynie dla problemu SPEŁNIALNOŚCI (pytanie, czy dla danej formuły rachunku zdań istnieje podstawienie wartości logicznych do zmiennych zdaniowych, nadające całej formule wartość *true*). NP–zupełność wszystkich innych znanych takich problemów wykazano według powyższej procedury poprzez wielomianową redukcję ze spełnialności – redukcję bezpośrednią lub prowadzącą poprzez inne problemy pośrednie.



Co–problemem dla danego problemu decyzyjnego A (zapis co–A) nazywamy negację problemu A. Dla przykładu: "dany ciąg bitów jest kodem liczby pierwszej" i "dany ciąg bitów jest kodem liczby złożonej" są dla siebie co–problemami. Podobnie "dana macierz koduje graf hamiltonowski" i "...nie koduje grafu hamiltonowskiego". Oczywiście problem co–co–A=A (podwójna negacja). Jest oczywiste, że jeśli problem decyzyjny A należy do P, to również $co–A \in P$ – rozwiązuje go bowiem ta sama procedura wielomianowa, co A, tylko że jej odpowiedź negujemy. W przypadku klasy NP nie wiadomo, czy zachodzi podobna relacja – bowiem negacja odpowiedzi algorytmu niedeterministycznego nie musi prowadzić do algorytmu niedeterministycznego rozwiązującego co–problem (patrz interpretacja odpowiedzi algorytmu niedeterministycznego). Jest tak dlatego, że algorytm niedeterministyczny opowiada jedynie na pytanie o istnienie jakiegoś obiektu (którego postać można np.

odgadnąć korzystając z niedeterminizmu), a nie na pytanie o jego nieistnienie (wymagające sprawdzenia wszystkich możliwych ścieżek obliczeń).

Pytanie, czy klasa *co–NP*, czyli zbiór co–problemów dla problemów z NP jest równa NP stanowi jeden z głównych nierozstrzygniętych problemów teorii złożoności. Drugim, równie ważnym problemem otwartym jest pytanie o to, czy P=NP? Większość badaczy skłania się do przypuszczenia, że w obu przypadkach równość nie zachodzi, jak dotąd jednak hipotezy powyższe nie doczekały się dowodu.



Zadanie 7.7.
Udowodnij, że problem: "czy dana formuła zdaniowa jest tautologią" należy do co–NP.

Rozwiązanie.
Wystarczy pokazać, że jego negacja: "czy dana formuła zdaniowa nie jest tautologią" należy do NP. Ten ostatni problem decyzyjny oznaczmy przez B. Zauważmy, że operacja na danych *p* dla B (formuła zdaniowa) polegająca na jej zanegowaniu (formułę *p* zamieniamy na ~(*p*)) jest wielomianową redukcją z problemu B do problemu SPEŁNIALNOŚCI, jako że *p* nie jest tautologią wtedy i tylko wtedy, gdy ~(*p*) jest spełnialna. SPEŁNIALNOŚĆ należy jednak do NP (jest nawet NP–zupełna), zatem na mocy jednego z poprzednich zadań, również $B \in NP$.

Zadanie 7.8*.
Udowodnij, że gdyby pewien problem NP–zupełny A należał do co–NP, to mielibyśmy NP=co–NP.

Rozwiązanie.
Jeśli A jest NP–zupełny i $A \in$ co–NP, czyli istnieje problem $B \in NP$, którego A jest negacją, to z definicji NP–zupełności każdy problem $C \in NP$ spełnia $C \alpha A$ i A=co–B. To zaś oznacza, że co–$C \alpha B$ (redukcję wielomianową $C \alpha$co–B i co–$C \alpha B$ przeprowadza ta sama procedura). Zatem na mocy jednego z poprzednich zadań mamy co–$C \in NP$, czyli co–$NP \subseteq NP$. Zanegowanie problemów z klas znajdujących się po obu stronach tej inkluzji dowodzi inkluzji odwrotnej $NP=$co–co–$NP \subseteq$co–NP, co kończy dowód.

Raz jeszcze warto podkreślić silnie ograniczający warunek w definicji α–redukcji, nie pozwalający procedurze przekształcającej dane na modyfikowanie odpowiedzi zwrotnej z problemu, do którego redukujemy. Potocznie przyjmuje się, że jeżeli problem decyzyjny A może być łatwo rozwiązany przy użyciu procedury rozwiązującej problem decyzyjny B, to zachodzi $A \alpha B$. Nic bardziej błędnego! Jeżeli A jest NP–zupełny, zaś B to jego co–problem, to oba zagadnienia mają jednakową złożoność obliczeniową – dysponując algorytmem rozwiązującym B możemy rozstrzygnąć A (i odwrotnie) wywołując najpierw procedurę dla B, a później jeszcze jedną

operację (zanegowanie odpowiedzi). Nie jest to jednak α–redukcja i faktycznie redukowalność AαB na mocy poprzedniego zadania implikowałaby równość NP=co–NP. Można też bezpośrednio skonstruować problem decyzyjny A (niestety spoza klasy NP), dla którego Aαco–A na pewno nie zachodzi.

Rozważmy następujący problem:

Czy złożoności obliczeniowe wszystkich algorytmicznie rozstrzygalnych zagadnień decyzyjnych można oszacować z góry?

Szkic rozwiązania.
Sprecyzujmy najpierw postawione w treści pytanie. Weźmy dowolną funkcję określoną na zbiorze liczb naturalnych, o wartościach dodatnich, dla której istnieje algorytm obliczania jej wartości np. wielomiany, $n!$, $\exp(2^n)$, lub inną. Wykażemy istnienie problemu decyzyjnego rozstrzygalnego za pomocą pewnego deterministycznego algorytmu, dla którego jednak każdy taki algorytm będzie musiał mieć złożoność nie dającą się wyrazić jako $O(f(N))$. A zatem $O(f(N))$ będzie za małą liczbą operacji do rozstrzygnięcia problemu. Najpierw jednak sformalizujmy pojęcie problemu decyzyjnego. Można założyć, że danymi wejściowymi dla każdego problemu (po dokładnym sprecyzowaniu tego, jak rozumiemy dane wejście) jest dowolny skończony ciąg bajtów, na który nasz problem odpowiada TAK (akceptuje) lub NIE (odrzuca). Niech więc $\Sigma=\{0,...,255\}$ będzie zbiorem wszystkich możliwych wartości dających się zapisać w jednym bajcie, zaś Σ^* będzie zbiorem wszystkich skończonych ciągów złożonych z elementów Σ, a więc wszystkich możliwych zestawów danych dla dowolnych problemów decyzyjnych. Wówczas problemy decyzyjne można utożsamić z dowolnymi podzbiorami zbioru Σ^*. Faktycznie, mając dany problem decyzyjny, możemy przypisać mu podzbiór zbioru Σ^* złożony z tych ciągów bajtów, na które (po potraktowaniu ich jak dane wejściowe) problem nasz odpowie TAK. Odwrotnie, dowolnemu zbiorowi $A\subseteq\Sigma^*$ ciągów bajtowych odpowiada problem decyzyjny: "Dla danego ciągu bajtów x sprawdź czy $x\in A$?". Do końca tego wywodu będziemy więc utożsamiać problemy decyzyjne z podzbiorami zbioru Σ^*.
Ciągi bajtów nie muszą oznaczać jedynie danych. Za pomocą pewnych takich ciągów (plików tekstowych) opisujemy algorytmy podając ich kody np. w języku PASCAL. W jaki sposób opisać dowolny algorytm rozwiązujący dowolny problem decyzyjny? Wystarczy podać jego kod pascalowy, przy czym można przyjąć, że jedynymi instrukcjami wejściowymi dopuszczalnymi do użytku jest wczytanie ze standardowego wejścia ciągu bajtów stanowiących dane. Algorytm nasz ma udzielić odpowiedzi na problem decyzyjny, a więc jedynymi instrukcjami wyjścia będą exit(0) – odpowiedź NIE i exit(1)– odpowiedź TAK (jeżeli żadna nie zostanie wywołana – domyślnie przyjmiemy exit(0)). Metoda ta pozwala zapisać dowolny algorytm rozwiązujący problem decyzyjny (nie troszcząc się nawet o ich np. własność stopu), my jednak chcielibyśmy znaleźć metodę opisu nie wszystkich algorytmów, lecz tych, których złożoność jest $O(f(N))$. Dlatego zmodyfikujemy naszą notację i uznajemy, że ciąg bajtów υ postaci napisu:

υ='liczba1;liczba2;kod programu P spełniający powyższe ograniczenia'

opisuje problem decyzyjny, albo inaczej zbiór ciągów bajtowych x, dla których opisany program P uruchomiony na danych x odpowie TAK (czyli wywoła

exit(1)) nie później, niż w ciągu pierwszych liczba1+liczba2*$f(|x|)$ instrukcji (po tym czasie domyślnie interpretujemy odpowiedź na NIE). Ten problem decyzyjny (inaczej, podzbiór Σ^*) oznaczymy przez L_υ. Założymy też, że dla ciągów bajtów υ nie będących opisanej postaci będzie $L_\upsilon=\emptyset$. Przy takiej notacji każdy ciąg bajtów υ opisuje pewien problem decyzyjny $L_\upsilon\subseteq\Sigma^*$.
Wykażemy, że każdy problem decyzyjny rozstrzygalny algorytmem o złożoności $O(f(N))$ jest równy L_υ dla jakiegoś napisu υ. Faktycznie, jeżeli P jest kodem tego algorytmu, to istnieją liczby a i b, takie, że dla każdych danych x program ten zakończy działanie (akceptując lub odrzucając daną), po wykonaniu liczby operacji mniejszej od $a+bf(|x|)$, czyli algorytm ten można opisać np. napisem $\upsilon='a;b;P'$.
Co więcej mając dane dwa ciągi bajtów u i w, możemy algorytmicznie sprawdzić, czy $w\in L_u$. Wystarczy zweryfikować, czy u jest wymaganej postaci, co potrafi każdy kompilator (jeśli nie jest – odpowiedź jest zawsze NIE), a następnie wykonać opisany program przez $a+bf(|w|)$ pierwszych kroków (tą ostatnią liczbę z założenia potrafimy zawsze obliczyć).
Rozważmy teraz zbiór L złożony z ciągów bajtów nie należących do opisywanych przezeń zbiorów, czyli:

$$L=\{w\in\Sigma^*:\ w\notin L_w\}.$$

Rozważmy problem decyzyjny L (lub jak kto woli, problem, czy dany ciąg bajtów należy do L). Na mocy powyższego, L jest algorytmicznie rozstrzygalny, bowiem $w\in L \Leftrightarrow w\notin L_w$, co jak powiedziano potrafimy zweryfikować. Ale zagadnienia tego nie można rozstrzygnąć algorytmem o złożoności $O(f(N))$, w przeciwnym razie na mocy powyższego istniałby opis L, czyli ciąg bajtów υ, dla którego $L=L_\upsilon$ i mielibyśmy:

$$\upsilon\in L_\upsilon \Leftrightarrow \upsilon\in \mathrm{L} \Leftrightarrow \upsilon\notin L_\upsilon$$

zgodnie z definicją L. Sprzeczność! Podsumowując, mamy

Twierdzenie: Dla każdej algorytmicznie obliczalnej funkcji f istnieje rozstrzygalny problem decyzyjny, którego jednak nie rozstrzygnie żaden algorytm o złożoności czasowej wynoszącej tylko $O(f(N))$.

Analogiczne twierdzenie jest prawdziwe dla złożoności pamięciowej.

Twierdzenie: Jakąkolwiek algorytmicznie obliczalną funkcje f nie wybrać, zawsze istnieje rozstrzygalny problem decyzyjny, którego nie rozstrzygnie żaden algorytm zużywający tylko $O(f(N))$ pamięci.

Szkic dowodu.
Załóżmy przez sprzeczność, że $f>0$ jest funkcją obliczalną algorytmicznie o wartościach całkowitych, taką że każdy rozstrzygalny problem decyzyjny można rozwiązać procedurą używającą $O(f(N))$ bitów pamięci, czyli nie więcej niż $cf(N)$ dla pewnej stałej c. Liczba możliwych stanów całej pamięci o tym rozmiarze nie przekracza $2^{cf(N)}$, a skoro obliczenie przebiega w sposób deterministyczny, każdy stan wyznacza jednoznacznie stan kolejny, co więcej – stany pamięci nie mogą się powtórzyć (inaczej program pętliłby się), więc i liczba kroków nie przekracza $2^{cf(N)}=o(2^{Nf(N)})$. Jest to sprzeczne z poprzednim twierdzeniem, mówiącym że istnieje problem decyzyjny, dla którego nawet $O(2^{Nf(N)})$ kroków jest niewystarczające.

Zadanie 7.9.
Udowodnij, że istnieją rozstrzygalne problemy decyzyjne spoza klasy NP.

Rozwiązanie.
Wystarczy wziąć problem decyzyjny, do którego rozwiązania nie wystarczy złożoność czasowa $O(2^{2^N})$.

Zauważmy, że gdyby z treści powyższego zadania wykreślić słowo "rozstrzygalne", to odpowiednim przykładem byłoby dowolne zagadnienie nierozstrzygalne np. problem STOPU.

Zadanie 7.10.
Decyzyjny PROBLEM STOPU formułuje się następująco: „Dany jest kod programu nie pobierającego żadnych danych wejściowych. Sprawdź, czy program ten ukończy pracę w skończonej liczbie kroków, czy też będzie działał w nieskończoność?". Wywnioskuj z powyższych twierdzeń, że problem ten jest algorytmicznie nierozstrzygalny.

Wskazówka:
Każdy problem rozstrzygalny jest α-redukowalny do problemu STOPU (patrz zadanie 8.1).
Rozwiązanie.
Załóżmy przez sprzeczność, że problem STOPU jest rozstrzygalny pewną procedurą P o czasie obliczeń $O(f(N))$, gdzie f jest algorytmicznie obliczalną funkcją niemalejącą (np. za f można przyjąć łączną liczbę kroków wykonanych przez procedurę P uruchamianą kolejno na wszystkich danych o długości nie większej od N). Niech A będzie dowolnym problemem decyzyjnym, zaś w wielomianem rosnącym szacującym z góry złożoność programu dokonującego α-redukcji z A do problemu STOPU. Przeprowadzając najpierw α-redukcję, a później uruchamiając naszą hipotetyczną procedurę możemy rozwiązać problem A w czasie $O(w(N)+ f(w(N)))$, czyli $o(2^N f(2^N))$. Ten czas wystarczałby do rozwiązania każdego problemu rozstrzygalnego – znów mamy sprzeczność z naszym twierdzeniem.

Jak powiedziano, pierwszym problemem NP–zupełnym była SPEŁNIALNOŚĆ, gdzie dana jest formuła logiczna, złożona ze zmiennych zdaniowych, połączonych w pewien sposób tradycyjnymi spójnikami logicznymi, zaś pytamy o możliwość przypisania doń takich wartości logicznych, by miała ona wartość true. Definiuje się formuły logiczne w tzw. *postaci normalnej* (*CNF*), wówczas musi mieć ona kształt skończonego iloczynu logicznego klauzul: $c_1 \wedge \ldots \wedge c_k$, gdzie każda klauzula jest postaci sumy logicznej pewnej ilości zmiennych zdaniowych w postaci prostej lub zanegowanej np. $c_i=(p_1 \vee p_4 \vee \sim p_8 \vee \ldots)$. Wymaga się ponadto, by w obrębie pojedynczej klauzuli żadna zmienna nie powtarzała się (nawet gdyby raz miała wystąpić bez, a raz z negacją). Dalej, dla dowolnego naturalnego k mówimy, że formuła logiczna w postaci normalnej jest typu k–*CNF*, jeżeli każda z jej klauzul zawiera dokładnie k składników.

Zadanie 7.11.
Dla dowolnego naturalnego k przedstaw wielomianową redukcję problemu spełnialności dla formuł logicznych w postaci k–CNF do spełnialności dla formuł (k+1)–CNF.

Rozwiązanie.
Naturalnym wydaje się założenie, że rozmiar danych opisujących formułę jest proporcjonalny do liczby wszystkich wystąpień zmiennych zdaniowych, spójników logicznych i nawiasów znajdujących się w jej zapisie. Niech dana formuła k–CNF będzie w postaci iloczynu logicznego klauzul $\phi=c_1\wedge c_2\wedge...\wedge c_m$. Wprowadzamy m nowych zmiennych zdaniowych $q_1,...,q_m$ i tworzymy formułę ϕ' w postaci (k+1)–CNF zastępując każdą starą klauzulę dwiema nowymi o k+1 składnikach każda:

$$\phi' = (c_1\vee q_1)\wedge(c_1\vee\sim q_1)\wedge(c_2\vee q_2)\wedge(c_2\vee\sim q_2)\wedge..\wedge(c_m\vee q_m)\wedge(c_m\vee\sim q_m).$$

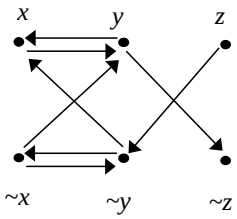
Widać, że jeśli istnieje przypisanie wartości logicznych nadające wartość true formule ϕ, to podobne przypisanie uzyskamy dla ϕ' przenosząc wartości starych zmiennych zdaniowych, dla nowych zaś $q_1,...,q_n$ przyjmujemy dowolne wartościowanie. Odwrotnie, jeżeli ϕ' jest spełnialna, to przy odpowiednim wartościowaniu zmiennych zdaniowych (starych i nowych) każdy fragment $(c_i\vee q_i)\wedge(c_i\vee\sim q_i)$ musi mieć wartość true, co oznacza, że bez względu na wartość logiczną q_i klauzula c_i musi mieć wartość true i ϕ jest również spełnialne. Równoważność ta dowodzi, że redukcja jest poprawna i w oczywisty sposób wielomianowa.

Istnieje wielomianowa redukcja przekształcająca daną formułę logiczną ϕ w formułę ϕ' będącą w postaci 3–CNF, taką że spełnialność ϕ jest równoważna spełnialności dla ϕ' (nie oznacza to, że obie formuły są logicznie równoważne – zazwyczaj ϕ' używa innego zbioru zmiennych zdaniowych niż ϕ, podobnie jak w poprzednim zadaniu). Dlatego dla wszystkich $k\geq 3$ problemy spełnialności dla formuł w postaci k–CNF są NP–zupełne. Najczęściej w dowodach NP–zupełności bardziej zaawansowanych problemów jako zagadnienia bazowego używa się spełnialności dla 3–CNF. Oczywiście spełnialność przy 1–CNF jest wielomianowa, gdyż wówczas każda klauzula zawiera tylko jedną zmienną (w postaci prostej lub zanegowanej) i wystarczy sprawdzić, czy pewne dwie klauzule nie są tą samą zmienną – raz z negacją, a raz bez.

Zadanie 7.12*.
Udowodnij, że spełnialność dla formuł logicznych w postaci 2–CNF jest problemem wielomianowym.

Rozwiązanie.
Niech dana formuła używa zmiennych zdaniowych $q_1,...,q_n$. Bezpośrednie sprawdzanie wartości logicznej ϕ przy wszystkich sposobach przypisania true/false do zmiennych q_i wymagałoby 2^n prób, czyli byłoby wykładnicze. Określmy zamiast tego zbiór literałów (a więc zmiennych logicznych lub ich bezpośrednich negacji) $V=\{q_1,...,q_n,\sim q_1,...,\sim q_n\}$ i przyjmijmy dla dowolnego literału $a\in V$ upraszczające zapis oznaczenie $\sim a$ określające $\sim q$ jeżeli a był zmienną $a=q$ oraz q gdy a był negacją zmiennej $a=\sim q$ (pozwoli to nam operować tylko na elementach zbioru V bez dopisywania wielokrotnych i w rezultacie znoszących się wzajemnie negacji tej samej zmiennej, takich jak $\sim\sim\sim q$). Następnie tworzymy digraf $D(\phi)$ o zbiorze wierzchołków V, którego łuki wprowadzamy na podstawie klauzul formuły ϕ.

Mianowicie dla każdej klauzuli postaci $(a \vee b)$ dodajemy dwa łuki: z $\sim a$ do b i z $\sim b$ do a. Intuicyjnie, alternatywa $a \vee b$ jest równoważna każdej z dwóch implikacji: $\sim a \rightarrow b$ oraz $\sim b \rightarrow a$. Dla przykładu przedstawiamy digraf formuły $(x \vee y) \wedge (x \vee \sim y) \wedge (\sim x \vee y) \wedge (\sim y \vee \sim z)$:



Przyporządkowanie wartości true/false do zmiennych będziemy opisywać etykietując literały $a \in V$ wartościami w(a). Zgodnie z powyższym ϕ jest spełnialna wtedy i tylko wtedy, gdy istnieje poetykietowanie wszystkich wierzchołków V etykietami z {0,1} spełniające warunki:

- dla każdego $a \in V$ zachodzi: w(a)=0 wtedy i tylko wtedy, gdy w($\sim a$)=1,
- dla żadnego łuku z a do b nie zachodzi równocześnie w(a)=1 i w(b)=0.

Łuk z a do b spełniający warunek drugi nazwiemy *spełnionym*, co oznacza, że odpowiadająca mu klauzula będzie prawdziwa. W dalszej kolejności udowodnimy następujący fakt:

(i) Formuła ϕ jest spełnialna wtedy i tylko wtedy, gdy dla każdego literału $a \in V$ w $D(\phi)$ nie istnieje droga z a do $\sim a$ lub z $\sim a$ do a.

Jeśli tak, to przy danej formule ϕ wystarczy utworzyć digraf $D(\phi)$, a następnie dla każdego wierzchołka a sprawdzić istnienie dróg z a do $\sim a$. Sprowadza się to do co najwyżej $2n$–krotnego wywołania np. algorytmu Dijkstry, czyli spełnialność dla formuł 2–CNF jest wielomianowa.

Dowód faktu (i). Załóżmy, że dla pewnego literału a istnieją w $D(\phi)$ drogi z a do $\sim a$ i odwrotnie. Wówczas ϕ nie może być spełnialna, gdyż przypisanie w(a)=1 implikowałoby w($\sim a$)=1 (wszystkie łuki drogi z a do $\sim a$ muszą być spełnione), zaś przypisanie w(a)=0 czyli w($\sim a$)=1 implikowałoby w(a)=1 (ten sam argument dla drogi odwrotnej). Pozostaje wykazać, że niemożność przejścia w $D(\phi)$ z a do $\sim a$ lub odwrotnie dla wszystkich $a \in V$ implikuje istnienie właściwego poetykietowania. Dowód przeprowadzimy przez indukcję względem liczby zmiennych logicznych n. Przy n=1 jest to w oczywisty sposób prawdziwe (nie istnieje formuła w 2–CNF z jedną zmienną). Niech więc n>1 i $D(\phi)$ spełnia nasze założenie. Zatem nie istnieje droga z a do $\sim a$ dla pewnego literału a. Niech U będzie zbiorem tych wierzchołków, do których można dojść z a (uwzględniamy też drogę długości 0), zaś U'={$\sim b$: $b \in U$}. Zauważmy, że U nie może zawierać żadnego literału c wraz z $\sim c$, bowiem istnienie dróg z a do c i z a do $\sim c$ oznaczałoby istnienie drogi z c do $\sim a$ (patrz: konstrukcja digrafu), czyli również z a do $\sim a$. Dlatego U i U' są rozłączne i możemy przyjąć w(c)=1 dla wszystkich $c \in U$ oraz w(c)=0 dla wszystkich $c \in U$'. Teraz bez względu na sposób dalszego przypisania etykiet wierzchołkom z $V \setminus (U \cup U')$ wszystkie łuki, których przynajmniej jeden koniec leży w $U \cup U$' są spełnione. W przeciwnym razie musiałby istnieć łuk o początku w U i końcu poza U lub końcu w U' i początku poza U' – obie te sytuacje są sprzeczne z definicją zbioru U. Wystarczy więc poetykietować wierzchołki z $V \setminus (U \cup U')$ tak, by wszystkie łuki o obu końcach w tym zbiorze były spełnione. Jest to możliwe na mocy założenia indukcyjnego, bowiem $D(\phi)$ po usunięciu wierzchołków z $U \cup U$' staje się digrafem $D(\phi')$ formuły ϕ' powstałej z ϕ przez skreślenie klauzul zawierających literały z $U \cup U$', ponadto założenie dowodzonej implikacji musi pozostać prawdziwe dla $D(\phi')$ będącego podgrafem $D(\phi)$.

8. Redukcje wielomianowe i problemy NP–zupełne

W tym rozdziale przedstawimy przykłady najczęściej spotykanych problemów NP–zupełnych oraz ich wielomianowe przypadki szczególne. Zaprezentujemy metody konstrukcji wielomianowych redukcji oraz dowodzenia ich poprawności w celu wykazywania NP–zupełności innych zagadnień decyzyjnych.

Zadanie 8.1.
Dane są następujące problemy decyzyjne:

1) MNOŻENIE MACIERZY (MM): Dane są trzy macierze liczbowe A,B,C; sprawdź czy $C = AB$?
2) ODWRACANIE MACIERZY (OM): Dane są dwie macierze liczbowe A,B; sprawdź czy $B = A^{-1}$?
3) IZOMORFIZM GRAFÓW (IG): Dane są dwa grafy G i H o tej samej liczbie wierzchołków, sprawdź czy są one izomorficzne?
4) Problem STOPU (S): Dany jest kod algorytmu A: sprawdź, czy A zatrzyma się po wykonaniu skończonej liczby kroków?
5) KLIKA (K): Dany jest graf G i liczba naturalna k; sprawdź czy graf pełny K_k jest podgrafem G?
6) 3–SPEŁNIALNOŚĆ (3S): Dana jest formuła logiczna w postaci 3–CNF, sprawdź czy jest ona spełnialna?

Podaj statusy tych problemów i zbadaj wszystkie relacje wielomianowej redukcji pomiędzy nimi.

Rozwiązanie.
Problemy MM i OM są rzecz jasna wielomianowe (P). Problemy K i 3S są znanymi problemami NP–zupełnymi. Problem IG należy do NP – prosty algorytm niedeterministyczny o czasie działania $O(n^2)$ „zgaduje" przenumerowanie wierzchołków grafów, po czym sprawdza, czy ich macierze sąsiedztwa po zastosowaniu odgadniętej zmiany numerów są identyczne:

```
function izo(G: array[1..n,1..n] of boolean, H: array[1..n,1..n] of boolean):boolean;
var
        i,j:integer;
        permutacja: array[1..n] of integer;
begin
        for i := 1 to n do begin
                permutacja[i] := 1;
                for j := 1 to n–1 do permutacja[i] := permutacja[i]+rand;
        end;
        for i := 1 to n–1 do
                for j :=  i+1 to n do
                        if permutacja[i] = permutacja [j]
```

```
                then return false;      // wylosowany ciąg nie jest permutacją
        for i := 1 to n–1 do
                for j :=  i+1 to n do
                        if G[i,j] ≠ H[permutacja[i], permutacja[j]]
                        then return false;      // wylosowana permutacja nie ustala izomorfizmu
        return true;
end;
```

Do dziś nie wiadomo, czy weryfikacja izomorfizmu grafów jest wielomianowa, czy NP-zupełna. Na użytek tego zadania będziemy umieszczali ją „pomiędzy" obydwoma klasami.

> Oczywiście jest to jedynie „roboczo" przyjęta konwencja. Gdybyśmy wiedzieli, że IG nie należy ani do P, ani do NP-zupełnych, wiedzielibyśmy także, że P≠NP.

Wreszcie wiadomo, że problem STOPU S jest nierozstrzygalny, zatem nie może on należeć do NP.

Rozważymy teraz zachodzące pomiędzy naszymi problemami relacje wielomianowej redukcji α. Przede wszystkim każdy problem decyzyjny jest w relacji α z samym sobą (zachodzi AαA), co jest oczywiste – odpowiednia procedura redukcji po prostu nie zmienia danych.

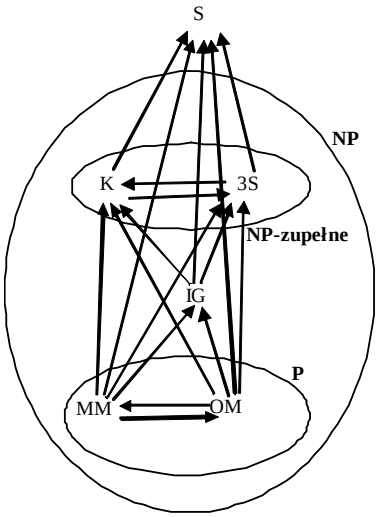
Ponadto każdy problem wielomianowy jest w relacji α z każdym problemem nietrywialnym (tj. takim, dla którego istnieje zestaw danych x_{tak}, dla którego odpowiedź na problem jest TAK, oraz dane x_{nie}, dla których odpowiedź jest NIE). Udowodnimy ten fakt. Dla dowolnego problemu A∈P i nietrywialnego B procedura redukcji AαB działa następująco: dla danych *x* dla problemu A rozwiązujemy problem A w czasie wielomianowym, i w przypadku, gdy odpowiedź jest TAK, zwracamy zestaw danych x_{tak}, w przeciwnym razie zwracamy x_{nie}. Powyższa uwaga dowodzi, że w naszym zadaniu zachodzą relacje: MMαOM, MMαIG, MMαS, MMαK, MMα3S, OMαMM, OMαIG, OMαS, OMαK, OMα3S.

Ponadto do problemów NP–zupełnych redukują się wszystkie inne problemy z NP, co daje dodatkowo relacje: IGαK, 3SαK, IGα3S, Kα3S. Nie wiemy nic o relacjach 3SαIG i KαIG – ich istnienie jest bowiem równoważne NP–zupełności IG.

Wreszcie z całą pewnością nie zachodzi relacja SαA dla żadnego innego problemu A z naszego zadania – bowiem problemy te są rozstrzygalne i istnienie takiej relacji implikowałoby rozstrzygalność S. Natomiast wszystkie problemy A z naszego zadania (podobnie jak wszystkie rozstrzygalne problemy decyzyjne) spełniają AαS. Dla przykładu przedstawimy redukcję LZαS. Daną dla problemu STOPU jest kod algorytmu, zatem dla liczby naturalnej *k* (dana dla LZ) procedura redukcji zwraca napis następującej postaci:

```
'program;
var     i:integer;
        zlozona:boolean;
begin
        zlozona:=false;
        for i:=2 to k–1 do if k mod i = 0 then zlozona:=true;
        if not zlozona then repeat until false;
end. '
```

w którym napis **k** zastępujemy wartością liczby *k*. Widać od razu, że liczba *k* jest złożona wtedy i tylko wtedy, gdy wygenerowany kod opisuje program, który po skończonym czasie zakończy pracę.
Na poniższym rysunku zaznaczono wszystkie redukcje wielomianowe, o których wiadomo, że na pewno zachodzą. Pominięto jedynie redukcje trywialne postaci AαA.



Zadanie 8.2.
NP–zupełny problem decyzyjny CYKLU HAMILTONA ma następującą postać: "Dla danego grafu *G* (np. w postaci macierzy sąsiedztwa) o *n* wierzchołkach sprawdź, czy w *G* istnieje cykl prosty wykorzystujący wszystkie wierzchołki (cykl Hamiltona)".

Problem NAJDŁUŻSZEGO CYKLU (wersja decyzyjna) ma postać: "Dla danego w postaci macierzy sąsiedztwa grafu *G* o *n* wierzchołkach i liczby naturalnej *L* sprawdź, czy w *G* istnieje cykl prosty o długości co najmniej *L*".

Wykaż, że ten ostatni problem jest NP–zupełny.

Rozwiązanie.
Najpierw wykażemy, że problem NAJDŁUŻSZEGO CYKLU jest w NP. Faktycznie, niedeterministyczny algorytm wielomianowy może odgadnąć sekwencję wierzchołków o długości co najmniej *L* i nie więcej niż *n* oraz zwykłym algorytmem wielomianowym sprawdzić, czy odgadnięte wierzchołki są różne i czy odwiedzane w wygenerowanej sekwencji tworzą cykl w *G*.

Następnie pokażemy redukcję CYKL HAMILTONA α NAJDŁUŻSZY CYKL. Procedura redukcji działa następująco:

Dla danego grafu *G* (dana dla problemu CYKLU HAMILTONA) generujemy parę: Graf *G* i liczba *n* (dana dla NAJDŁUŻSZEGO CYKLU).

Oczywiście jest to procedura wielomianowa. Do wykazania jej poprawności wystarczy zauważyć, że jeśli w grafie G istnieje cykl Hamiltona, to jest to cykl o długości n (odpowiedź na drugi problem jest TAK) i odwrotnie, istnienie w G cyklu prostego o długości co najmniej n (a więc dokładnie n) oznacza, że będzie to cykl Hamiltona.

Status problemu często zależy od tego, co traktujemy jako dane dla niego, a co jest wielkością ustaloną. Określmy problem DŁUGIEGO CYKLU DC: „dany jest graf G i liczba naturalna k; sprawdź, czy w G istnieje podgraf C_k?". Analogicznie jak wyżej stwierdzamy, że DC jest NP–zupełny. Zdefiniujmy teraz rodzinę podproblemów zagadnienia DC. Dla dowolnej liczby naturalnej $k>2$ definiujemy problem DC_k: „dany jest graf G; sprawdź, czy w G istnieje podgraf C_k?". Tym razem k jest ustalone, a daną do problemu jest jedynie graf. Zauważmy, że dla każdego k problem DC_k jest wielomianowy – wiemy, że można go rozwiązać w czasie $O(n^k)$.

Zadanie 8.3.
Problem KOMIWOJAŻERA (wersja decyzyjna) ma następującą postać: "Dany jest pełny graf n–wierzchołkowy (każde dwa wierzchołki łączy krawędź), przy czym każdej krawędzi e przypisana jest dodatnia waga w(e) – jej długość. Ponadto dana jest liczba dodatnia L. Czy w grafie tym da się znaleźć cykl prosty przechodzący przez wszystkie wierzchołki, którego suma wag krawędzi (długość ważona) będzie $\leq L$?". Udowodnij, że problem ten jest NP–zupełny.

Rozwiązanie.
Przynależność problemu KOMIWOJAŻERA do NP jest prosta – algorytm niedeterministyczny tworzymy podobną metodą, jak w poprzednim zadaniu. Przedstawimy teraz redukcję CYKL HAMILTONA α PROBLEM KOMIWOJAŻERA:

Dla danego grafu n–wierzchołkowego G (dane dla CYKLU HAMILTONA) tworzymy graf pełny oparty na tych samych wierzchołkach, w którym krawędzie występujące w G obciążamy wagą 1, zaś nowe krawędzie obciążamy przez 2. Ponadto przyjmujemy $L=n$.

Oczywiście redukcja jest wielomianowa, pozostaje wykazać jej poprawność. Jeżeli w grafie G istnieje cykl Hamiltona, to w grafie pełnym cykl ten składa się z krawędzi o wadze 1 i jego ważona długość wyniesie n, zatem odpowiedź na problem KOMIWOJAŻERA jest TAK. Odwrotnie, jeżeli odpowiedź na problem KOMIWOJAŻERA jest TAK, to w grafie pełnym istnieje cykl prosty o długości n i długości ważonej $\leq n$. Ale w grafie tym jedynymi wagami krawędzi są 1 i 2, czyli wszystkie krawędzie tego cyklu muszą mieć wagę 1, zatem występują one w grafie G tworząc tam cykl Hamiltona.

Zadanie 8.4.
Dany jest graf planarny G. Zbadaj statusy problemów decyzyjnych polegających na sprawdzeniu, czy:
a) $\chi(G) \leq 1$,

b) $\chi(G) \leq 2$,
c) $\chi(G) \leq 3$,
d) $\chi(G) \leq 4$,
e) $\chi(G) \geq 1$,
f) $\chi(G) \geq 2$,
g) $\chi(G) \geq 3$,
h) $\chi(G) \geq 4$.

Wskazówka:
- Zweryfikowanie planarności podanego grafu jest wielomianowe.
- Problem 3–KOLOROWANIA grafu pozostaje NP–zupełny, nawet jeżeli ograniczymy się do grafów planarnych.

Rozwiązanie.
Problem a) jest wielomianowy, graf można pokolorować jednym kolorem wtedy i tylko wtedy, gdy jest to graf pusty (nie ma krawędzi) – co można zweryfikować w czasie liniowym.

Istnieje również liniowy algorytm sprawdzający, czy dany graf jest dwudzielny – omawialiśmy go w poprzednich rozdziałach, stąd b) jest wielomianowy.

Na mocy wskazówki problem c) jest NP–zupełny, zaś d) jest trywialnym problemem wielomianowym, dla którego zgodnie z twierdzeniem o 4 barwach odpowiedź zawsze brzmi TAK.

Podobnie problem e) jest trywialny – każdy graf wymaga do pokolorowania co najmniej jednego koloru.

Problemy f) i g) są negacjami odpowiednio problemów a) i b), czyli są wielomianowe. Problem h) jest negacją NP–zupełnego problemu c), zatem należy on do co–NP i jest NP–trudny, ale nie wiadomo, czy jest w NP (gdyby tak było, mielibyśmy co–NP=NP).

Zadanie 8.5.
Dla dowolnej liczby naturalnej L definiujemy problem decyzyjny KOL_L: "Dla danego grafu n–wierzchołkowego G sprawdź, czy $\chi(G) \leq L$?". Zbadaj statusy problemów KOL_L dla wszystkich liczb naturalnych L.

Rozwiązanie.
Oczywistym jest, że $\text{KOL}_L \in \text{NP}$ dla wszystkich L. Podobnie jak w poprzednim zadaniu dowodzimy, że KOL_1 i KOL_2 są liniowe. Wiemy też, że KOL_3 jest NP–zupełny. Wykażemy, że KOL_L jest też NP–zupełny dla każdego $L>3$. Dowód przeprowadzimy przez indukcję pokazując istnienie wielomianowej redukcji $\text{KOL}_L \alpha \text{KOL}_{L+1}$. Przebiega ona następująco:

Dla danego grafu G (dana dla KOL_L) tworzymy graf G' dodając do G nowy wierzchołek v i łącząc go krawędziami ze wszystkimi wierzchołkami z G.

Łatwo zauważyć, że każde pokolorowanie G x kolorami można przedłużyć do $(x+1)$–kolorowania G' (nowy wierzchołek dostaje nowy kolor) i odwrotnie, obcięcie pokolorowania G' do wierzchołków z G daje pokolorowanie G używające o jeden kolor mniej (nowy wierzchołek musiał mieć inny kolor, niż wszystkie wierzchołki z G). Zatem $\chi(G)+1=\chi(G')$, czyli:

$$\chi(G)\leq L \Leftrightarrow \chi(G')\leq L+1.$$

Redukcja jest poprawna.

Zadanie 8.6.
Dla dowolnej liczby naturalnej L definiujemy problem decyzyjny RK_L: "Dla danego grafu n–wierzchołkowego G sprawdź, czy można znaleźć w nim k wierzchołkowo rozłącznych grafów pełnych pokrywających razem wszystkie jego wierzchołki?". Zbadaj statusy problemów RK_L dla wszystkich liczb naturalnych L.

Rozwiązanie.
Załóżmy, że zbiorem wierzchołków grafu G jest V i że G można przedstawić w postaci sumy L grafów pełnych o parami rozłącznych zbiorach wierzchołków odpowiednio równych V_i, czyli $V=V_1\cup...\cup V_L$. Oznacza to, że dla każdego i = 1,...,L pomiędzy każdą parą wierzchołków z V_i istnieje krawędź. W dopełnieniu grafu G, czyli w grafie G' zbiory V_i są niezależne, czyli G' można pokolorować wierzchołkowo L barwami nadając kolor i wierzchołkom z V_i. Odwrotnie, istnienie L–barwnego pokolorowania wierzchołkowego dla G' oznacza istnienie rozłącznych zbiorów niezależnych V_i w G' spełniających $V=V_1\cup...\cup V_L$ (tworzą je wierzchołki o tych samych barwach), a zatem w G są to zbiory wierzchołków klik pokrywających V. Ostatecznie, pytanie z problemu RK_L dla grafu G jest równoważne pytaniu o to, czy G' można pokolorować używając dokładnie L barw.

> Warto przypomnieć, że graf n–wierzchołkowy G można pokolorować legalnie używając dokładnie k barw wtedy i tylko wtedy, gdy zachodzi $n \geq k \geq \chi(G)$.

Zatem na mocy poprzedniego zadania RK_L jest wielomianowy dla L=1,2 i NP–zupełny dla $L\geq 3$.

> O ile np. problem decyzyjny „sprawdź, czy dla danego grafu G zachodzi $\chi(G)\leq 4$?" jest NP–zupełny, to nie wiemy nic o NP–zupełności podobnie brzmiącego problemu X: „sprawdź, czy dla danego grafu G zachodzi $\chi(G)=4$?". Oczywiście problem „sprawdź, czy dany graf G można pokolorować używając dokładnie czterech barw?" jest NP–zupełny.

Zadanie 8.7.
Udowodnij, że jeśli problem decyzyjny X: „dla danego grafu G sprawdź, czy $\chi(G)=4$?" jest z klasy NP, to NP=co–NP.

Rozwiązanie.
Jak wiemy, pytanie „czy $\chi(G)\leq 3$" jest NP–zupełne nawet po ograniczeniu się do grafów planarnych, czyli takich, dla których zawsze zachodzi $\chi(G)\leq 4$. A zatem dla grafu planarnego G to, że $\chi(G)=4$ jest równoważne temu, że nie zachodzi $\chi(G)\leq 3$. Stąd problem decyzyjny X

po ograniczeniu się do grafów planarnych jest negacją problemu NP–zupełnego, więc jego przynależność do NP implikowałaby równość NP=co–NP na mocy zadania 7.8.

Zadanie 8.8.
Dane są następujące problemy decyzyjne.

- KLIKA (K): Dany jest graf G i liczba naturalna L; sprawdź czy graf pełny K_L jest podgrafem G?
- ZBIÓR NIEZALEŻNY (ZN): Dany jest graf G i liczba naturalna L; sprawdź czy w G istnieje L–wierzchołkowy zbiór niezależny (jego wierzchołki mają nie być połączone żadnymi krawędziami)?
- POKRYCIE WIERZCHOŁKOWE (PW): Dany jest graf G i liczba naturalna L; sprawdź, czy w G istnieje L–wierzchołkowe pokrycie (pokryciem wierzchołkowym nazywamy podzbiór zbioru wierzchołków, taki że każda krawędź grafu ma przynajmniej jeden ze swych końców w tym podzbiorze)?

Pokaż, że między każdą parą tych problemów zachodzi relacja α.

Rozwiązanie.
Wykażemy najpierw, że KαZN. Redukcja jest następująca:

Dla danego grafu n–wierzchołkowego G i liczby L (dana dla K) zwracamy dopełnienie G' grafu G (zanegowanie wszystkich krawędzi w macierzy sąsiedztwa) i tę samą liczbę L.

Dowód poprawności. Jeżeli U jest L–wierzchołkowym zbiorem rozpinającym podgraf K_L w G, to w G' wierzchołki te nie będą połączone krawędziami tworząc zbiór niezależny. Odwrotnie, L–wierzchołkowy zbiór niezależny w G' ma tą własność, że w G każde dwa jego wierzchołki będą połączone krawędzią tworząc podgraf K_L.

Teraz wykażemy ZNαPW. Oto redukcja:

Dla danego grafu L–wierzchołkowego G i liczby L (można ograniczyć się do przypadku $L \leq n$, w przeciwnym razie odpowiedź na ZN jest NIE) zwracamy ten sam graf G i liczbę $n–L$.

Dowód poprawności. Niech V będzie zbiorem wszystkich wierzchołków w G, a U pewnym jego podzbiorem złożonym z wierzchołków niezależnych. Wtedy każda krawędź musi mieć przynajmniej jeden z końców w zbiorze $V\backslash U$ – w przeciwnym razie U nie byłby niezależny. Zatem $V\backslash U$ jest pokryciem wierzchołkowym. Odwrotnie, jeżeli U jest pokryciem wierzchołkowym, to nie może być krawędzi o obu końcach nie należących do U (byłaby ona niepokryta), czyli $V\backslash U$ jest niezależny. Krótko mówiąc $U \subseteq V$ jest pokryciem wierzchołkowym wtedy i tylko wtedy, gdy jego dopełnienie $V\backslash U$ jest zbiorem niezależnym. Mamy stąd równoważność: w G istnieje L–wierzchołkowy zbiór niezależny wtedy i tylko wtedy, gdy jest tam $(n–L)$–wierzchołkowe pokrycie.

Łatwo zauważyć, że odwrotne redukcje ZNαK i PWαZN przeprowadzają te same procedury, co odpowiednio KαZN i ZNαPW (jest to sytuacja dość nietypowa!). Zatem korzystając z przechodniości α mamy od razu KαPW i PWαK, co kończy rozwiązanie.

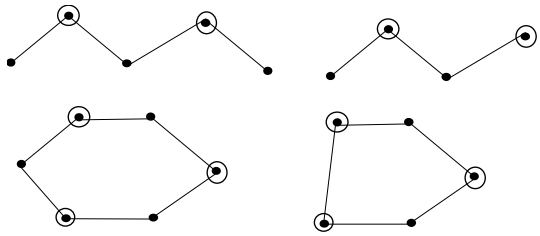
Jest oczywiste, że wszystkie trzy problemy należą do NP, zatem udowodnienie NP–zupełności jednego z nich dowodzi NP–zupełność pozostałych. Faktycznie wykazano, że są one NP–zupełne.

Z reguły dla problemów decyzyjnych definiuje się ich wersje optymalizacyjne – są to podobnie brzmiące problemy, dla których odpowiedzi nie muszą być postaci TAK/NIE (a więc nie są one decyzyjne) i które stanowią naturalne uogólnienia swoich decyzyjnych odpowiedników (tzn. ich rozwiązanie pozwala na łatwe rozwiązanie wersji decyzyjnej). Na przykład dosyć trudno znaleźć sensowny odpowiednik optymalizacyjny dla problemu STOPU – dany algorytm może się tylko zatrzymać lub nie ... ale dla problemu SPEŁNIALNOŚCI możemy w przypadku, gdy formuła jest spełnialna, zażądać dodatkowo zwrócenia jakiegoś przyporządkowania wartości true/false do zmiennych zdaniowych, które nadaje formule wartość true. Wersje optymalizacyjne problemów KOLOROWANIA (wierzchołkowego lub krawędziowego) polegają na znalezieniu dla danego grafu optymalnego pokolorowania właściwego typu. Za wersję optymalizacyjną problemu CYKLU HAMILTONA można przyjąć znalezienie takiego cyklu w przypadku, gdy odpowiedź na część decyzyjną brzmi TAK. Optymalizacyjny problem KOMIWOJAŻERA polega na znalezieniu najkrótszego w sensie ważonym cyklu n–wierzchołkowego w obciążonym wagami dodatnimi grafie pełnym. Wreszcie dla problemów z ostatniego zadania w wersjach optymalizacyjnych szukamy największego podgrafu pełnego (odpowiednio największego zbioru niezależnego) oraz najmniejszego pokrycia wierzchołkowego. Jest tak dlatego, że jeśli dany podzbiór zbioru wierzchołków rozpina w nim podgraf pełny, wówczas to samo dotyczy każdego jego podzbioru, itd. ... aż do pojedynczych wierzchołków. Zatem moce podzbiorów rozpinających podgrafy pełne w danym grafie stanowią przedział zaczynający się od 1 i kończący na pewnej wartości maksymalnej, właśnie której znalezienie jest obliczeniowo trudne. To samo dotyczy zbiorów niezależnych. Natomiast z pokryciami wierzchołkowymi jest dokładnie odwrotnie. Nadzbiór pokrycia wierzchołkowego jest też pokryciem wierzchołkowym, itd. .. aż do największego pokrycia wierzchołkowego, którym jest zbiór wszystkich wierzchołków.

Zadanie 8.9.
Zbuduj algorytm znajdujący najmniejsze pokrycie wierzchołkowe dla grafów o maksymalnym stopniu $\Delta \leq 2$. Czas działania winien wynosić $O(n)$.

Rozwiązanie.
Mamy $2m \leq \Delta n \leq 2n$. Możemy rozbić graf na składowe spójności w czasie $O(n+m)$, czyli $O(n)$ i szukać pokrycia wierzchołkowego w każdej z nich. Dlatego dalej ograniczymy się jedynie do grafów spójnych. Każdy wierzchołek pokrycia może stykać się z co najwyżej dwiema krawędziami, czyli każde pokrycie wierzchołkowe U spełnia $2|U| \geq m$, więc $|U| \geq \lceil m/2 \rceil$.

Znajdziemy pokrycia o liczności $\lceil m/2 \rceil$, a więc najmniejsze możliwe. Graf nasz może być tylko cyklem lub ścieżką, parzystej lub nieparzystej długości. We wszystkich przypadkach wybieramy do pokrycia kolejno co drugi wierzchołek, aż do zakończenia ścieżki lub ponownego napotkania pierwszego wierzchołka. Dla ścieżki, którą rozpoznajemy po wierzchołku stopnia 1 jako pierwszego wybieramy sąsiada tego wierzchołka. Poniższy rysunek ilustruje poprawność konstrukcji.



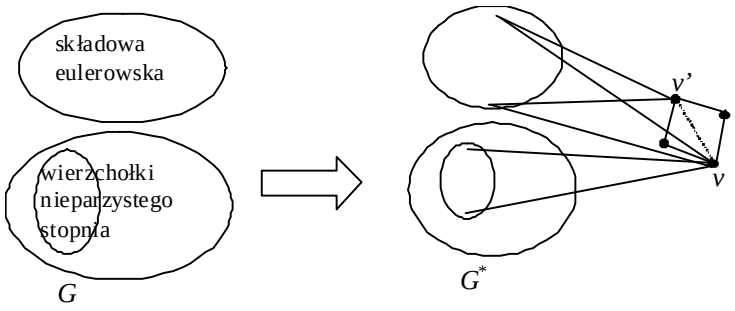
A zatem problem decyzyjny POKRYCIA WIERZCHOŁKOWEGO staje się wielomianowy, jeżeli ograniczymy się do grafów o $\Delta \leq 2$.

Zadanie 8.10*.
Wykaż, że problem POKRYCIA WIERZCHOŁKOWEGO pozostaje NP–zupełny, jeśli ograniczymy się do grafów eulerowskich.

Rozwiązanie.
Przypominamy, że graf posiada cykl Eulera wtedy i tylko wtedy, gdy jest spójny, a wszystkie jego wierzchołki mają parzyste stopnie. Niech PWE oznacza problem PW dla grafów eulerowskich. Przeprowadzimy redukcję PWαPWE. Jedno z możliwych rozwiązań zostało opisane poniżej:

Dla grafu G (możemy założyć, że nie ma on wierzchołków stopnia zero) i liczby L (dane dla PW) tworzymy jego nadgraf G^* oraz liczbę L+2 (dane dla PWE).

Graf G^* powstaje następująco: dodajemy do G rozłączny z nim cykl C_4 i jeden jego wyróżniony wierzchołek v łączymy ze wszystkimi wierzchołkami, które w G mają stopień nieparzysty. Ponadto jeżeli w G występują składowe spójności o wszystkich wierzchołkach z parzystymi stopniami (a więc będące grafami eulerowskimi) – wówczas wszystkie ich wierzchołki łączymy krawędziami z v i wierzchołkiem v' przeciwległym do v w dodanym cyklu. Jeżeli po tym wszystkim v ma stopień nieparzysty, łączymy go krawędzią z v'. Konstrukcję obrazuje poniższy rysunek.



Dla uzasadnienia poprawności konstrukcji zauważmy najpierw, że G^* jest eulerowski. Faktycznie jest on spójny, gdyż wszystkie składowe spójności G zostały połączone z tym samym cyklem C_4. Zbadajmy teraz parzystość stopni jego wierzchołków. Wierzchołki stopnia nieparzystego z G mają nowego sąsiada v, a te, które leżały w składowych eulerowskich (i miały stopień parzysty) – dwóch nowych sąsiadów v i v'. Wreszcie liczba wierzchołków stopni nieparzystych w G musi być parzysta (bo suma stopni wierzchołków w grafie jest parzysta na mocy lematu o uściskach dłoni), a stopień v w G^* wynosi:

$$\deg(v) = 2+\text{liczba wierzchołków stopnia nieparzystego w } G + \text{liczba wierzchołków w składowych eulerowskich.}$$

Podobnie stopień v' wynosi:

$$\deg(v') = 2 + \text{liczba wierzchołków w składowych eulerowskich,}$$

zatem stopnie v i v' mają jednakową parzystość i ostatecznie sytuację ratuje ewentualne dodanie łączącej je krawędzi.

Wykażemy teraz, że moc najmniejszego pokrycia wierzchołkowego w G jest o dwa mniejsza od tejże liczby wyliczonej dla G^*. Zakończy to dowód poprawności redukcji.

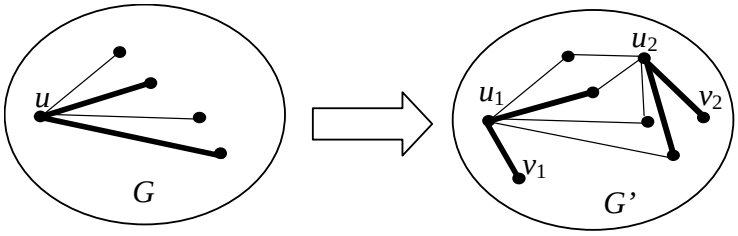
Jeśli U jest pokryciem wierzchołkowym dla G, to zbiór $U\cup\{v,v'\}$ pokrywa wszystkie krawędzie G^*.

Odwrotnie, niech U' będzie pokryciem wierzchołkowym dla G^*, a V zbiorem wierzchołków grafu G. Wiadomo, że $V\cap U$' musi być pokryciem dla G, gdyż żaden wierzchołek G^* spoza V nie pokryje żadnej krawędzi grafu G. Ale do U' muszą też należeć przynajmniej dwa wierzchołki z dodanego cyklu C_4, celem pokrycia jego krawędzi, a zatem mamy $|V\cap U'|\leq|U'|-2$, co kończy dowód.

Zadanie 8.11.
Problem decyzyjny ŚCIEŻKI HAMILTONA (SH) ma następującą postać: "Dla danego grafu G o n wierzchołkach sprawdź, czy w G istnieje ścieżka prosta przechodząca przez wszystkie wierzchołki (ścieżka Hamiltona)". Udowodnij NP–zupełność tego problemu.

Rozwiązanie.
Istnieje wiele sposobów rozwiązania, na przykład przez redukcję CYKL HAMILTONA α ŚCIEŻKA HAMILTONA. Dla danego grafu n–wierzchołkowego G (dana dla CYKLU

HAMILTONA) bierzemy dowolny jego wierzchołek u (jego stopień jest co najmniej 2, w przeciwnym razie cykl Hamiltona nie istnieje i procedura redukcji zwraca dowolny graf niepółhamiltonowski) i zastępujemy go dwiema kopiami u_1 i u_2 o tych samych zbiorach sąsiadów co w przypadku u. Następnie doczepiamy do nich dwa wierzchołki stopnia 1 – odpowiednio v_1 i v_2. Tak powstały graf G' stanowi daną dla zagadnienia ŚCIEŻKI HAMILTONA.



Poprawność. Widać, że jeśli w G istnieje cykl Hamiltona, to musi on przechodzić przez wierzchołek u i dwóch jego sąsiadów. Zamieniamy ten cykl na ścieżkę Hamiltona w G' zastępując obie krawędzie tego cyklu wychodzące z u dwiema krawędziami wychodzącymi z u_1 i u_2 do tych sąsiadów, a następnie dodajemy krawędzie wychodzące z v_1 i v_2.
Odwrotnie, jeśli w G' istnieje ścieżka Hamiltona, to musi ona mieć oba końce w wierzchołkach stopnia 1, czyli w v_1 i v_2, a kolejnymi po nich wierzchołkami na tej ścieżce muszą być u_1 i u_2. Usuwając wierzchołki v_i i sklejając ze sobą u_i uzyskujemy znów graf G, w którym krawędzie ścieżki Hamiltona skleiły się w cykl.

Identyczna redukcja dowodzi, że zmodyfikowany problem ścieżki Hamiltona, w postaci: "dla danego grafu G z dwoma wyróżnionymi wierzchołkami $u \neq v$ sprawdź, czy w G istnieje ścieżka Hamiltona o końcach u i v?" – także jest NP-zupełny. Zagadnienia cyklu oraz ścieżki Hamiltona (w obu wersjach) okazują się być NP-zupełne także dla grafów skierowanych.

Zadanie 8.12*.
Załóżmy, że P=NP i dysponujemy wielomianową procedurą:

function *kolor*(*G*: **array** [1..*n*,1..*n*] **of boolean**, *k*:**integer**):**boolean**;

która dla danej macierzy sąsiedztwa grafu G i liczby k zwraca odpowiedź na pytanie: „czy $\chi(G) \leq k$?" Korzystając z niej opracuj wielomianowy algorytm zwracający optymalne pokolorowanie wierzchołkowe podanego na wejście grafu G.

Rozwiązanie.
Jak powiedzieliśmy, zazwyczaj posiadanie procedury rozwiązującej wersję optymalizacyjną zagadnienia pozwala na proste rozwiązanie wersji decyzyjnej. W tym zadaniu mamy do czynienia z sytuacją odwrotną. Mając daną procedurę rozwiązującą wersję decyzyjną dla problemu KOLOROWANIA WIERZCHOŁKOWEGO GRAFÓW (odpowiadającą na pytanie, czy

pokolorowanie daną liczbą barw jest możliwe) mamy znaleźć optymalne pokolorowanie grafu. Przy tym złożoność algorytmu ma być wielomianowa, co oznacza, że możemy w nim wykonać wielomianową ilość prostych operacji oraz wielomianową ilość razy uruchomić funkcję *kolor* na grafach *n*–wierzchołkowych (złożenie wielomianów jest wielomianem).

Najpierw określimy liczbę chromatyczną danego grafu *G*, czyli $\chi(G)$, jako najmniejsze $k \le n$, dla którego *kolor*(*G*,*k*) zwróci wartość true. W tym celu wystarczy wywołać tę funkcję co najwyżej *n* razy. Następnie zbudujemy nadgraf *G** grafu *G* oparty na tych samych *n* wierzchołkach, utworzony jak następuje:

1. początkowo przyjmij $G^*=G$,
2. ustaw wszystkie krawędzie dopełnienia *G'* grafu *G* w pewien ciąg,
3. kolejno dla każdej krawędzi *e* z tego ciągu sprawdź, czy $G^* \cup \{e\}$ pozostaje $\chi(G)$–barwny i jeśli tak – dodaj krawędź *e* do *G**.

Operacje wykonane dotychczas wymagają co najwyżej $O(n^2)$ wywołań funkcji *kolor* i tyle samo operacji elementarnych. Z konstrukcji *G** wynika, że $\chi(G^*)=\chi(G)$ oraz, że jest on nadgrafem *G*, czyli każde jego optymalne pokolorowanie będzie optymalne dla *G*. Jak jednak znaleźć to $\chi(G^*)$–kolorowanie dla *G**? Rozważmy dowolne takie pokolorowanie *G** i dwa jego niesąsiednie wierzchołki *u* i *v*. W chwili, gdy w punkcie 3 sprawdzaliśmy możliwość dołączenia do tworzonego *G** krawędzi {*u*,*v*} (która musiała wystąpić gdzieś w ciągu krawędzi z *G'*) musiało okazać się, że graf taki wymagałby więcej niż $\chi(G)$ barw (inaczej krawędź zostałaby dodana, a *u* i *v* byłyby sąsiadami). W takim razie *u* i *v* w pokolorowanym *G** muszą mieć jednakowe barwy, gdyby bowiem było inaczej, to można by było dodać {*u*,*v*} do *G** nie zwiększając jego liczby chromatycznej i tym bardziej byłoby to możliwe w trakcie wykonywania testu w punkcie 3. Zatem w optymalnym pokolorowaniu *G** każde dwa wierzchołki nie są sąsiadami wtedy i tylko wtedy, gdy mają jednakowe kolory. Ostatecznie *G** jest grafem pełnym $\chi(G)$–dzielnym i jego optymalne pokolorowanie (a więc i optymalne dla *G*) uzyskamy stosując zachłanny algorytm kolorowania wierzchołków, który w oczywisty sposób jest wielomianowy.

Fakt: Problem PODZIAŁU ZBIORU (PZ) sformułowano następująco: "Dany jest zbiór *n* różnych liczb naturalnych $A=\{a_1,...,a_n\}$. Sprawdź, czy zbiór ten można rozbić na sumę dwóch rozłącznych zbiorów *B* i *C* (tzn. $B \cap C=\emptyset$ i $B \cup C=A$) o równych sumach elementów?". Problem ten jest NP–zupełny.

Zadanie 8.13.

Podaj status następujących problemów decyzyjnych:

P1: "Dany jest zbiór *n* różnych liczb naturalnych $A=\{a_1,...,a_n\}$ oraz liczba całkowita *L*. Sprawdź, czy zbiór ten można rozbić na sumę dwóch rozłącznych zbiorów *B* i *C*, których różnica sum jest co do modułu nie większa niż *L*, czyli $|\Sigma_{a \in B} a - \Sigma_{a \in C} a| \le L$?".

P2: "Dany jest zbiór *n* różnych liczb naturalnych $A=\{a_1,...,a_n\}$ oraz liczba całkowita *L*. Sprawdź, czy zbiór ten można rozbić na sumę dwóch rozłącznych zbiorów *B* i *C*, których różnica sum jest nie mniejsza niż *L*, czyli $|\Sigma_{a \in B} a - \Sigma_{a \in C} a| \ge L$?".

Rozwiązanie.

- Problem P1 jest oczywiście NP–zupełny jako naturalne uogólnienie problemu PODZIAŁU ZBIORU. Mianowicie przyjęcie $L=0$ oznacza szukanie rozbicia zbioru na dwa podzbiory

o jednakowych sumach. Redukcja PZαP1 polega więc na prostym umieszczeniu w danych dla P1 liczby L=0 obok zbioru liczb A (dana dla PZ).

- Problem P2 jest wielomianowy. Aby sprawdzić istnienie rozbicia zbioru liczb na podzbiory B i C o sumach elementów różniących się przynajmniej o L wystarczy zbadać ten warunek dla rozbicia, w którym różnica ta osiąga wartość maksymalną, tj. B=A i C=$\emptyset$. A zatem wystarczy sprawdzić, czy $\Sigma_{a\in A}\, a \geq L$, co oczywiście jest wielomianowe.

Zadanie 8.14.
Problem PLECAKOWY sformułowano następująco: "Dany jest zbiór n przedmiotów B=$\{b_1,...,b_n\}$ o znanych rozmiarach r(b_1),...,r(b_n) i wartościach w(b_1),...,w(b_n) będących liczbami naturalnymi. Ponadto mamy dane limity (liczbowe) R i W. Sprawdź, czy do plecaka o rozmiarze R można włożyć takie przedmioty, by ich łączna wartość nie była mniejsza od W?". Inaczej mówiąc pytamy o istnienie podzbioru $B'\subseteq B$, takiego że $\Sigma_{v\in B'}$ r(v)$\leq R$ i $\Sigma_{v\in B'}$ w(v)$\geq W$. Udowodnij, że problem PLECAKOWY jest NP–zupełny.

Rozwiązanie.
Jest oczywiste, że problem ten należy do NP (algorytm niedeterministyczny zgaduje, które przedmioty wybieramy, a następnie weryfikuje oba warunki). Wykażemy redukcję wielomianową z problemu PODZIAŁU ZBIORU:

> Niech więc A=$\{a_1,...,a_n\}$ będą liczbami danymi w PZ. W czasie wielomianowym obliczamy ich sumę S i jeżeli jest ona nieparzysta (odpowiedź dla PZ na pewno brzmi NIE) – zwracamy jakikolwiek zestaw danych dla problemu plecakowego, który stanowi jedyny przedmiot o rozmiarze większym od R i W>0. W przeciwnym razie zwracamy zbiór przedmiotów B=$\{b_1,...,b_n\}$ o rozmiarach równych wartościom równym kolejnym liczbom z ciągu A: r(b_i)=w(b_i)=a_i oraz przyjmujemy R=W=S/2.

Dowód poprawności redukcji:

- Jeżeli istnieje podział A na dwie połówki o równych sumach, to weźmy jedną z tych połówek – zbiór A'. Suma jego elementów musi być S/2 i odpowiadające tym elementom przedmioty w B mają łączną wartość i łączny rozmiar równy S/2. Zatem odpowiedź na problem PLECAKOWY jest TAK.
- Jeżeli istnieje rozwiązanie dla problemu PLECAKOWEGO, to z konstrukcji zestawu danych (rozmiary przedmiotów równe wartościom) wynika, że wartości włożonych do plecaka przedmiotów muszą się sumować dokładnie do S/2. Zatem w zbiorze liczb A istnieje podzbiór A' o sumie elementów S/2, czyli A' oraz $A\backslash A$' jest szukanym rozbiciem zbioru A. Odpowiedź na PZ jest TAK.

Zadanie 8.15.
Problem PAKOWANIA PUDEŁEK określony jest następująco. "Dany jest zbiór n przedmiotów B=$\{b_1,...,b_n\}$ o znanych rozmiarach r(b_1),...,r(b_n) będących liczbami naturalnymi oraz dodatni rozmiar każdego z pudełek R i ich liczba L. Sprawdź, czy L pudełek o rozmiarze R wystarczy do zapakowania wszystkich przedmiotów ze zbioru B, tak by żaden przedmiot nie "wystawał" ze swego pudełka". Inaczej mówiąc pytamy o istnienie rozbicia zbioru B na L podzbiorów

$B_i \subseteq B$ $i=1,...,L$ parami rozłącznych ($B_i \cap B_j = \emptyset$ dla $i \neq j$), takich że dla każdego z nich $\Sigma_{b \in B_i} r(b) \leq R$. Udowodnij, że problem PAKOWANIA PUDEŁEK jest NP–zupełny.

Rozwiązanie.
Znów przynależność do NP jest oczywista (algorytm niedeterministyczny odgaduje przypisanie przedmiotów do pudełek, a następnie sprawdza jego poprawność), skupimy się więc na redukcji PZαPAKOWANIE PUDEŁEK:

> Dla danego ciągu liczb $\{a_1,...,a_n\}$ (dane dla PZ) obliczamy ich sumę S. Jeżeli jest ona nieparzysta (odpowiedź dla PZ jest NIE) – zwracamy zestaw złożony z jednego przedmiotu i rozmiaru większego od R (L i R dowolne). W przeciwnym razie tworzymy zbiór przedmiotów $B=\{b_1,...,b_n\}$ o rozmiarach równych kolejnym liczbom z $\{a_1,...,a_n\}$, tj. $r(b_i)=a_i$. Ponadto przyjmujemy rozmiar pudełek $R=S/2$ oraz ich liczbę $L=2$.

Dowód poprawności redukcji:

- Istnienie rozbicia zbioru $\{a_1,...,a_n\}$ na dwa podzbiory A_1 i A_2 o równych sumach oznacza, że przedmioty odpowiadające elementom z A_1 mają sumę rozmiarów równą $S/2$, czyli dadzą się zapakować do jednego pudełka. Podobnie drugie pudełko można wypełnić przedmiotami o rozmiarach z A_2.
- Jeżeli przedmioty z B dadzą się zapakować do dwóch pudełek o rozmiarze $S/2$ (suma rozmiarów wszystkich przedmiotów jest S), to oba pudełka muszą być "całkowicie wypełnione", czyli znaleźliśmy rozbicie zbioru przedmiotów na dwa rozłączne zbiory o sumach rozmiarów $S/2$. Wyznacza to przepołowienie zbioru $\{a_1,...,a_n\}$.

Z przedstawionego dowodu wynika, że PAKOWANIE PUDEŁEK pozostaje NP–zupełne nawet wtedy, gdy ograniczymy się do danych, w których liczba pudełek wynosi 2. Analogicznie można pokazać, że dla każdego ustalonego $L \geq 2$ problem ten, ograniczony do sytuacji, w których liczba pudełek jest równa L pozostaje NP–zupełny. Redukcja przebiega analogicznie, jak w powyższym rozwiązaniu, tylko że na jej wyjście kierujemy dodatkowo $L-2$ przedmioty o rozmiarze $S/2$.

---

9. Zagadnienia NP–trudne i algorytmy suboptymalne

Problemy NP–trudne nader często pojawiają się w praktycznych zagadnieniach optymalizacyjnych. W rozdziale tym przedstawimy główne metody radzenia sobie z takimi „niepodatnymi" problemami. Są to: algorytmy pseudowielomianowe oraz absolutnie– lub względnie–przybliżone o pewnych gwarancjach dokładności. Niekiedy daje się udowodnić niemożność skonstruowania algorytmu przybliżonego określonego typu (o ile tylko P≠NP). Przedstawimy kilka przykładów takich zagadnień.

Zadanie 9.1.
Przedstawimy następujący algorytm rozwiązujący NP–zupełny problem PODZIAŁU ZBIORU (PZ) (tablica przekazana jako parametr procedury zawiera dane liczby $\{a_1,...,a_n\}$). Wykaż jego poprawność. Czy jest to algorytm wielomianowy?

```
function polowa(zbior:array[1..n] of integer):boolean;
var
        i,S:integer;
begin
        S := 0;
        for i := 1 to n do S := S+zbior[i];
        if S mod 2 = 1 then return false;

        zaalokuj pomocniczą tablicę tab:array[0..S] of boolean;
        for i := 1 to S do tab[i] := false;
        tab[0] := true;

        for i := 1 to n do                  // główna pętla
                for j := S downto zbior[i] do
                        if tab[j– zbior[i]] then tab[j] := true;

        return tab[S/2];
end;
```

Rozwiązanie.
Początkowo algorytm oblicza sumę S elementów podanego zbioru liczb (jeżeli jest nieparzysta, wiadomo że rozwiązania nie ma). Następnie alokujemy tablicę *tab* o wartościach typu **boolean** i polach numerowanych od 0 do S. Początkowo tylko pole *tab*[0] tej tablicy zawiera wartość true. Zasada działania głównej pętli jest następująca: chcemy, by po zakończeniu i–tego przebiegu głównej pętli w *tab* zachodziło *tab*[x]=true dokładnie dla pól o tych numerach x, że liczba x jest sumą pewnego podzbioru sekwencji $a_1,...,a_i$ (relacja ta jest niezmiennikiem pętli). Faktycznie, zachodzi ona jeszcze przed pierwszym jej wykonaniem, kiedy możemy przyjąć i=0 (jedynie 0 jest sumą zbioru pustego liczb). Dla kolejnych przebiegów wykonamy rozumowanie indukcyjne względem liczby obiegów tej pętli. Jeżeli po i–krotnym wykonaniu głównej pętli **for** tablica *tab* zawiera true dokładnie w polach o numerach będących sumami elementów pewnych podzbiorów $\{a_1,...,a_i\}$, to jakie inne wartości mogą pojawić się jako sumy podzbiorów $\{a_1,...,a_i,a_{i+1}\}$? Tylko takie, które są o a_{i+1} większe od pewnych sum podzbiorów zbioru $\{a_1,...,a_i\}$ – odpowiadające im pola są oznaczane wartościami true w wewnętrznej instrukcji pętli głównej. A zatem po wyjściu z tej

pętli w *tab* wartości true występują dokładnie w polach o numerach będących sumami liczb z $\{a_1,...,a_n\}$, czyli podział zbioru jest możliwy tylko, gdy $tab[S/2]$=true.

Złożoność algorytmu nie jest wielomianowa. Najwięcej czasu zajmuje pętla główna, której wnętrze wymaga liczby operacji proporcjonalnej do S. Mamy więc czas pracy $O(n\Sigma_i a_i)$, co można też wyszacować przez $O(n^2 max(a_i))$. W oszacowaniu pojawiają się więc liczby z danego zbioru, a nie ich rozmiary (czyli liczby bitów potrzebnych do ich zapisania). Rozmiarem danych jest $N=\Sigma_{i=1,...,n}(\lfloor \log_2 a_i \rfloor+1)$ i wielkość $n\Sigma_i a_i$ nie da się oszacować żadnym wielomianem zmiennej N. Można się o tym łatwo przekonać np. tworząc dla liczby naturalnej k zestaw danych o $n=k$ i $a_1 \approx ... \approx a_n \approx 2^k$. Widzimy, że rozmiar $N<(k+1)^2$ jest wielomianowy względem k, lecz czas działania programu $n\Sigma_i a \approx k^2 2^k$ jest wykładniczy.

Jeżeli czas działania algorytmu daje się przedstawić jako wielomian, którego argumentami poza rozmiarem danych są też liczby występujące w tych danych, to algorytm taki nazywa się *pseudowielomianowym*. Jeżeli liczby występujące w typowych danych dla problemu nie są duże – algorytmy pseudowielomianowe, mogą okazać się równie użyteczne w praktyce, co prawdziwe wielomianowe (mimo że w rzeczywistości ich złożoność może być wykładnicza).

Posługując się wygodnym, teoretycznym modelem maszyny liczącej, mogącej operować na zmiennych całkowitych dowolnej wielkości, można zaproponować realizację powyższego algorytmu, w której tablicę *tab* zastępuje pojedyncza zmienna typu **integer**. Jej kolejne bity odpowiadałyby polom tablicy, zaś wszystkie operacje byłyby realizowane poprzez bitowe: alternatywę, koniunkcję oraz przesunięcia. Wówczas łączna liczba operacji wyszłaby ... $O(n)$. Nie jest to oczywiście dowód, że P=NP, a tylko że niefrasobliwe przyjmowanie założenia upraszczającego uznającego operacje na liczbach (bez względu na ich długość) za operacje elementarne może prowadzić na manowce. Precyzyjne oszacowanie, uwzględniające liniowy czas wykonania operatora bitowego przywraca rezultat uzyskany w zadaniu 9.1.

Zadanie 9.2.
Problem SUMY PODZBIORU (wersja optymalizacyjna) sformułowano następująco: "Dany jest zbiór n różnych liczb naturalnych $A=\{a_1,...,a_n\}$ oraz liczba naturalna K. Spośród wszystkich podzbiorów $B \subseteq A$ o sumie elementów nie przekraczającej L ($\Sigma_{a \in B} a \leq L$) znajdź ten o maksymalnej sumie". Udowodnij, że jeżeli P≠NP, to dla żadnej liczby K nie może istnieć wielomianowy algorytm K–absolutnie przybliżony dla tego problemu.

Rozwiązanie.
Problem SUMY PODZBIORU w wersji decyzyjnej (SP) brzmi: "Dany jest zbiór n liczb naturalnych $A=\{a_1,...,a_n\}$ oraz liczba całkowita L. Czy istnieje podzbiór zbioru A o sumie elementów równej L?". Jest on NP–zupełny, jako oczywiste uogólnienie problemu PODZIAŁU ZBIORU. Wykażemy, że istnienie wielomianowego algorytmu K–absolutnie przybliżonego dla wersji optymalizacyjnej implikowałoby istnienie wielomianowego algorytmu rozstrzygającego SP. Procedura ta polegałaby na przemnożeniu wszystkich elementów zbioru A oraz limitu L przez K+1 i wywołaniu dla takich danych hipotetycznego algorytmu K–absolutnie przybliżonego dla wersji optymalizacyjnej. Wykażemy, że odpowiedź na SP jest TAK, wtedy i tylko wtedy, gdy algorytm przybliżony wywołany dla danych

$\{(K+1)a_1,...,(K+1)a_n\}$ i limitu $L(K+1)$ znajdzie podzbiór o sumie dokładnie równej $L(K+1)$. A zatem moglibyśmy w wielomianowym czasie zweryfikować problem NP–zupełny.

- Jest oczywiste, że jeśli algorytm przybliżony znajdzie w powyższych danych podzbiór X o sumie $L(K+1)$, to zbiór złożony z elementów X podzielonych przez $(K+1)$ jest podzbiorem zbioru A o sumie L, czyli odpowiedź na SP jest TAK.
- Odwrotnie, załóżmy, że odpowiedź na SP jest TAK. Istnieje zatem podzbiór zbioru A o sumie elementów L, wtedy optymalne rozwiązanie problemu optymalizacyjnego dla nowych danych ma sumę elementów dokładnie równą $L(K+1)$. Algorytm przybliżony zgodnie ze swoją definicją musi zwrócić podzbiór X zbioru $\{(K+1)a_1,...,(K+1)a_n\}$ o sumie elementów równej Ia, przy tym $|L(K+1)–\text{Ia}|\le L$. Ponadto musi być $\text{Ia}\le L(K+1)$, czyli $L(L+1)–K\le\text{Ia}\le L(K+1)$. Wreszcie Ia jest wielokrotnością $K+1$, jako że wszystkie elementy z X dzielą się przez $K+1$. Uwzględniając ten fakt jedyną dopuszczalną wartością jest $\text{Ia}=L(K+1)$.

Zadanie 9.3.
Dla problemu SUMY PODZBIORU w wersji optymalizacyjnej (patrz zadanie poprzednie) proponujemy następujący wielomianowy algorytm przybliżony: "Usuń z ciągu $a_1,...,a_n$ liczby większe od K, a pozostałe posortuj w porządek niemalejący i w tej kolejności umieszczaj je w tworzonym zbiorze „zachłannie” (tzn. nie wyjmując elementów wcześniej tam umieszczonych), o ile tylko ich suma nie przekracza L. Drugie rozwiązanie utwórz wybierając tylko jeden element – największy nie przekraczający L. Zwróć jako wynik to z obu rozwiązań, które ma większą sumę elementów". Wykaż, że jest to algorytm 2–względnie przybliżony.

Rozwiązanie.
Niech Io będzie wartością rozwiązania optymalnego, Ia – wartością otrzymaną z opisanego algorytmu, a S – sumą liczb w zbiorze A. Możemy od razu założyć, że liczby $a_1,...,a_n$ występują w kolejności niemalejącej i nie przekraczają L. Zgodnie z definicją wystarczy wykazać, że $\text{Io}/\text{Ia}\le 2$, czyli że $\text{Ia}\ge\text{Io}/2$. Rozważmy serię przypadków:

- Jeżeli cały ciąg $a_1+...+a_n\le L$, to nasz algorytm też zwróci cały ten ciąg jako swoje rozwiązanie, czyli Ia=Io.
- Jeżeli w ciągu $a_1,...,a_n$ istnieje element $a_i\ge K/2$, to nasz algorytm zwróci rozwiązanie $\text{Ia}\ge L/2\ge\text{Io}/2$.
- Jeżeli oba powyższe przypadki nie zachodzą, to istnieje pierwszy element naszego ciągu (powiedzmy i–ty), który "już się nie mieści w limicie", czyli $a_1+...+a_{i-1}\le L$ oraz $a_1+...+a_i>L$, wreszcie $a_i<L/2$. Łatwo widać, że dwie ostatnie nierówności dowodzą, że: $a_1+...+a_{i-1}\ge L/2\ge\text{Io}/2$, a nasz algorytm zwróci rozwiązanie o wartości $a_1+...+a_{i-1}$.

Podobnie konstruuje się 2–przybliżony algorytm dla optymalizacyjnej wersji problemu PLECAKOWEGO (włóż przedmioty o maksymalnej możliwej wartości przy ustalonym górnym limicie ich łącznego rozmiaru). Oba rozwiązania (wybieramy to o większej wartości) tworzymy poprzez:

1. posortowanie przedmiotów według niemalejącej wartości stosunku wartość/rozmiar i wybranie ich „zachłannie” w tej kolejności, dopóki jeszcze mieszczą się w plecaku,
2. wybranie "najwartościowszego" przedmiotu mieszczącego się w plecaku.

Łatwo zauważyć, że wersja optymalizacyjna problemu SUMY PODZBIORU jest szczególnym przypadkiem optymalizacyjnego problemu PLECAKOWEGO – takim

mianowicie, w którym wszystkie przedmioty mają rozmiary równe swoim wartościom (tj. funkcje r i w są równe) i liczby te są różne dla różnych przedmiotów. A zatem problem PLECAKOWY również niedopuszcza wielomianowych algorytmów *K*-absolutnie przybliżonych dla żadnego *K* (o ile P≠NP), choć istnieją dlań procedury 2-względnie przybliżone, a nawet jeszcze dokładniejsze.

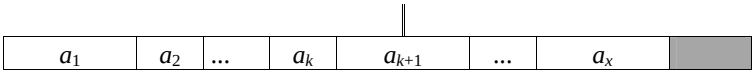
Modyfikacja kodu z zadania 9.1 (zastanów się, jaka?) pozwala też tworzyć pseudowielomianowe procedury rozwiązujące wersję decyzyjną i optymalizacyjną problemu PLECAKOWEGO np. o czasach działania $O(n^2 max\ \mathrm{r}(b_i)))$ lub $O(n^2 max\ \mathrm{w}(b_i)))$.

Problem MAKSYMALNEGO ZAŁADUNKU określony jest następująco. "Dany jest zbiór *n* przedmiotów $B=\{b_1,...,b_n\}$ o znanych rozmiarach $\mathrm{r}(b_1),...,\mathrm{r}(b_n)$ będących liczbami naturalnymi oraz dodatni rozmiar każdego z pudełek *R*. Ponadto znamy liczbę pudełek *L*. Znajdź upakowanie maksymalnej możliwej liczby przedmiotów". Zatem rozwiązaniem poprawnym jest dowolna rodzina *L* podzbiorów *B* postaci $B_i \subseteq B$ $i=1,...,L$ parami rozłącznych ($B_i \cap B_j = \emptyset$ dla $i \neq j$), takich że dla każdego z nich $\Sigma_{b \in B_i} \mathrm{r}(b) \leq R$. Wartością funkcji kryterialnej (którą maksymalizujemy) jest łączna liczba upakowanych przedmiotów $\Sigma_{i=1,...,L} |B_i|$. Łatwo zauważyć, że jest to problem NP–trudny (w dowodzie znów korzystamy z NP-zupełności PODZIAŁU ZBIORU).

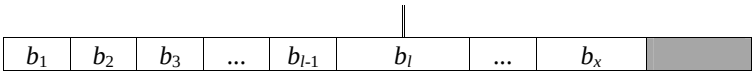
Zadanie 9.4.
Wykaż, że następujący zachłanny algorytm maksymalizacji załadunku do dwóch pudełek (podproblem powyższego, w którym ograniczamy się do danych z $L=2$) jest 1–absolutnie przybliżony:
Ustaw przedmioty w kolejności niemalejącego rozmiaru i wkładaj je kolejno do pierwszego pudełka tak, długo jak to możliwe. Począwszy od pierwszego przedmiotu, który nie zmieści się w pierwszym pudełku rozpocznij analogiczną procedurę dla drugiego pudełka.

Rozwiązanie.
Możemy przyjąć, że przedmioty $b_1,...,b_n$ są już ustawione w kolejności niemalejącego rozmiaru. Rozważmy dowolne rozwiązanie optymalne naszego zagadnienia, przedstawione symbolicznie na poniższym rysunku (pionowa kreska oddziela pudełka, szare obszary to „przestrzeń niewykorzystana", zaś literami a_i oznaczono ciąg przedmiotów wybranych z *B* w naszym optymalnym rozwiązaniu):

Będziemy teraz przekształcać rozwiązanie optymalne w pewien sposób. Najpierw zapomnijmy o „barierce" oddzielającej pudełka i „scalamy" je razem:



Następnie zamieniamy nasze x przedmiotów z optymalnego rozwiązania na najmniejszych x przedmiotów ustawionych w niemalejącej kolejności. Oczywiście łączny rozmiar wybranych przedmiotów po tej zamianie nie zwiększy się:



Jeżeli teraz „barierka oddzielająca pudełka" nie rozcina żadnego przedmiotu (inaczej, niż na powyższym rysunku) – wówczas wprowadzając ją uzyskujemy bezpośrednio rozwiązanie zwrócone przez nasz algorytm przybliżony, który nie może już zmieścić elementu b_{x+1} na mocy optymalności ciągu a_i. W przeciwnym razie „przesuwamy blok" przedmiotów $b_l,...,b_x$ w prawo do barierki, usuwając te przedmioty, które wyjdą poza „prawy koniec" prawego pudełka. Znów uzyskujemy rozwiązanie odpowiadające algorytmowi suboptymalnemu.

Ile przedmiotów może zostać usuniętych z powodu przekroczenia zakresu drugiego pudełka? Co najwyżej jeden, bowiem przesuwany blok przedmiotów został przemieszczony o odległość nie przekraczającą $r(b_l)$, zaś wszystkie kolejne przedmioty mają rozmiary większe lub równe $r(b_l)$. A zatem w kilku krokach przekształciliśmy nieznane rozwiązanie optymalne w rozwiązanie zwracane przez badany algorytm wielomianowy, przy czym wartość funkcji kryterialnej (liczby zapakowanych przedmiotów) zmniejszyła się o co najwyżej 1. Dlatego algorytm jest 1–absolutnie przybliżony.

Warto zauważyć, że zagadnienie MAKSYMALNEGO ZAŁADUNKU to jedna z prostych wersji optymalizacyjnych zdefiniowanego w zadaniu 8.15. problemu decyzyjnego PAKOWANIA PUDEŁEK. Problem ten w naturalny sposób prowadzi też do innych (bynajmniej nie równoważnych sobie) zagadnień decyzyjnych. Możemy np. zapytać o maksymalny możliwy łączny rozmiar przedmiotów, które jeszcze zmieszczą się w L pudełkach, lub o minimalną ilość pudełek wystarczającą do upakowania wszystkich przedmiotów. Zatem dla pewnych zagadnień decyzyjnych trudno znaleźć optymalizacyjny odpowiednik, inne zaś można łatwo skojarzyć z wieloma takimi zagadnieniami.

Zadanie 9.5.
Optymalizacyjna wersja problemu KOLOROWANIA GRAFÓW brzmi następująco: "Dany jest graf n–wierzchołkowy G. Pokoloruj go poprawnie wierzchołkowo używając najmniejszej możliwej liczby barw". Udowodnij, że jeżeli P≠NP, to nie może istnieć wielomianowy algorytm 1,33–względnie przybliżony dla tego zagadnienia.

Rozwiązanie.
Jak pamiętamy, problem decyzyjny KOL_3 polegający na sprawdzeniu, czy dla danego grafu G zachodzi $\chi(G)\leq 3$ jest NP–zupełny. Wykażemy, że istnienie algorytmu przybliżonego opisanego w treści zadania implikowałoby możliwość rozwiązania KOL_3 w czasie

wielomianowym. Rozwiązanie to polegałoby po prostu na pokolorowaniu danego grafu za pomocą algorytmu 1,33–przybliżonego i policzeniu barw użytych w uzyskanym pokolorowaniu. Niech bowiem algorytm przybliżony zwróci pewne pokolorowanie *c*.

- Jeżeli $\chi(G)>3$, to oczywiście liczba kolorów w *c* też jest większa od 3,
- Jeżeli $\chi(G)\leq 3$, to z nierówności $Ia/Io\leq 1{,}33$ wynika $Ia \leq 1{,}33Io=1{,}33\chi(G)\leq 1{,}33\times 3<4$, czyli liczba barw użytych w *c* jest $Ia\leq 3$.

Zadanie 9.6.
Problem KLIKI (wersja optymalizacyjna) brzmi następująco: "W danym grafie *n*–wierzchołkowym *G* znajdź podgraf pełny rozpięty na maksymalnej możliwej liczbie wierzchołków". Udowodnij, że jeśli P≠NP, to dla żadnej liczby *K* nie może istnieć wielomianowy algorytm *K*–absolutnie przybliżony dla tego problemu.

Rozwiązanie.
Wykażemy, że istnienie takiego algorytmu oznaczałoby możliwość poznania dokładnej wartości liczby klikowej w czasie wielomianowym, czyli rozwiązywałoby NP–zupełny problem KLIKI. Algorytm ten miałby następującą postać: Dla danego grafu *G* przez *G*' oznaczymy graf powstały z *K*+1 rozłącznych kopii grafu *G*, połączonych tak, by każde dwa wierzchołki pochodzące z różnych kopii były połączone krawędzią. Wówczas każda klika w *G*' ma zbiór wierzchołków, który przekrojony ze zbiorem wierzchołków dowolnej ze wspomnianych kopii *G* daje w tej kopii klikę. A zatem zachodzi następująca zależność pomiędzy liczbami klikowymi:

$$\omega(G')=(K+1)\omega(G).$$

Uruchomiony dla *G*' hipotetyczny algorytm przybliżony musiałby zwrócić podgraf pełny K_k o liczbie wierzchołków $k\leq\omega(G')$ oraz $k\geq\omega(G')-K$. Z tych dwóch nierówności, wiedząc, że $\omega(G')$ dzieli się przez *K*+1 można jednoznacznie określić $\omega(G')$, a więc i $\omega(G)$.

Podobnie dowodzimy niemożność konstrukcji wielomianowego algorytmu *L*–absolutnie przybliżonego dla problemów MAKSYMALNEGO ZBIORU NIEZALEŻNEGO i KOLOROWANIA GRAFÓW (o ile tylko P≠NP).

Można jednak wykazać znacznie więcej: nieistnienie dla żadnej stałej *K* *K*–względnie przybliżonego algorytmu wielomianowego dla któregokolwiek z tych trzech problemów, o ile tylko P≠NP. Wymienione zagadnienia optymalizacyjne są więc trudne nawet przy aproksymacji (absolutnej i względnej)!

A oto inny prosty przykład zagadnienia trudnego nawet przy aproksymacji:

Zadanie 9.7.
Udowodnij, że jeśli P≠NP, to dla żadnej liczby *K* nie może istnieć wielomianowy algorytm *K*–względnie przybliżony dla optymalizacyjnej wersji problemu KOMIWOJAŻERA.

Rozwiązanie.
Wykażemy, że istnienie takiego algorytmu oznaczałoby możliwość rozwiązania problemu CYKLU HAMILTONA w czasie wielomianowym, co implikowałoby P=NP. Mając dany graf *n*-wierzchołkowy *G* stworzylibyśmy graf pełny K_n o tym samym zbiorze wierzchołków

i krawędziach obciążonych wagami liczbowymi w sposób analogiczny, jak w α-redukcji przedstawionej w rozwiązaniu zadania 8.3. Tym razem jednak krawędziom nie występującym w G nadajemy wagi $(K+1)n$ zamiast 2. Jeżeli G zawiera cykl Hamiltona, to wyznacza on w K_n drogę komiwojażera o łącznym koszcie n, więc hipotetyczna procedura K-względnie przybliżona uruchomiona na tym grafie pełnym musiałaby zwrócić rozwiązanie w postaci cyklu o koszcie nie większym od Kn. Cykl ten nie mógłby zatem zawierać żadnej krawędzi nie występującej w G – reasumując, procedura K-względnie przybliżona byłaby zmuszona zwrócić jakiś cykl Hamiltona w G, rozwiązując tym samym (w połączeniu z wcześniejszą α-redukcją) problem CYKLU HAMILTONA w czasie wielomianowym. Sprzeczność.

Zadanie 9.8.
Zaprojektuj algorytm wielomianowy 1–absolutnie przybliżony dla problemu KOLOROWANIA GRAFÓW, działający dla przypadku grafów planarnych.

Rozwiązanie.
To, czy graf można pokolorować jednym kolorem (brak krawędzi) albo dwoma (dwudzielność) można sprawdzić w czasie liniowym. Jeśli tak nie jest – kolorujemy go algorytmem z dowodu twierdzenia o czterech barwach (czas pracy $O(n^2)$). Mamy wtedy $\chi(G)=3$ lub $\chi(G)=4$, a procedura użyje co najwyżej 4 kolorów, a zatem jest ona 1–absolutnie przybliżona.

Zadanie 9.9.
Wykaż, że algorytm SL jest 4–absolutnie przybliżony dla problemu z poprzedniego zadania.

Rozwiązanie.
Przypomnimy najpierw sposób działania SL. W danym grafie ustawiamy wszystkie wierzchołki w ciąg jak następuje: wierzchołek o minimalnym stopniu trafia na koniec, a z grafu usuwamy wszystkie incydentne z nim krawędzie. Dla pozostałej części grafu powtarzamy procedurę, ale skreślany wierzchołek ustawiamy bezpośrednio przed ostatnio znalezionym, itd. proces powtarzamy aż do wyczerpania się grafu. Następnie kolorujemy początkowy graf zachłannie według otrzymanej sekwencji: pierwszy wierzchołek dostaje kolor 1, a kolejne zawsze uzyskują najmniejszy kolor nie używany dotąd przez sąsiadów. Istnieje implementacja tego algorytmu (podobnie jak wielu klasycznych heurystyk sekwencyjnych) o czasie działania $O(n+m)$, gdzie dane podajemy w postaci listy sąsiadów. Jeżeli przypomnimy sobie następujący fakt:

> W każdym grafie planarnym istnieje wierzchołek stopnia nie większego niż 5 (tzw. pąk).

oraz zauważamy, że ze sposobu tworzenia sekwencji wierzchołków wynika, iż na każdym etapie wierzchołek właśnie kolorowany w podgrafie rozpiętym przez niego i jego pokolorowanych sąsiadów ma najmniejszy stopień (czyli sąsiadów tych jest co najwyżej 5), to uzyskamy:

> Dla każdego grafu planarnego algorytm SL użyje co najwyżej sześciu barw.

Możliwe wartości $\chi(G)$ to liczby od 1 do 4, zatem do zakończenia rozwiązania wystarczy zauważyć, że dla grafów o $\chi(G)=1$ (a więc bezkrawędziowych) procedura SL jest optymalna, zaś przy $\chi(G)\geq 2$ będzie zawsze $|Ia-\chi(G)|\leq 4$.

Zadanie 9.10.
Opracuj wielomianowy 4–względnie przybliżony algorytm dla problemu MAKSYMALNEGO ZBIORU NIEZALEŻNEGO w grafach planarnych.

Rozwiązanie.
Kolorujemy dany graf planarny algorytmem używającym co najwyżej czterech barw (patrz na ramkę po zadaniu 5.7), a następnie zwracamy jako wynik zbiór wierzchołków A, które uzyskały najczęściej występujący kolor. Oczywiście rozwiązanie to jest poprawne, gdyż wierzchołki o jednakowej barwie nie mogą sąsiadować, a zatem tworzą zbiór niezależny. Pozostaje wykazać 4–względną przybliżoność. Problem jest maksymalizacyjny, czyli ma zajść $|A|\geq|Io|/4$, gdzie Io jest rozwiązaniem optymalnym tzn. największym możliwym zbiorem niezależnym wierzchołków. My jednak wiemy więcej, mianowicie że zachodzi nierówność $|A|\geq n/4$, gdzie n jest liczbą wszystkich wierzchołków grafu – bowiem wierzchołki te zostały podzielone na co najwyżej 4 zbiory (o różnych kolorach), z których największym jest A. Teza została udowodniona.

Zadanie 9.11.
Optymalizacyjna wersja problemu KOLOROWANIA KRAWĘDZIOWEGO GRAFÓW brzmi następująco: "Dany jest graf n–wierzchołkowy G. Pokoloruj go krawędziowo używając najmniejszej możliwej liczby barw". Problem ten jest NP–trudny. Wykaż, że prosty algorytm zachłanny: „wybieraj krawędzie w dowolnej kolejności, nadając każdej z nich najmniejszy wolny (tzn. nie wykorzystany przez krawędzie sąsiednie) kolor” jest 2–przybliżony.

Rozwiązanie.
Krawędź może stykać się z co najwyżej $2(\Delta-1)$ innymi krawędziami, dlatego algorytm nasz nie użyje więcej, niż $2\Delta-1$ barw. Tymczasem w pokolorowaniu optymalnym potrzeba co najmniej Δ kolorów, algorytm nasz jest więc 2–przybliżony.

Istnieje jeszcze dokładniejsza, bo aż 1–absolutnie przybliżona procedura wielomianowa rozwiązująca powyższe zagadnienie. Jedyne możliwe wartości indeksu chromatycznego grafu to Δ i $\Delta+1$ i właśnie rozróżnienie pomiędzy tymi dwoma wielkościami jest NP–trudne (co więcej, pozostaje takim po ograniczeniu problemu do grafów o maksymalnym stopniu $\Delta=3$). Jednak algorytm z dowodu twierdzenia Vizinga, wykonujący się w czasie $O(m\Delta)$ (patrz: ramka po zadaniu 5.7) używa co najwyżej $\Delta+1$ barw, a zatem myli się on nie więcej niż o 1.

Zadanie 9.12.
Zaproponuj wielomianowy algorytm 4/3-przybliżony dla KOLOROWANIA KRAWĘDZIOWEGO GRAFÓW.

Rozwiązanie.
Grafy o indeksie chromatycznym nie przekraczającym 2 mają składowe spójności postaci: izolowanych wierzchołków, ścieżek bądź cykli parzystych (w innym grafie musimy użyć przynajmniej trzech barw). Krawędzie każdej z tych składowych można łatwo pokolorować

optymalnie – naprzemiennie barwami 1 i 2. Dla grafów innej postaci używamy procedury z twierdzenia Vizinga i wówczas Ia/$\chi'(G) \le (\chi'(G)+1)/\chi'(G) = 1+1/\chi'(G) \le 1+1/3 = 4/3$.

Podobnie jak w zadaniu 9.5 widzimy, że współczynnika 4/3 dla KOLOROWANIA KRAWĘDZIOWEGO także nie można poprawić (o ile P≠NP). Zatem nie może też istnieć wielomianowy schemat aproksymacyjny dla tego zagadnienia. Z drugiej strony kolorowanie krawędziowe, choć też NP-trudne, okazuje się łatwiejsze od wierzchołkowego w tym sensie, że (jak widzieliśmy) możemy skonstruować dlań procedury wielomianowe o gwarantowanej stałej dokładności absolutnej i względniej. Jeszcze inaczej zachowuje się (również NP-trudny) optymalizacyjny problem PLECAKOWY: choć nie dopuszcza algorytmów o stałym błędzie absolutnym (patrz ramka pod zadaniem 9.3), posiada on *wielomianowy schemat aproksymacyjny*, tj. dla dowolnie małego współczynnika $\varepsilon>0$ istnieje algorytm wielomianowy $(1+\varepsilon)$-przybliżony. Inaczej mówiąc, możemy dowolnie blisko (w sensie błędu względnego) zbliżyć się do optimum nie tracąc wielomianowości!

Zadanie 9.13.**
Zaprojektuj wielomianowy schemat aproksymacyjny dla optymalizacyjnej wersji problemu PLECAKOWEGO.

Wskazówka: użyj dokładnego algorytmu pseudowielomianowego (patrz ramka pod zadaniem 9.3).

Rozwiązanie.
Niech $B=\{b_1,...,b_n\}$ będzie zbiorem przedmiotów o rozmiarach $r(b_1),...,r(b_n)$ i wartościach $w(b_1),...,w(b_n)$ zaś R – rozmiarem plecaka (można założyć, że wszystkie $r(b_i) \le R$, przedmioty nie mieszczące się w plecaku bez zmniejszenia ogólności pomijamy w rozważaniach). Przez C oznaczymy poszukiwany przez nas podzbiór przedmiotów, które razem mieszczą się w plecaku ($\Sigma_{b\in C}\, r(b) \le R$) i mają największą możliwą łączną wartość $Io=\Sigma_{b\in C}\, w(b)$. Zdefiniujmy pewien współczynnik x oraz zmodyfikowane wartości przedmiotów równe $w'(b)=\lfloor w(b)/x \rfloor$ dla wszystkich $b\in B$. A zatem dla dowolnego przedmiotu b zachodzi:

$$\text{(i)} \qquad w(b) \ge x w'(b) \ge w(b) - x$$

Projektowana procedura polega na zastosowaniu wspomnianego dokładnego algorytmu pseudowielomianowego dla nowych w' – zwrócony przezeń zbiór przedmiotów C' będzie naszym rozwiązaniem. Jaką dokładność względną zapewnia taki algorytm? Otóż:

$$Ia=\Sigma_{b\in C'}\, w(b) \ge x \Sigma_{b\in C'}\, w'(b) \ge x \Sigma_{b\in C}\, w'(b) \ge \Sigma_{b\in C}\, w(b) - nx = Io - nx$$

gdzie druga nierówność wynika z faktu, że zbiór C' jest rozwiązaniem optymalnym przy wartościach przedmiotów $w'(b_1),...,w'(b_n)$, a pozostałe są konsekwencją (i). Jeżeli przyjmiemy nasz parametr $x=\delta \max w(b_i)/n$, to

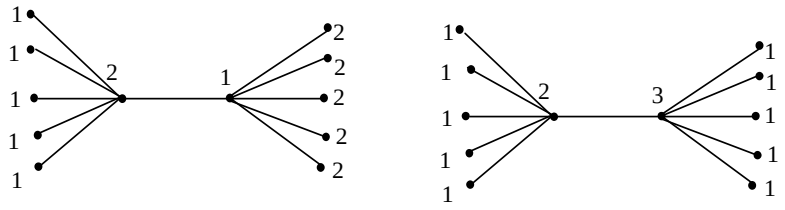
$$Ia \geq Io - \delta \max w(b_i) \geq (1-\delta)Io,$$

gdyż rozwiązanie optymalne nie może być gorsze od uzyskanego przez wybranie jednego przedmiotu. Stąd:

$$Io/Ia \leq 1/(1-\delta) = 1+\delta/(1-\delta)$$

Ułamek $\delta/(1-\delta)$ dąży do 0 gdy δ dąży do 0, więc chcąc uzyskać procedurę o Io/Ia dowolnie bliskim 1 wystarczy przyjąć odpowiednio małe δ i określić: $x=\delta \max w(b_i)/n$, $w'(b)=\lfloor w(b)/x \rfloor$ dla $b \in B$. Pozostaje sprawdzić, czy czas działania będzie wielomianowy. Wynosi on $O(n^2 \max w'(b_i)))$, ale $n^2 \max w'(b_i) \leq n^3/\delta$, co kończy dowód.

Problem SUMACYJNEGO KOLOROWANIA GRAFU (SKG) sformułowano następująco: "Dany jest graf *G* i liczba naturalna *L*. Sprawdź, czy jest możliwe legalne pokolorowanie wierzchołków grafu *G*, w którym jako kolorów użyto liczb naturalnych, tak że suma przypisanych barw po wszystkich wierzchołkach nie przekracza limitu *L*?". W wersji optymalizacyjnej rozwiązaniem poprawnym jest dowolne legalne pokolorowanie wierzchołkowe liczbami naturalnymi, ale kryterium optymalizacyjnym nie jest liczba różnych użytych barw (tak jak w modelu klasycznym), lecz ich suma. Problem ten jest NP–trudny nawet po ograniczeniu się do grafów dwudzielnych. Okazuje się, że w niektórych grafach kolorowanie o optymalnej sumie musi używać więcej niż $\chi(G)$ barw. Przykładowo rozważmy pokolorowanie poniższego drzewa o $2k$ liściach dla $k \geq 3$ (rysunek przedstawia sytuację z $k=5$):



Pokolorowanie po lewej stronie jest jedynym (z dokładnością do zamiany barw) optymalnym w sensie klasycznym, ale wartość funkcji kryterialnej dla modelu sumacyjnego wynosi tu $3k+3$. Tymczasem używające dodatkowej barwy 3 pokolorowanie po prawej stronie jest optymalne w sensie sumacyjnym – wartość sumy barw to $2k+5$. Jednak w przypadku grafów dwudzielnych kolorowania klasyczne są w pewnym sensie suboptymalne w stosunku do modelu sumacyjnego.

Zadanie 9.14.
Udowodnij że następujący algorytm o złożoności liniowej: „dany graf dwudzielny pokoloruj wierzchołkowo kolorami 1 i 2, a następnie (o ile to konieczne) zamień kolory tak, by barwa 2 występowała nie częściej od 1” jest 3/2–przybliżonym algorytmem dla problemu SUMACYJNEGO KOLOROWANIA grafów dwudzielnych. Wykaż, że współczynnika dokładności względnej 3/2 nie można zmniejszyć dla tego algorytmu.

Rozwiązanie.
Niech n będzie liczbą wierzchołków grafu, zaś n_1 i n_2 ilościami wierzchołków odpowiednio barwy 1 i 2. Oczywiście $n = n_1+n_2$ i $n_2 \le n_1$, czyli $n_2 \le n/2$. Rozwiązanie optymalne ma wartość funkcji kryterialnej $Io \ge n$, bo każdy wierzchołek ma kolor równy co najmniej 1, zaś wartość sumy barw dla pokolorowania suboptymalnego, to $Ia = n_1+2n_2 = n+n_2 \le 3/2n$. Zatem faktycznie $Ia/Io \le 3/2$, co dowodzi pierwszej części zadania. Aby pokazać, że współczynnika 3/2 nie można poprawić, zwróćmy raz jeszcze uwagę na graf z ostatniej ramki. Kolorowanie po lewej stronie odpowiada rozwiązaniu zwracanemu przez nasz algorytm, zaś po prawej mamy pewne pokolorowanie o lepszej wartości sumy barw. Widzimy, że stosunek funkcji kryterialnych $(3k+3)/(2k+5)$ wraz ze wzrostem k dąży do 3/2, czyli nie można go oszacować z góry mniejszą stałą.

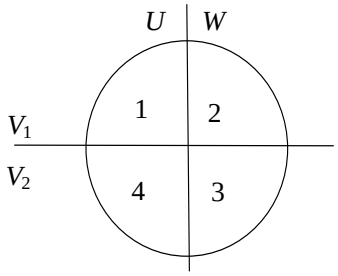
Problem MAKSYMALNEGO PRZEKROJU (MP) sformułowano następująco: "Dany jest graf G o zbiorze wierzchołków V, oraz liczba naturalna L. Sprawdź, czy jest możliwe rozbicie zbioru wierzchołków na dwie rozłączne części: $V=V_1 \cup V_2$, takie by ilość krawędzi łączących oba zbiory (czyli o jednym końcu w V_1 i drugim w V_2) była nie mniejsza od L?". W wersji optymalizacyjnej rozwiązanie poprawne stanowi dowolne rozbicie V na sumę dwóch rozłącznych części, zaś funkcją kryterialną (którą maksymalizujemy) jest liczba krawędzi łączących wierzchołki z różnych zbiorów. Wersja decyzyjna tak określonego problemu jest NP–zupełna, a zatem wariant optymalizacyjny jest NP–trudny.

Zadanie 9.15*.
Dany jest następujący algorytm rozwiązujący problem MAKSYMALNEGO PRZEKROJU w sposób przybliżony: „Podziel zbiór wierzchołków na dwie rozłączne części U i W w sposób dowolny, a następnie poszukaj dowolnego wierzchołka v, którego przeniesienie do drugiego zbioru zwiększy wartość funkcji kryterialnej. Dokonaj przeniesienia i powtarzaj taką operację tak długo, aż nie znajdzie się żaden odpowiedni wierzchołek v".

Wykaż, że algorytm ten ma wielomianową złożoność, oraz że jest on 2–przybliżony dla naszego zagadnienia. Udowodnij, że współczynnika dokładności względnej 2 nie można zmniejszyć dla tego algorytmu.

Rozwiązanie.
Wraz z każdą operacją przeniesienia wierzchołka zwiększamy wartość funkcji kryterialnej, którą stanowi moc pewnego zbioru krawędzi (tych, łączących wierzchołki z różnych zbiorów). Wielkość ta nie może przekraczać liczby wszystkich krawędzi m, dlatego pętla wyszukująca wierzchołek v wykona się nie więcej niż m razy i czas działania algorytmu faktycznie jest wielomianowy.

Przystąpimy teraz do dowodu 2–przybliżoności. Najogólniejszą sytuację możliwą do zastania po zakończeniu pracy programu przedstawiono symbolicznie na poniższym rysunku. Kółko oznacza zbiór wszystkich wierzchołków V, obszary oddzielone pionową linią to zbiory U i W zwrócone przez algorytm przybliżony, zaś rozwiązanie optymalne (być może inne, niż U i W) dają zbiory oddzielone linią poziomą. Sytuacja jest opisana przez rozbicie zbioru wierzchołków na cztery rozłączne podzbiory, które zostały symbolicznie ponumerowane.



Niech $E(ij)$ będzie liczbą krawędzi o jednym końcu w obszarze o numerze i, a drugim w j. Zatem wartość kryterium dla rozwiązania optymalnego, to:

$$\mathrm{Io} = E(13)+E(14)+E(23)+E(24),$$

zaś program nasz zwróci rozbicie o wartości:

$$\mathrm{Ia} = E(12)+E(13)+E(24)+E(34).$$

Mamy wykazać, że $\mathrm{Io} \le 2\mathrm{Ia}$, co jest równoważne nierówności:

(i) $E(14)+E(23) \le E(12)+E(34)+\mathrm{Ia}.$

Weźmy teraz dowolny wierzchołek v z U i zauważmy, że liczba krawędzi wychodzących z niego do wierzchołków z W jest większa lub równa liczbie krawędzi wychodzących do innych wierzchołków z U (wynika to z zasady działania algorytmu przybliżonego – gdyby było inaczej, wierzchołek ten znalazłby się w zbiorze W). Sumując taką nierówność po wszystkich wierzchołkach z U mamy:

$$2E(14) \le 2E(U) \le \mathrm{Ia},$$

gdzie $E(U)$ jest liczbą krawędzi łączących wierzchołki z U, a czynnik 2 wynika z faktu, że podczas sumowania każdą krawędź z U zliczyliśmy dwukrotnie. Podobnie rozważając wierzchołki z W otrzymamy:

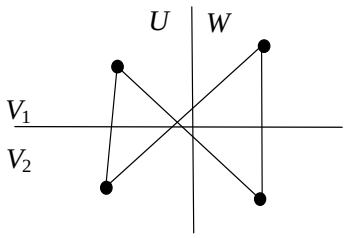
$$2E(23) \le 2E(W) \le \mathrm{Ia},$$

co po dodaniu stronami da nam nierówność:

$$E(14)+E(23) \le \mathrm{Ia}$$

w oczywisty sposób implikującą (i) i tym samym kończącą dowód 2–przybliżoności.

Na zakończenie rozważmy cykl C_4:



Podział na zbiory V_1 i V_2 ma funkcję kryterialną równą 4, podczas gdy algorytm nasz może zwrócić podział na U i W, jako że przeniesienie żadnego wierzchołka do drugiego zbioru nie spowoduje tu wzrostu wartości funkcji kryterialnej, w tym wypadku równej 2. Widzimy więc, że współczynnika dokładności względnej 2 nie można polepszyć.

Bardzo podobny problem MINIMALNEGO PRZEKROJU, w którym chcemy rozbić zbiór wierzchołków danego grafu na dwie rozłączne i niepuste części $V=V_1 \cup V_2$, tak by ilość łączących je krawędzi była najmniejsza z możliwych – jest wielomianowy. Zagadnienie to jest ściśle związane z badaniem *spójności krawędziowej* grafu: pytamy o minimalną możliwą liczbę krawędzi, które trzeba usunąć aby rozspoić dany graf spójny. Wyznaczanie spójności – zarówno krawędziowej, jak i wierzchołkowej grafów – jest możliwe w czasie wielomianowym.

Literatura

[1] A. Aho, J. Hopcroft, J. Ullman, „*Projektowanie i analiza algorytmów*”, Helion (2003).
[2] T. Cormen, C. Leiserson, R. Rivest , C. Stein, „*Wprowadzenie do algorytmów*”, PWN (2005).
[3] M. Garey, D. S. Johnson, „*Computers and Intractability. A Guide to the Theory of NP–Completeness*”, Freeman (1979).
[4] J. Hopcroft, J. Ullman, „*Wprowadzenie do teorii automatów, języków i obliczeń*”, PWN (1994).
[5] M. Kubale, „*Introduction to Computational Complexity and Algorithmic Graph Coloring*”, GTN (1998).
[6] M. Kubale, „*Łagodne wprowadzenie do analizy algorytmów*”, PG, Gdańsk (1999).
[7] M. Sysło, J. Kowalik, N. Deo, „*Algorytmy optymalizacji dyskretnej*”, PWN (1995).