Wykład ze Wstępu do Informatyki

Rok 2015-2016

Marek Zawadowski
Wydział Matematyki, Informatyki i Mechaniki
Uniwersytet Warszawski

31 maja 2015

Spis treści

1 Wstęp **3**
1.1 Literatura . . . 3
1.2 Zaliczenie i egzamin . . . 3
1.3 Historia Informatyki . . . 3

2 Wprowadzenie **4**
2.1 Algorytm Euklidesa . . . 4
2.2 Problem algorytmiczny . . . 4
2.3 Sortowanie liczb . . . 5
2.4 Analiza złożoności algorytmu . . . 5
2.5 Wieże Hanoi . . . 7
2.6 Wyszukiwanie słowa w słowniku . . . 7
2.7 Tablice rzeczywistego czasu działania algorytmów . . . 8
2.8 Komputer od środka . . . 9

3 Język Pascal **11**
3.1 Języki programowania wysokiego poziomu . . . 11
3.2 Diagramy składniowe . . . 11
3.3 Formalna definicja języka imperatywnego . . . 12
3.4 Zmienne . . . 13
3.5 Typy proste . . . 14
3.6 Typy strukturalne . . . 15
3.7 Przegląd instrukcji języka Pascal . . . 16
3.8 Procedury . . . 22
3.9 Procedury rekurencyjne . . . 26
3.10 Poprawność programów . . . 30

4 Podstawowe metody programowania **32**
4.1 Metoda powrotów (prób i błędów) . . . 32
4.2 Metoda 'dziel i rządź' . . . 36
4.3 Sortowanie przy pomocy porównań . . . 39
4.4 Programowanie dynamiczne . . . 41

4.5 Algorytmy zachłanne 45
4.6 Algorytm sortowania przez kopcowanie (heapsort) 48
4.7 Podsumowanie 51

5 Reprezentacja liczb na komputerze 53
5.1 Systemy liczbowe o różnych podstawach 53
5.2 Reprezentacja stałopozycyjna liczb całkowitych 54
5.3 Operacje arytmetyczne stałopozycyjne 56
5.4 Reprezentacja zmiennopozycyjna liczb rzeczywistych 57
5.5 Arytmetyka zmiennopozycyjna 58
5.6 Wybrane problemy numeryczne 60

6 Dynamiczne struktury danych 63
6.1 Podstawowe dynamiczne struktury danych 63
6.2 Typy wskaźnikowe 66
6.3 Implementacja list 69
6.4 Drzewa binarnych poszukiwań (BST) 73
6.5 Drzewa czerwono-czarne 78
6.6 Struktury danych dla rodziny zbiorów rozłącznych 90

7 Algorytmy grafowe 93
7.1 Grafy i reprezentacje grafów 93
7.2 Składowe spójne grafu niezorientowanego 95
7.3 Przeszukiwanie grafu wszerz (BFS) 97
7.4 Przeszukiwanie grafu w głąb (DFS) 102
7.5 Sortowanie topologiczne 106
7.6 Silnie spójne składowe grafu 108
7.7 Minimalne drzewo rozpinające 113
7.8 Znajdowanie najkrótszej ścieżki w grafie z wagami 119

8 Złożoność algorytmów 122
8.1 Problemy decyzyjne 122
8.2 Algorytmy weryfikujące 123
8.3 Redukowalność problemów i problem PNP 124
8.4 Problemy nieobliczalne 126
8.5 Metody przybliżone 127

1 Wstęp

1.1 Literatura

1. Ogólne wprowadzenie: D. Harel, Rzecz o istocie informatyki

2. Algorytmy:

   - T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms (Wprowadzenie do Algorytmów)
   - L. Banachowski, K. Diks, W. Rytter, Algorytmy i struktury danych
   - A.V. Aho, J.E. Hopcroft, J.D. Ulman, Projektowanie i analiza algorytmów komputerowych

3. Język Pascal:

   - M.Iglewski, J.Madey, S.Matwin, Pascal
   - R.K. Kott, Programowanie w języku Pascal

1.2 Zaliczenie i egzamin

Zaliczenie: program i kolokwium. Egzamin: pisemny, po obu semestrach. Szczegóły na stronie http://www.mimuw.edu.pl/ zawado/WInfo.html

1.3 Historia Informatyki

IV w. p.n.e. Euklides: algorytm Euklidesa (pierwszy niebanalny algorytm).

IX w n.e. Algorismus (Muhammad ibn Musa al-Kwarizmi =Muhammad syn Musy z Kworyzmu), algorytmy dodawania odejmowania, mnożenia, i dzielenia liczb dziesiętnych.

XIX w. n.e. Joseph Jaquard, maszyna tkacka sterowana algorytmem. Charls Babbage, maszyna różnicowa do obliczania wzorów matematycznych i projekt maszyny analitycznej, mechanicznego prototypu komputera.

1920-30 r. Alan Turing, Emil Post, John von Neuman, Kurt Gödel, Alnzo Church, Stephen Kleene: badania pojęcia funkcji obliczalnej.

1945 r. J. von Neuman, pierwszy komputer (U Pensylvenia) (?)

196- r. Informatyka staje sie nową dziedziną wiedzy.

2 Wprowadzenie

2.1 Algorytm Euklidesa

- Dane wejściowe: dwie liczby naturalne $m, n > 0$.
- Wynik: $NWD(m, n)$.

Opis algorytmu. Odejmuj liczbę większą od mniejszej aż do wyrównania liczb.
Zapis algorytmu.

```
a:=m; b:=n;
dopóki a<> b wykonuj   {NWD(a,b)=NWD(m,n), a,b>=1}
   jeśli a<b to b:=b-a
   w przeciwnym przypadku a:=a-b
wypisz(a)
```

W algorytmie użyliśmy następujących *operacji elementarnych*:

1. cztery instrukcje przypisania;
2. iteracja nieograniczona (pętla while);
3. instrukcja warunkowa;
4. instrukcja wejścia wyjścia.

W nawiasach { } zapisaliśmy *niezmiennik pętli*, tzn. takie zdanie które jeśli jest prawdziwe przy wejściu do pętli to pozostaje prawdziwe po każdym pełnym wykonaniu pętli.

Czy program robi to co chcemy? Tak:

1. Po każdym wykonaniu pętli prawdziwy jest niezmiennik pętli $NWD(a, b) = NWD(m, n), a, b \geq 1$.
2. Po wyjściu z pętli $a = b$. Zatem $a = NWD(a, b)$.
3. Pętla wykonuje się co najwyżej $m+n-2$ razy (w szczególności nie może działać w nieskończoność).

Ad 1. Wystarczy pokazać, że jeśli $a > b$ to $NWD(a, b) = NWD(a - b, b)$. W tym celu wystarczy pokazać, że liczba k dzieli a i b wiw gdy dzieli $(a - b)$ i b.

Ad 2. Oczywiste.

Ad 3. Każde wykonanie instrukcji warunkowej zmniejsza sumę $a+b$ o co najmniej 1. Z drugiej strony mamy $a \geq 1$, $b \geq 1$. Zatem instrukcja warunkowa może być wykonana co najwyżej $m + n - 2$ razy.

2.2 Problem algorytmiczny

Problem (zadanie) algorytmiczny polega na

- scharakteryzowaniu wszystkich poprawnych danych wejściowych;
- scharakteryzowaniu oczekiwanych wyników jako funkcji danych wejściowych.

Rozwiązanie algorytmiczne polega na podaniu *algorytmu* tzn. takiego *opisu działań przy pomocy operacji elementarnych*, który zastosowany do poprawnych danych wejściowych daje oczekiwane wyniki.

Rozróżniamy *wykonywanie algorytmów* od *działania twórczego.*

Problemy dotyczące algorytmów:

1. *Język:* jaki jest zbiór instrukcji elementarnych?
2. *Rozstrzygalność:* czy istnieje algorytm rozwiązujący dany problem?
3. *Analiza poprawności:* czy algorytm działa poprawnie, tzn. robi to co ma robić?
4. *Analiza złożoności:* czy algorytm działa szybko?
5. *Analiza numeryczna:* czy algorytm działa dokładnie?

2.3 Sortowanie liczb

- Dane wejściowe: liczba naturalna n i ciąg liczb $a_1, a_2, \ldots, a_n$.
- Wynik: permutacja $a'_1, a'_2, \ldots, a'_n$ ciągu $a_1, a_2, \ldots, a_n$ taka, że $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Przykład. Dane: $2, 7, 4, 5, 1$. Wynik: $1, 2, 4, 5, 7$.

Jak do tego problemu podejść systematycznie? Na przykład tak jak sortujemy rozdane karty w brydżu.

Zapis algorytmu (sortowanie przez wkładanie). ($A[j]$ - j-ty element).

```
dla j:=2 do n wykonuj
  k:=A[j];
  i:=j-1;
  dopóki i>0 oraz A[i]>k wykonuj
    A[i+1]:=A[i];
    i:=i-1;
  A[i+1]:=k;
```

Przykład.

$$\begin{array}{rccccc} Dane: & 2 & 7 & 4 & 5 & 1 \\ & 2 & 4 & 7 & 5 & 1 \\ & 2 & 4 & 5 & 7 & 1 \\ Wynik: & 1 & 2 & 4 & 5 & 7 \end{array}$$

2.4 Analiza złożoności algorytmu

Analiza złożoności algorytmu jest to przewidywanie ile *zasobów* potrzeba do wykonania algorytmu.

Zasoby:

1. pamięć;
2. połączenia komunikacyjne;
3. czas działania (dla nas najważniejszy).

Żeby analizować złożoność algorytmu musimy coś wiedzieć o tym jak on będzie wykonywany przez maszynę.

My będziemy zakładać, że programy są wykonywane na *maszynie o dostępie swobodnym (RAM)*: tzn. instrukcje są wykonywane jedna po drugiej (nigdy dwie na raz) i program nie może się modyfikować w trakcie działania.

Czas działania zależy od rozmiaru danych wejściowych. 5 liczb nasz algorytm sortuje szybciej niż 1000. Także dla dwóch ciągów równej długości algorytm może wykonywać różną liczbę instrukcji w zależności od tego jak bardzo różnią się one od ciągu posortowanego. Na ogół, czas działania algorytmu wyrażany jest jako funkcja rozmiaru danych wyjściowych.

1. *Rozmiar danych wejściowych* jest to funkcja przyporządkowująca poprawnym danym wejściowym algorytmu liczbę naturalną.

2. *Czas działania algorytmu* jest to funkcja przyporządkowująca danym wejściowym liczbę podstawowych operacji wykonywanych przez algorytm na tych danych.

3. *(Pesymistyczna, czasowa) złożoność algorytmu* jest to funkcja z $\mathbf{N}$ w $\mathbf{N}$ przyporządkowująca liczbie naturalnej n najdłuższy czas działania algorytmu na danych o rozmiarze n.

Dla problemu sortowania rozmiar danych to długość ciągu.

```
nr                                   czas
 1 dla j:=2 do n wykonuj            | n
 2   k:=A[j];                       | n-1
 3  i:=j-1;                         | n-1
 4  dopóki i>0 oraz A[i]>k wykonuj  | t_2+...+t_n
 5    A[i+1]:=A[i];                 | (t_2-1)+...+(t_n-1)
 6    i:=i-1;                       | (t_2-1)+...+(t_n-1)
 7  A[i+1]:=k;                      | n-1
```

- t_j - liczba wykonań linii 4 przy ustalonym j.
- t_j - jest największy gdy macierz jest uporządkowana w porządku malejącym, wtedy $t_j = j$.
- $T(n)$ - złożoność algorytmu.

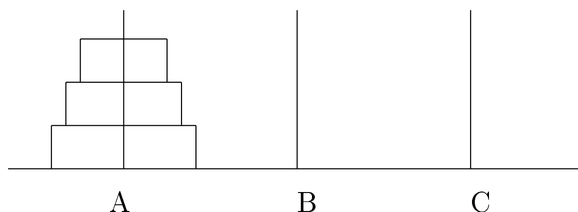
$$T(n) = 3(n-1) + n + 2\sum_{j=2}^{n}(j-1) + \sum_{j=2}^{n} j =$$

$$= 3(n-1) + n + 2\frac{n(n-1)}{2} + \frac{n(n+1)}{2} - 1 = \frac{3}{2}n^2 + \frac{7}{2}n - 4$$

To jest ciągle 'za dokładnie', składniki $\frac{7}{2}n$ i -4 oraz stała $\frac{3}{2}$ nie mają większego znaczenia, przy dużych n. To co jest ważne to n^2. Mówimy, że algorytm sortowania przez wkładanie ma złożoność (pesymistyczną, czasową) $O(n^2)$.

Notacja $O(f(n))$. Niech $f : \mathbf{N} \to \mathbf{N}$ funkcja. Mówimy, że *funkcja $g : \mathbf{N} \to \mathbf{N}$ jest (klasy) $O(f(n))$* (piszemy $g \in O(f(n))$ lub wręcz $g = O(f(n))$) jeśli istnieją stałe $a, b \in \mathbf{R}$ takie, że dla $n > b$, $g(n) \leq a * f(n)$ ('g przyjmuje wartości nie większe niż f z dokładnością do stałej').

2.5 Wieże Hanoi



Problem wież Hanoi. Przenieść pojedynczo n krążków z wieży A na wieże B używając wieży C tak by nigdy krążek większy nie leżał na mniejszym.

Opis algorytmu. Aby przenieść n krążków z A na B przez C

1. przenieś $n-1$ krążków z A na C używając B;
2. przenieś krążek z A na B;
3. przenieś $n-1$ krążków z C na B używając A.

Zapis algorytmu.

```
procedura przenies(m,X,Y,Z); {przenosi krążki z X na Y używając Z}
  jeśli m=1 to przestaw(X,Y)
            w przeciwnym przypadku
               przenies(m-1,X,Z,Y);
               przestaw(X,Y)
               przenies(m-1,Z,Y,X);

przenies(n,A,B,C) {wywołanie początkowe}
```

Przykład. $n = 3$. ...

Ile przestawień wykona algorytm by przestawić n krążków?

- a_n - liczba przestawień n krążków.

Równanie rekurencyjne:

$$\begin{cases} a_1 = 1 \\ a_{n+1} = a_n + 1 + a_n = 2a_n + 1 \end{cases}$$

Rozwiązanie: $a_n = 2^n - 1$.

Dowód indukcyjny. Dla $n = 1$, $2^1 - 1 = 1 = a_1$. Załóżmy, że $a_n = 2^n - 1$. Wtedy

$$a_{n+1} = 2a_n + 1 = 2(2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1$$

Liczba przestawień jest proporcjonalna do ilości wszystkich operacji wykonywanych przez algorytm. Zatem cały algorytm działa w czasie $O(2^n)$.

2.6 Wyszukiwanie słowa w słowniku

Problem wyszukiwania słowa w słowniku.

- Dane wejściowe: liczba naturalna n i ciąg słów $w_1, \ldots, w_n$ uporządkowany w porządku leksykograficznym (alfabetycznym) oraz słowo w.

- Wynik: TAK, gdy dla pewnego $1 \leq i \leq n$, $w = w_i$; NIE, w przeciwnym przypadku.

Przykład. Dane: $\vec{w} =' a','ala','b','bela','hela'$, $w =' bela'$. Wynik: TAK.

Poniższa procedura mem sprawdza czy słowo występuje w słowniku pomiędzy słowami w_{m1} i w_{m2}.

```
procedura mem(m1,m2);
  jeśli m1=m2 to
    jeśli w=w_m1 to wypisz('TAK')
      w przeciwnym przypadku wypisz('NIE')
    w przeciwnym przypadku
      m3:= (m1+m2) div 2;
      jeśli w>w_m3 to mem(m3+1,m2)
        w przeciwnym przypadku mem(m1,m3)

mem(1,n) (wywołanie początkowe)
```

Rozmiar danych: długość ciągu.
p_n -liczba porównań słów dla słownika długości n.
Równanie rekurencyjne:

$$\begin{cases} p_1 = 1 \\ p_{2n} = p_n + 1 \end{cases}$$

Rozwiązanie: $p_n \sim \log n$.
Liczba porównań jest rzędu $\log n$. Algorytm działa w czasie $O(\log n)$.

2.7 Tablice rzeczywistego czasu działania algorytmów

W poniższej tabeli przedstawiony jest rozmiar zadań jakie można rozwiązać w ciągu jednej sekundy, minuty, godziny.

Zakładamy, że do wykonania operacji podstawowej potrzebna jest jedna milisekunda ($= 10^{-3}$s).

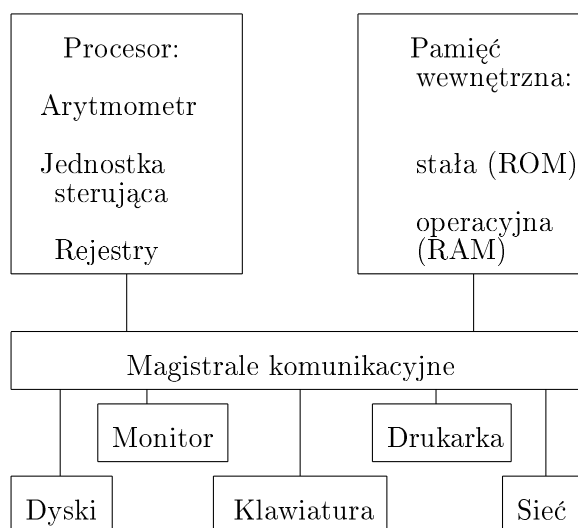
nr	Algorytm	Złożoność	Maksymalny rozmiar zadania		
			1 sekunda	1 minuta	1 godzina
A1	szukanie słowa w słowniku	$O(\log n)$	2^{1000}	-	-
A2	znajdowanie maksimum w tablicy	$O(n)$	1000	$6 * 10^4$	$3.6 * 10^6$
A3	sortowanie przez 'scalanie', 'kopcowanie'	$O(n * \log n)$	140	4893	$2 * 10^5$
A4	sortowanie przez 'wkładanie',	$O(n^2)$	31	244	1897
A5		n^3	10	39	153
A6	Wieże Hanoi	$O(2^n)$	9	15	21

A teraz przypuśćmy, że zwiększymy szybkość komputera 10 razy. Poniższa tablica pokazuje o ile zwiększy się maksymalny rozmiar zadania który można rozwiązać po przyspieszeniu.

nr	Algorytm	Złożoność	Maksymalny rozmiar zadania przed przyspieszeniem.	Maksymalny rozmiar zadania po przyspieszeniu.
A1	szukanie słowa w słowniku	$O(\log n)$	s_1	s_1^{10}
A2	znajdowanie maksimum w tablicy	$O(n)$	s_2	$10 * s_2$
A3	sortowanie przez 'scalanie', 'kopcowanie'	$O(n * \log n)$	s_3	około $10 * s_3$ dla dużych n
A4	sortowanie przez 'wkładanie',	$O(n^2)$	s_4	$3.16 * s_4$
A5		n^3	s_5	$2.15 * s_5$
A6	Wieże Hanoi	$O(2^n)$	s_6	$s_6 + 3.3$

2.8 Komputer od środka

Schemat logiczny komputera



- *Procesor* przetwarza informacje i steruje pozostałymi elementami systemu.
- *Pamięć* służy do przechowywania informacji.
- *Układy wejścia-wyjścia* (Dyski, Monitor, Klawiatura, Drukarka, Sieć) umożliwiają komunikację komputera ze światem zewnętrznym.
- *Magistrale komunikacyjne* łączą moduły komputera.

Komputer działa powtarzając *cykle rozkazowe.* Na jeden cykl rozkazowy składa się wiele operacji. W pewnym przybliżeniu można je przedstawić następująco:

1. pobranie kolejnego rozkazu z komórki pamięci wskazywanej przez licznik rozkazów;

2. sprawdzenie czy rozkaz wymaga pobrania danych, jeśli tak, to wyznaczenie miejsc w pamięci z których należy pobrać dane i umieszczenie danych w rejestrach komputera;
3. wykonanie rozkazu (arytmometr);
4. wysłanie wyniku pod właściwy adres w pamięci;
5. zmiana zawartości licznika rozkazów, tak by wskazywał kolejny rozkaz dla procesora;
6. obsługa przerwań (o ile takie mają miejsce);
7. przejście do kroku 1. w celu wykonania następnego cyklu rozkazów.

Od pomysłu algorytmu do wykonania programu przez maszynie jest szereg kroków do wykonania. Pierwsze kroki są wykonywane człowieka a następne przez maszynę. Można wyszczególnić następujące etapy tego procesu:

1. Pomysł algorytmu (człowiek);
2. Algorytm (człowiek);
3. Program w języku wysokiego poziomu (programista);
4. Program w języku adresów symbolicznych, asemblerze, (kompilacja, maszyna);
5. Kod maszynowy (dalsza kompilacja, maszyna);
6. Wykonanie kodu na komputerze (maszyna).

3 Język Pascal

3.1 Języki programowania wysokiego poziomu

Języki programowania wysokiego poziomu są to sformalizowane języki służące do zapisu algorytmów.

Typy języków programowania wysokiego poziomu:

1. *imperatywne*: Pascal, C, Basic, Fortran, Cobol, APL, Algol, Forth, ...
2. *funkcyjne*: ML, Miranda, Haskel, ...
3. *programowanie w logice*: Prolog.
4. *programowanie zorientowane obiektowo*: SmallTalk, C^{++}...
5. *programowanie równoległe*: Occam, Concurrent Pascal, ...

Na opis języka programowania składa się:

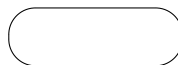
1. Precyzyjna *składnia* tzn. dokładne określenie co jest dopuszczalnym programem w tym języku.
2. Jednoznaczna *semantyka* tzn. jednoznaczny opis każdego wyrażenia dozwolonego składniowo.
   (a) *Semantyka operacyjna*: opis stanu komputera przed i po wykonaniu instrukcji.
   (b) *Semantyka denotacyjna*: opis funkcji przekształcającej dane wejściowe w dane wyjściowe.

3.2 Diagramy składniowe

Składnię języka programowania można opisywać graficznie przy pomocy *diagramów składniowych* lub tekstowo przy pomocy notacji BNF. My opiszemy składnie języka Pascal graficznie.

W diagramie składniowym obiekt definiowany występuje jako podpis do rysunku definiującego. Symbole:

1. Blok owalny

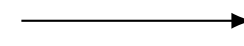
   

   obejmuje symbole oznaczające same siebie.

2. Blok prostokątny

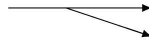
   obejmuje pojęcie zdefiniowane gdzie indziej.

3. Strzałka



wskazuje kolejność symboli w napisie złożonym.

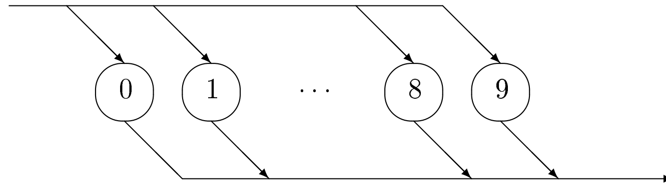
4. Rozgałęzienie



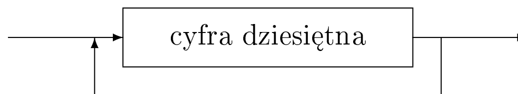
oznacza alternatywę definicyjną - można wybrać dowolną ze strzałek.

Przykłady

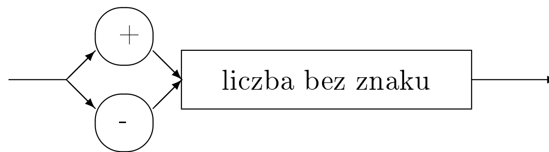
cyfra dziesiętna



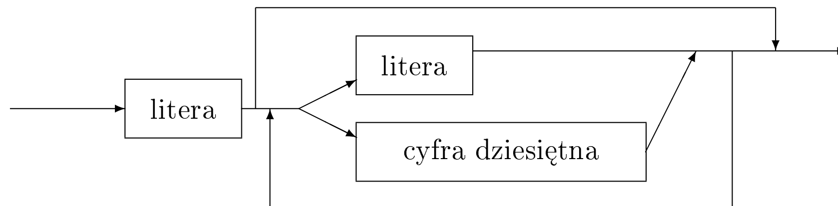
liczba bez znaku



liczba



identyfikator



3.3 Formalna definicja języka imperatywnego

Na tekst programu składają się

1. opis struktur danych, tzn. opis obiektów na których działa algorytm; w programie: definicje i deklaracje.
2. opis procesu obliczeniowego; w programie: instrukcje.

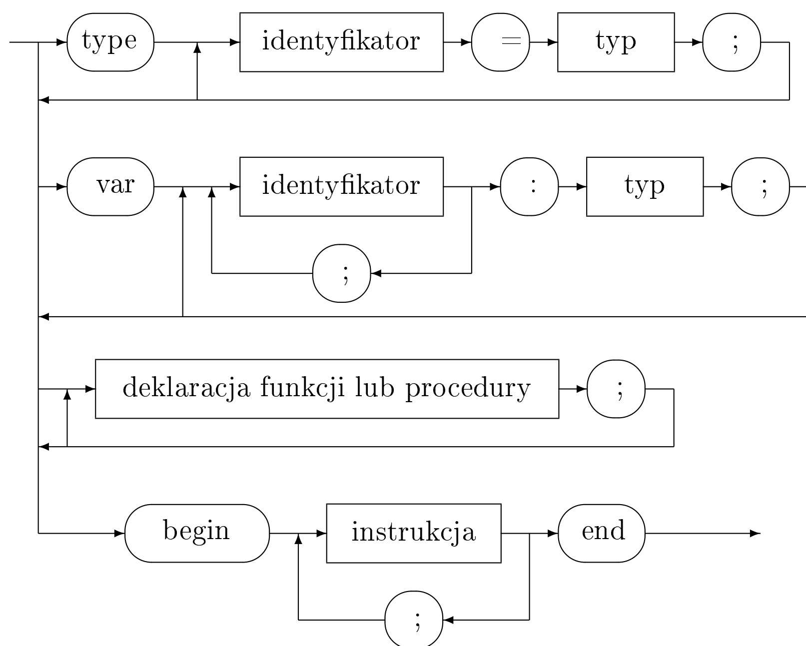
Czasem zawiera się to w następującej 'równości':

```
program = algorytm + struktury danych
```

Formalną definicja (fragmentu) języka Pascal można przedstawić tak:

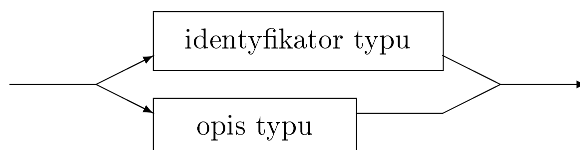
program

blok



W powyższym diagramie występują kolejno sekcja definicji typów, sekcja deklaracji zmiennych, sekcja deklaracji funkcji i procedur oraz program główny.

typ



Jednak dalszy opis języka Pascal przedstawimy mniej formalnie.

3.4 Zmienne

1. Zmienna i jej nazwa: *zmienną* możemy utożsamiać z obszarem pamięci, w którym przechowywana jest pewna wartość (wartość tej zmiennej w postaci kodu dwójkowego). *Nazwa zmiennej (identyfikator)* to mnemotechniczny adres tego obszaru pamięci.

   NB. Dla różnych zmiennych obszar pamięci może być różny.

2. *Typ zmiennej* wyznacza wielkość obszaru pamięci przeznaczonego na daną zmienną. Aby poprawnie skompilować program musimy poinformować kompilator o zamiarze wykorzystania każdej zmiennej (wyjątki od tej reguły poznamy później). Taka informacja to deklaracja zmiennej (lub stałej).

Deklaracje zmiennych w języku Pascal mają następująca postać:

```
var nazwa1, nazwa2 : typ1;
    nazwa3 : typ3;
    ....
```

Słowo 'var' jest słowem kluczowym rozpoczynającym sekcję deklaracji zmiennych. Deklaracja zmiennych wprowadza identyfikatory zmiennych wymienione po lewej stronie deklaracji, i zapowiada, że będą one używane dla oznaczania wartości typu podanego po prawej stronie deklaracji.

Definicje *stałych* w języku Pascal mają następującą postać:

```
const stala1='opis stalej';
    ....
```

Słowo 'const' jest słowem kluczowym rozpoczynającym sekcję definicji stałych. Stałe, podobnie jak zmienne, przechowują wartości różnych typów ale nie mogą być modyfikowane podczas realizacji programu.

Przykład

```
const zakres=100;
      pi=3.14;
      liczba=17;
      znak='a';
      ciag_znakow='ala';
```

3.5 Typy proste

Podstawowymi typami języka Pascal są *typy proste*. Przy ich pomocy definiuje się bardziej złożone *typy strukturalne*.

Typy standardowe

typ	identyfikator typu	przykładowe elementy typu	funkcje i relacje
logiczny	boolean	true, false	and, or, not
całkowity	integer	-2, 1, 1000	+,-,*, div, mod,< !
znakowy	char	'a','1','+'	
rzeczywisty	real	10,1.7,1,2E4	+,-,*,/
łańcuchowy	string	'ala'	+,<

Typy *logiczny*, *całkowity* i *znakowy* są typami *porządkowymi*. Na elementach typu porządkowego T są określone funkcje *ord* przekształcającą typ T w typ *integer* oraz funkcje poprzednika i następnika

$$succ, pred : T \longrightarrow T$$

(*succ* nie jest zdefiniowany dla ostatniego elementu typu T, a *pred* nie jest zdefiniowany dla pierwszego elementu typu T).

Przykład deklaracji zmiennych:

```
var x,y,z : real;
    p,q : boolean;
    litera : char;
    s1,s2 : string;
    m,n : integer;
```

Używając funkcji i relacji tworzymy ze zmiennych i stałych *wyrażenia*.

Przykłady wyrażeń:

1. $(x + y)/z$ - wyrażenie typu *real*;
2. $(x + y) < z$ - wyrażenie typu *boolean*;
3. s1+s2 - wyrażenie typu *string*;
4. s1+litera - wyrażenie typu *string*;
5. (p and $((s1 + litera) < z)$) or not q - wyrażenie typu *boolean*;
6. n mod m - wyrażenie typu *integer*.

Typy wyliczeniowe

Typy *wyliczeniowe* są definiowane przez wyliczenie identyfikatorów elementów typu.

Przykład

```
type dzien_tygodnia = (pon,wt,sr,czw,pt,sob,niedz);
     kolor=(czerwony,zielony,niebieski);
```

Typy okrojone

Typy *okrojone* są definiowane przez ograniczenie typu porządkowego.

Przykład

```
type dzien_roboczy = (pon..pt);
     mala_liczba=(1..30);
```

Typy wyliczeniowe i okrojone też są typami *porządkowymi*.

3.6 Typy strukturalne

Typy strukturalne są definiowane z wcześniej zdefiniowanych typów przy pomocy tzw. konstruktorów typów (operacji na typach).

Tablice

Deklaracja tablicy:

```
type tablica=array[T1,...,Tn] of T;
```

gdzie $T1, \ldots Tn$ są typami porządkowymi a T dowolnym typem. Teoriomonogościowo tablicom typu `tablica` odpowiadają funkcje z produktu kartezjańskiego `T1`× ... ×`Tn` w `T`.

Przykład definicji typów tablicowych:

```
type tablica1 = array[1..10] of char;
     tablica2 = array[dzien_roboczy,mala_liczba] of real;
     tablica3 = array['a'..'z'] of integer;
```

Wtedy `tablica1[7]` jest znakiem, `tablica2[wt,3]` jest liczbą rzeczywistą, a `tablica3['c']` jest liczbą całkowitą.

Rekordy

Deklaracja rekordu:

```
type rekord1=record p1:T1;
                   p2:T2;
                   ...
                   pn:Tn
             end;
```

gdzie `rekord1` jest identyfikatorem definiowanego typu rekordowego `T1` ... `Tn` są identyfikatorami typów a `p1` są identyfikatorami pól. Wszystkie identyfikatory pól muszą być różne. Jeśli natomiast identyfikatory typów są równe, możemy pola odpowiednich typów umieścić na tej samej liście. Na przykład jeśli `T1` i `T2` są równe to powyższy rekord możemy zdefiniować też tak:

```
type rekord2=record p1,p2:T1;
                   p3:T3;
                   ...
                   pn:Tn
            end;
```

Teorio-mnogościowo rekordy typu `rekord1` to n-tki uporządkowane, elementy produktu kartezjańskiego T1× ... ×Tn.

Przykład

```
type student = record
                 nazwisko,imie:string;
                 rok,nr:integer;
                 srednia:real
               end;
var s1,s2:student;
```

Wtedy `s1.nazwisko` jest łańcuchem, `s1.rok` liczbą całkowitą, a `s1.srednia` liczbą rzeczywistą.

Zbiory, **Pliki**, na ćwiczeniach.

Typy wskaźnikowe *Typy wskaźnikowe* służą do konstrukcji dynamicznych struktur danych. Będą one omawiane w drugim semestrze.

Z typami związane są

1. operacje na elementach danego typu;
2. sposób dostępu do informacji przechowywanych w zmiennych i stałych danego typu;
3. konstrukcje programotwórcze służące do przeszukiwania elementów typów strukturalnych.

3.7 Przegląd instrukcji języka Pascal

Poniżej opiszemy kolejno podstawowe instrukcje języka Pascal i ich znaczenia.

Instrukcja przypisania

Instrukcja ma postać

```
a:=w
```

gdzie `a` jest zmienną a `w` jest wyrażeniem tego samego typu co zmienna `a`.

Przykład. Przy deklaracji zmiennych

```
var a,x:real;
      p:boolean;
    n,m:integer;
```

możemy na przykład dokonać takich podstawiań

```
a:=(x+n)/2;
n:=n*m;
n:=n+2;
p:=(a<x) or (n=0);
```

Opis logiczny instrukcji. Znaczenie tej instrukcji opisuje aksjomat

$$P(a\backslash w)\{\ a := w\ \}P \tag{1}$$

gdzie P jest dowolną formułą, a $P(a\backslash w)$ jest formułą powstałą z P przez zastąpienie wszystkich[1] wystąpień zmiennej a wyrażeniem w. Cała formula (1) oznacza, że jeśli przed wykonaniem instrukcji `a:=w` spełniony jest warunek $P(a\backslash w)$ to po wykonaniu instrukcji `a:=w` spełniony jest warunek P.

Przykłady zastosowania aksjomatu (1):

$$(b = (x + n) + 1)\{a := x + n\}(b = a + 1)$$

czyli, jeśli przed wykonaniem instrukcji $a := x + n$ zachodzi $(b = (x + n) + 1)$ to po jej wykonaniu zachodzi $(b = a + 1)$.

$$(n + 1 < k \wedge n + 1 > 0)\{n := n + 1\}(n < k \wedge n > 0)$$

czyli, jeśli przed wykonaniem instrukcji $n := n + 1$ zachodzi $(n + 1 < k \wedge n + 1 > 0)$ to po jej wykonaniu zachodzi $(n < k \wedge n > 0)$.

Innymi słowy instrukcja podstawiania jest wykonywana w ten sposób, że najpierw wyliczamy wartość wyrażenia po prawej stronie a potem wstawiamy wyliczoną wartość na zmienną po lewej stronie.

Do opisu znaczenia instrukcji przypisania użyliśmy *zapisu logicznego.* Ogólnie zapis logiczny ma postać

$$P\{\ I\ \}Q$$

gdzie P i Q są formułami a I instrukcją, i oznacza, że jeśli przed wykonaniem instrukcji I prawdziwa jest formuła P to po jej wykonaniu (o ile wykonywanie instrukcji się zakończy) prawdziwa jest formuła Q. P nazywamy *warunkiem początkowym* a Q *warunkiem końcowym.*

Prawdziwa jest następująca *reguła wnioskowania*

$$\frac{P\{\ I\ \}Q, \quad Q\{\ J\ \}R}{P\{\ I; J\ \}R} \tag{2}$$

Ta reguła, jak i inne reguły wnioskowania, pozwala wywnioskować formułę pod kreską o ile ustalimy prawdziwość formuł nad kreską. W tym przypadku reguła (2) wyraża to, że `;` łączy dwie instrukcje nic w nich nie zmieniając.

[1] Są tu pewne ograniczenia które nas w praktyce nie będą dotyczyć. Gdy formuła P ma kwantyfikatory to wyrażenie w wstawiamy na tak zwane *wolne* wystąpienia zmiennej a. Ponadto, takie podstawienie nie może wiązać żadnej ze zmiennych występujących w wyrażeniu w.

Często użyteczna jest reguła która nieznacznie uogólnia regułę (2)

$$\frac{P \Rightarrow P', \quad P'\{\ I\ \}Q, \quad Q \Rightarrow Q', \quad Q'\{\ J\ \}R, \quad R \Rightarrow R'}{P\{\ I; J\ \}R'} \tag{3}$$

Przykład. Zamianę wartość zmiennych x i y dowolnego typy T można wykonać używając dodatkowej zmiennej z typu T w następujący sposób:

```
z:=x;
x:=y;
y:=z
```

Można to wykazać używając opisu znaczenia instrukcji podstawiania (1) oraz reguły (2) w następujący sposób:

$$(x = a \wedge y = b)\ \{z := x\}\ (z = a \wedge y = b)$$
$$(z = a \wedge y = b)\ \{x := y\}\ (z = a \wedge x = b)$$
$$(z = a \wedge x = b)\ \{y := z\}\ (y = a \wedge x = b)$$

W praktyce takie rozumowania jest lepiej prowadzić od końca.

Zatem, używając dwukrotnie reguły (2), otrzymujemy

$$(x = a \wedge y = b)\ \{z := x;\ x := y;\ y := z\}\ (y = a \wedge x = b)$$

Gdy T jest typem całkowitym lub rzeczywistym to możemy zamienić wartości zmiennych x i y nie używając dodatkowej zmiennej, w następujący sposób:

```
x:=x+y;
y:=x-y;
x:=x-y
```

By pokazać, że powyższe trzy instrukcje rzeczywiście wymieniają wartości zmiennych x i y znowu użyjemy aksjomatu (1) zaczynając od końca (gdzie wstawiamy formułę którą chcemy udowodnić $(y = a \wedge x = b)$):

$$((x+y)-((x+y)-y)) = b \wedge (x+y)-y = a\ \{x := x+y\}\ x-(x-y) = b \wedge x-y = a$$
$$x - (x - y) = b \wedge x - y = a\ \{y := x - y\}\ (x - y) = b \wedge y = a)$$
$$(x - y) = b \wedge y = a)\ \{x := x - y\}\ x = b \wedge y = a)$$

Formuła pierwsza $((x + y) - ((x + y) - y)) = b \wedge (x + y) - y = a$ (otrzymana na końcu) nie jest tą o którą nam chodzi. Ale zauważmy, że prawdziwa jest formuła

$$(x = a \wedge y = b) \Rightarrow ((x + y) - ((x + y) - y)) = b \wedge (x + y) - y = a$$

A teraz używając reguły (3) otrzymujemy żądany wynik:

$$(x = a \wedge y = b)\ \{x := x + y;\ y := x - y;\ y := x - y\}\ (x = b \wedge y = a)$$

Instrukcja pusta

Instrukcja pusta to pusty ciąg znaków i znaczy 'nic nie rób'. Jej znaczenie opisuje się aksjomatem

$$P\{\ \ \}P$$

dla dowolnej formuły P. Jeśli napiszemy ciąg instrukcji I_1;   ; I_2 to jest to złożenie trzech instrukcji przy czym drugą instrukcją jest instrukcja pusta.

Instrukcja złożona

Instrukcja złożona łączy ciąg instrukcji w jedną instrukcję i ma postać `begin I1; ... ; In end` gdzie `I1`, ... , `In` są instrukcjami. Znaczenie tej instrukcji opisuje reguła

$$\frac{P_i \; \{ \; I_i \; \} \; P_{i+1} \quad i = 1, \ldots, n}{P_1 \; \{ \; \texttt{begin} \; I_1; \ldots; I_n \; \texttt{end} \; \} \; P_{n+1}} \tag{4}$$

Przykład. Fragment programu

```
begin
  x:=x+2;
  y:=2*x;
end;
```

jest złożeniem trzech instrukcji, z których ostatnia jest instrukcją pustą.

Instrukcje warunkowe

Mamy dwie instrukcje warunkowe. Pierwsza ma postać

```
if w then I
```

gdzie `w` jest wyrażeniem typu boolowskiego a `I` jest instrukcją. Znaczenie tej instrukcji opisuje reguła

$$\frac{P \wedge w \; \{ \; I \; \} \; Q \quad P \wedge \neg w \Rightarrow Q}{P \;\; \{ \texttt{if} \; w \; \texttt{then} \; I \} \;\; Q} \tag{5}$$

tzn. instrukcja oznacza 'jeśli `w` to wykonaj `I`'. Druga instrukcja warunkowa jest rozszerzeniem pierwszej i ma postać

```
if w then I1 else I2
```

gdzie `w` jest wyrażeniem typu boolowskiego a `I1` i `I2` są instrukcjami. Znaczenie tej instrukcji opisuje reguła

$$\frac{P \wedge w \; \{ \; I1 \; \} \; Q \quad P \wedge \neg w \; \{ \; I2 \; \} \; Q}{P \;\; \{ \texttt{if} \; w \; \texttt{then} \; I1 \texttt{else} \; I2 \} \;\; Q} \tag{6}$$

tzn. instrukcja oznacza 'jeśli `w` to wykonaj `I1` w przeciwnym wypadku wykonaj `I2`'.

Przykład. Jeśli x jest typu całkowitego to mamy

$$T \; \{ \; \texttt{if} \; x < 0 \; \texttt{then} \; abs := -x \; \texttt{else} \; abs := x \; \} \; abs = |x| \tag{7}$$

gdzie T oznacza formułę zawsze prawdziwą. Z (1) i (3) mamy

$$\frac{\begin{array}{c}(T \wedge x < 0) \Rightarrow (-x = -x \wedge x < 0), \qquad (abs = -x \wedge x < 0) \Rightarrow (abs = |x|) \\ ((-x = -x \wedge x < 0) \; \{ \; abs := -x \; \} \; (abs = -x \wedge x < 0)\end{array}}{(T \wedge x < 0) \; \{ \; abs := -x \; \} \; (abs = |x|)}$$

i podobnie można pokazać, że

$$(T \wedge \neg(x < 0)) \; \{ \; abs := x \; \} \; (abs := |x|).$$

Zatem (7) wynika z powyższych dwóch formuł na mocy reguły (6).

Iteracja warunkowa (pętla while)

Instrukcja iteracji warunkowej ma postać

```
while w do I
```

gdzie `w` jest wyrażeniem typu boolowskiego a `I` jest instrukcją.

Przykład.

```
while x<y do begin
  x:=x+2;
  y:=y+1
end
```

Znaczenie tej instrukcji opisuje reguła

$$\frac{P \wedge w \; \{ \; I \; \} \; P}{P \; \{ \; \texttt{while} \; w \; \texttt{do} \; I \; \} \; P \wedge \neg w} \tag{8}$$

Formułę P spełniającą przesłanki tej reguły nazywamy *niezmiennikiem pętli*. Instrukcja ta powoduje, że instrukcja I jest wykonywana dopóki warunek w jest spełniony. Zauważmy, że w regule (8) warunek P występuje zarówno w warunku początkowym jak i końcowym (stąd jego nazwa *niezmiennik*). Umiejętny wybór takiego niezmiennika jest zazwyczaj najistotniejszym problemem przy dowodzeniu częściowej poprawności programów.

Przykład. Pokażemy, że po wykonaniu poniższego fragmentu programu zmienna `iloczyn` ma wartość $m * n$. Wszystkie zmienne są typu całkowitego.

```
if n<0 then a:=-n
       else a:=n;
k:=0;
x:=0;
while k<a do begin
  x:=m+x;
  k:=k+1
end;
if n<0 then iloczyn:=-x
       else iloczyn:=x
```

Mamy

$$T \; \{ \; \texttt{if n<0 then a:=-n else a:=n} \; \} \; (a = |n|)$$

oraz

$$(a = |n|) \; \{ \; \texttt{k:=0; x:=0} \; \} \; (k = 0 \wedge x = 0 \wedge (a = |n|)).$$

Prawdziwa jest następująca formuła

$$(k = 0 \wedge x = 0 \wedge (a = |n|)) \Rightarrow (k \leq a \wedge x = k * m)$$

Ponadto dla formuły $Q = (k \leq a \wedge x = k * m)$ mamy

$$Q \wedge (k < a) \; \{ \; \texttt{x:=m+x;k:=k+1} \; \} \; Q$$

Zatem Q jest niezmiennikiem pętli i na mocy reguły (8) mamy

$$Q \; \{ \; \texttt{while k<a do begin x:=m+x;k:=k+1 end} \; \} \; (\neg(k < a) \wedge Q).$$

Używając reguły (3) otrzymujemy

$$Q \{ \texttt{while k<a do begin x:=m+x;k:=k+1 end} \} (k = a \wedge Q).$$

Ponadto mamy też

$$(x = a * m \wedge a = |n|) \{ \texttt{if n<0 then iloczyn:=-x else iloczyn:=x} \} \\ (iloczyn = n * m).$$

Łącząc powyższe formuły przy pomocy reguły (3) otrzymujemy tezę.

Iteracja ograniczona (pętla for)

Instrukcja iteracji ograniczonej ma postać

```
for x:=t1 to t2 do I
```

gdzie `x` jest zmienną typu porządkowego, `t1` i `t2` są wyrażeniami tego samego typu co zmienna `x` a `I` jest instrukcją.

Przykład.

```
for i:=1 to 10 do begin
  x:=x*i;
  y:=y+i
end;
```

Znaczenie tej instrukcji opisuje reguła (zakładamy, że instrukcja I nie zmienia wartości zmiennej i oraz $t1 \leq succ(t2)$)

$$\frac{P(t1), \quad P(i) \{ \ I \ \} \ P(succ(i)) \quad i = t1, \ldots, t2}{P(t1) \{ \ \texttt{for } i := t1 \texttt{ to } t2 \texttt{ do } I \ \} \ P(succ(t2))} \tag{9}$$

Podobnie jak w przypadku pętli while formułę P spełniającą przesłanki tej reguły nazywamy *niezmiennikiem pętli.* Instrukcja ta powoduje, że instrukcja I jest wykonywana kolejno dla wszystkich wartości x of $t1$ do $t2$.

Instrukcje wejścia-wyjścia

Instrukcje wejścia-wyjścia służą do komunikacji ze 'światem zewnętrznym'. Instrukcja

```
write(w)
```

wypisuje na ekran wartość wyrażenia w typu standardowego. Instrukcja

```
read(x)
```

wczytuje wartość z klawiatury na zmienną x typu standardowego.

Przykłady. Podajemy poniżej dwa proste przykłady programów. Pierwszy `nwd` oblicza największy wspólny dzielnik a drugi `srednia` średnią arytmetyczną n liczb.

```
Program nwd;
  var n,m:integer;
  begin
    read(n);
    read(m);
    while n<>m do
      if n>m then n:=n-m
             else m:=m-n;
    write(n)
  end.
```

```
Program srednia;
  var n,i:integer;
      x,s:real;
  begin
    read(n);
    s:=0;
    for i:=1 to n do begin
      read(x);
      s:=s+x;
    end;
    write(s/n)
  end.
```

3.8 Procedury

Programy nawet w języku wysokiego poziomu, jeśli nie są podzielone na mniejsze moduły szybko stają się nieczytelne. By temu zapobiec może je dzielić na mniejsze moduły, które wykonują poszczególne fragmenty zadania i mają bardziej przejrzystą formę. Do modularyzacji większych programów służą *procedury*. Rozważmy następujące zadanie.

Zadanie.

- Dane: tablica A liczb całkowitych.
- Wynik: Liczba wystąpień liczby 1 po liczbie 0.

Dla tablicy $A = [1, 0, 0, 7, 1, 1, 0, 6, 7, 1, 1]$ wynik powinien być 2. Można to zadanie rozwiązać tak:

```
const m=100;
var A : array[1..m] of integer;
  s,n,i : integer;
begin
  for i:=1 to m do                  {wczytanie wartości A}
    read(A[i]);
  s:=0; n:=1;                       {inicjalizacja zmiennych}
  while n<m do begin                {pętla główna}
    while (n<m) and (A[n]<>0) do    {szukanie kolejnego 0 w A}
      n:=n+1;
    if A[n]=0 then begin            {jeśli znalazl 0 to...}
      while (n<m) and (A[n]<>1) do {szukanie kolejnego 1 w A}
        n:=n+1;
      if A[n]=1 then s:=s+1;  {jeśli znalazl 1 to zwiększamy s}
    end;
  end;
  write(s);                   {wypisanie wyników}
end.
```

W nawiasach klamrowych zapisane są komentarze wyjaśniające co robią poszczególne fragmenty programu. Można jednak ten program zapisać, używając procedur, tak:

```
const m=100;
var A : array[1..m] of integer;
```

```
  s,n : integer;

procedure dane;{wczytanie wartości A}
  var i:integer;
  begin
    for i:=1 to m do
    read(A[i]);
  end;

procedure szukaj(var j:integer;x:integer);
{szukanie w tablicy A wartosci x od miejsca j}
  begin
    while (j<m) and (A[j]<>x) do
      j:=j+1;
  end;

 begin{prgram glowny}
   dane;  {wywolanie procedury wczytujacej dane}
   s:=0; n:=1;         {inicjalizacja zmiennych}
   while n<m do begin               {pętla główna}
      szukaj(n,0);   {wywolanie procedury szukaj z parametrami
                     aktualnymi n oraz 0}
     if A[n]=0 then begin           {jeśli znalazl 0 to...}
       szukaj(n,1); {wywolanie procedury szukaj z parametrami
                     aktualnymi n oraz 1}
       if A[n]=1 then s:=s+1;  {jeśli znalazl 1 to zwiększamy s}
     end;
   end;
   write(s);                   {wypisanie wyników}
 end.
```

Mając taki program można go teraz łatwo poprawić by szukał różnych kombinacji liczb na przykład $0, 1$ i $2, 3$.

```
const m=100;
var A : array[1..m] of integer;

procedure dane;{wczytanie wartości A}
  var i:integer;
  begin
    for i:=1 to m do
    read(A[i]);
  end;

procedure szukaj(var j:integer;x:integer);
{szukanie w tablicy A wartosci x od miejsca j}
  begin
    while (j<m) and (A[j]<>x) do
      j:=j+1;
  end;
```

```
function kombinacja(k,l:integer):integer;
{liczy ile razy k wystepuje przed l w tablicy A}
  var s,n:integer;
  begin
    s:=0; n:=1;         {inicjalizacja zmiennych}
    while n<m do begin            {pętla główna}
      szukaj(n,k);   {wywolanie procedury szukaj z parametrami
                     aktualnymi n oraz k}
     if A[n]=k then begin          {jeśli znalazl k to...}
       szukaj(n,l); {wywolanie procedury szukaj z parametrami
                     aktualnymi n oraz l}
       if A[n]=l then s:=s+1;  {jeśli znalazl l to zwiększamy s}
     end;
   end;
   kombinacja:=s;
 end;

 begin {program glowny}
   dane;  {wywolanie procedury wczytujacej dane}
   write(kombinacja(0,1));   {wypisanie wyników}
   write(kombinacja(2,3))
 end.
```

W ten sposób pierwszy program został podzielony na mniejsze moduły, które wykonują jasno określone podzadania. A sam program główny został zredukowany do 'spisu treści'.

Parametry i zmienne związane z procedurami

Ponieważ procedury mogą zależeć od parametrów różnego rodzaju i można w nich deklarować dodatkowe zmienne, które istnieją tylko w czasie wykonywania tej procedury, w poniższych tabelach zestawiamy i opisujemy te nowo napotkane 'twory'.

Zmienne w procedurach	
Globalne (zmienne używane w procedurze ale nie zadeklarowane w tej procedurze i nie będące parametrami formalnymi)	**Lokalne** (zmienne zadeklarowane w procedurze i istniejące tylko podczas działania tej procedury)

Zmienne:

- *Zmienne globalne* są zadeklarowane w nagłówku programu i istnieją w czasie całego działania programu.
- *Zmienne lokalne* są deklarowane po nagłówku procedury (i słowie `var`) i istnieją tylko w czasie działania tej procedury.

- W ciele procedury dostępne są zarówno zmienne lokalne zadeklarowane w tej procedurze jaki i zmienne globalne. Wyjątek od tej reguły stanowi sytuacja, w której zmienna globalna ma ten sam identyfikator co zmienna lokalna. Wtedy zmienna globalna nie jest dostępna w tej procedurze[2].

Parametry procedur	
Formalne (wyliczone w nagłówku procedury i używane w ciele procedury)	**Aktualne** (wyliczone przy każdym wołaniu procedury)
wołane przez wartość	*wyrażenia* typów odpowiadających parametrom formalnym
wołane przez zmienną	*zmienne* typów odpowiadających parametrom formalnym

Parametry:

- *Parametry formalne* procedury to zmienne zadeklarowane w nagłówku procedury (w nawiasie, po identyfikatorze procedury).
- *Parametry aktualne* procedury to zmienne lub wyrażenia, które są parametrami przy wołaniach procedury.
- Parametry formalne *wołane przez wartość* zachowują się w ciele procedury jak zmienne lokalne z ta różnicą, że są *inicjalizowane* przed rozpoczęciem wykonywania procedury przez wartości parametrów aktualnych.
- Parametry formalne *wołane przez zmienną* (w nagłówku procedury ich deklarację poprzedza słowo `var`) zachowują się w procedurze podobnie do zmiennych globalnych. Dokładniej, parametr aktualny odpowiadający parametrowi formalnemu wołanemu przez zmienną *musi* być zmienną i wszystkie operacje dotyczące tego parametru formalnego w czasie wykonywania procedury są wykonywane na odpowiadającym mu parametrze aktualnym.

Uwaga. Z powyższego opisu wynika, że wartość parametru aktualnego wołanemu przez zmienną może być aktualizowana w czasie działania procedury. Możemy zatem, przy pomocy zmiennych wołanych przez zmienną przekazywać szereg wartości dowolnych typów, które obliczymy w czasie działania procedury.

Przykład.

```
var t:array[1..100] of integer;
    a,b:integer;
procedure cos (var x:integer;y:integer);
```

[2]Takie zjawisko nazywa się *zasłanianiem zmiennych*

```
 var b,t:integer;
 begin
   x:=x+1; y:=y+1; a:=a+1; b:=y+1; t:=a+1;
 end;

begin
  a:=20; b:=10;
  cos(a,b)
  writeln(a); writeln(b);
end.
```

W procedurze `cos`, x jest parametrem formalnym wołanym przez zmienną, y jest parametrem formalnym wołanym przez wartość, b i t są zmiennymi lokalnymi a zmienna a jest globalna. Zmienne globalne: tablicowa t i całkowita b są niedostępne w procedurze `cos`, ponieważ są *zasłonięte* przez zmienne lokalne o tym samym identyfikatorze i typie całkowitym. W instrukcji wołania procedury `cos(a,b)`, a i b są parametrami aktualnymi. W wyniku wykonania programu na ekranie zostaną wypisane liczby:

```
22
10
```

3.9 Procedury rekurencyjne

Procedury rekurencyjne to takie, które wołają same siebie.

Jedną z najprostszych funkcji, którą wygodnie jest definiować rekurencyjnie jest funkcja *silnia*. Można ją zdefiniować tak:

$$n! = \begin{cases} 1 & \text{gdy } n = 0 \\ n * (n-1)! & \text{gdy } n > 0 \end{cases}$$

Tę matematyczną definicje można łatwo przetłumaczyć na funkcję w języku Pascal:

```
function silnia(n:integer):integer;
 begin
   if n=0 then silnia:=1
          else silnia:=n*silnia(n-1)
 end;
```

Do administrowania obliczeniami programu używającego procedur używamy *stosu odwołań*. *Stos*[3] jest jedną z najprostszych dynamicznych struktury danych. Na *stosie* można dokonywać trzech operacji:

1. włożyć element na wierzch *stosu*;
2. zdjąć element z wierzchu *stosu*;
3. sprawdzić czy *stos* jest pusty.

Stos liter można sobie wyobrażać tak:

[3] Stos na przykład książek ma to do siebie, że można wkładać książki na wierzch i z wierzchu je zdejmować, nie można natomiast wyjmować ich ze środka, bez naruszania całej konstrukcji.

C
A
B
A

Ważne jest, że liczba elementów na stosie jest w zasadzie nieograniczona, jego wielkość może się zmieniać *dynamicznie* w trakcie wykonywania programu w zależności od bieżących potrzeb.

Spróbujmy teraz przeanalizować jak jest wykonywana instrukcja `silnia(5)` umieszczona w pewnym miejscu programu głównego, które oznaczymy przez $\otimes$. By obliczyć wartość `silnia(5)` musimy przerwać wykonywanie kolejnych instrukcji programu, wykonać procedurę *silnia* z parametrem o wartości 5 a następnie powrócić do wykonywania programu w miejscu w którym je przerwaliśmy. By umożliwić taki powrót wkładamy na stos adres miejsca powrotu $\otimes$ i aktualne wartości zmiennych. Teraz stos wygląda tak: 1

$\otimes$, wart. zm.

Rozpoczynając wykonywanie ciała funkcji *silnia* zmienna n ma wartość 5. Zatem $n \neq 0$ i pozostaje do wykonania instrukcja `silnia:=n*silnia(n-1)` przy wartości zmiennej n równej 5. W tym celu należy obliczyć wartość wyrażenia `n*silnia(n-1)`. Wartość n znamy ale by obliczyć wartość `silnia(n-1)` musimy ponownie wywołać procedurę *silnia* tym razem od parametru aktualnego $n - 1 = 5 - 1 = 4$. Przed wywołaniem `silnia(n-1)` trzeba zapamiętać gdzie mamy wrócić z tego wołania i jakie wartości mają mieć wtedy zmienne. Oznaczmy miejsce `silnia(n-1)` w ciele procedury *silnia* jako $\oplus$. Zatem w tym przypadku musimy zapamiętać adres miejsca powrotu $\oplus$ i wartość zmiennej n równą 5. Te dane wkładamy na stos, który wygląda teraz tak:

$\oplus$, $n = 5$
$\otimes$, wart. zm.

Mając tak zabezpieczony powrót rozpoczynamy obliczanie procedury *silnia* z wartością początkową parametru n równą 4. Ponieważ $4 \neq 0$, znowu musimy wywołać procedurę *silnia*, ale tym razem od parametru aktualnego $n = 4 - 1 = 3$. Przed wywołaniem `silnia(n-1)` znów trzeba zapamiętać gdzie mamy wrócić z tego wołania i jakie wartości mają mieć wtedy zmienne. Zatem wkładamy na stos adres miejsca powrotu i wartość zmiennej n. Teraz stos wygląda tak:

$\oplus$, $n = 4$
$\oplus$, $n = 5$
$\otimes$, wart. zm.

... i rozpoczynamy obliczanie `silnia(3)`. W ten sposób dojdziemy w końcu do sytuacji w której stos wygląda tak:

$\oplus$, $n = 1$
$\oplus$, $n = 2$
$\oplus$, $n = 3$
$\oplus$, $n = 4$
$\oplus$, $n = 5$
$\otimes$, wart. zm.

... i rozpoczynamy obliczanie `silnia(0)`. W tym przypadku wartość funkcji *silnia* jest obliczana bezpośrednio w ciele procedury bez potrzeby dalszych odwołań. Po skończeniu obliczenia `silnia(0)`, na wierzchu stosu znajduje się informacja 'co dalej'. Zdejmujemy zatem z wierzchu stosu dane $\oplus$ i $n = 1$, tak, że stos wygląda teraz tak:

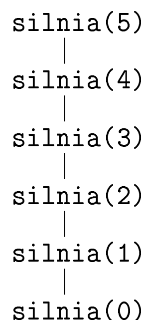
$\oplus$, $n = 2$
$\oplus$, $n = 3$
$\oplus$, $n = 4$
$\oplus$, $n = 5$
$\otimes$, wart. zm.

i kontynuujemy obliczenie w miejscu $\oplus$ z wartością $n = 1$, tzn. kontynuujemy obliczanie wartości funkcji *silnia* z wartością $n = 1$ i obliczaną właśnie wartością `silnia(0)` równą 1. Teraz wyliczamy, że `silnia(1)` ma wartość 1 i kończymy wykonywanie tego wołania funkcji *silnia* i ponownie wracamy do obliczania w miejscu i z wartością zmiennych, które są zapamiętane teraz na wierzchu stosu. Zdejmujemy zatem z wierzchu stosu dane $\oplus$ i $n = 2$ i kontynuujemy obliczenie funkcji *silnia* aż do momentu gdy zakończymy obliczanie wartości `silnia(5)` równej 120. W tym momencie stos wygląda tak:

$\otimes$, wart. zm.

Opróżniamy teraz stos, wracamy do miejsca $\otimes$ i wartości zmiennych z przed wołania `silnia(5)` oraz wartością `silnia(5)` równą 120 i kontynuujemy wykonywanie programu.

Możemy drzewo odwołań dla wołania `silnia(5)` przedstawiać tak:



tzn. `silnia(5)` woła `silnia(4)` aż do `silnia(0)`, a ta ostatnia jest wyliczania bez żadnych dodatkowych wołań.

Zła rekurencja. Rekurencyjne procedury są zwykle bardziej czytelne i łatwiejsze do zaprogramowania jednak, ponieważ ich implementacja używa *stosu*, zwykle używają większej pamięci i są nieco wolniejsze od procedur nierekurencyjnych, o ile takie istnieją. Na przykład funkcję *silnia* można obliczyć iteracyjnie:

```
function silnia1(n:integer):integer;
 var s,i:integer;
 begin
   s:=1;
   for i:= 1 to n do
     s:=s*i;
   silnia1:=s
 end;
```

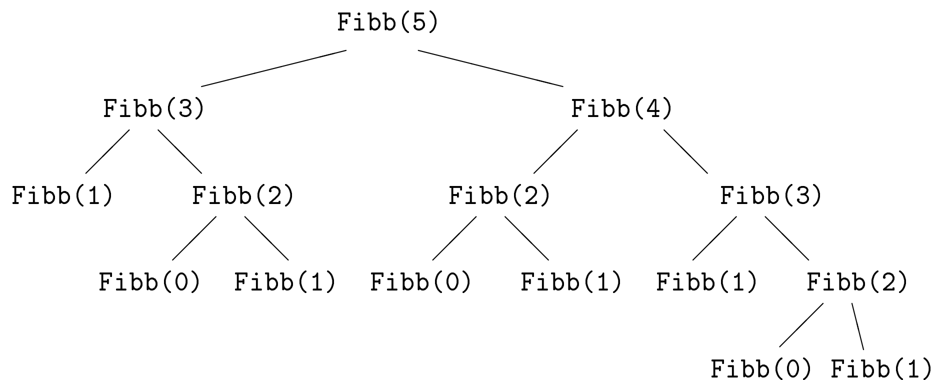
Tak zapisana funkcja jest nieco mniej czytelna ale obliczania przy jej użyciu będzie nieco efektywniejsze. Natomiast w przypadku obliczania ciągu Fibbonacciego zdefiniowanego następująco:

$$f_n = \begin{cases} 0 & \text{gdy } n = 0 \\ 1 & \text{gdy } n = 1 \\ f_{n-2} + f_{n-1} & \text{gdy } n > 1 \end{cases}$$

różnica szybko robi się o wiele istotniejsza. Jeśli zapiszemy tę funkcję rekurencyjnie:

```
function Fibb(n:integer): integer;
 begin
   if n=0 then Fibb:=0
          else if n=1 then Fibb:=1
                      else Fibb:=Fibb(n-2)+Fibb(n-1)
 end;
```

to wygląda ona elegancko ale jest *bardzo nieefektywna*, gdyż wiele wartości będzie wyliczała wielokrotnie. Drzewo odwołań dla wołania `Fibb(5)` będzie wyglądało tak:



A zatem `Fibb(3)` będzie wywołany dwa razy `Fibb(2)` i `Fibb(0)` będzie wywołany trzy razy, a `Fibb(1)` będzie wywołany pięć razy! Taka procedura nie tylko pochłania więcej pamięci ale i wielokrotnie więcej czasu od procedury iteracyjnej:

```
function Fibb1(n:integer):integer;
 var x,y,i:integer;
 begin
   if n<2 then Fibb1:=n
          else begin x:=1; y:=0
            for i:=1 to n-1 do begin
              x:=x+y;
              y:=x-y
            end
            Fibb1:=x
          end;
 end;
```

Zauważmy, że niezmiennikiem pętli `for` w procedurze `Fibb1` jest formuła $x = f_{i+1} \wedge y = f_i$. A stąd łatwo zauważyć, że funkcja `Fibb1` też oblicza wartości ciągu Fibbonacciego.

Zatem jeśli jest proste rozwiązanie iteracyjne to lepiej unikać rekurencji.

3.10 Poprawność programów

Jak się przekonać, że napisany przez nas program jest poprawny, tzn. dla poprawnych danych wejściowych daje poprawne wyniki? Testowanie jest pewną wskazówką ale trudno przetestować program dla wszystkich danych wejściowych. Potrzebny jest *dowód poprawności.*

Mówimy, że program jest *poprawny* jeśli daje poprawne wyniki dla wszystkich możliwych danych wejściowych. Dowody poprawności zwykle składają się z dwóch części: *dowodu częściowej poprawności* i *własności stopu.*

- *Warunek początkowy* określa własności jakie muszą spełniać poprawne dane wejściowe.
- *Warunek końcowy* określa własności jakie muszą spełniać poprawne dane wyjściowe, wyniki.
- Program, lub fragment programu S jest *częściowo poprawny ze względu na warunek początkowy* p i *warunek końcowy* q jeśli o ile dane wejściowe spełniają warunek p i program się zatrzyma to dane wyjściowe spełniają warunek q. Notacja: $p\{S\}q$.
- Program, lub fragment programu S jest *poprawny ze względu na warunek początkowy* p i *warunek końcowy* q jeśli o ile dane wejściowe spełniają warunek p to program się zatrzyma i dane wyjściowe spełniają warunek q.

Przykład 1.
Pokażemy, że fragment programu $S1$

```
silnia:=1;
k:=1;
while k<n do begin
  k:=k+1;
  silnia:=silnia*k
end;
```

jest poprawny ze względu na warunek początkowy $p = (n > 0)$ i warunek końcowy $q = (silnia = n!)$.

Zauważmy, że jeśli p jest spełniony przed rozpoczęciem wykonywania $S1$ to $r = (silnia = k!) \wedge (k \leq n)$ jest spełniony po wykonaniu pierwszych dwóch instrukcji z $S1$.

Pokażemy, że formuła r jest niezmiennikiem pętli *while*. Oznaczmy przez S parę instrukcji `k:=k+1;silnia:=silnia*k`.

Załóżmy, że warunki r i $(k < n)$ są spełniane. Oznaczmy przez $silnia_0$ oraz k_0 wartości zmiennych *silnia* i k przed wykonaniem S a przez $silnia_1$ i k_1 wartości tych zmiennych po wykonaniu S. Pierwsza instrukcja zwiększa wartość k o jeden, a stąd $k_1 = k_0 + 1$. Ponieważ $k_0 < n$ to $k_1 = k_0 + 1 \leq n$. Druga instrukcja mnoży wartość zmiennej *silnia* przez (nową) wartość k. Zatem

$$silnia_1 = silnia_0 * k_1 = k_0! * (k_0 + 1) = (k_0 + 1)! = k_1!$$

Czyli pokazaliśmy, że

$$r \wedge (k < n)\{S\}r$$

i z reguły (8) otrzymujemy, że

$$r\{\texttt{while}\ k < n\ \texttt{do}\ S\}(k \geq n) \wedge r$$

Zatem po wyjściu z pętli *while* (o ile to nastąpi) mamy, że $k = n \wedge silnia = k!$. Stąd q. Pokazaliśmy, że $p\{S1\}q$, tzn. że $S1$ jest częściowo poprawny ze względu na warunek początkowy p i warunek końcowy q.

Ponadto pętla *while* zatrzyma się po $n - 1$ przejściach z wartością $k = n$, gdyż przed wejściem do pętli $k = 1$ oraz każde wykonanie pętli zwiększa wartość zmiennej k o jeden. Zatem $S1$ jest poprawny ze względu na warunek początkowy p i warunek końcowy q.

Przykład 2. Pokażemy, że fragment programu $S2$

```
z:=x; y:=1; m:=n;
while m>0 do begin
  if (m mod 2) = 1 then y:=z*y;
  m:=m div 2;
  z:=z*z
end
```

jest poprawny ze względu na warunek początkowy $p = (n > 0)$ i warunek końcowy $q = (y = x^n)$.

Niezmiennikiem pętli jest formuła $Q = (x^n = y * z^m \wedge m \geq 0)$. Jeśli oznaczymy przez z_1, y_1, m_1, wartości zmiennych z, y, m przed wykonaniem jednego obrotu pętli a przez z_2, y_2, m_2, wartości tych zmiennych po wykonaniu jednego obrotu pętli to mamy:

- jeśli $m_1 = 2 * k$ to $y_2 = y_1$ oraz
$$x^n = y_1 * {z_1}^{m_1} = y_1 * (z_1 * z_1)^k = y_2 * {z_2}^{m_2};$$
- jeśli $m_1 = 2 * k + 1$ to $y_2 = y_1 * z_1$ oraz
$$x^n = y_1 * {z_1}^{m_1} = (y_1 * z_1) * (z_1 * z_1)^k = y_2 * {z_2}^{m_2}.$$

Oczywiście, przed wejściem do pętli niezmiennik jest spełniony. Po wyjściu z pętli $m = 0$ i wtedy $x^n = y * z^m = y * 1 = y$. Stąd $p\{S2\}q$.

Ponieważ każdy obrót pętli zmniejsza wartość dodatnią m o co najmniej 1, pętla będzie wykonana co najwyżej n razy. Zatem $S2$ się zatrzyma dla dowolnego $n > 0$.

4 Podstawowe metody programowania

4.1 Metoda powrotów (prób i błędów)

Problem ustawienia n hetmanów.

- Dane: liczba naturalna n.
- Wynik: ustawienie n hetmanów na szachownicy $n \times n$ tak by żadne dwa hetmany się nie szachowały.

Zastanówmy się jak taki problem można rozwiązać w miarę efektywnie dla $n = 8$.

Pomysł 1 Przejrzeć wszystkie ustawienia hetmanów na szachownicy i sprawdzać czy są poprawnie ustawione.

Ustawień jest $\binom{64}{8} \sim 4 \cdot 10^9$. To jest za dużo!

Pomysł 2 Pomysł pierwszy można łatwo poprawić ograniczając nieco przestrzeń którą mamy przeszukiwać. W każdym wierszu może stać tylko jeden hetman. Wektor $(i_1, \ldots, i_8)$ dla $1 \leq i_1, \ldots, i_8 \leq 8$ reprezentuje ustawienie hetmanów na polach o współrzędnych $((1, i_1), \ldots, (8, i_8))$. Takich wektorów jest $8^8 \sim 10^7$. Dużo!

```
program1
 for i1:=1 to 8 do
  for i2:=1 to 8 do
    ......
   for i8:=1 to 8 do
     sprawdz czy (i1,...,i8) reprezentuje poprawne ustawienie
```

Pomysł 3 Nietrudno zauważyć, że możemy i tę przestrzeń ograniczyć. W każdym wierszu i każdej kolumnie może stać tylko jeden hetman. Wystarczy zatem przejrzeć wektory $(i_1, \ldots, i_8)$ będące permutacjami zbioru $\{1, \ldots, 8\}$. Permutacji jest $8! \sim 10^4$. Sporo!

```
program2
 proba:=pierwsza permutacja;
 while proba nie jest ostatnią permutacją
       oraz nie jest rozwiązaniem  do
   proba:=następna_permutacja(proba);
```

Pomysł 4 (Metoda powrotów) Rozszerzamy częściowe poprawne rozwiązanie aż do uzyskania pełnego poprawnego rozwiązania. Jeśli się nie da rozszerzyć częściowego rozwiązania to wracamy i poprawiamy częściowe rozwiązanie w pierwszym możliwym miejscu.

Przykład $n = 8$.

H							
		H					
				H			
	H						
			H				*

Na tej szachownicy hetmany zostały ustawione w kolejnych wierszach w pierwsze pole od lewej strony które nie jest szachowane przez poprzednio ustawione hetmany. W szóstym wierszu wszystkie pola są szachowane, zatem musimy piątym wierszu przestawić hetmana na następne nieszachowane pole w tym wierszu (ósme zaznaczone $*$) i próbować dalej...

Szkic procedury realizującej taki algorytm. (Konkretne algorytmy z powrotami, w tym ten rozwiązujący nasz problem, są 'ukonkretnieniem' tego schematu.)

```
procedure probuj;
 begin
   zapoczątkuj wybieranie kandydatów;
   while (proba nieudana) and (sa jeszcze kandydaci) do begin
     wybierz następnego kandydata;
     if  kandydat jest dopuszczalny  then begin
       dopisz kandydata do częściowego rozwiązania;
       if  rozwiazanie niepełne  then begin
         probuj wykonać nastepny krok (rekursja);
         if proba nieudana then
           usun kandydata z częściowego rozwiazania
       end;
     end;
   end;
 end;
```

Żeby napisać szczegółowo procedurę musimy najpierw zastanowić się nad reprezentacją danych odpowiednią do naszych potrzeb. Powinniśmy mieć tablicę `het` n liczb całkowitych reprezentującą obecnie rozważane częściowe rozwiązanie, tzn

$$het[i] = 0 \text{ gdy w } i\text{-tym wierszu nie ma hetmana,}$$

$$het[i] = j > 0 \text{ gdy } i\text{-tym wierszu na } j\text{-tym miejscu stoi hetman.}$$

Ponadto musimy pamiętać kolumny oraz lewe $\swarrow$ i prawe $\searrow$ 'skosy', na których już stoi hetman. Będziemy te informacje pamiętać w tablicach boolowskich `kol`, `lewy`, `prawy`, tak, że

$$kol[i] = \begin{cases} true & i\text{-ta kolumna jest wolna} \\ false & \text{w } i\text{-tej kolumnie stoi hetman} \end{cases}$$

$$lewy[i] = \begin{cases} true & \text{gdy skos o sumie współrzednych } i \text{ jest wolny} \\ false & \text{gdy na skosie o sumie współrzednych } i \text{ stoi hetman} \end{cases}$$

$$prawy[i] = \begin{cases} true & \text{gdy skos o różnicy współrzednych } i \text{ jest wolny} \\ false & \text{gdy na skosie o różnicy współrzednych } i \text{ stoi hetman} \end{cases}$$

Oprócz tablic potrzebujemy jeszcze zmiennej `OK` która będzie pamiętała czy już znaleźliśmy rozwiązanie czy nie, tzn.

$$OK = \begin{cases} true & \text{gdy znaleźlismy już rozwiazanie} \\ false & \text{w przeciwnym przypadku.} \end{cases}$$

Zatem deklaracja istotnych zmiennych powinna wyglądać tak:

```
const n=8;
var het   : array[1..n] of integer;
    kol   : array[1..n] of boolean;
    lewy  : array[2..2*n] of boolean;
    prawy : array[1-n..n-1 of boolean;
    OK    : boolean;
```

procedury tak:

```
procedure probuj(i:integer; var q : boolean);
 var k:integer;
 begin
   k:=0; {k - kolejna próbowana pozycja}
   while (not q) and (k<n) do begin
     k:=k+1;
     if kol[k] and lewy[k+i] and prawy[k-i] then begin
       het[i]:=k; kol[k]:=false;
       lewy[k+i]:=false; prawy[k-i]:=false;
       if i<n then begin
           probuj(i+1,q);
           if not q then begin
             het[i]:=0; kol[k]:=true;
             lewy[k+i]:=true; prawy[k-i]:=true;
           end;
       end
       else q:=true;
     end;
   end;
 end;
```

a program główny tak:

```
begin
 for i:=1 to n do het[i]:=0;
 for i:=1 to n do kol[i]:=true;
 for i:=2 to 2*n do lewy[i]:=true;
 for i:=1-n to n-1 do prawy[i]:=true;
 OK:=false;
 probuj(1,OK);
 if OK then wypisz(het)
       else write('Nie ma rozwiązań.')
end.
```

Procedura `wypisz` powinna wypisywać rozwiązanie, na ekran lub do pliku, liczbowo lub graficznie.

Jeśli byśmy chcieli wypisać nie jedno a wszystkie rozwiązania to można to zrobić jeszcze prościej modyfikując procedurę `probuj` tak:

```
procedure probuj1(i:integer);
 var k:integer;
 begin
   for k:=1 to n do
```

```
    if kol[k] and lewy[k+i] and prawy[k-i] then begin
        het[i]:=k; kol[k]:=false;
        lewy[k+i]:=false; prawy[k-i]:=false;
        if i<n then probuj1(i+1)
               else wypisz(het);
        het[i]:=0; kol[k]:=true;
        lewy[k+i]:=true; prawy[k-i]:=true;
    end;
end;
```

4.2 Metoda 'dziel i rządź'

Metoda 'dziel i rządź' polega na rozwiązaniu większego problemu poprzez podzielenie go na mniejsze podproblemy dla których znajdujemy rozwiązanie a następnie za ich pomocą znajdujemy rozwiązanie całego problemu. Metodę tę zilustrujemy prezentując algorytm *sortowania przez scalanie.*

Problem sortowania.

- Dane wejściowe: liczba naturalna n i ciąg liczb $a_1, a_2, \ldots, a_n$.
- Wynik: permutacja $a'_1, a'_2, \ldots, a'_n$ ciągu $a_1, a_2, \ldots, a_n$ taka, że $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Zanim zaprezentujemy algorytm sortowania przez scalanie najpierw rozważmy problem scalania dwóch ciągów:

Problem scalania.

- Dane wejściowe: dwa niemalejące ciągi liczb $a_1, a_2, \ldots, a_n$, $b_1, b_2, \ldots, b_m$.
- Wynik: permutacja $c_1, c_2, \ldots, c_{n+m}$ ciągu $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m$ taka, że $c_1 \leq c_2 \leq \ldots \leq c_{n+m}$.

Procedurę scalania można opisać tak.

1. porównujemy najmniejsze elementy dwóch ciągów, mniejszy z nich wpisujemy do ciągu wynikowego i usuwamy z ciągu scalanego,
2. powtarzamy 1. aż jeden z ciągów scalanych będzie pusty,
3. dopisujemy pozostałe elementy na koniec ciągu wynikowego.

Na przykład dla ciągów scalanych

$$a_1, a_2, \ldots, a_n, \qquad b_1, b_2, \ldots, b_m$$

jeśli mamy

$$a_1 \leq b_1 \quad a_2 \not\leq b_1 \quad a_2 \not\leq b_2 \quad a_2 \leq b_3 \ldots$$

to wynikowy ciąg scalony wygląda tak:

$$a_1, b_1, b_2, a_2 \ldots$$

Zauważmy, że scalanie wymaga co najwyżej $n + m - 1$ porównań.

Teraz, przy definicji typu `tablica=array[1..n] of integer`, możemy tak napisać procedurę sortującą przez scalanie.

```
procedure sortowanie(var A:tablica;min,max:integer);
 var m:integer;
 begin
   if min<max then begin
     m:=(min+max) div 2;
     sortowanie(A,min,m);
     sortowanie(A,m+1,max);
     scalanie(A,min,m,max);
   end;
 end;
```

By posortować tablicę T typu `tablica` należy wywołać `sortowanie(T,1,n)`. Natomiast sama procedura scalająca wygląda tak:

```
procedure scalanie(var A:tablica;min,m,max:integer);
 var i,j,k,l,p:integer;  B:tablica;
 begin
   i:=min; j:=m+1; k:=min;
   while (i<= m) and (j<= max) do begin
     if A[i]<= A[j] then begin B[k]:=A[i]; i:=i+1 end
                    else begin B[k]:=A[j]; j:=j+1 end;
     k:=k+1;
   end;
   if i<=m then begin l:=i; p:=m end
           else begin l:=j; p:=max end;
   for i:=0 to (p-l) do B[k+i]:=A[l+i];
   for i:=min to max do A[i]:=B[i];
 end;
```

Mając powyższy przykład przed oczami podsumujmy zasadnicze kroki metody 'dziel i rządź':

1. *dziel* problem na mniejsze podproblemy;
2. *rozwiąż* podproblemy rekurencyjnie; jeśli są one dostatecznie małe to rozwiąż je bezpośrednio;
3. *połącz* rozwiązania podproblemów w rozwiązanie całego problemu.

Policzymy teraz złożoność algorytmu sortowania przez scalanie. Niech $T(n)$ oznacza maksymalną liczbę porównań elementów tablicy `A` przy wykonywaniu procedury `sortowanie` dla tablicy o rozmiarze n. (Dla uproszczenia zakładamy, że $n = 2^k$.)

$$T(n) = \begin{cases} 0 & \text{gdy } n = 1 \\ 2 \cdot T(n/2) + (n-1) & \text{gdy } n > 1 \end{cases}$$

Notacja $O(f(n))$, $\Omega(f(n))$ i $\Theta(f(n))$. Niech $f : \mathbf{N} \to \mathbf{N}$ funkcja. Przypomnijmy, że funkcja $g : \mathbf{N} \to \mathbf{N}$ *jest (klasy)* $O(f(n))$ jeśli istnieją stale $a, b \in \mathbf{R}$ takie, że dla $n > a$, $g(n) \leq b \cdot f(n)$. Jeśli funkcja '$g(n) \in O(f(n))$' mówimy często, że $g(n)$ jest $O(f(n))$ i piszemy: $g(n) = O(f(n))$.

Mówimy, że funkcja $g : \mathbf{N} \to \mathbf{N}$ *jest (klasy)* $\Omega(f(n))$ jeśli istnieją stale $a, b \in \mathbf{R}$ takie, że dla $n > a$, $b \cdot f(n) \leq g(n)$. Podobnie jeśli funkcja '$g(n) \in \Omega(f(n))$' mówimy często, że $g(n)$ jest $\Omega(f(n))$ i piszemy: $g(n) = \Omega(f(n))$.

Mówimy, że funkcja $g : \mathbf{N} \to \mathbf{N}$ *jest (klasy)* $\Theta(f(n))$ jeśli g jest klasy $O(f(n))$ i $\Omega(f(n))$.

Mamy

Fakt 4.1 *Niech a, b, c będą liczbami rzeczywistymi dodatnimi. Rozwiązaniem równania rekurencyjnego*

$$T(n) = \begin{cases} b & \textit{gdy } n = 1 \\ a \cdot T(n/c) + b \cdot n & \textit{gdy } n > 1 \end{cases}$$

dla n postaci c^k, *(gdzie* $k \in N$*) jest funkcja*

$$T(n) = \begin{cases} O(n) & \textit{gdy } a < c \\ O(n \ln n) & \textit{gdy } a = c \\ O(n^{\log_c a}) & \textit{gdy } a > c \end{cases}$$

Zauważmy, że jeśli zinterpretujemy liczby a, b, c jako

1. $\frac{1}{c}$ - rozmiar jednego podproblemu;
2. a - liczba podproblemów;
3. $b \cdot n$ - czas budowania rozwiązania problemu rozmiaru n z rozwiązań podproblemów;

to powyższy Fakt można użyć do obliczania złożoności wielu algorytmów zbudowanych metodą 'dziel i rządź', także algorytmu sortowania przez scalanie. W tym przypadku $a = b = c = 2$ a zatem $T(n) = O(n \cdot \ln(n))$. Później pokażemy też, że $T(n) = \Theta(n \cdot \ln(n))$.

Dowód Faktu 4.1:
Niech $r = \frac{a}{c}$, $n = c^m$. Wtedy

$$T(n) = n \cdot b \cdot \sum_{i=0}^{m} r^i \qquad \text{gdzie} \quad (m = \log_c n). \tag{10}$$

Równość (10) udowodnimy przez indukcję po m.

Dla $m = 0$, mamy $n = c^0 = 1$ oraz

$$T(n) = T(1) = b = n \cdot b \cdot \sum_{i=0}^{0} r^i.$$

Czyli (10) zachodzi dla $m = 0$.

Załóżmy teraz, że (10) zachodzi dla $n = c^m$. Wtedy używając definicji rekurencyjnej T i założenia indukcyjnego mamy

$$T(c^{m+1}) \stackrel{def\,T}{=} a \cdot T(c^m) + b \cdot c^{m+1} \stackrel{ind.}{=}$$

$$= a \cdot n \cdot b \cdot \sum_{i=0}^{m} r^i + b \cdot c^{m+1} =$$

$$= \frac{a}{c} \cdot c^{m+1} \cdot b \cdot \sum_{i=0}^{m} r^i + b \cdot c^{m+1} =$$

$$= b \cdot c^{m+1}(\frac{a}{c} \cdot \sum_{i=0}^{m} r^i + 1) =$$

$$= b \cdot c^{m+1}(\sum_{i=1}^{m+1} r^i + r^0) =$$

$$= b \cdot c^{m+1} \cdot \sum_{i=0}^{m+1} r^i$$

Załóżmy, że $a < c$. Wtedy szereg $\sum_{i=0}^{\infty} r^i$ jest zbieżny i jego suma jest równa $\frac{1}{1-r}$. Zatem

$$T(n) = n \cdot b \cdot \sum_{i=0}^{m} r^i \leq \frac{1}{1-\frac{a}{c}} \cdot b \cdot n = O(n).$$

Jeżeli $a = c$ to $r^i = 1$ dla dowolnego $i \in N$. Wtedy dla $n = c^m$, mamy $\sum_{r=0}^{m} r^i = \log_c n$ oraz

$$T(n) = b \cdot n \cdot \log_c n = O(n \ln n).$$

Niech $a > c$. Mamy $m = \log_c n$ oraz

$$T(n) = n \cdot b \cdot \sum_{i=0}^{\log_c n} r^i = b \cdot n \cdot \frac{r^{1+\log_c n} - 1}{r-1} =$$

$$= \frac{b}{r-1} \cdot (n \cdot (\frac{a}{c})^{1+\log_c n} - n) =$$

$$= \frac{b}{r-1} \cdot (n \cdot (\frac{a^{1+\log_c n}}{c \cdot n}) - n) =$$

$$= \frac{a \cdot b}{c \cdot (r-1)} \cdot (a^{\log_c n} - \frac{c \cdot n}{a}) =$$

$$= const \cdot (n^{\log_c a} - \frac{c \cdot n}{a}) = O(n^{\log_c a})$$

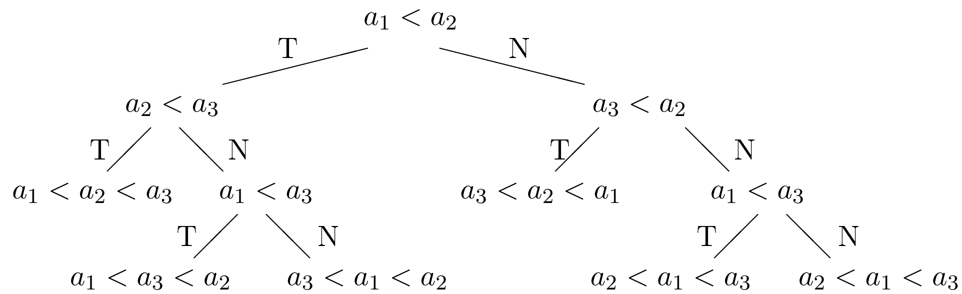
gdzie *const* jest stałą a przedostatnia równość wynika z równości $a^{\log_c n} = n^{\log_c a}$. Ponieważ $a > c$ to $\log_c a > 1$.

Q.E.D.

4.3 Sortowanie przy pomocy porównań

W poprzednim podrozdziale pokazaliśmy, że można skonstruować algorytm, który sortuje tablicę n liczb w czasie $O(n \ln n)$, poprzez wskazanie konkretnego algorytmu który ma te własność ('sortowanie przez scalanie'). Czy można zrobić to szybciej? Odpowiedź może zależeć od różnych szczegółów. Na przykład, jeżeli założymy, że w tablicy sortowanej jest stosunkowo niewiele różnych wartości np. co najwyżej 2 lub 3 lub 4 to można posortować taka tablicę w czasie $O(n)$. Z drugiej strony jeśli chcemy pokazać, że nie ma żadnego algorytmy, który sortuje w czasie mniejszym niż $O(n \ln n)$, to musimy dowieść więcej (niż tylko wskazać algorytm i pokazać, że ma żądane własności), musimy pokazać że '*każdy* algorytm robi 'coś innego' (albo nie sortuje albo nie tak szybko jak byśmy chcieli). By pokazać, że nie ma programu sortującego szybciej niż $O(n \ln n)$ założymy, że sortowanie odbywa się metodą porównań, tzn. porównujemy elementy tablicy między sobą i w zależności od wyniku porównania coś przestawiamy lub nie.

Do abstrakcyjnej analizy algorytmów sortujących przy pomocy porównań użyjemy *drzewa decyzyjnego*. Wierzchołki wewnętrzne drzewa są etykietowane decyzjami do podjęcia. Podczas obliczenia, w zależności od ich wyniku przesuwamy się w lewo lub prawo. Liście drzewa są etykietowane poprawnymi rozwiązaniami. Drzewo decyzyjne dla sortowania trzech elementów a_1, a_2, a_3 może wyglądać tak:



Jedno obliczenie w takim drzewie odpowiada jednej gałęzi drzewa. W liściach drzewa znajdują się wszystkie możliwe odpowiedzi, tzn. permutacje zbioru $\{a_1, a_2, a_3\}$ (każda co najmniej raz). Zatem wysokość drzewa - to pesymistyczna złożoność algorytmu. Mamy

Lemat 4.2 *Drzewo decyzyjne dla dowolnego algorytmu sortującego n elementów przy pomocy porównań ma wysokość $\Omega(n \ln n)$.*

Dowód: Niech $H(n)$ będzie wysokością drzewa decyzyjnego dla pewnego algorytmu sortującego przy pomocy porównań. Takie drzewo ma co najmniej $n!$ liści. Ponieważ pełne drzewo binarne o wysokości h ma 2^h liści to dowolne drzewo o wysokości h ma co najwyżej 2^h liści. Stąd

$$n! \leq 2^{H(n)}$$

i

$$\log_2(n!) \leq H(n).$$

Ponieważ $n! \geq (\frac{n}{3})^n$ to

$$H(n) \geq \log_2(n!) \geq \log_2(\frac{n}{3})^n = n \cdot \log_2 n - n \cdot \log 3 = \Omega(n \cdot \ln n).$$

Q.E.D.

Wniosek 4.3 *Każdy algorytm sortującą tablicę n elementów przy pomocy porównań ma pesymistyczną złożoność czasową $\Omega(n \ln n)$.*

4.4 Programowanie dynamiczne

Programowanie dynamiczne (Pd) stosuje się do problemów optymalizacyjnych, w których musimy dokonywać serii wyborów w celu znalezienia optymalnego rozwiązania. Dokonując wyborów, często musimy się wielokrotnie odwoływać do rozwiązania tego samego podproblemu. Jeśli wszystkich podproblemów nie jest 'zbyt dużo' to Pd jest metodą efektywną. Algorytm Pd rozwiązuje problem tylko raz i zapamiętuje rozwiązanie do ponownego wykorzystania.

Konstrukcja algorytmu:

1. Charakteryzacja struktury optymalnego rozwiązania.
2. Rekurencyjna definicja optymalnego rozwiązania (top-down).
3. Obliczenie kosztu i sposobu konstrukcji optymalnego rozwiązania metodą 'od dołu do góry' (bottom-up).
4. Konstrukcja optymalnego rozwiązania przy użyciu informacji obliczonej w punkcie 3.

Niech $\vec{x} =< x_1, \ldots, x_m >$, $\vec{y} =< y_1, \ldots, y_n >$ i $\vec{z} =< z_1, \ldots, z_k >$ będą ciągami. Wtedy

1. $\vec{z}$ jest *podciągiem* ciągu $\vec{x}$, jeżeli istnieje rosnący ciąg liczb naturalnych $< i_1, \ldots, i_k >$ taki, że $z_j = x_{i_j}$ dla $j = 1, \ldots, k$.
2. $\vec{z}$ jest *wspólnym podciągiem* ciągów $\vec{x}$ i $\vec{y}$ jeśli jest podciągiem $\vec{x}$ i $\vec{y}$.
3. $NWP(\vec{x}, \vec{y})$ oznacza zbiór najdłuższych wspólnych podciągów ciągów $\vec{x}$ i $\vec{y}$.

Przykład.

$$\vec{x} =< A, B, C, B, D, A, B >, \quad \vec{y} =< B, B, D, C, B, A, A >$$

$$\vec{z} =< B, D, A >, \quad \vec{z'} =< D, C > .$$

Wtedy $\vec{z}$ jest podciągiem $\vec{x}$ i $\vec{y}$ a $\vec{z'}$ jest podciągiem $\vec{y}$ ale nie $\vec{x}$.

Problem znajdowania najdłuższego wspólnego podciągu.

- Dane wejściowe: dwa ciągi $\vec{x}$ i $\vec{y}$.
- Wynik: $\vec{z} \in NWP(\vec{x}, \vec{y})$.

Charakteryzacja najdłuższego wspólnego podciągu.

Ciąg długości n ma 2^n podciągów. Zatem sprawdzenie wszystkich podciągów jest kosztowne. Ale problem NWP ma własność optymalnej podstruktury.

Niech $\vec{x} =< x_1, \ldots, x_m >$ będzie ciągiem, $i \leq n$. *i-tym prefiksem* ciągu $\vec{x}$ nazywamy ciąg $\vec{x}_i =< x_1, \ldots, x_i >$.

Przykład.

$$\vec{x} =< A, B, C, B, D, A, B >, \quad \vec{x}_4 =< A, B, C, B >, \quad \vec{x}_0 = \emptyset$$

Mamy

Fakt 4.4 *Niech* $\vec{x} =< x_1, \ldots, x_m >$ *i* $\vec{y} =< y_1, \ldots, y_n >$ *będą ciągami oraz* $\vec{z} =< z_1, \ldots, z_k > \in NWP(\vec{x}, \vec{y})$. *Wtedy*

1. *Jeżeli* $x_m = y_n$ *to* $z_k = x_m = y_n$ *oraz* $\vec{z}_{k-1} \in NWP(\vec{x}_{m-1}, \vec{y}_{n-1})$.
2. *Jeżeli* $x_m \neq y_n$ *oraz* $z_k \neq x_m$ *to* $\vec{z} \in NWP(\vec{x}_{m-1}, \vec{y})$.
3. *Jeżeli* $x_m \neq y_n$ *oraz* $z_k \neq y_n$ *to* $\vec{z} \in NWP(\vec{x}, \vec{y}_{n-1})$.

Dowód: Ad 1. Załóżmy, że $x_m = y_n$. Jeżeli $z_k \neq x_m$ to ciąg $< z_1, \ldots, z_k, x_m >$ jest wspólnym podciągiem $\vec{x}$ i $\vec{y}$. Zatem $\vec{z} \notin NWP(\vec{x}, \vec{y})$ a to jest sprzeczne z założeniem. Stąd $z_k = x_m$. Podobnie można pokazać, że $z_k = y_n$.

Pokażemy, że $\vec{z}_{k-1} \in NWP(\vec{x}_{m-1}, \vec{y}_{n-1})$. Oczywiście $\vec{z}_{k-1}$ jest podciągiem $\vec{x}_{m-1}$ i $\vec{y}_{n-1}$. Przypuśćmy, że istnieje dłuższy wspólny podciąg $\vec{w} =< w_1, \ldots, w_k >$ ciągów $\vec{x}_{m-1}$ i $\vec{y}_{n-1}$. Wtedy ciąg $< w_1, \ldots, w_k, x_m >$ jest wspólnym podciągiem $\vec{x}$ i $\vec{y}$ dłuższym od $\vec{z}$. I znowu mamy sprzeczność.

Ad 2. Jeśli $z_k \neq x_m$ to $\vec{z}$ jest podciągiem $\vec{x}_{m-1}$. Jeśli by istniał dłuższy podciąg $\vec{x}_{m-1}$ i $\vec{y}$ to byłby on też podciągiem $\vec{x}$ i $\vec{y}$. A to jest sprzeczne z założeniem.

Ad 3. Ten przypadek jest podobny do 2.

Q.E.D.

Wniosek 4.5 *Najdłuższy wspólny podciąg dwóch ciągów zawiera najdłuższe wspólne podciągi prefiksów tych ciągów.*

Rekurencyjna definicja rozwiązania.

Z podanego faktu wynika, że znalezienie $\vec{z} \in NWP(\vec{x}, \vec{y})$ sprowadza się do znalezienia

$$\vec{z'} \in NWP(\vec{x}_{m-1}, \vec{y}_{n-1}) \text{ gdy } x_m = y_n$$

oraz dwóch podproblemów znalezienia

$$\vec{z_1} \in NWP(\vec{x}_{m-1}, \vec{y}), \quad \vec{z_2} \in NWP(\vec{x}, \vec{y}_{n-1}) \text{ gdy } x_m \neq y_n$$

Ten który jest dłuższy należy do $NWP(\vec{x}, \vec{y})$.

Niech $c(i, j)$ =długość najdłuższego wspólnego podciągu ciągów $\vec{x}_i$ i $\vec{y}_j$. Wtedy

$$c(i,j) = \begin{cases} 0 & \text{gdy } i = 0 \text{ lub } j = 0, \\ c(i-1, j-1) + 1 & \text{gdy } x_i = y_j, \\ max(c(i-1, j), c(i, j-1)) & \text{gdy } x_i \neq y_j. \end{cases}$$

Tę definicje można łatwo przetłumaczyć na funkcję rekurencyjnie obliczającą c ale złożoność jej będzie wykładnicza.

Obliczanie kosztu i sposobu konstrukcji optymalnego rozwiązania.

Możemy napisać funkcję, która oblicza koszt rozwiązania optymalnego 'od dołu do góry'. To znaczy liczy długość najdłuższego wspólnego podciągu prefiksów ciągów $\vec{x}$ i $\vec{y}$ od najkrótszych aż do całych ciągów $\vec{x}$ i $\vec{y}$. By po takim obliczeniu móc odtworzyć najdłuższy wspólny podciąg, przy okazji liczenia $c(i, j)$ dla $i, j > 0$ musimy zapamiętać z której klauzuli definicji funkcji c korzystaliśmy i jeżeli z trzeciej to która z wartości $c(i-1, j)$, $c(i, j-1)$ była większa.

Innymi słowy funkcja ta rozwiązuje następujący problem:

- Dane wejściowe: dwa ciągi $\vec{x}$ i $\vec{y}$.

- Wynik: dwie tablice; c jak wyżej i b taką, że $b(i,j)$ ='wskazówka' jak obliczać optymalne rozwiązanie dla $\vec{x}_i$ i $\vec{y}_j$.

Tablice b i c (zwłaszcza b) będą nam pomocne do znalezienia rozwiązania.

```
procedure dlugosc_NWP(x,y);
 begin
   m:=dlugosc(x);
   n:=dlugosc(y);
   for i:=1 to m do c[i,0]:=0;
   for i:=1 to n do c[0,i]:=0;
   for i:=1 to m do
     for j:=1 to n do
       if x[i]=y[j] then begin
         c[i,j]:=c[i-1,j-1]+1;  {skracamy oba ciagi}
         b[i,j]:='xy'
       end else
         if c[i-1,j]>=c[i,j-1] then begin
           c[i,j]:=c[i-1,j];  {skracamy x}
           b[i,j]:='x'
         end else begin
           c[i,j]:=c[i,j-1];  {skracamy y}
           b[i,j]:='y'
         end;
 end;
```

Przykład. Poniższa tablica opisuje wartości tablic b i c obliczone dla ciągów $\vec{x} =< A,B,C,B,D,A,B >$ i $\vec{y} =< B,D,C,A,B,A >$ ($m = 7$, $n = 6$).

	$j \rightarrow$	0	1	2	3	4	5	6
$i \downarrow$		$\vec{y} \rightarrow$	B	D	C	A	B	A
0	$\vec{x} \downarrow$	0	0	0	0	0	0	0
1	A	0	$x,0$	$x,0$	$x,0$	$xy,1$	$y,1$	$xy,1$
2	B	0	$xy,1$	$y,1$	$y,1$	$x,1$	$xy,2$	$y,2$
3	C	0	$x,1$	$x,1$	$xy,2$	$y,2$	$x,2$	$x,2$
4	B	0	$xy,1$	$x,1$	$x,2$	$x,2$	$xy,3$	$y,3$
5	D	0	$x,1$	$xy,2$	$x,2$	$x,2$	$x,3$	$x,3$
6	A	0	$x,1$	$x,2$	$x,2$	$xy,3$	$x,3$	$xy,4$
7	B	0	$xy,1$	$x,2$	$x,2$	$x,3$	$xy,4$	$x,4$

W prawym dolnym rogu mamy $c[7,6] = 4$. Zatem najdłuższy wspólny podciąg $\vec{x}$ i $\vec{y}$ ma długość 4. Z tego rogu podróżując w górę gdy $b[i,j] =' x'$, w lewo gdy $b[i,j] =' y'$, oraz po skosie $b[i,j] =' xy'$, zbieramy kolejne elementy najdłuższego wspólnego podciągu $\vec{x}$ i $\vec{y}$. W ten sposób znajdujemy, że $< B,C,B,A > \in NWP(\vec{x},\vec{y})$.

Konstrukcja optymalnego rozwiązania.

Mając tablicę b możemy teraz wypisać ciąg $\vec{z} \in NWP(\vec{x},\vec{y})$.

```
procedure drukuj_NWP(i,j);
 begin
  if (i<>0) and (j<>0) then begin
    if b[i,j]='xy' then begin
```

```
      drukuj_NWP(i-1,j-1);
      write(x[i])
    end else
      if b[i,j]='x' then drukuj_NWP(i-1,j)
                    else drukuj_NWP(i,j-1);
  end;
end;
```

Teraz instrukcja `drukuj_NWP(dlugosc(x),dlugosc(y))` drukuje $\vec{z} \in NWP(\vec{x}, \vec{y})$.

4.5 Algorytmy zachłanne

Typowe algorytmy optymalizacyjne dokonują szeregu wyborów w celu znalezienia optymalnego rozwiązania. Są jednak liczne problemy przy rozwiązywaniu których nie trzeba stosować metody programowania dynamicznego a wystarczą prostsze i bardziej efektywne algorytmy. *Algorytmy zachłanne* wybierają to co w danym momencie wygląda najlepiej w nadziei, że doprowadzi to do optymalnego globalnego rozwiązania. Metodę tę zilustrujemy rozwiązując następujący problem.

Problem zgodnego wyboru zajęć

Mamy salę wykładową i zbiór propozycji na jej wykorzystanie do prowadzenia różnych zajęć w określonych godzinach. Chcemy wybrać maksymalną liczbę zajęć nie kolidujacych ze sobą. Formalnie problem ten wygląda tak:

- Dane wejściowe: zbiór n przedziałów $P = \{[s_i, f_i) : 1 \leq i \leq n\}$, taki, że $s_i, f_i \in R$ $s_i < f_i$ dla $i = 1, \ldots, n$.
- Wynik: podzbiór $A \subseteq \{1, \ldots, n\}$ o maksymalnej liczbie elementów taki, że dla $i, j \in A$, $i \neq j$ mamy $[s_i, f_i) \cap [s_j, f_j) = \emptyset$.

Opis algorytmu.

1. Porządkujemy przedziały względem ich prawych końców, tak by $f_i \leq f_j$ dla $1 \leq i < j \leq n$.
2. Wybieramy po kolei takie zajęcia, które nie kolidują z już wybranymi i się najwcześniej kończą:

```
function wybor_zajec;
  begin
    A:={1};
    j:=1; {j - ostatnio wybrany element}
    for i:=2 to n do
      if s[i]>=f[j] then begin
        A:=A+{i};
        j:=i;
      end;
    zwroc(A);
  end;
```

Fakt 4.6 *Powyższy algorytm znajduje optymalne rozwiązanie problemu zgodnego wyboru zajęć.*

Dowód. Niech $B \subseteq \{1, \ldots, n\}$ będzie optymalnym rozwiązaniem, $|B| = k$, oraz niech $A \subseteq \{1, \ldots, n\}$ będzie zbiorem skonstruowanym przez algorytm. Niech A_l będzie zbiorem pierwszych l elementów z A, B^l zbiorem powstałym z B po usunięciu pierwszych l elementów.

Zauważmy, że sposób konstrukcji zbioru A gwarantuje, że różne przedziały o indeksach z A są rozłączne.

Pokażemy przez indukcję po $l = 1, \ldots, k$, że

1. A_l jest określone i $A_l \cup B^l$ jest optymalnym rozwiązaniem.

2. $max(A_l) < min(B^l)$.

Wtedy dla $k = l$ mamy $B^k = \emptyset$ i $A_k \cup B^k \subseteq A$ jest optymalnym rozwiązaniem. Ponieważ A jest jakimś zgodnym rozwiązaniem to otrzymujemy $A_k = A$ jest optymalnym rozwiązaniem.

Pozostaje pokazać 1. i 2.

$l = 1$. Niech $q = min(B)$. Mamy pokazać, że $A_1 \cup B^1 = \{1\} \cup B - \{q\}$ jest optymalnym rozwiązaniem. Ponieważ $1 \leq q$ to $f_1 \leq f_q$ (f_i są uporządkowane niemalejąco) i przedział $[s_1, f_1)$ jest rozłączny z przedziałami $[s_i, f_i)$ dla $i \in B - \{q\} = B^1$. Zatem zbiór $A_1 \cup B^1 = \{1\} \cup B^1$ zawiera indeksy zgodnych zajęć i ma k elementów. Czyli jest optymalnym rozwiązaniem.

Załóżmy teraz, że $1 \leq l < k$, $A_l \cup B^l$ jest optymalnym rozwiązaniem oraz, że $max(A_l) = p < q = min(B^l)$. Jeśli $i \in B^l$ to $p < i$ i $f_p \leq f_i$. Ale $A_l \cup B^l$ jest zgodnym wyborem zajęć więc mamy też $f_p \leq s_i$. Stąd oraz założenia $l < k$ mamy

$$\emptyset \neq B^l \subseteq \{j : f_p \leq s_j\} \neq \emptyset.$$

Niech $m = min\{j : f_p \leq s_j\}$. Wtedy $A_{l+1} = A_l \cup \{m\}$, $m \leq q$ (bo m jest minimum zbioru do którego należy q) oraz $f_m \leq f_q \leq s_i$ dla $i \in B^l - \{q\} = B^{l+1}$. Zatem $A_{l+1} \cup B^{l+1} = (A_l \cup \{m\}) \cup (B^l - \{q\})$ też jest zgodnym rozwiązaniem a ponieważ ma k elementów to jest optymalnym rozwiązaniem. Mamy też $max(A_{l+1}) = m \leq q < min(A^{l+1})$.

Q.E.D.

By można było stosować algorytmy zachłanne, pożądane jest by problem miał dwie cech:

1. *Własność zachłannego wyboru:* globalne optymalne rozwiązanie może być otrzymane przez lokalne optymalne (zachłanne) wybory.

2. *Optymalna podstruktury:* (jak w Pd) optymalne rozwiązanie problemu zawiera optymalne rozwiązanie podproblemów.

Uwaga. Nie każda możliwa strategia wyboru zachłannego dla problemu *wyboru zajęć* daje optymalne rozwiązanie. Na przykład wybór zajęć niekolidujących z już wybranymi, które:

1. trwają najkrócej,

2. kolidują z najmniejszą liczbą pozostałych zajęć

nie daje optymalnego rozwiązania. Poniższe przykłady wyjaśniają dlaczego. W tym przypadku

wybierając najkrótsze zajęcia wybierzemy tylko dwa, gdy można by wybrać trzy, a w tym

wybierając zajęcia kolidujące z najmniejszą liczbą pozostałych zajęć, zaczniemy od zajęć oznaczonych $*$ i w efekcie wybierzemy trzy, gdy można by wybrać cztery.

Problemy plecakowe.
Problemy plecakowe dotyczą pakowania do plecaka przedmiotów które mają w sumie największą wartość nie przekraczając jednak dopuszczalnego obciążenia plecaka. W jednej wersji musimy pakować całe przedmioty a w drugiej możemy pakować także tylko części przedmiotów. Mimo, że problemy wyglądają podobnie jeden może być łatwo rozwiązany przy pomocy algorytmu zachłannego a drugi nie ma efektywnego rozwiązania. Formalnie problemy te można sformułować tak.

Problem plecakowy 0-1.

- Dane wejściowe: n przedmiotów, i-ty przedmiot jest wart v_i złotych i waży w_i kilogramów, w jest dopuszczalnym ciężarem plecaka.
- Wynik: podzbiór $A \subseteq \{1, \ldots, n\}$ taki, że $\sum_{i \in A} w_i \leq w$ oraz wartość $\sum_{i \in A} v_i$ jest maksymalna.

Problem plecakowy ułamkowy.

- Dane wejściowe: n przedmiotów, i-ty przedmiot jest wart v_i złotych i waży w_i kilogramów, w jest dopuszczalnym ciężarem plecaka.
- Wynik: podzbiór $A \subseteq \{1, \ldots, n\}$ oraz liczby $r_i \in R$ dla $i \in A$ takie, że $\sum_{i \in A} r_i \cdot w_i \leq w$ oraz wartość $\sum_{i \in A} r_i \cdot v_i$ jest maksymalna.

Dla problemu $0-1$ nie jest znane istotnie lepsze rozwiązanie niż przeglądanie wszystkich możliwych podzbiorów przedmiotów. Natomiast problem ułamkowy można rozwiązać 'zachłannie' w następujący sposób:

1. Porządkujemy przedmioty względem malejącej wartości $\frac{v_i}{w_i}$.
2. Wybieram pierwsze k przedmiotów tak, że $\sum_{i=1}^{k} w_i \leq w < \sum_{i=1}^{k+1} w_i$ oraz $w - \sum_{i=1}^{k} w_i$ z $k+1$-go przedmiotu.

Kod Huffmana.
Kod Huffmana służy do kompresji plików tak, że każdy symbol z pliku jest pamiętany przez jedyne binarne słowo kodujące, ale różne słowa kodujące mogą się różnić długością. Bardziej oszczędzimy miejsce gdy symbol częściej występujący będziemy kodować jako krótszy ciąg a symbol występujący rzadziej jako dłuższy ciąg.

Żeby można było odkodować tak zakodowany text, żadne słowo kodujące nie może być prefiksem innego słowa kodującego.

Problem kodowania znaków

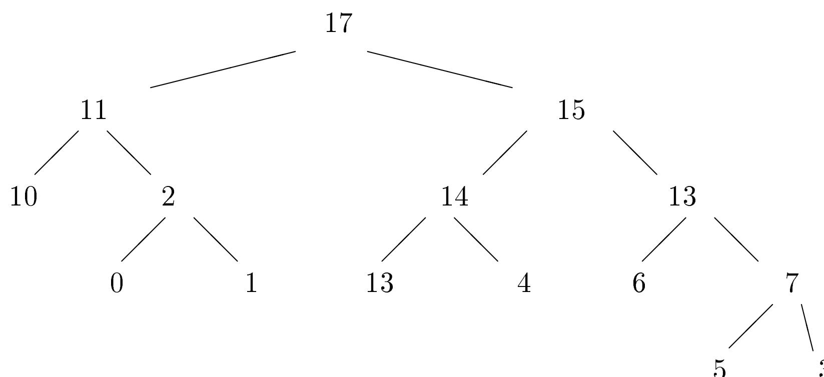
- Dane wejściowe: alfabet $\mathcal{A} = \{a_1, \ldots, a_n\}$, oraz częstości wystąpienia poszczególnych liter tego alfabetu, to znaczy ciąg liczb naturalnych $c_1, \ldots, c_n$.
- Wynik: ciąg binarnych słów kodujących $w_1, \ldots, w_n$, z których żadne nie jest prefiksem innego, dla którego długość kodu $\sum_{i=1}^{n} c_i \cdot |w_i|$ jest minimalna.

4.6 Algorytm sortowania przez kopcowanie (heapsort)

Algorytm sortowania przez kopcowanie podobnie do algorytmu sortowania przez scalanie działa w czasie $O(n \ln(n))$ ale ma tę własność, że sortuje 'w miejscu' tzn. nie używa dodatkowej tablicy. Algorytm ten używa struktury *kopca*.

Kopiec jest to drzewo binarne etykietowane liczbami rzeczywistymi (lub innymi obiektami na których jest zdefiniowany liniowy porządek), takie, że etykieta każdego wierzchołka jest niemniejsza od etykiety synów tego wierzchołka.

Przykład. Kopiec może wyglądać na przykład tak:

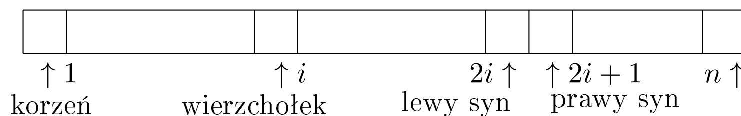


W szczególności wierzchołki na każdej ścieżce są uporządkowane w porządku nierosnącym i etykieta korzenia ma maksymalną wartość.

Opis algorytmu sortowania przez kopcowanie.

1. Tworzymy kopiec etykietowany elementami sortowanego ciągu.
2. Największą etykietę w drzewie (znajdującą się w korzeniu) usuwamy a na jej miejsce wstawiamy etykietę z jednego z liści. Liść ten usuwamy i poprawimy drzewo tak by znowu tworzyło kopiec.
3. Krok 2. powtarzamy dopóki drzewo jest niepuste.
4. Elementy usuwane z kopca tworzą ciąg posortowany.

Kopiec o n wierzchołkach można wygodnie reprezentować w n-elementowej tablicy A:



Wtedy:

1. $A[1]$ jest etykietą korzenia;
2. synowie i-tego wierzchołka mają numery $2i$ oraz $2i + 1$;
3. i jest liściem gdy $n < 2i$;
4. *warunek kopca* wyraża nierówność: $A[i] \leq A[\lfloor i/2 \rfloor]$ dla $i = 1, \ldots, n$.

Program będzie używał dwóch procedur: `kopcuj` oraz `buduj_kopiec`. Procedura `buduj_kopiec` buduje kopiec. Wywołanie procedury `kopcuj(i,j)` odtwarza strukturę kopca we fragmencie tablicy od i-tego do j-tego miejsca zakładając, że w elementy tablicy od $i+1$-ego do j-tego miejsca spełniają własność kopca.

```
procedure kopcuj(i,j);
begin
 l:=2*i; r:=2*i+1;  {l, r -synowie i}
 if (l<=j) and (A[l]>A[i]) then w:=l
                           else w:=i;
 if (r<=j) and (A[r]>A[w]) then w:=r;
 {po tych zamianach A[w] ma najwieksza wartosc z A[l], A[r], A[i]}
 if w<>i then begin
   zamien(A[i],A[w]);
   kopcuj(w,j)
 end;
end;
```

Procedura `kopcuj(i,j)` bez odwołań rekurencyjnych wykonuje stałą liczbę porównań. Jeśli procedura `kopcuj(i,j)` woła `kopcuj(w,j)` to $j \geq w \geq 2i$. Zatem procedura `kopcuj(i,j)` woła siebie $O(\ln(n))$ i działa w czasie $O(\ln(n))$.

```
procedure buduj_kopiec; {tworzenie kopca}
begin
 for i:=n div 2 downto 1 do
   kopcuj(i,n)
end;
```

Program główny, używając powyższych procedur, realizuje algorytm sortowania przez kopcowanie:

```
begin{program główny}
 buduj_kopiec;
 for i:=n  downto 2 do begin
   zamien(A[i],A[1]);
   kopcuj(1,i-1)
 end;
end.
```

Lemat 4.7 *Załóżmy, że elementy tablicy* $A[i+1], \ldots, A[j]$ *spełniają warunek kopca,* $1 < i < j$. *Wtedy*

1. *Po wykonaniu procedury* `kopcuj(i,j)` *elementy* $A[i], \ldots, A[j]$ *spełniają warunek kopca.*

2. *Procedura* `kopcuj(i,j)` *wraz z odwołaniami rekurencyjnymi wykonuje co najwyżej tyle porównań co wysokość i-teg wierzchołka w drzewie o j wierzchołkach.*

Dowód: Ćwiczenie.

Lemat 4.8 *Procedura* `buduj_kopiec` *działa w czasie* $O(n)$.

Dowód: Procedura `kopcuj(i,n)` jest wywoływana raz dla każdego wierzchołka $1 \leq i \leq n$. Dla i-tego wierzchołka `kopcuj(i,n)` wykonuje (wraz z odwołaniami rekurencyjnymi) co najwyżej tyle porównań co wysokość i-tego wierzchołka. Zatem w procedurze `buduj_kopiec` liczba porównań jest co najwyżej taka jak suma wysokości wszystkich wierzchołków drzewa.

Niech $W(n)$ będzie sumą wysokości wierzchołków drzewa binarnego o n wierzchołkach.

Pełne drzewo binarne o wysokości m ma $2^{m+1}-1$ wierzchołków i 2^m liści.

Niech $h_n(i)$ będzie liczbą wierzchołków o wysokości i w drzewie o n wierzchołkach. Mamy

$$h_n(i) \leq h_{n'}(i) \text{ dla } n \leq n'.$$

Niech $2^m \leq n \leq n' = 2^{m+1}-1$. Wtedy

$$h_{n'}(i) = 2^{m-i}.$$

Stąd

$$W(n) = \sum_{i=1}^{m} i \cdot h_n(i) \leq \sum_{i=1}^{m} i \cdot h_{n'}(i) = \sum_{i=1}^{m} i \cdot 2^{m-i} =$$

$$= 2^m \cdot \sum_{i=1}^{m} \frac{i}{2^i} \leq c \cdot n = O(n)$$

Q.E.D.

Twierdzenie 4.9 *Algorytm sortowania przez kopcowanie działa w czasie $O(n\ln(n))$.*

Dowód: Procedura `buduj_kopiec` Wykonuje $O(n)$ porównań. Następnie algorytm wykonuje n-krotnie procedurę `kopcuj(1,i)` dla $i = 1, \ldots, n$. Procedura `kopcuj(1,i)` wykonuje co najwyżej $O(\ln(i))$ porównań. Zatem cały algorytm działa w czasie $O(n\ln(n))$.

Q.E.D.

4.7 Podsumowanie

Mamy cztery metody konstrukcji rozwiązań problemów optymalizacyjnych:

1. Metoda powrotów: rozszerzamy częściowe poprawne rozwiązanie aż do uzyskania pełnego poprawnego rozwiązania. Jeśli się nie da rozszerzyć częściowego rozwiązania to wracamy i poprawiamy częściowe rozwiązanie w pierwszym możliwym miejscu.

   Przykłady.

   (a) Ustawienia Hetmanów
   (b) Konik szachowy
   (c) Kwadrat magiczny

2. Metoda 'dziel i rządź' : problem dzielimy na mniejsze podproblemy, które rozwiązujemy rekurencyjnie, i z tych rozwiązań podproblemów budujemy rozwiązanie całego problemu. By taka metoda była efektywna problemy powinny być w 'sensowny sposób' rozłączne. *Przykłady.*

   (a) Sortowanie przez scalanie
   (b) Wypełnienie tablicy $2^n \times 2^n$ kształtem L tak by dokładnie jedno ustalone pole zostało nie pokryte.

3. Programowanie dynamiczne: Jeśli przy rozwiązaniu problemu musimy się odwoływać wielokrotnie do rozwiązania tych samych podproblemów to metoda 'dziel i rządź' może nie być efektywna. Jeśli podproblemów jest 'stosunkowo niewiele' (wielomianowo wiele) w stosunku do rozmiaru problemu to można je wszystkie 'rozwiązać' (lub jakoś zapamiętać sposób ich rozwiązania) od dołu go góry w sposób efektywny.

   Konstrukcja algorytmu:

   (a) Charakteryzacja struktury optymalnego rozwiązania.
   (b) Rekurencyjna definicja optymalnego rozwiązania.
   (c) Obliczenie kosztu i sposobu konstrukcji optymalnego rozwiązania metodą 'od dołu do góry' (bottom-up).
   (d) Konstrukcja optymalnego rozwiązania przy użyciu informacji obliczonej w punkcie 3.

   By taka metoda była skuteczna liczba podproblemów nie może być zbyt dużą. *Przykłady.*

   (a) Wybór najdłuższego wspólnego podciągu,
   (b) Wybór rozmieszczenia nawiasów przy mnożeniu macierzy,
   (c) Triangulacja wielokąta,
   (d) Obliczanie dwumianu Newtona.

4. Algorytmy zachłanne: wybierają to co w danym momencie wygląda najlepiej w nadziei, że doprowadzi to do optymalnego globalnego rozwiązania. Są bardzo efektywne.

   Przykłady.

(a) Zgodny wybór zajęć,
(b) Ułamkowy problem plecakowy,
(c) Algorytmy grafowe.

5 Reprezentacja liczb na komputerze

5.1 Systemy liczbowe o różnych podstawach

System dziesiętny

Cyfry: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$

Liczba 764.5 oznacza $7 * 10^2 + 6 * 10^1 + 4 * 10^0 + 5 * 10^{-1}$.

System ten jest wygodny dla człowieka a dla maszyny mniej. Reprezentacja cyfry dziesiętnej zajmuje cztery bity pamięci komputera:

cyfra	*reprezentacja*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

W ten sposób marnujemy część pamięci ponieważ są kombinacje (np. 1101), które nie oznaczają żadnej cyfry dziesiętnej.

System dwójkowy

Cyfry: $0, 1$

Liczba $(1010.1.5)_2$ oznacza $1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 + 1 * 2^{-1}$.

System dwójkowy dobrze pasuje do maszyny gdyż reprezentacja jednej cyfry zajmuje dokładnie jeden bit. n-cyfrowa liczba (bez znaku) pamiętana jest w słowie n-bitowym. Natomiast dla człowieka jest on zbyt rozwlekły.

Przypomnijmy, że istnieje prosty sposób konwersji zapisu dziesiętnego liczby na dwójkowy:

1. część całkowitą dzielimy przez 2 i bierzemy reszty,
2. część ułamkową mnożymy przez 2 i bierzemy części całkowite (aż do uzyskania żądanej precyzji).

Przykład. Dla liczby $x = (43.625)_{10}$ liczymy

$$\begin{array}{r|l} 43 & \\ 21 & 1 \\ 10 & 1 \\ 5 & 0 \\ 2 & 1 \\ 1 & 0 \\ 0 & 1 \end{array} \qquad \begin{array}{r|l} & 625 \\ 1 & 250 \\ 0 & 500 \\ 1 & 000 \end{array}$$

i otrzymujemy

$$x = (101011.101)_2$$

W przypadku niektórych liczb reprezentowanych w systemie dziesiątkowym możemy uzyskać tylko przybliżone reprezentacje dwójkowe. Na przykład dla liczby $y = (69.76)_{10}$ powyższa procedura się nie zakończy

$$
\begin{array}{r|l}
69 & \\
34 & 1 \\
17 & 0 \\
8 & 1 \\
4 & 0 \\
2 & 0 \\
1 & 0 \\
0 & 1
\end{array}
\qquad\qquad
\begin{array}{r|l}
 & 76 \\
1 & 52 \\
1 & 04 \\
0 & 08 \\
0 & 16 \\
0 & 32 \\
0 & 64 \\
1 & 28 \\
\dots & \dots
\end{array}
$$

i otrzymujemy wynik przybliżony

$$y \approx (1000101.1100001)_2.$$

System szesnastkowy

Cyfry: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$

Liczba $E9B.F$ oznacza $14 * 16^2 + 9 * 16^1 + 11 * 16^0 + 15 * 16^{-1}$.

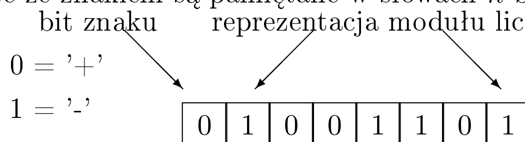
System ten w zasadzie łączy dobre cechy systemu dwójkowego i dziesiętnego. Jest bardziej zwięzły od dwójkowego i lepiej wykorzystuje pamięć niż system dziesiątkowy. Reprezentacja cyfry szesnastkowej zajmuje cztery bity pamięci komputera:

cyfra	*reprezentacja*	*cyfra*	*reprezentacja*
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

i każda kombinacja 4-bitowa odpowiada pewnej cyfrze szesnastkowej. Dlatego często system szesnastkowy jest stosowany do zapisu liczb liczb dwójkowych. Na przykład kody ASCII znaków są często podawane przy użyciu dwucyfrowych liczb szesnastkowych. Natomiast ludzkie przyzwyczajenie do operowania w systemie dziesiątkowym jest tak ugruntowane, że żaden inny system pozycyjny nie może mieć szerszego znaczenia w komunikacji między ludźmi.

5.2 Reprezentacja stałopozycyjna liczb całkowitych

Liczby całkowite ze znakiem są pamiętane w słowach n-bitowych. Ustalmy $n = 8$.



Nieujemna liczba całkowita jest pamiętana jako znak i moduł liczby zapisany w systemie dwójkowym. Powyższe słowo reprezentuje liczbę 77. Natomiast jeśli liczba jest ujemna to istnieje kilka sposobów jej reprezentacji:

1. **znak-moduł**: wygodny dla człowieka, ale przy operacjach arytmetycznych trzeba porównywać znaki i 0 ma dwie reprezentacje[4] ('dodatnią' i 'ujemną').

2. **znak-uzupełnienie do** 1: ta reprezentacja jest mniej wygodna dla człowieka i w niej też 0 ma dwie reprezentacje.

3. **znak-uzupełnienie do** 2: ta reprezentacja jest jeszcze mniej wygodna dla człowieka ale w niej 0 (i każda inna reprezentowana liczba) ma tylko jedną reprezentację, ponadto operacje arytmetyczne są wykonywane w prosty sposób.

Ze względu na przytoczone powyżej zalety zajmiemy się bliżej sposobem reprezentacji liczb jako znak-uzupełnienie do 2. Ten jaki jest w praktyce używany do reprezentacji liczb całkowitych na komputerach.

Uzupełnienie do 1 *liczb całkowitych* otrzymujemy negując wszystkie bity. Dla

$$x = 10011000$$

uzupełnienie do 1 liczby x jest równe

$$01100111$$

Uzupełnienie do 2 *liczb całkowitych* otrzymujemy negując wszystkie bity i dodając 1. Dla x jak wyżej uzupełnienie do 2 liczby x jest równe

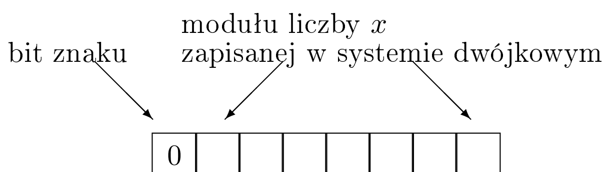
$$\begin{array}{r} 01100111 \\ + \qquad\qquad 1 \\ \hline 01101000 \end{array}$$

tzn. by otrzymać uzupełnienie do 2 liczby całkowitej zostawiamy z prawej strony wszystkie zera i pierwszą jedynkę a resztę bitów negujemy:

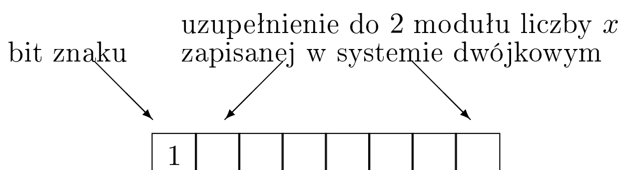
$$01101000$$

Reprezentacja liczb całkowitych w systemie znak-uzupełnienie do 2 ($n = 8$):

1. dla $0 \leq x \leq 127$



2. dla $-128 \leq x \leq -1$



Przykład. Słowo

[4] To, że 0 ma dwie reprezentacje nie jest tylko sprawą estetyki. Gdy jeden obiekt ma różne reprezentacje, sprawdzenie równości dwóch obiektów staje się niepotrzebnie skomplikowana procedurą.

0	1	0	1	0	0	0	0

reprezentuje liczbę dodatnią 80 natomiast słowo

1	1	0	1	0	0	0	0

reprezentuje liczbę ujemną $-(0110000)_2 = -48$.

5.3 Operacje arytmetyczne stałopozycyjne

Reprezentując liczby w systemie znak-uzupełnienie do 2, przy dodawaniu nie trzeba zwracać uwagi na znak liczby. Wyniki otrzymujemy poprawne i dokładne o ile argumenty i wartości mają swoje reprezentacje jako słowa n-bitowe w systemie znak-uzupełnienie do 2.

1. *Zmiana znaku* Uzupełniamy do 2 liczbę (łącznie z bitem znaku):

$x = 26 \sim$	0	0	0	1	1	0	1	0
$-x = -26 \sim$	1	1	1	0	0	1	1	0

2. *Dodawanie* Dodajemy dwie liczby łącznie z bitem znaku i ewentualne przeniesienie pomijamy. Wynik reprezentuje sumę w systemie znak-uzupełnienie do 2:

$x = 24 \sim$	0	0	0	1	1	0	0	0
$y = -40 \sim$	1	1	0	1	1	0	0	0
$x + y = -16 \sim$	1	1	1	1	0	0	0	0

3. *Odejmowanie* Dodajemy odjemną do liczby przeciwnej do odjemnika.

4. *Mnożenie i dzielenie* Nie tak prosto. Najlepiej mnożyć i dzielić moduły a potem ustalać znak. Jeśli mnożnik jest dodatni to można mnożyć jak zwykle.

Uwaga. Zakres liczb całkowitych reprezentowanych w komputerze jest zwykle dość *mały* a operacje są wykonywane *dokładnie.* Dla liczb rzeczywistych jest odwrotnie, zakres jest *duży* a operacje są wykonywane w sposób przybliżony.

5.4 Reprezentacja zmiennopozycyjna liczb rzeczywistych

Opiszemy teraz *reprezentację zmiennopozycyjną liczb rzeczywistych t-cyfrową, dziesiętną.*

Dowolną liczbę rzeczywistą $x \neq 0$ można przedstawić w postaci *znormalizowanej*:

$$x = m * 10^c$$

gdzie $0.1 \leq |m| < 1$, c - liczba całkowita. Liczbę m nazywamy *mantysą* a liczbę c - *cechą* liczby x.

Dla x równego 0 przyjmuje się $m = 0$ i $c = 0$.

Przykład. Dla

$$x = 0.00354$$

po znormalizowaniu otrzymujemy

$$x = 0.354 * 10^{-2}$$

Mantysa i cecha są pamiętane w komputerze w postaci stałopozycyjnej dziesiętnej. W arytmetyce t-cyfrowej mantysa ma t-cyfr i jest na ogół wielkością *zaokrągloną*. Zakres cech decyduje o *zakresie* liczb rzeczywistych reprezentowanych w komputerze. Liczba cyfr mantysy decyduje o precyzji liczb pamiętanych na komputerze.

Przykład. Przyjmijmy, że w naszej reprezentacji mantysa jest 4-cyfrowa a cecha 1-cyfrowa.

liczba x	*przyblienie* $\overline{x}$
$a_1 = \frac{1}{6} = 0.166(6)$	$\overline{a_1} = 0.1667 * 10^0$
$a_2 = 42.5368$	$\overline{a_2} = 0.4254 * 10^2$
$a_3 = 6\ 042\ 875$	$\overline{a_3} = 0.6043 * 10^7$
$a_4 = -7.67443$	$\overline{a_4} = -0.7674 * 10^1$

Błąd bezwzględny przybliżenia jest to moduł różnicy między wartością rzeczywistą a wartością przybliżoną:

$$\varepsilon\overline{a} = |a - \overline{a}|$$

Mamy następujące oszacowanie błędu zaokrąglenia mantysy:

$$|m - \overline{m}| \leq 0.5 * 10^{-t}$$

A stąd, dla liczby dla $a = m * 10^c$, błąd bezwzględny wartości przybliżonej $\overline{a}$ można oszacować tak:

$$\varepsilon\overline{a} = |a - \overline{a}| = |m * 10^c - \overline{m} * 10^c| \leq 0.5 * 10^{-t} * 10^c$$

Błąd względny przybliżenia jest to błąd bezwzględny podzielony przez moduł dokładnej wartości liczby:

$$\delta\overline{a} = |\frac{a - \overline{a}}{a}| = \frac{\varepsilon\overline{a}}{|a|}$$

Dla $a = m * 10^c$, mamy następujące oszacowanie błędu względnego:

$$\delta\overline{a} = |\frac{a - \overline{a}}{a}| \leq \frac{0.5 * 10^{-t} * 10^c}{m * 10^c} \leq \frac{0.5 * 10^{-t} * 10^c}{0.1 * 10^c} = 5 * 10^{-t}$$

Wielkość $u = 5 * 10^{-t}$ jest *względną dokładnością komputera.*

Ponieważ

$$|\frac{a - \overline{a}}{a}| \leq u$$

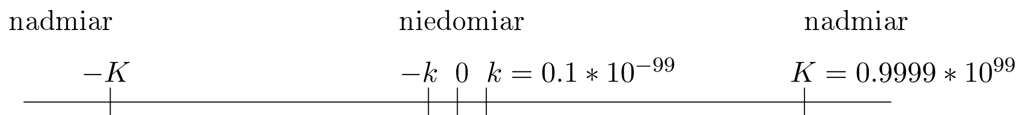
to dla pewnego ρ mamy

$$\overline{a} = a(1 + \rho)$$

gdzie $|\rho| \leq u$.

Uwaga. Zwykle błąd względny jest lepszą miara przybliżenia.

Oś liczb rzeczywistych na komputerze prezentuje się mniej więcej tak (cecha 2-cyfrowa, mantysa 4-cyfrowa):



Przedział $[-K, K]$ stanowi zakres liczb reprezentowanych na komputerze. Poza tym przedziałem występuje *nadmiar.* Liczby o module większym od K nie mają reprezentacji. Przedział $[-k, k]$ jest nazywany *niedomiarem.* Liczby z tego zakresu są reprezentowane przez 0, k, lub $-k$ z dużym błędem względnym.

5.5 Arytmetyka zmiennopozycyjna

Nie tylko wartości liczb zmiennopozycyjnych są przybliżone ale operacje arytmetyczne wykonywane na przybliżeniach liczb też nie muszą mieć dokładnej reprezentacji. Zatem przy wykonywaniu działań arytmetycznych w arytmetyce zmiennopozycyjnej błędy przybliżeń się kumulują. Opiszemy teraz te operacje i oszacujemy błędy jakie przy nich powstają.

1. *Mnożenie* wykonujemy w następujący sposób
   (a) mnożymy mantysy,
   (b) dodajemy cechy,
   (c) normalizujemy.

   Przykład. Obliczymy reprezentację zmiennopozycyjną iloczynu $a = 0.375 * 10^2$ i $b = 0.225 * 10^3$. Obliczamy iloczyn mantys

$$\begin{array}{r} 0.375 \\ 0.225 \\ \hline 1875 \\ 750 \\ 750 \\ \hline 0.084375 \end{array}$$

   i po normalizacji otrzymujemy $\overline{a * b} = 0.8438 * 10^4$. Zatem

$$\overline{a * b} = a * b(1 + \rho),$$

   dla $|\rho| \leq u$.

2. *Dzielenie* wykonujemy podobnie

   (a) dzielimy mantysy,
   (b) odejmujemy cechy,
   (c) normalizujemy.

3. *Dodawanie* jest nieco bardziej skomplikowane i wykonujemy je w następujący sposób

   (a) wyrównujemy cechy zwiększając mniejszą i przesuwając mantysę mniejszej w prawo,
   (b) dodajemy mantysy,
   (c) normalizujemy.

   Przykład. Obliczymy reprezentację zmiennopozycyjną sumy $a = 0.357 * 10^3$ i $b = 0.268 * 10^4$. Obliczamy sumę 'wyrównanych' mantys

$$\begin{array}{r} 0.036 \\ \underline{0.268} \\ 0.304 \end{array}$$

   i otrzymujemy $\overline{a+b} = 0.304 * 10^4$. Można pokazać, że istnieją ρ_1, ρ_2 takie, że

$$\overline{a+b} = a(1+\rho_1) + b(1+\rho_2)$$

   oraz $|\rho_1|, |\rho_2| \leq u$.

4. *Odejmowanie* jest wykonywane podobnie do dodawania.

Oszacujemy teraz błąd względny mnożenia i dodawania.
Dla mnożenia mamy

$$\left|\frac{a*b - \overline{a*b}}{a*b}\right| = \left|\frac{a*b - a*b(1+\rho)}{a*b}\right| = |\rho| \leq u$$

Zatem błąd względny mnożenia nie przekracza względnej dokładności komputera. Jeśli już liczby mnożone są przybliżone to błędy się kumulują.

$$\left|\frac{(a(1+\rho_1)*b(1+\rho_2))(1+\rho) - a*b}{a*b}\right| =$$

$$= \left|\frac{a*b((1+\rho_1+\rho_2+\rho_1*\rho_2))(1+\rho) - a*b}{a*b}\right| =$$

$$= |(\rho_1+\rho_2+\rho_1*\rho_2)(1+\rho)| \approx 2*u$$

Czyli błąd wyniku mnożenia jest rzędu $2*u$, o ile błąd reprezentacji argumentów nie przekraczał względnej dokładności komputera.

Dla dodawania mamy

$$\left|\frac{(a(1+\rho_1) + b(1+\rho_2)) - (a+b)}{a+b}\right| \leq$$

$$\leq \frac{|a| * |\rho_1| + |b| * |\rho_2|}{|a + b|} \leq \frac{|a| + |b|}{|a + b|} * u$$

Jeśli liczby a i b mają przeciwne znaki i nieznacznie różniące się moduły to $|a| + |b|$ może być duże podczas gdy $|a + b|$ będzie bardzo małe i w konsekwencji $\frac{|a|+|b|}{|a+b|}$ może być bardzo duże. W takim przypadku możemy po jednej operacji dodawania znacznie utracić dokładność obliczenia. Taką sytuację obrazuje następujący przykład.

Przykład. Obliczymy różnicę liczb $a = 10.0726$ i $b = 10.0789$ w arytmetyce 5-cyfrowej . Mamy przybliżenia

$$\overline{a} = 0.10073 * 10^2 \quad \text{i} \quad \overline{b} = 0.10079 * 10^2$$

Błąd względny przybliżeń a i b

$$\delta\overline{a} = \left|\frac{a - \overline{a}}{a}\right| = \left|\frac{4 * 10^{-6} * 10^2}{10.0726}\right| < 4 * 10^{-5} \leq u(= 5 * 10^{-5})$$

$$\delta\overline{b} = \left|\frac{b - \overline{b}}{b}\right| = \frac{10^{-6} * 10^2}{10.0789} < 10^{-5} \leq u$$

nie przekracza względnej dokładności komputera. Natomiast błąd różnicy

$$\left|\frac{(b - a) - \overline{\overline{b} - \overline{a}}}{(b - a)}\right| = \left|\frac{6.3 * 10^{-3} - 6 * 10^{-3}}{6.3 * 10^{-3}}\right| =$$

$$= \frac{0.3}{6.3} > 4 * 10^{-2} = \frac{4}{5} * 10^3 * u$$

W szczególności zmiennopozycyjne działania mnożenia i dodawania są przemienne ale nie są łączne!

5.6 Wybrane problemy numeryczne

W tej sekcji pokażemy kilka typowych problemów związanych obliczeniami zmiennopozycyjnymi.

Obliczanie pierwiastków równania kwadratowego

Równanie kwadratowe

$$x^2 + px + q = 0$$

ma pierwiastki

$$x_1 = \frac{-p - \sqrt{p^2 - 4q}}{2} \quad \text{i} \quad x_1 = \frac{-p + \sqrt{p^2 - 4q}}{2}$$

o ile wyróżnik $\Delta = p^2 - 4q$ jest nieujemny. Obliczymy te pierwiastki w arytmetyce 4-cyfrowej dla

$$p = -6.433 \quad \text{i} \quad q = 0.009474.$$

Obliczenie x_1:

1. $\alpha = \overline{(6.433)^2} = 41.38$,

2. $\beta = \overline{4q} = 0.03790$,

3. $\gamma = \overline{\alpha - \beta} = 41.34$,

4. $\sigma = \overline{\sqrt{\gamma}} = 6.430$,

5. $\varphi_1 = \overline{-p - \sigma} = 6.433 - 6.430 = 0.003$,

6. $\overline{x_1} = \overline{\frac{\varphi_1}{2}} = 1.5 * 10^{-3}$.

Ponieważ przybliżona wartość pierwiastka x_1 w arytmetyce 5-cyfrowej wynosi

$$x_1 \approx 1.4731 * 10^{-3}$$

to widzimy, że nasze obliczenie x_1 dało nam tylko jedną dokładną cyfrę znaczącą i błąd względny wynosi

$$\delta\overline{x_1} \approx 10^{-1}.$$

Korzystając z poprzednich obliczeń możemy obliczyć x_2:

1. $\varphi_2 = \overline{-p + \sigma} = \overline{6.433 + 6.430} = 12.86$,

2. $\overline{x_2} = \overline{\frac{\varphi_2}{2}} = 6.43$.

Ponieważ przybliżona wartość pierwiastka x_2 w arytmetyce 5-cyfrowej wynosi

$$x_2 \approx 6.4313$$

to widzimy, że nasze obliczenie x_2 dało nam dokładne trzy cyfry znaczące.

Przyczyna utraty dokładności w obliczeniu x_1 tkwi w obliczeniu φ_1 w kroku 5. Obliczamy różnicę liczb o bardzo bliskich wartościach. Natomiast problem ten nie istnieje przy obliczaniu φ_2 w kroku 2. obliczenia x_2. Tam liczymy sumę tych liczb nie tracąc istotnie na dokładności.

Mając dobre przybliżenie x_2, możemy też obliczyć x_1 ze wzoru Viete'a $x_1 * x_2 = q$:

$$\overline{x_1} = \overline{\left(\frac{0.009474}{6.430}\right)} = 0.001473 = 1.473 * 10^{-3}$$

i wtedy mamy cztery cyfry dokładne oraz błąd względny

$$\delta\overline{x_1} \approx 10^{-4}.$$

Morał z tych rozważań jest taki, że jeżeli $p > 0$ to x_1 jest liczone dokładniej a jeżeli $p < 0$ to x_2 jest liczone dokładniej.

Schemat Hornera obliczania wielomianu

Normalna procedura obliczenia wartości wielomianu n-tego stopnia

$$W_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0,$$

sugerowana przez sposób zapisu wielomianu, polega na obliczeniu

1. $x^2, x^3, \ldots, x^n$ - $n-1$ mnożeń,

2. $a_1 x, a_2 x^2, \ldots, a_n x^n$ - n mnożeń,

3. $a_0 + a_1x + \ldots + a_{n-1}x^{n-1} + a_nx^n$ n -dodawań.

Zatem takie obliczenie wymaga $2n - 1$ mnożeń i n dodawań.

Natomiast istnieje inny sposób obliczania tej samej wartości, zaproponowany przez Hornera. Liczymy:

1. $P_n(x) = a_nx + a_{n-1}$ - 1 mnożenie i 1 dodawanie,
2. $P_{n-1}(x) = (a_nx + a_{n-1})x + a_{n-2}$ - 1 mnożenie i 1 dodawanie,
3. ...
4. $P_1 = P_2x + a_0$ - 1 mnożenie i 1 dodawanie.

Zatem takie obliczenie wymaga tylko n mnożeń i n dodawań. Oprócz tego, że ta metoda obliczania wartości wielomianu jest szybsza to jest też dokładniejsza.

6 Dynamiczne struktury danych

'Zbiory matematyczne' nie zmieniają się, natomiast 'zbiory informatyczne' mogą się zwiększać, zmniejszać lub zmieniać w inny sposób. Są to *'zbiory dynamiczne'*.

Sposób w jaki reprezentujemy 'zbiory dynamiczne' zależy od tego jakie operacje chcemy na nich wykonywać. Mamy dwa rodzaje operacji: *modyfikacje* i *pytania*.

Modyfikacje (przykłady):

1. `dodaj(S,x)` - dodaj element (wskazywany przez) x do zbioru S;
2. `usun(S,x)` - usuń element (wskazywany przez) x ze zbioru S.

Pytania (przykłady):

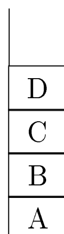
1. `szukaj(S,k)` - sprawdź czy element x należy do zbioru S; jeśli tak to wskaż ten element, jeśli nie to wskaż Nil;
2. `minimum(S)` - (pytanie na zbiorze liniowo uporządkowanym) zwraca element najmniejszy zbioru S;
3. `maksimum(S)` - (pytanie na zbiorze liniowo uporządkowanym) zwraca element największy zbioru S;
4. `nastepny(S,x)` - (pytanie na zbiorze liniowo uporządkowanym) zwraca element następny po x w zbioru S;
5. `poprzedni(S,x)` - (pytanie na zbiorze liniowo uporządkowanym) zwraca element poprzedni przed x w zbioru S.

Zwykle potrzeba tylko części spośród tych operacji. Ważne jest by operacje wykonywane były szybko, tzn. w czasie stałym lub co najwyżej logarytmicznym w stosunku do rozmiaru zbioru S.

6.1 Podstawowe dynamiczne struktury danych

Stos

Stos jest najprostszą strukturą dynamiczną (LIFO - last in first out). Można wkładać elementy tylko na wierzch stosu i zdejmować elementy tylko z wierzchu stosu. Poza tym można testować czy stos jest pusty. Wszystkie operacje są wykonywane w czasie stałym. Stos można sobie wyobrażać tak:



Operacje na stosie:

1. dodaj (na wierzch) `Push(S,x)`;
2. zdejmij (z wierzchu) `Pop(S)`;

3. test pustości stosu

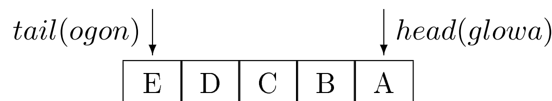
$$\texttt{empty(S)} = \begin{cases} \texttt{true} & \text{gdy } S \text{ jest pusty} \\ \texttt{false} & \text{w przeciwnym przypadku.} \end{cases}$$

Jak już wcześniej pokazaliśmy, przy pomocy stosów można implementować procedury rekurencyjne. Stos S, jeśli ma ograniczoną wysokość, można implementować w tablicy:

```
S : array[1..n] of typ;
topS : integer;

function empty :boolean;
begin
  empty:=(topS=0)
end;

procedure push(x:typ);
begin
  topS:=topS+1;
  S[topS]:=x;
end;

procedure pop(var x:typ);
begin
  x:=S[topS];
  topS:=topS-1;
end;
```

Kolejka

Kolejka jest strukturą podobną do kolejki w sklepie (FIFO - first in first out). Można wkładać elementy tylko na koniec kolejki i zdejmować elementy tylko z początku kolejki. Poza tym można testować czy kolejka jest pusta. Wszystkie operacje są wykonywane w czasie stałym. Kolejkę można sobie wyobrażać tak:



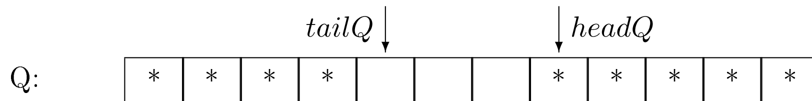
Operacje na kolejce:

1. dodaj (na koniec) `dodaj(Q,x)`;
2. zdejmij (z początku) `usun(Q)`;
3. test pustości kolejki

$$\texttt{empty(Q)} = \begin{cases} \texttt{true} & \text{gdy } S \text{ jest pusty} \\ \texttt{false} & \text{w przeciwnym przypadku.} \end{cases}$$

Kolejkę Q, jeśli ma ograniczoną długość, można implementować w tablicy, ale w bardziej skomplikowany sposób niż stos.

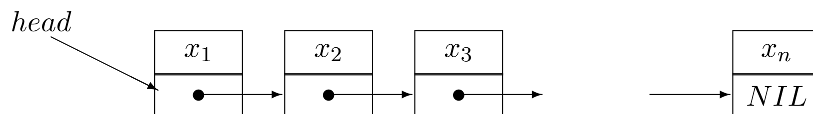
```
Q : array[0..n] of typ;
headQ,tailQ : integer;
```



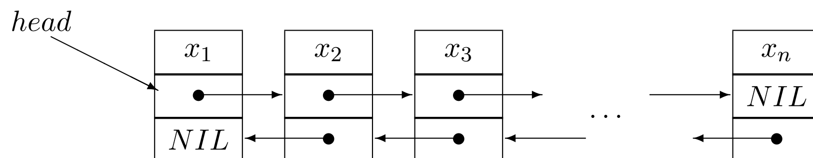

```
function empty :boolean;
begin
  empty:=(headQ=tailQ)
end;

procedure wstaw(x:typ);
begin
  Q[tailQ]:=x;
  if tailQ=n then tailQ:=0
             else tailQ:=tailQ+1;
end;

procedure usun(var x:typ);
begin
  x:=Q[headQ];
  if headQ=n then headQ:=0
             else headQ:=headQ+1
end;
```

Listy.

Lista to zbiór elementów, z których każdy (z wyjątkiem ostatniego) wskazuje na następny i na każdy element (z wyjątkiem pierwszego) wskazuje jakiś element. Listę jednokierunkową można sobie wyobrażać tak:

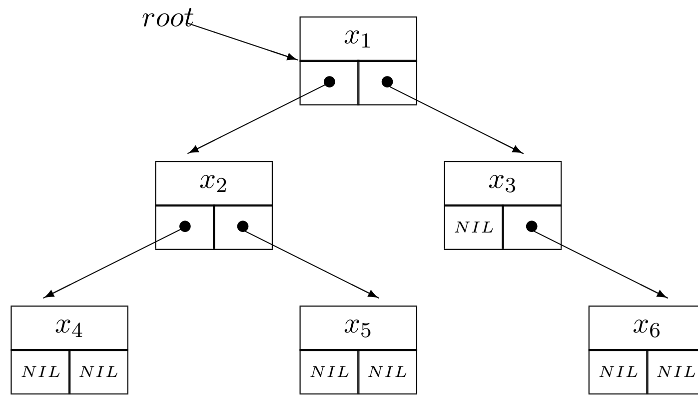


Lista zajmuje proporcjonalnie wiele miejsca do liczby elementów na liście. Listę dwukierunkową można sobie wyobrażać tak:



Drzewa binarne

Drzewo binarne (o ile jest niepuste) ma korzeń i każdy wierzchołek drzewa ma co najwyżej dwa następniki, lewy i prawy. Ponadto z korzenia do każdego wierzchołka istnieje dokładnie jedna droga. Drzewo binarne można sobie wyobrażać tak:



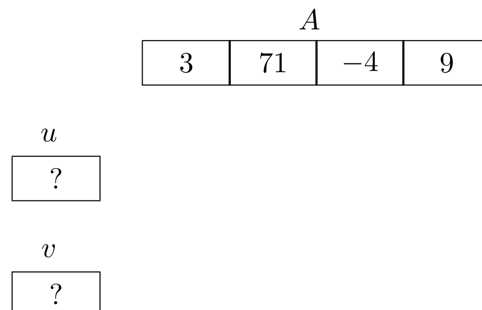
6.2 Typy wskaźnikowe

Zmienne typów dotychczas poznanych istnieją przez cały czas wykonywania tej części programu (program główny lub procedura), w której są zadeklarowane. Mają one na stałe przydzieloną pamięć, do której odwołujemy się za pomocą identyfikatora zmiennej. Są to *zmienne statyczne*. Takie zmienne nie nadają się do reprezentowania 'struktur dynamicznych', na przykład tych, które zostały opisane powyżej, (o ile nie nałożymy z góry ograniczeń dotyczących rozmiaru tych struktur). Do reprezentowania takich struktur służą typy *wskaźnikowe* i wskazywane przez zmienne tych typów *zmienne dynamiczne*. Zmienne dynamiczne można tworzyć i usuwać w dowolnym miejscu programu a odwołujemy się do nich nie przez identyfikator zmiennej lecz przez zmienną wskaźnikową wskazującą tą zmienną.

Przykład deklaracji typu wskaźnikowego.

```
type Tab=array[1..10] of integer; {typ tablicowy, przykladowy typ}
     Wskaznik=^Tab; {typ wskaznikow do elementow typu Tab}

var A:Tab; {deklaracja zmiennej typu Tab}
    u,v,w:Wskaznik; {deklaracja zmiennych typu
                 Wskaznik wskazującego typ Tab}
```

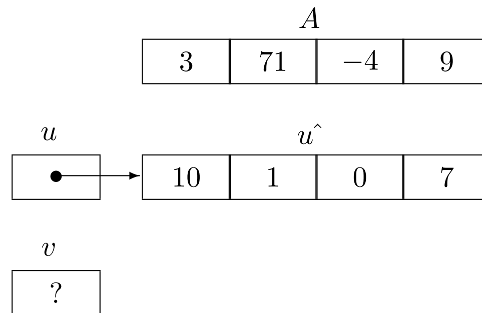
Zmienne typu wskaźnikowego mogą mieć wartość nieokreśloną (np. zaraz po ich deklaracji) mogą zawierać adres zmiennej dynamicznej typu Tab lub mogą być równe stałej `Nil`, nie wskazującej żadnej zmiennej. Po takich deklaracjach w pamięci komputera mamy zarezerwowane takie obszary pamięci (wartości we wszystkich tablicach są losowe):



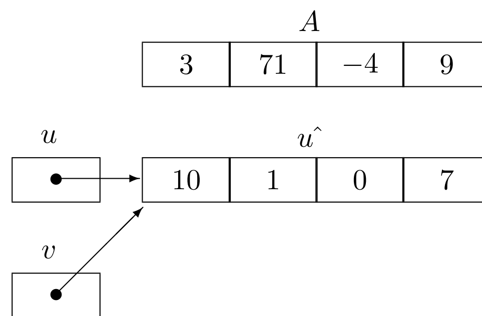
Po wykonaniu kolejno instrukcji

```
new(u);             {1}
v:=u;               {2}
new(v);             {3}
w:=u; u:=v; v:=w;   {4}
u^:=A               {5}
dispose(v); v:=Nil  {6}
new(u);             {7}
```

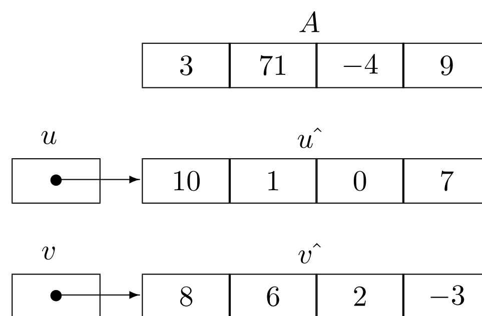
pamięć komputera będzie wyglądała jak następuje. Po wykonaniu instrukcji 1:



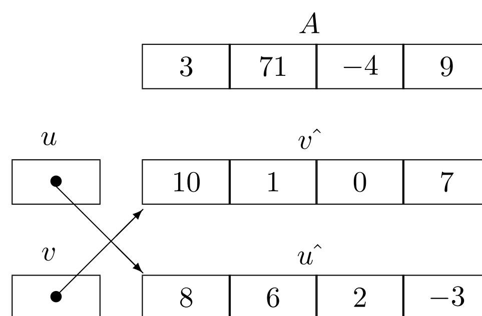
Po wykonaniu instrukcji 2:



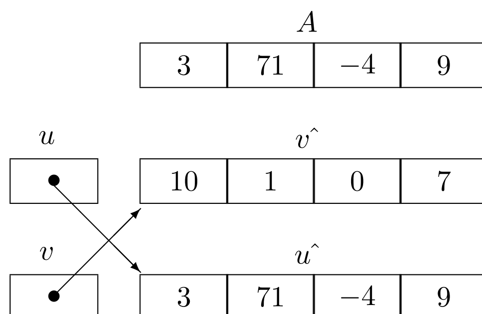
Po wykonaniu instrukcji 3:



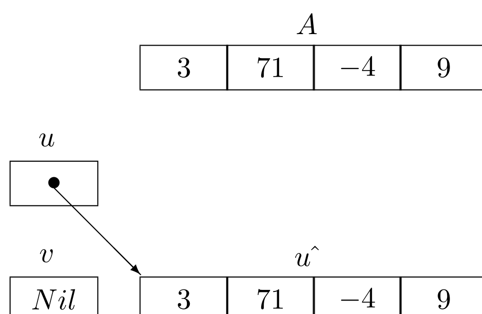
Po wykonaniu linii 4:



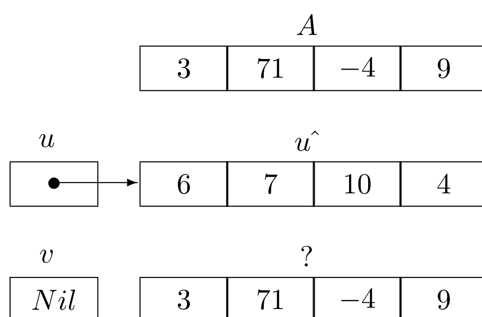
Po wykonaniu instrukcji 5:



Po wykonaniu linii 6:



Po wykonaniu instrukcji 7:



A teraz dokładnie opiszemy procedury `new`, tworzącą zmienną dynamiczną i `dispose` usuwającą zmienną dynamiczną.

Zakładamy następującą deklarację:

```
var u,v : ^Typ
```

Procedura `new(v)`:

1. tworzy nową zmienną (dynamiczną) typu `Typ` całkowicie nieokreśloną (rezerwuje miejsce w pamięci komputera na zmienną typy `Typ`);
2. tworzy nowy wskaźnik typu `^Typ` i przypisuje go zmiennej `v` (adres nowo utworzonej zmiennej dynamicznej zostaje przypisany zmiennej `v`);
3. Do utworzonej zmiennej można się odwołać przez `v^`.

Procedura `dispose(v)`:

1. usuwa zmienną (dynamiczną) typu `Typ` wskazywaną przez `v` (zwalnia miejsce w pamięci komputera zajmowane przez tę zmienną); jeśli `v` nie wskazuje zmiennej wystąpi błąd;
2. wszystkie zmienne wskazujące na `v^` mają wartość nieokreśloną.

Operacje na zmiennych i wartościach typów wskaźnikowych:

1. do każdego typu wskaźnikowego należy stała `Nil` nie wskazująca na żadną zmienną dynamiczną;
2. instrukcje przypisania: `v:=u` oraz `v:=Nil`;
3. relacje: `v=u` oraz `v<>u`;
4. funkcje mogą mieć wartości typów wskaźnikowych;
5. parametry formalne i aktualne procedur mogą być typu wskaźnikowego.

W następnym paragrafie pokażę jak można implementować listy używając typów wskaźnikowych. Ale typy wskaźnikowe mogą się też przydać do innych celów, na przykład do deklaracji dużych zmiennych. Deklaracja

```
type Wektor = array[1..10000] of real;
var A,B,C:Wektor;
```

nie zostanie zaakceptowana przez kompilator Turbo Pascala ponieważ ma on ograniczenie na łączny rozmiar zmiennych deklarowanych w sekcji deklaracji zmiennych `var` a zmienne typu Wektor zajmują dużo pamięci. Natomiast deklaracja

```
type Wektor = array[1..10000] of real;
     Wsk=^Wektor
var u,v,w:Wsk;
```

zostanie zaakceptowana przez kompilator Turbo Pascala ponieważ zmienne typu `Wsk` pamiętają tylko adresy i zajmują mało pamięci. Teraz na początku programu po instrukcjach

```
new(u); new(v); new(w);
```

mamy dostęp do trzech zmiennych dynamicznych typu Wektor: `u^, v^, w^`.

6.3 Implementacja list

Elementami listy są rekordy zawierające wartość (lub wartości) i wskaźnik do następnego elementu na liście.

Lista jednokierunkowa

```
type lista=^element;
     element=record
               nazwisko : string;
               wiek : integer;
               next :lista
             end;
```

Pokażemy jak wykonać następujące operacje na liście:

1. tworzenie pustej listy; (inicjalizacja)
2. tworzenie nowej zmiennej typu element;
3. wstawienie zmiennej wskazywanej przez zmienną wskaźnikową do listy;
4. znajdowanie wskaźnika do elementu o szukanym polu;
5. drukowanie wszystkich elementów listy;
6. usuwanie wskazanego elementu z listy (tylko dla list dwukierunkowych).

```
procedure ini;
 begin head:=Nil end;

function nowy: lista;
 var v:lista;
 begin
   new(v); v^.next:=Nil;
   read(v^.nazwisko);
   read(v^.wiek);
   nowy:=v
 end;

procedure dodaj(v:lista);
  begin
    v^.next:=head;
    head:=v
  end;

function szukaj(s:string):lista;
  var v:lista;
  begin
    v:=head; szukaj:=Nil;
    while (v<>Nil) do
      if v^.nazwisko=s then begin
         szukaj:=v; v:=Nil
        end
      else v:=v^.next
  end;

procedure druk;
var v:lista;
begin v:=head;
  while v<>Nil do begin
    writeln(v^.Nazwisko,' ',v^.wiek);
    v:=v^.next
  end;
end;
```

Teraz fragment programu:

```
ini;
for i:=1 to 10 do dodaj(nowa);
druk;
```

tworzy listę 10-elementową i ją drukuje.

Lista dwukierunkowa

```
type lista2=^element2;
     element2=record
               nazwisko : string;
               wiek : integer;
               next,prev :lista
             end;
```

Procedury `ini`, `druk`, `szukaj` dla listy dwukierunkowej są takie same jak dla listy jednokierunkowej. Natomiast procedury `nowy`, `dodaj` należy odpowiednio zmodyfikować:

```
function nowy2 : lista2;
var v:lista2;
begin
  new(v); v^.next:=Nil; v^.prev:=Nil;
  read(v^.nazwisko);
  read(v^.wiek);
  nowy2:=v
end;

procedure dodaj2(v:lista2);
  begin
    if head<>Nil then head^.prev:=v;
    v^.next:=head;
    head:=v;
    v^.prev:=Nil;
  end;
```

Ponadto na liście dwukierunkowej możemy łatwo usuwać wskazany element.

```
procedure usun2(x:lista2);
begin
  if x^.prev<>Nil then x^.prev^.next:=x^.next
                 else begin head:=x^.next;
                            if head<>Nil then head^.pre:=Nil
                      end;
  if x^.next<>Nil then x^.next^.prev:=x^.prev
end;
```

Lista dwukierunkowa z wartownikiem Procedury `dodaj2` i `usun2` są nieco skomplikowane, gdyż musimy sprawdzać czy lista jest pusta przy dodawaniu oraz czy są przed i za usuwanym elementem są inne elementy przy usuwaniu. Można te

procedury uprościć dodając do listy wartownika 'sztuczny element', który powoduje, że lista nigdy nie jest pusta. W efekcie, na liście z wartownikiem, dodawanie i usuwanie odbywa się bez sprawdzania żadnych warunków.

```
procedure ini3;
 begin new(head); head^.next:=head; head^.prev:=head
 end;

procedure dodaj3(v:lista2);
  begin
    v^.next:=head^.next;
    v^.prev:=head;
    v^.next^.prev:=v;
    head^.next:=v;
  end;

procedure usun3(x:lista2);
begin
   x^.prev^.next:=x^.next;
   x^.next^.prev:=x^.prev
end;

function szukaj3(s:string):lista2;
  var v:lista2;
  begin
    v:=head^.next;
    while (v<>head) and (v^.nazwa<>s) do
      v:=v^.next;
    if v<>head then szukaj3:=v
               else szukaj3:=Nil;
  end;

procedure druk3;
var v:lista2;
begin v:=head^.next;
  while v<>head do begin
    writeln(v^.Nazwisko,' 'v^.wiek);
    v:=v^.next
  end;
end;
```

Teraz, poniższa procedura czyści listę:

```
procedure empty;
var v:lista2;
begin
  while head<>head^.next do begin
    v:=head^.next;
    usun3(v);
    dispose(v)
```

```
    end;
  end;
```

Mając takie listy można łatwo implementować stosy i kolejki.

6.4 Drzewa binarnych poszukiwań (BST)

Drzewo binarne jest to drzewo, w którym każdy wierzchołek ma co najwyżej dwa następniki, lewy i prawy. Ponadto, o ile jest niepuste, posiada *korzeń* - jedyny wierzchołek, który nie jest następnikiem żadnego wierzchołka. Będziemy używali pojęć: *ojciec*, *lewy i prawy syn*, *lewe i prawe poddrzewo*, *potomek*, *przodek*.

Drzewo binarnych poszukiwań jest to drzewo binarne etykietowane, w którym każdy wierzchołek ma etykietę wyróżnioną zwaną *kluczem*. Klucze wierzchołków są typu, na którym określony jest porządek liniowy. Ponadto dla każdego wierzchołka v klucze w lewym poddrzewie są niewiększe od klucza wierzchołka v, a klucze w prawym poddrzewie są niemniejsze od klucza wierzchołka v.

W Pascalu drzewa binarnych poszukiwań implementuje się używając typów wskaźnikowych.

```
type wsk=^wierzcholek;
     wierzcholek=record
                    klucz:integer;
                    lewy,prawy,ojciec:wsk
                 end
```

Pokażemy procedury na drzewach:

1. znajdowanie wierzchołka o danym kluczu;
2. drukowanie rekordów według wzrastającej wartości kluczy;
3. znajdowanie wierzchołka o kluczu minimalnym;
4. znajdowanie wskaźnika do następnego wierzchołka;
5. dodawanie nowego wierzchołka;
6. usuwanie wierzchołka z drzewa;
7. inne przykładowe procedury rekurencyjne na drzewach.

1. **Znajdowanie wierzchołka o danym kluczu.** Procedura rekurencyjna `szukaj` zwraca wskaźnik do wierzchołka o kluczu `s` w drzewie o korzeniu `w` lub `Nil` jeśli nie ma takiego klucza. Procedura porusza się po drzewie, w lewo lub w prawo, z zależności od tego czy napotkane klucze na ścieżce są mniejsze czy większe od poszukiwanego. Jeśli dojdzie do końca ścieżki nie znajdując po drodze szukanego wierzchołka to znaczy, że takiego wierzchołka w ogóle nie ma w drzewie.

```
function szukaj(w:wsk;s:integer):wsk;
begin
  if w=Nil then szukaj:=Nil else
    if w^.klucz=s then szukaj:=w else
      if w^.klucz<s then szukaj:=szukaj(w^.prawy,s)
        else szukaj:=szukaj(w^.lewy,s)
end;
```

2. **Drukowanie rekordów według wzrastającej wartości kluczy.** Procedura rekurencyjna `drukuj` odwiedza wszystkie wierzchołki drzewa w porządku niemalejącym kluczy, tzn. dla danego wierzchołka zawsze odwiedza najpierw wierzchołki jego lewego poddrzewa, potem ten wierzchołek a na końcu wierzchołki jego prawego poddrzewa. Odwiedziwszy wierzchołek drukuje go.

```
procedure drukuj(w:wsk);
begin
  if w<>Nil then begin
    drukuj(w^.lewy);
    write(w^.klucz);
    drukuj(w^.prawy);
  end;
end;
```

3. **Znajdowanie wierzchołka o kluczu minimalnym.** Procedura `minimum` zwraca wskaźnik do wierzchołka o kluczu minimalnym drzewie o korzeniu `w` lub `Nil` jeśli drzewo jest puste. Procedura przechodzi po wierzchołkach w lewo tak długo aż nie trafi na wierzchołek, który nie ma lewego syna.

```
function minimum(w:wsk):wsk;
begin
  if w<>Nil then
    while w^.lewy<>Nil do w:=w^.lewy;
  minimum:=w
end;
```

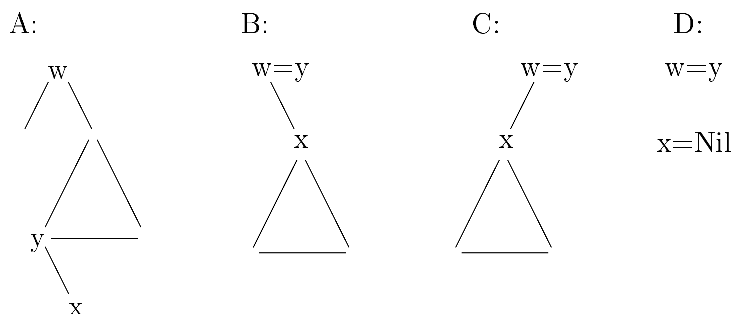
4. **Znajdowanie wskaźnika do następnego wierzchołka.** Procedura `next` zwraca wskaźnik do wierzchołka o kluczu następnym po kluczu wierzchołka wskazywanego przez `w` lub `Nil` jeśli nie ma takiego klucza. Procedura działa w ten sposób, że zwraca minimum poddrzewa o korzeniu `w^.prawy` o ile jest ono niepuste, a jeśli jest puste to wraca do góry po kolejnych przodkach aż do momentu gdy znajdzie wierzchołek, dla którego poprzedni wierzchołek jest lewym synem.

```
function next(w:wsk):wsk;
var y:wsk;
begin
  if w^.prawy<>Nil then next:=minimum(w^.prawy)
  else begin
    y:=w^.ojciec; next:=Nil;
    while y<>Nil do
      if w<>y^.lewy then begin
         w:=y; y:=y^.ojciec;
        end
      else begin
        next:=y; y:=Nil
      end;
  end;
end;
```

5. **Dodawanie nowego wierzchołka.** Procedura rekurencyjna `dodaj` dodaje nowy wierzchołek wskazywany przez `w` do drzewa binarnych poszukiwań o korzeniu wskazywanym przez `r` zachowując strukturę drzewa binarnego. Nowy wierzchołek jest zawsze dodawany jako nowy liść w drzewie. Procedura woła parametr korzenia `r` przez zmienną ponieważ, gdy drzewo jest puste, jest on modyfikowany.

```
procedure dodaj(var r:wsk;w:wsk);
begin
   w^.lewy:=Nil; w^.prawy:=Nil;
  if r=Nil then
    begin w^.ojciec:=Nil; r:=w end
  else
    if r^.klucz<w^.klucz then
      if r^.prawy<>Nil then dodaj(r^.prawy,w)
      else
        begin r^.prawy:=w; w^.ojciec:=r end
     else
       if r^.lewy<> Nil then dodaj(r^.lewy,w)
       else begin r^.lewy:=w; w^.ojciec:=r end
end;
```

6. **Usuwanie wierzchołka z drzewa.** Jeżeli wierzchołek wskazywany przez `w`, który mamy usunąć ma co najwyżej jedno niepuste poddrzewo (lewe lub prawe) to go usuwamy a to poddrzewo (o ile takie istnieje) podczepiamy bezpośrednio do jego ojca. Jeżeli wierzchołek wskazywany przez `w`, który mamy usunąć ma dwa niepuste poddrzewa to usuwamy wierzchołek następny (wtedy on ma co najwyżej jedno niepuste poddrzewo) a dane z tego wierzchołka przepisujemy do wierzchołka wskazywanego przez `w`. Ponadto, jeśli usuwamy korzeń, to musimy zmodyfikować wskaźnik do korzenia `r` oraz gdy usunięty wierzchołek jest różny od wskazywanego przez `w` to musimy zmodyfikować `w` by wskazywał usunięty wierzchołek. W procedurze używamy dodatkowych zmiennych `x,y` typu `wsk`, `y` wskazuje rzeczywiście usuwany wierzchołek a `x` jego jedynego syna (o ile takiego posiada). Procedura woła oba parametry przez zmienną, gdyż zarówno wskaźnik do korzenia jaki i do usuwanego wierzchołka może być zmieniony.



Powyższy rysunek przedstawia wartości zmiennych `x` i `y` w poddrzewie o wierzchołku wskazywanym przez `w` w różnych możliwych przypadkach. A gdy `w` ma dwóch synów, B, C gdy `w` ma jednego syna, D gdy `w` nie ma synów.

```
procedure usun(var r,w:wsk);
```

```
var x,y:wsk;
begin
  if (w^.lewy=Nil) or (w^.prawy=Nil) then y:=w
                                      else y:=next(w);
  {y wskazuje na wierzcholek ktory latwo usunac,
  bo ma co najwyzej jedno podrzewo}
  if y^.lewy<>Nil then x:=y^.lewy
                 else x:=y^.prawy;
  {x wskazuje na jedynego syna y, o ile taki istnieje}
  if x<>Nil then x^.ojciec:=y^.ojciec;
  {o ile x istnieje, to modyfikujemy ojca x tak
  by teraz byl nim ojciec y}
  if y^.ojciec=Nil then r:=x
  {jesli y wskazuje korzen to modyfikujemy korzen}
  else
    if y=y^.ojciec^.lewy then y^.ojciec^.lewy:=x
    else y^.ojciec^.prawy:=x;
    {jesli y nie wskazuje korzenia to modyfikujemy ojca
    y tak by teraz jego synem (z wlasciwej strony) byl x}
  if y<>w then begin
    w^.klucz:=y^.klucz
    ....
  end;
  {jesli usuniety wierzcholek wskazywany przez y jest
  rozny od wierzcholka wskazywanego przez w to
  przepisujemy wszystkie dane z pola klucz (i innych
  pol przechowujacych dane o ile takie istnieja) ale
  nie z pol 'administrujacych drzewem': lewy, prawy,
  ojciec}
  w:=y;
  {zwracamy na w wskaznik usunietego wierzcholka}
end;
```

7. **Inne przykładowe procedury rekurencyjne na drzewach.** Jeśli funkcję rekurencyjną można obliczyć znając jej wartości dla obu synów, to definicja takiej funkcji jest zwykle prosta, tak jak w pierwszych dwóch przykładach. Natomiast gdy tak nie jest, jak w przypadku trzecim, to musimy skonstruować nową funkcje, która będzie miała tę własność. W praktyce taka funkcja oblicza wiele wartości, z których jedna jest przez nas poszukiwaną a inne są tylko pomocnicze.

   (a) **Obliczanie liczby wierzchołków drzewa.**

   ```
   function rozmiar(w:wsk):integer;
   begin
     if w=Nil then rozmiar:=0
              else rozmiar:=1+rozmiar(w^.lewy)+rozmiar(w^.prawy)
   end;
   ```

   (b) **Obliczanie wysokości drzewa.**

   ```
   function wysokosc(w:wsk):integer;
   ```

```
  var l,p:integer;
  begin
    if w=Nil then wysokosc:=0
              else begin
                l:=wysokosc(w^.lewy);
                p:=wysoksc(w^.prawy);
                if l<p then l:=p;
                wysokosc:=l+1
              end;
  end;
```

(c) **Znajdowanie wskaźnika do wierzchołka drzewa, dla którego różnica wysokości poddrzew, lewego i prawego, jest największa.** Funkcja `roznica` ma jako parametr tylko korzeń drzewa i zwraca jako wartość tylko żądany wskaźnik. Ponieważ do obliczenia wartości funkcji w danym wierzchołku potrzebujemy więcej informacji dotyczącej obu poddrzew, funkcja `roznica` używa procedury lokalnej `roznica1`, która zwraca potrzebne informacje przez parametry wołane przez zmienną. Ciało samej funkcji `roznica` zawiera tylko jedno wywołanie procedury `roznica1` z odpowiednimi parametrami i wstawienie jednego z nich jako wartości funkcji.

```
function roznica(w:wsk):wsk;
  var u:wsk;
    d,h:integer;
  procedure roznica1(r:wsk; var u:wsk; var d,h:integer);
   var u1,u2:wsk;
   d1,d2,h1,h2:integer;
   begin
     if r=Nil then begin u:=Nil; d:=0; h:=0 end
     else begin
       roznica1(r^.lewy,u1,d1,h1);
       roznica1(r^.prawy,u2,d2,h2);
       {obliczamy rekurencyjnie wartosci dla obu
       poddrzew}
       if h1>h2 then begin h:=h1+1; d:= h1-h2 end
                else begin h:=h2+1; d:=h2-h1 end;
       if (d>d1) and (d>d2) then u:=r
       {jestesmy w miejscu gdzie jest najwieksza
       roznica poddrzew}
       else if d1>d2 then begin d:=d1; u:=u1 end
                     else begin d:=d2; u:=u2 end;
       {wstawiamy warosci poprzednie}
     end;
   end;
   begin{cialo funkcji roznica}
     roznica1(w,u,d,h);
     roznica:=u
   end;
```

6.5 Drzewa czerwono-czarne

Definicja i własności

W drzewach binarnych poszukiwań wiele operacji takich jak *dodaj*, *usuń*, *następnik*, *minimum* jest wykonywanych w czasie proporcjonalnym do wysokości drzewa $O(h)$. Jeśli wysokość drzewa jest porównywalna z liczba wierzchołków (przypadek ekstremalny ale możliwy) to te operacje są wykonywane bardzo nieefektywnie, podobnie jak na listach. Drzewa czerwono-czarne to pewien rodzaj drzew binarnych poszukiwań, w których mamy dodatkowy klucz *kolor* (*czerwony* lub *czarny*). Ponadto wymagamy by patrząc wyłącznie na czarne wierzchołki to drzewo było pełnym drzewem binarnym (warunek 4 poniżej) oraz żeby czerwonych wierzchołków było 'nie za dużo' (warunek 3 poniżej). Ponieważ pełne drzewa binarne są 'idealnie' zbalansowane to na drzewa czerwono-czarne można patrzeć jak na pełne drzewa binarne 'lekko' zaburzone czerwonymi wierzchołkami. Takie drzewa są 'zbalansowane' w tym sensie, że ich wysokość jest rzędu $O(\ln n)$ (gdzie n jest liczbą wierzchołków drzewa), w szczególności wszystkie wspomniane wyżej operacje są wykonywane w czasie $O(\ln n)$. Sposób pokolorowania wierzchołków drzewa czerwono-czarnego gwarantuje, że żadna ścieżka z korzenia do liścia nie jest więcej niż dwa razy dłuższa niż jakakolwiek inna.

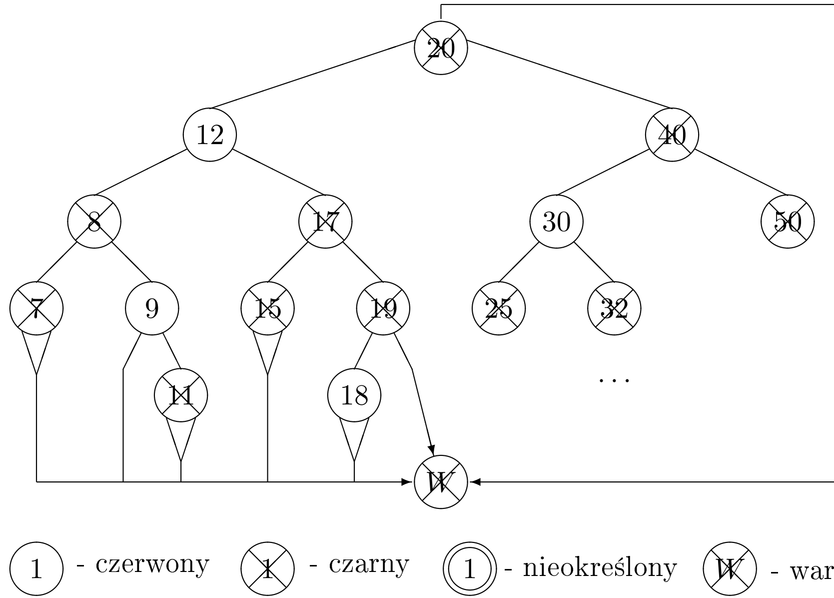
Najpierw mała modyfikacja drzewa binarnych poszukiwań. Dokładamy do drzewa *wartownika*. Podobnie jak w liście z wartownikiem, wartownik w drzewie to dodatkowy wierzchołek, na który wskazują wszystkie wskaźniki, które w normalnym drzewie binarnych poszukiwań wskazują na Nil. Wartownik jest zawsze ojcem korzenia. Ojcem wartownika może być każdy liść o ile tak to wcześniej zdefiniujemy. W szczególności teraz każdy *wierzchołek właściwy*, tzn. różny od wartownika, ma dwóch synów (być może obaj są tym samym wartownikiem).

Tak zmodyfikowane drzewo binarnych poszukiwań jest drzewem *czerwono-czarnym* jeśli następujące warunki:

1. Każdy wierzchołek jest czerwony lub czarny.
2. Wartownik jest czarny.
3. Jeśli wierzchołek jest czerwony obaj jego synowie są czarni.
4. Każda ścieżka z jednego wierzchołka do dowolnego liścia zawiera tą samą liczbę czarnych wierzchołków.

W szczególności możemy mówić o *czarnej wysokości* wierzchołka x, oznaczenie $ch(x)$, jako liczbie czarnych wierzchołków na dowolnej ścieżce z tego wierzchołka do wartownika, nie licząc wierzchołka x. Będziemy zawsze zakładali, że korzeń drzewa jest czarny.

Przykład 1.



Wskaźniki z wierzchołków o numerach 25, 32, i 50 do wartownika zostały pominięte.

Lemat 6.1 *Drzewo czerwono-czarne z n wierzchołkami właściwymi ma wysokość nie większą niż* $2\lg_2(n+1)$.

Dowód: Najpierw udowodnimy przez indukcję po wysokości wierzchołka, następujący fakt: drzewo czerwono-czarne o korzeniu x ma co najmniej $2^{ch(x)}-1$ wierzchołków właściwych. Dla wartownika w, mamy $ch(w) = 0$ oraz $2^{ch(w)} - 1 = 2^0 - 1 = 0$. I rzeczywiście drzewo o korzeniu w nie ma wierzchołków właściwych. Niech teraz x będzie wierzchołkiem właściwym. Wtedy jego synowie xl i xp mają czarną wysokość albo $ch(x)$ lub $ch(x) - 1$ w zależności od tego czy ich kolor jest czerwony czy czarny. Ponieważ wysokość synów wierzchołka x jest mniejsza niż wierzchołka x to z założenia indukcyjnego mamy, że drzewa o korzeniach w xl i xp mają co najmniej $2^{ch(x)-1} - 1$ wierzchołków. Zatem drzewo o korzeniu w x ma $2(2^{ch(x)-1} - 1) + 1 = (2^{ch(x)} - 2) + 1 = 2^{ch(x)} - 1$. To kończy dowód faktu.

By zakończyć dowód Lematu niech teraz h będzie wysokością drzewa o korzeniu r. Z warunku 3. definicji drzewa czerwono-czarnego co najmniej polowa wierzchołków na dowolnej ścieżce z korzenia do liścia (nie licząc korzenia) jest czarna. Zatem $ch(r) \geq \frac{h}{2}$. Zatem z faktu mamy $n \geq 2^{\frac{h}{2}} - 1$. Stąd $2 \cdot \lg_2(n+1) \geq h$. Q.E.D.

Z tego Lematu natychmiast wynika, że wspomniane na początku operacje na drzewie o n wierzchołkach są wykonywane w czasie $O(\lg(n))$.

Rotacja

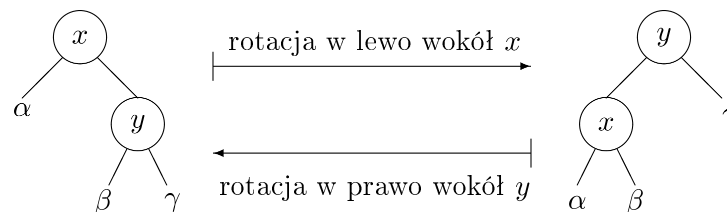
Dodawanie i usuwanie wierzchołków w drzewie czerwono-czarnym w taki sposób jak to robiliśmy w drzewie binarnych poszukiwań może zaburzyć strukturę drzewa czerwono-czarnego. By przywrócić tę strukturę musimy zmienić kolory niektórych wierzchołków i dokonać pewnych lokalnych zamian wskaźników. Zamian wskaźników dokonujemy przy pomocy rotacji: w lewo i w prawo, które nie naruszają struktury drzewa binarnych poszukiwań.

Zanim napiszemy procedurę dla rotacji zdefiniujemy nasze typy:

```
type kolory=(czerwony,czarny);
     wsk =^wierzcholek;
     wierzcholek=record
                  klucz:real;
                  ....
                  kolor:kolory;
                  ojciec,lewy,prawy:wsk
          end;
  var korzen,wartownik:wsk; {wskaznik do korzenia}
```

Zawsze jest prawdziwa zależność `korzen^.ojciec=wartownik` ale używamy zmiennej `wartownik` dla większej przejrzystości kodu procedur.

Przykład 2. Efekt działania procedur `rotacja_w_lewo` oraz `rotacja_w_prawo` przedstawia graficznie poniższy diagram



Opiszemy procedurę dla rotacji w lewo wokół wierzchołka `x`. Procedura dla rotacji w prawo jest symetryczna. Zakładamy, że `x^.prawy` jest wierzchołkiem właściwym, tzn. `x^.prawy<>wartownik`.

```
procedure rotacja_w_lewo(var r,x:wsk);
 var y:wsk;
 begin
  y:=x^.prawy; {y staje sie prawym synem x}
  x^.prawy:=y^.lewy;
  {podczepiamy lewo poddrzewo y jako prawe poddrzewo x}
  x^.prawy^.ojciec:=x
  {modyfikujemy ojca x^.prawy nawet gdy jest on wartownikiem}
  y^.ojciec:=x^.ojciec; {modyfikujemy ojca y}
  if y^.ojciec=wartownik then korzen:=y
  else if x=x^.ojciec^.lewy then x^.ojciec^.lewy:=y
                            else x^.ojciec^.prawy:=y;
  y^.lewy:=x;
  x^.ojciec:=y;
 end;
```

Dodawanie wierzchołków

Dodawanie wierzchołków do drzew czerwono-czarnych odbywa się w dwóch krokach. Na początku dodajemy wierzchołek tak jak do normalnego drzewa binarnych poszukiwań procedurą `dodaj` odpowiednio zmodyfikowaną by działała na drzewie z wartownikiem, i malujemy go na czerwono. To powoduje, że drzewo pozostaje drzewem binarnych poszukiwań oraz wszystkie warunki drzewa czerwono-czarnego, z wyjątkiem być może warunku 3, są spełnione. Warunek 3 nie będzie spełniony o

ile dołączyliśmy nowy wierzchołek (czerwony) jako syna czerwonego wierzchołka. Następnie procedura `dodaj_cc` poprawia drzewo przy pomocy przemalowań i rotacji wierzchołków tak by nie niszcząc struktury drzewa binarnych poszukiwań przywrócić strukturę drzewa czerwono-czarnego. Przez cały czas poprawiania drzewa wszystkie warunki, z wyjątkiem warunku 3, są spełnione. Natomiast warunek 3 nie jest spełniony dla co najwyżej jednej pary wierzchołków dla x i jego ojca. W każdym obrocie pętli `while` dokonuje się pewnych przemalowań oraz albo x staje się swoim dziadkiem przesuwając do góry o dwa pare wierzchołków, które nie spełniają warunku 3 (przypadek 1), albo dokonuje się jednej (przypadek 3) lub dwóch (przypadek 2) rotacji i pętla `while` się kończy.

W pętli `while` należy rozważyć sześć przypadków. Z tym, że trzy z nich są symetryczne. Przykład 3 poniżej pokazuje trzy przypadki gdy ojciec x jest lewym synem swojego dziadka, tzn. gdy `x^.ojciec=x^.ojciec^.ojciec^.lewy`. Przypadki gdy ojciec x jest prawym synem swojego dziadka, tzn. gdy `x^.ojciec=x^.ojciec^.ojciec^.prawy` są analogiczne.

Przykład 3. Założenie dla wszystkich trzech rozważanych przypadków (pozostałe przypadki są symetryczne):

1. ojciec wierzchołka x jest lewym synem swojego ojca;

2. y jest wujem x.

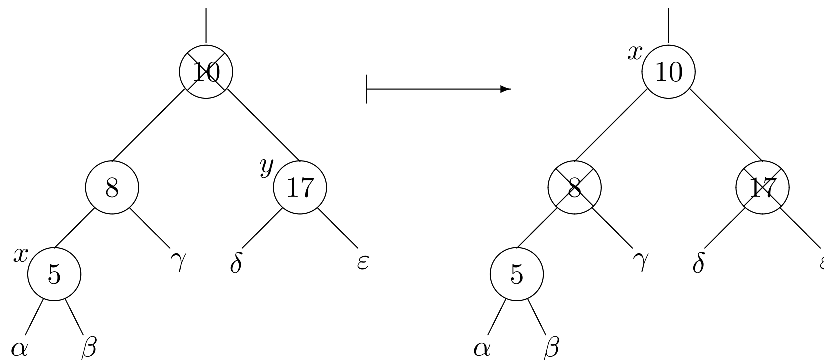
Przypadek 1. Założenie:

- y jest czerwony.

Akcja:

- zamieniamy kolory dziadka x i jego synów;

- x staje się swoim dziadkiem.



Przypadek 2. Założenie:

- y jest czarny;

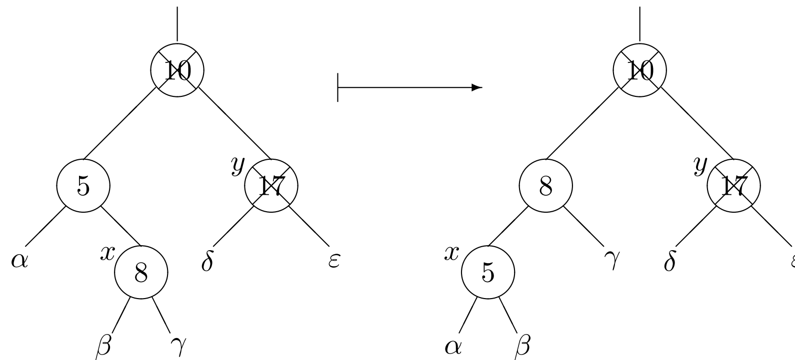
- x jest prawym synem.

Akcja:

1. x staje się swoim ojcem;

2. rotacja w lewo wokół x;

3. ... i przechodzimy do Przypadku 3.

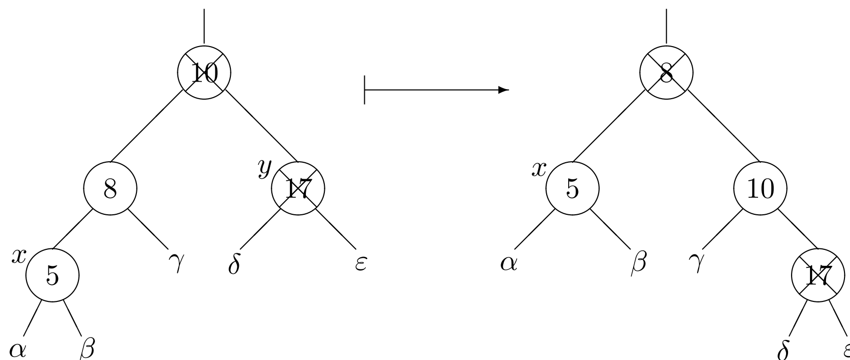


Przypadek 3. Założenie:

- y jest czarny;
- x jest lewym synem.

Akcja:

1. zamieniamy kolory ojca i dziadka x;
2. rotacja w prawo wokół dziadka x;
3. ... i koniec.



r wskazuje na korzeń drzewa a x na wstawiany wierzchołek.

```
procedure dodaj(var r,x:wsk);
begin
  x^.lewy:=wartownik; x^.prawy:=wartownik;
  if r=wartownik then
    begin x^.ojciec:=wartownik; r:=x end
  else
    if r^.klucz<x^.klucz then
      if r^.prawy<>wartownik then dodaj(r^.prawy,x)
      else begin r^.prawy:=x; x^.ojciec:=r end
    else
      if r^.lewy<> wartownik then dodaj(r^.lewy,x)
      else begin r^.lewy:=x; x^.ojciec:=r end;
end;
```

```
procedure dodaj_cc(var r,x:wsk);
begin
  dodaj(r,x);
  x^.kolor:=czerwony;
  while (x<>r) and (x^.ojciec^.kolor=czerwony) do
    if x^.ojciec=x^.ojciec^.ojciec^.lewy then begin
      y:=x^.ojciec^.ojciec^.prawy; {y jest teraz wujem x}
      if y^.kolor=czerwony then begin
      {przypadek 1: wuj y jest czerwony}
        x^.ojciec^.kolor:=czarny;
        y^.kolor:=czarny;
        x^.ojciec^.ojciec^.kolor:=czerwony;
        x:=x^.ojciec^.ojciec;
      end else begin
                 if x=x^.ojciec^.prawy then begin
                 {przypadek 2: wuj y jest czarny, x jest prawym synem}
                   x:=x^.ojciec;
                   rotacja_w_lewo(r,x);
                 end;
                 {przypadek 3: wuj y jest czarny, x jest lewym synem}
                 x^.ojciec^.kolor:=czarny;
                 x^.ojciec^.ojciec^.kolor:=czerwony;
                 rotacja_w_prawo(r,x^.ojciec^.ojciec);
               end;
      end
    end else begin
      (to samo co dla warunku 'then' ale ze zamiana 'lewy' z 'prawy')
    end;
    r^.kolor:=czarny; {kolor korzenia zawsze ma byc czarny}
end;
```

Opiszemy teraz dokładniej powyższe procedury. Procedura `dodaj` wstawia wierzchołek tak jak do drzewa binarnych poszukiwań pamiętając, że teraz jest to drzewo z wartownikiem. Następnie malujemy wstawiony wierzchołek na czerwono. O ile przez wstawienie wierzchołka zaburzyliśmy strukturę drzewa czerwono-czarnego to w pętli `while` procedury `doddaj_cc` przywracamy tę strukturę. Pętla `while` działa tak długo jak długo `x` i jego ojciec są czerwoni (jedyne miejsce w całym drzewie gdzie warunek 3 może nie być spełniony) oraz `x` nie jest korzeniem. Ponieważ korzeń nie może być czerwony więc `x` ma dziadka `x^.ojciec^.ojciec`. W pętli `while` należ rozpatrzeć sześć przypadków w tym trzy pierwsze są symetryczne do pozostałych trzech i zależą od tego czy wierzchołek `x^.ojciec` ojciec `x` jest lewym czy prawym synem swojego dziadka. Część procedury opisana powyżej jest dla przypadku gdy ojciec `x` jest lewym synem swojego dziadka. Drugi przypadek jest symetryczny. Zatem wewnątrz pętli `while` wierzchołek `x` jest czerwony i jego ojciec też. Ponadto, ponieważ `x` ma dziadka, to ma też wuja `y`, który jest synem dziadka, ale nie ojcem. Przypadek 1 zachodzi gdy wuj `y` jest czerwony. Wtedy zamieniamy kolory dziadka `x` i jego synów. To eliminuje istniejący problem z warunkiem 3 ale jeśli pradziadek `x` jest czerwony dziadek i pradziadek tworzą nową parę, która nie spełnia warunku 3. Dlatego trzeba przesunąć problem do góry i teraz nowym `x`'em staje się dziadek `x`. W przypadkach 2 i 3 wuj `y` jest czarny. Przypadek 2 zachodzi gdy `x` jest prawym synem. W tym

przypadku przesuwamy `x` na jego ojca i dokonujemy rotacji w lewo wokół `x`. To nas doprowadza do Przypadku 3, który zachodzi gdy `x` jest lewym synem. Oczywiście, jeśli od razu `x` był lewym synem to dochodzimy do niego bez przechodzenia przez Przypadek 2. W Przypadku 3 zamieniamy kolory ojca i dziadka `x` dokonujemy rotacji wokół dziadka `x` i kończymy pętlę `while`. Na koniec 'na wszelki wypadek' malujemy korzeń na czarno jeśli przy powyższych operacjach stał się czerwony.

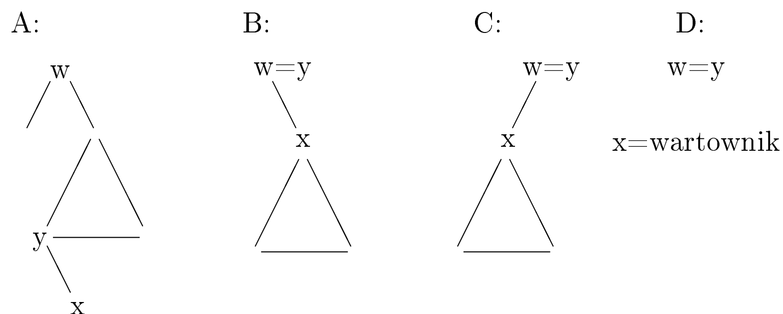
Lemat 6.2 *Procedura* `dodaj_cc` *na drzewie czerwono-czarnym o n wierzchołkach działa w czasie $O(\lg(n))$.*

Dowód: Na mocy Lematu 6.1 wysokość drzewa czerwono-czarnego o n wierzchołkach jest $O(\lg(n))$. Ponieważ wstawienie wierzchołka do drzewa binarnych poszukiwań jak i pętla poprawiająca drzewo po wstawieniu nowego wierzchołka działa w czasie $O(h)$ gdzie h jest wysokością drzewa to cała procedura działa w czasie $O(\lg(n))$. Q.E.D.

Usuwanie wierzchołków

Procedura usuwania wierzchołka z drzewa czerwono-czarnego `usun_cc` jest nieznaczną modyfikacją procedury `usun` usuwania wierzchołka z drzewa binarnych poszukiwań. Po usunięciu wierzchołka procedura `usun_cc` woła pomocniczą procedurę `popraw_cc`, która przy pomocy przemalowań i rotacji przywraca drzewu binarnych poszukiwań strukturę drzewa czerwono-czarnego.

Przykład 4.



Powyższy rysunek przedstawia wartości zmiennych `x` i `y` w poddrzewie o wierzchołku wskazywanym przez `w` w różnych możliwych przypadkach. A gdy `w` ma dwóch synów, B, C gdy `w` ma jednego syna, D gdy `w` nie ma synów.

r wskazuje na korzeń drzewa a w na usuwany wierzchołek.

```
procedure usun_cc(var r,w:wsk);
var x,y:wsk;
begin
  if (w^.lewy=wartownik) or (w^.prawy=wartownik) then y:=w
                                      else y:=next(w);
  {y wskazuje na wierzcholek ktory latwo usunac,
  bo ma co najwyzej jedno podrzewo}
  if y^.lewy<>wartownik then x:=y^.lewy
                else x:=y^.prawy;
  {x wskazuje na jedynego syna y, o ile taki istnieje}
  x^.ojciec:=y^.ojciec;
  {modyfikujemy ojca x (byc moze wartownika) tak by teraz
  byl nim ojciec y}
  if y^.ojciec=wartownik then r:=x
  {jesli y wskazuje korzen to modyfikujemy korzen}
  else
    if y=y^.ojciec^.lewy then y^.ojciec^.lewy:=x
    else y^.ojciec^.prawy:=x;
    {jesli y nie wskazuje korzenia to modyfikujemy ojca
    y tak by teraz jego synem (z wlasciwej strony) byl x}
  if y<>w then begin
    w^.klucz:=y^.klucz
    ....
  end;
  {jesli usuniety wierzcholek wskazywany przez y jest
  rozny od wierzcholka wskazywanego przez w to
  przepisujemy wszystkie dane z pola klucz (i innych
  pol przechowujacych dane o ile takie istnieja) ale
  nie z pol 'administrujacych drzewem': lewy, prawy, ojciec}
  if y^.kolor=czarny then popraw_cc(r,x);
  {jeśli usunięty wierzchołek był czarny to wolamy procedure
  przywracajaca strukture drzewa czrwono czarnego}
  w:=y;
  {zwracamy na w wskaznik usunietego wierzcholka}
end;
```

Ta procedura różni się od procedury `usun` trzema detalami. Wszystkie odwołania do stałej Nil są zamienione na odwołania do zmiennej `wartownik`. Przyporządkowanie ojcu `x` ojca `y` jest bezwarunkowe ponieważ `x` nie może być równe Nil a co najwyżej może być `wartownikiem` w którym to przypadku nic złego (ani dobrego też) się nie dzieje. W końcu po usunięciu `y`, o ile był to wierzchołek czarny, wołana jest procedura `popraw_cc` z parametrem `x` będącym jedynym synem `y` (lub wartownikiem jeśli `y` nie miał synów), która mam za zadanie odtworzenie struktury drzewa czerwono-czarnego.

Po usunięciu czarnego wierzchołka z drzewa warunek 4. definicji drzewa czerwono-czarnego nie jest spełniony przez ojca wierzchołka `y`. Możemy 'poprawić' ten defekt myśląc o wierzchołku `x` jak by był 'podwójnie czarny' tzn. natrafienie na ten wierzchołek na ścieżce liczy się podwójnie do czarnej wysokości. Teraz możemy myśleć, ze warunek 4 jest spełniony ale warunek 1 nie, bo `x` jest nie tyle czarny co 'podwójnie czarny'. To pozwala na myślenie, że procedura `popraw_cc` przywraca

spełniania warunku 1. Pętla `while` procedury `popraw_cc` przesuwa w stronę korzenia wierzchołek podwójnie czarny aż do momentu gdy

1. `x` wskazuje na wierzchołek czerwony, a wtedy maluje `x` na czarno i kończy działanie;
2. `x` wskazuje na korzeń, i wtedy dodatkowy kolor czarny na `x` może być po prostu zdjęty;
3. można dokonać odpowiednich przemalowań i rotacji.

Wewnątrz pętli `while` wierzchołek `x` jest 'podwójnie czarny' oraz nie jest korzeniem.

Przykład 5. Założenie dla czterech rozważanych przypadków:

1. x jest lewym synem swojego ojca;
2. z jest bratem x.

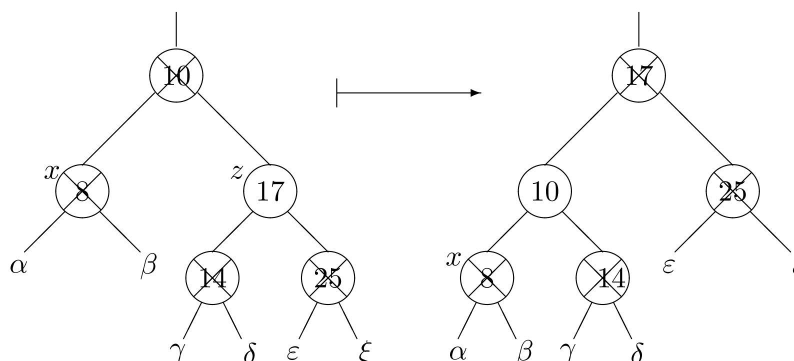
Pozostałe przypadki, gdy x jest prawym synem swojego ojca, są symetryczne. Zauważmy, że skoro x jest podwójnie czarny to jego brat z musi mieć dwóch synów.

Przypadek 1. Założenie:

- z jest czerwony.

Akcja:

- zamieniamy kolory z i ojca z;
- rotacja w lewo wokół ojca x;
- wtedy nowy brat x jest czarny... i przechodzimy do Przypadków 2,3,4.

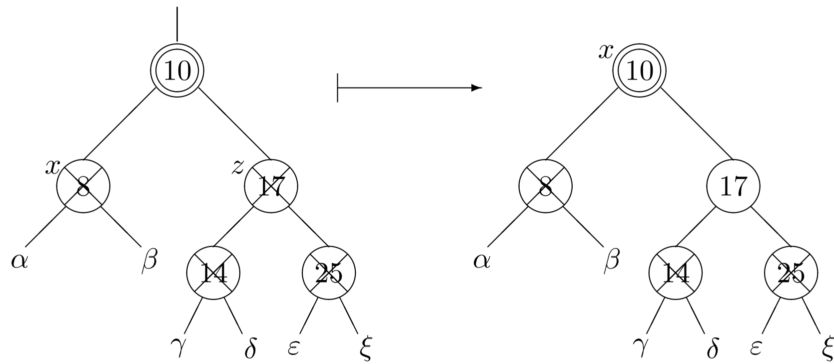


Przypadek 2. Założenie:

- z jest czarny;
- synowie z są czarni.

Akcja:

- malujemy z na czerwono;
- x staje się ojcem x.

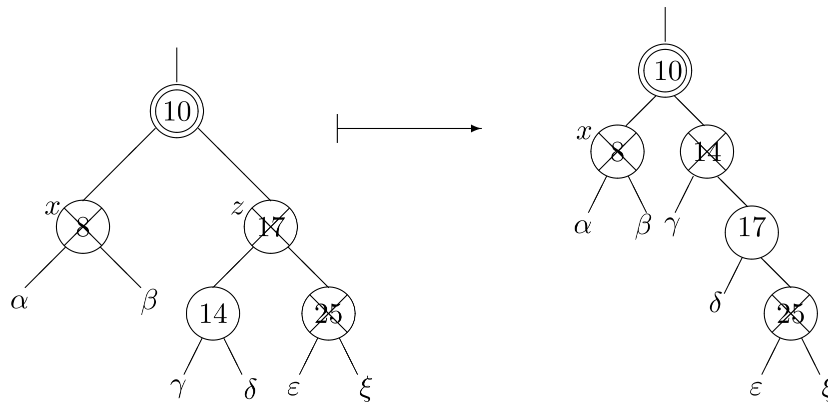


Przypadek 3. Założenie:

- z jest czarny;
- lewy syn z jest czerwony;
- prawy syn z jest czarny.

Akcja:

- zamieniamy kolory z i jego lewego syna;
- rotacja w prawo wokół z;
- ... i przechodzimy do Przypadku 4.

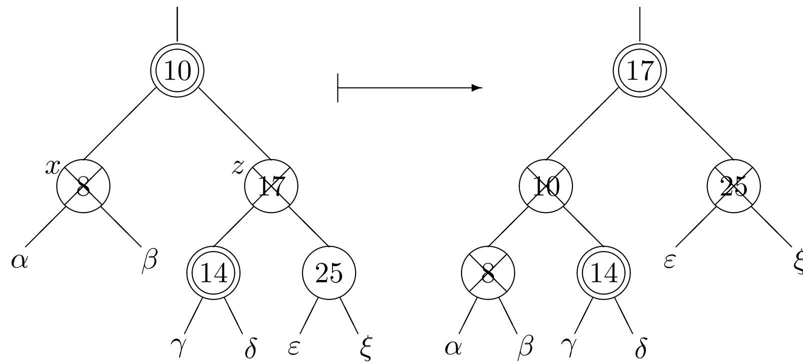


Przypadek 4. Założenie:

- z jest czarny;
- prawy syn z jest czerwony.

Akcja:

- zamieniamy kolory z i ojca z;
- malujemy prawego syna z na czarno;
- rotacja w lewo wokół ojca x;
- ... i koniec.




```
procedure popraw_cc(var r,x:wsk); var z:wsk;
 begin
   while (x<>r) and (x^.kolor=czarny) do
     if x=x^.ojciec^.lewy then begin
       z:=x^ojciec^.prawy;
       if z^.kolor=czerwony then begin
         {przypadek 1: brat x jest czerwony}
         z^.kolor:=czarny;
         x^.ojciec:=czerwony;
         rotacja_w_lewo(r,x^.ojciec);
         z:=x^ojciec^.prawy;
       end;
       if (z^.lewy^.kolor=czarny) and (z^.prawy^.kolor=czarny) then begin
         {przypadek 2: brat x jest czarny i jego synowie tez}
         z^.kolor:=czerwony;
         x:=x^ojciec
       end else begin
                   if (z^.prawy^.kolor=czarny) then begin
                     {przypadek 3: brat x jest czarny i jego prawy syn tez}
                     z^.lewy^kolor:=czarny;
                     z^.kolor:=czerwony;
                     rotacja_w_prawo(r,z);
                     z:=x^.ojciec^.prawy;
                   end;
                   {przypadek 4: brat x jest czarny a jego prawy syn czerwony}
                   z^.kolor:=x^.ojciec^.kolor;
                   x^.ojciec:=czarny;
                   z^.prawy^.kolor:=czarny;
                   rotacja_w_lewo(r,x^.ojciec);
                   x:=r;
                 end
     end else begin
      (to samo co dla warunku 'then' ale z zamiana 'lewy' z 'prawy')
     end;
   x^.kolor:=czarny;
 end;
```

Opiszemy teraz dokładniej działanie powyższej procedury. W pętli `while` należ rozpatrzeć osiem przypadków w tym cztery pierwsze są symetryczne do pozostałych

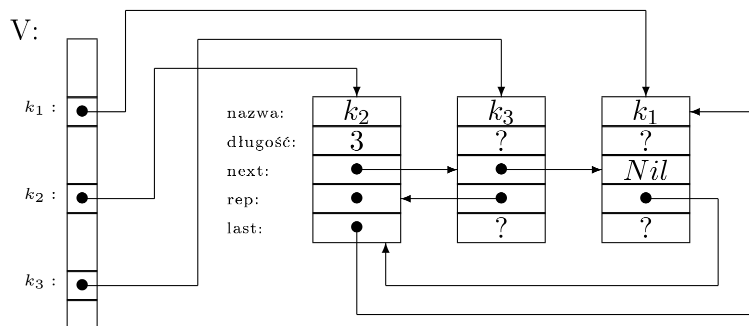
czterech i zależą od tego czy wierzchołek x jest lewym czy prawym synem swojego ojca. Część procedury opisana powyżej jest dla przypadku gdy x jest lewym synem swojego ojca. Drugi przypadek jest symetryczny. Zatem, tak jak wspomnieliśmy, wewnątrz pętli while wierzchołek x jest 'podwójnie czarny' oraz nie jest korzeniem. Ponieważ x jest 'podwójnie czarny' oraz drzewo spełnia warunek 4, to x musi mieć brata, którego będziemy oznaczali z, oraz z musi mieć dwóch synów. Przypadek 1 zachodzi gdy z jest czerwony. W tym przypadku zamieniamy kolory z i ojca z i dokonujemy rotacji wokół ojca z. Wtedy nowy brat x jest czarny i przechodzimy do Przypadków 2,3 i 4. Przypadek 2 zachodzi wtedy gdy z jest czarny i jego synowie też. W tym przypadku malujemy z na czerwono i x staje się swoim ojcem. Przypadek 3 zachodzi wtedy gdy z jest czarny, jego lewy syn jest czerwony a prawy syn jest czarny. W tym przypadku zamieniamy kolory z i jego lewego syna oraz dokonujemy rotacji w prawo wokół z. W ten sposób przechodzimy do Przypadku 4, który zachodzi gdy z jest czarny a jego prawy syn jest czerwony. W tym przypadku zamieniamy kolory z i jego ojca, prawego syna z malujemy na czarno i dokonujemy rotacji w lewo wokół x... i koniec.

6.6 Struktury danych dla rodziny zbiorów rozłącznych

Rozwiązywanie szeregu problemów wymaga grupowania elementów w zbiory rozłączne. W takiej sytuacji potrzebna jest nam struktura, która pozwala na szybkie wykonywanie trzech operacji:

1. `make-set(k)` - tworzenia zbioru, którego jedynym elementem jest k;
2. `union(k,l)` - sumowania dwóch zbiorów rozłącznych, do których należą elementy `k` i `l`;
3. `find(k)` - znajdowania reprezentanta zbioru, do którego należy element `k`.

Teraz opiszę taką strukturę. Zakładamy, że operacji `make-set` jest co najwyżej n. Każdy zbiór jest reprezentowany przez listę. Elementy listy mają wskaźniki do następnego elementu i do elementu reprezentującego dany zbiór. Dodawanie będzie polegało na łączeniu list. By przyspieszyć dodawanie list będziemy też pamiętali w reprezentancie zbioru długość listy (by dodawać krótszą listę do dłuższej) i wskaźnik do ostatniego elementu listy. Wskaźniki do elementów będziemy pamiętali w osobnej tablicy. Na obrazku można to przedstawić tak:



gdzie ? oznacza, że przechowywana w tym polu wartość nie jest aktualna.

Operacje `make-set` i `find` są wykonywane w czasie stałym, natomiast `union` jest wykonywana w czasie proporcjonalnym do mniejszego z sumowanych zbiorów.

```
type wsk=^element;
     element=record
               nazwa,dlugosc:integer;
               rep,next,last:wsk
             end;
var V:array[1..n] of wsk;
```

W $V[k]$ jest wskaźnik do elementu k lub `Nil`, jeśli jeśli k nie należy do żadnego zbioru.

```
procedure make-set(k:integer);
 var u:wsk;
 begin
   new(u); V[k]:=u;
   with u^ do begin
     nazwa:=k;
```

```
      next:=Nil;
      rep:=u;
      dlugosc:=1;
      last:=u
    end;
  end;

function find(k:integer):wsk;
  begin
    find:=V[k]^.rep
  end;

procedure union(k,l:integer);
  var u,w,z:wsk;
  begin
    u:=find(k); w:=find(l);
    {u i w wskazuja na reprezentantow zbiorow
    do ktorych naleza k i l, odpowiednio}
    if w^.dlugosc<u^.dlugosc then begin
      z:=u; u:=w; w:=z
    end;
    {teraz w wskazuje na reprezentanta dluzszej listy}
    w^.last^.next:=u; {polaczenie list}
    w^.dlugosc:=w^.dlugosc+u^.dlugosc; {uaktualnienie dlugosci}
    w^.last:=u^.last; {uaktualnienie ostatniego elementu}
    while u<>Nil do begin {uaktualnienie reprezentanta}
      u^.rep:=w;
      u:=u^.next;
    end;
  end;
```

Fakt 6.3 *n operacji* `make-set`*,* `union`*, i* `find` *wśród, których jest m operacji* `make-set` *jest wykonywanych w czasie $O(n + m\log(m))$.*

Dowód. Procedury `make-set` i `find` działają w czasie stałym. Zatem n takich operacji jest wykonywanych w czasie $O(n)$.

Pozostaje do pokazania, że jeśli wykonanych zostało m operacji `make-set` to łączny czas wykonania operacji `union` jest $O(m\log(m))$.

Procedura `union` modyfikuje szereg wartości reprezentanta sumy zbiorów i jeden wskaźnik (`next`) ostatniego elementu dłuższej listy. Te operacje są przy każdym wykonaniu procedury `union` wykonywane w czasie stałym. Ponieważ procedura union może być wykonana co najwyżej $m - 1$ razy (ponieważ wszystkich elementów we wszystkich zbiorach jest m), to łączny czas wykonania tej części procedury `union` jest $O(m)$.

Pozostaje do pokazania, że łączny czas wykonania drugiej części procedury `union` modyfikujący reprezentanta na krótszej liście jest $O(m\log(m))$. W tym celu policzymy ile razy łącznie może być modyfikowany reprezentant jednego elementu. Ponieważ modyfikujemy reprezentantów elementów tylko na krótszej liście, to po jednej modyfikacji lista, na której znajduje się element ma co najmniej dwa elementy, po dwóch modyfikacjach lista, na której znajduje się element ma co najmniej cztery elementy, ... po k modyfikacjach lista, na której znajduje się element ma co najmniej 2^k

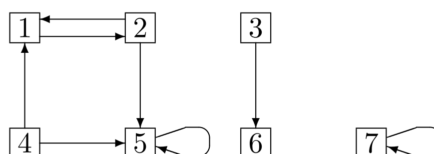
elementów. Ponieważ wszystkich elementów jest m to $2^k \leq m$. A stąd $k \leq \log_2(m)$. Czyli liczba modyfikacji reprezentanta każdego elementu jest niewiększa niż $\log_2(m)$. Stad łączna liczba modyfikacji reprezentantów wszystkich m elementów jest równa $O(m \log(m))$. Q.E.D.

7 Algorytmy grafowe

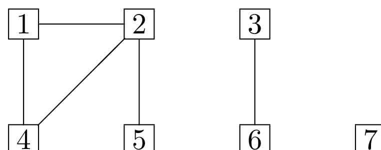
7.1 Grafy i reprezentacje grafów

Grafy są to bardzo proste struktury, które jednak pozwalają na modelowanie wielu 'rzeczywistych' sytuacji. Algorytmy grafowe pozwalają rozwiązywać problemy dotyczące takich sytuacji. Dlatego są jednymi najważniejszych podstawowych algorytmów i mają szerokie zastosowania. Poniżej zdefiniuję szereg podstawowych pojęć dotyczących grafów, pokażę kilka sytuacji, które można modelować przy pomocy grafów, oraz pokażę jak można reprezentować grafy w komputerze. W dalszym ciągu omówię najbardziej podstawowe algorytmy grafowe.

Graf zorientowany G jest parą uporządkowaną (V, E) taką, że V jest zbiorem (skończonym) *wierzchołków*, a E jest podzbiorem $V \times V$. Elementy E nazywany *krawędziami*. Czyli graf zorientowany jest to zbiór (skończony) z relacją binarną. Jeśli $e = (u, v) \in E$ to mówimy, że e jest krawędzią z u do v, lub krawędzią o początku u i końcu v.



Graf niezorientowany G jest parą uporządkowaną (V, E) taką, że V jest zbiorem (skończonym) *wierzchołków*, a E, zbiór *krawędzi*, jest podzbiorem zbioru par nieuporządkowanych z V, tzn. zbioru $\{(u, v) : u \neq v,\ u, v \in V\}$. W grafie niezorientowanym E można utożsamiać z relacją binarną na $V \times V$, która jest symetryczna i antyzwrotna, pamiętając jednak, że (u, v) i (v, u) oznacza tę samą krawędź. Jeśli $e = (u, v) \in E$ to mówimy, że e jest krawędzią z u do v, lub krawędzią o początku u i końcu v, lub krawędzią o końcach w u i v.



Wiele definicji dla obu rodzajów grafów jest identycznych, nawet jeśli ich znaczenie jest różne. Jeśli kontekst nie określa jasno czy dany graf jest zorientowany czy nie, to znaczy, że chodzi mi w takim przypadku o oba rodzaje grafów.

Stopniem wierzchołka v w grafie niezorientowanym G, nazywamy liczbę krawędzi G o początku w v.

Stopniem wyjściowym wierzchołka v w grafie zorientowanym G, nazywamy liczbę krawędzi G o początku w v. *Stopniem wejściowym* wierzchołka v w grafie zorientowanym G, nazywamy liczbę krawędzi G o końcu w v.

Ścieżką (lub drogą) długości k z wierzchołka u do wierzchołka v w grafie $G = (V, E)$, nazywamy ciąg wierzchołków $< v_0, v_1, \ldots, v_k >$, taki, że $v_0 = u$, $v_k = v$ oraz $(v_i, v_{i+1}) \in E$ dla $i = 0, \ldots, k$. Długością ścieżki jest liczba krawędzi na ścieżce. Ścieżka jest *prosta*, jeśli wszystkie wierzchołki na ścieżce są różne.

Cyklem, w grafie zorientowanym, nazywamy ścieżkę, która zaczyna się i kończy w tym samym wierzchołku i posiada co najmniej jedną krawędź. Cykl $< v_0, v_1, \ldots, v_k >$ jest *prosty*, jeśli wszystkie wierzchołki $v_1, \ldots, v_k$ są różne.

Ścieżka $< v_0, v_1, \ldots, v_k >$, w grafie niezorientowanym, jest *cyklem*, o ile $v_0 = v_k$ oraz wierzchołki $v_1, \ldots, v_k$ są różne.

Graf, w którym nie ma cykli nazywamy *acyklicznym*.

Jeśli istnieje ścieżka z u do v to v jest *osiągalny* z u i oznaczamy $u \mapsto v$

Graf niezorientowany jest *spójny*, jeśli każde dwa wierzchołki łączy ścieżka. *Składowe spójne* grafu niezorientowanego, są to klasy abstrakcji relacji osiągalności w grafie.

Graf $G' = (V', E')$ jest *podgrafem* grafu $G = (V, E)$, jeśli $V' \subseteq V$ i $E' \subseteq E$. G' jest podgrafem *pełnym* G jeśli $E' = E \cap (V \times V)$.

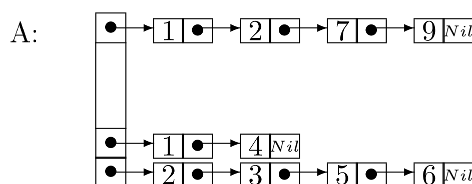
Macierze incydencji. Grafy można reprezentować przy pomocy macierzy incydencji. Przy takiej reprezentacji graf $G = (V, E)$, gdzie $V = \{v_1, \ldots v_n\}$ jest reprezentowany przez macierz M typu `array[1..n,1..n] of integer`, tak, że

$$M[i,j] = \begin{cases} 1 & \text{gdy } (v_i, v_j) \in E; \\ 0 & \text{w przeciwnym przypadku.} \end{cases}$$

Wtedy łatwo jest sprawdzić czy (v_i, v_j) jest krawędzią w grafie G. Ale zwykle jest to reprezentacja bardzo pamięciochłonna. Reprezentacja grafu o n wierzchołkach, niezależnie od liczby krawędzi, zużywa ona $O(n^2)$ pamięci.

Używając macierzy incydencji łatwo jest obliczyć liczbę ścieżek danej długości k pomiędzy wierzchołkami. W tym celu wystarczy obliczyć k-krotny iloczyn macierzy A przez siebie, tzn. A^k. Jeśli nas interesują ścieżki długości co najwyżej k, to możemy je obliczyć następująco $\sum_{i=1}^{k} A^i$.

Listy incydencji. Listy incydencji są często bardziej ekonomicznym sposobem reprezentowania grafów. Zwykle grafy pojawiające się w praktyce mają mały stopień (wyjściowy) wierzchołków i liczbę krawędzi rzędu $O(n)$ raczej niż $O(n^2)$. W takiej sytuacji lepiej je reprezentować przez listy incydencji. Przy takiej reprezentacji graf $G = (V, E)$, gdzie $V = \{v_1, \ldots v_n\}$ jest reprezentowany przez listy incydencji. To znaczy dla każdego wierzchołka v_i tworzymy osobną listę wierzchołków v_j takich, że $(v_i, v_j) \in E$. Wskaźniki do pierwszych elementów tych list przechowujemy w osobnej tablicy. Na rysunku może to wyglądać tak:



Na liście wskazywanej przez $A[i]$ znajdują się takie wierzchołki j, dla których $(i, j) \in E$.

Tak reprezentacja używa $O(|V| + |E|)$ pamięci, czyli proporcjonalnie wiele do rozmiaru grafu.

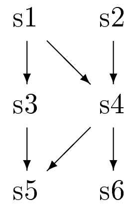
Przykłady. Grafy są wszędzie...

1. Sieć połączeń drogowych, komputerowych, telefonicznych (i wielu innych) na danym terenie to graf... (czasem zorientowany, czasem nie).

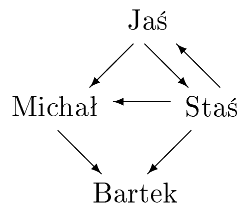
2. Schemat montażu dowolnego urządzenia to graf skierowany. Ogólniej, następstwo w procesach, które częściowo mogą być wykonywane równolegle jest grafem skierowanym. Na przykład, dla wykonania działań:

```
s1: a:=0;
s2: b:=1;
s3: c:=a+1;
s4: d:=b+a;
s5: e:=d+c;
s6: d:=d+1;
```

mamy taki graf następstwa:

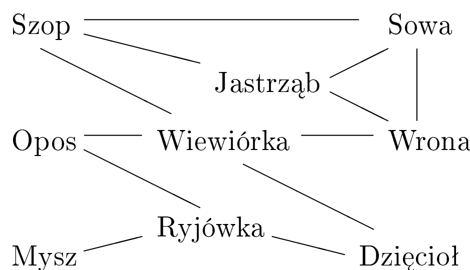


3. Relacje pomiędzy ludźmi można reprezentować przy pomocy grafów. Na przykład zależności służbowe w przedsiębiorstwach, czy graf 'wpływów' taki, że krawędź $a \to b$ jest w grafie o ile 'a może wpływać na zdanie b'.



4. Łańcuch żywieniowy zwierząt (Pies->Kot->Mysz etc.) jest grafem.

5. Współzawodnictwo zwierząt w ekosystemie o to samo pożywienie jest grafem modelującym zachodzące na siebie nisze ekologiczne. Krawędź $a - b$ jest w grafie o ile 'pożywienie a i b choć częściowo się pokrywają'.



6. Wyniki pojedynków drużyn w turnieju można reprezentować jako graf taki, że krawędź $a \to b$ jest w grafie o ile 'a wygrał z b'.

7.2 Składowe spójne grafu niezorientowanego

Podam teraz zastosowanie opisanej wcześniej struktury danych dla rodziny zbiorów rozłącznych. Poniższy algorytm oblicza składowe spójne grafu niezorientowanego.

Problem (znajdowanie składowych spójnych grafu niezorientowanego).

- Dane wejściowe: graf niezorientowany $G = (V, E)$.
- Wynik: składowe spójne grafu G.

```
procedure skladowe_spojne(G:graf);
 begin
  for v in V(G) do {tzn. 'dla kazdego wierzcholka grafu G'}
    make-set(v);
  {tworzymy zbiory jedno-elemenotowe dla kazdego wierzcholka grafu}
  for (u,v) in E do {tzn. 'dla kazdej krawedzi grafu G'}
   if find(u)<>find(v) then union(u,v);
 end;
```

W procedurze `skladowe_spojne` tworzymy zbiory jednoelementowe, których elementami są wszystkie wierzchołki grafu G. W trakcie działania procedury modyfikujemy te zbiory tak, by w każdym z nich były tylko takie wierzchołki pomiędzy którymi istnieje ścieżka. Procedura przegląda kolejne krawędzie grafu G. Jeśli napotka krawędź, która łączy wierzchołki z dwóch różnych zbiorów to je do siebie dodaje. Po zakończeniu procedury powstałe zbiory tworzą składowe spójne garfu G.

Bezpośrednią konsekwencją Faktu 6.3 jest

Fakt 7.1 *Dla grafu niezorientowanego* $G = (V, E)$*, procedura* `skladowe_spojne` *działa w czasie* $O(n + m\log(m))$*, gdzie* $|V| = m$ *i* $|E| = n$*.*

Po wykonaniu procedury `skladowe_spojne` sprawdzenie czy dwa wierzchołki są w tej samej składowej wygląda tak:

```
function ta_sama_skladowa(u,v):boolean;
 begin
   ta_sama_skladowa := (find(u)=find(v))
 end;
```

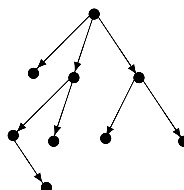
7.3 Przeszukiwanie grafu wszerz (BFS)

Drzewo jest to zorientowany graf acykliczny taki, że

1. o ile nie jest pusty zawiera dokładnie jeden wierzchołek, który nie jest końcem żadnej krawędzi; wierzchołek ten nazywa się *korzeniem*;

2. istnieje droga od korzenia do każdego wierzchołka grafu;

3. każdy wierzchołek, z wyjątkiem korzenia, jest końcem dokładnie jednej krawędzi.



Niech $D = (V, E)$ będzie drzewem. Jeśli $(v, w) \in E$ to v jest *poprzednikiem* w a w jest *następnikiem* v. Jeśli istnieje droga z v do w, to w jest *potomkiem* v (a v jest *przodkiem* w). Jeśli ponadto $v \neq w$, to w jest *potomkiem właściwym* v. *Liściem* nazywamy wierzchołek bez potomków właściwych. Podgraf pełny drzewa D zawierający wierzchołek v wraz z jego potomkami nazywamy *poddrzewem* drzewa D o korzeniu v. *Głębokością* wierzchołka v w drzewie nazywamy długość drogi od korzenia do v. *Wysokością* wierzchołka v w drzewie nazywamy długość najdłuższej drogi od v do jakiegoś liścia. *Wysokością drzewa* nazywamy wysokość korzenia tego drzewa.

Algorytm przeszukiwania grafu wszerz systematycznie bada krawędzie grafu G, by dotrzeć do każdego wierzchołka osiągalnego z s. Oblicza drzewo przeszukiwania wszerz z wierzchołka s i odległość od s do każdego wierzchołka osiągalnego z s.

Drzewem przeszukiwania wszerz grafu $G = (V, E)$ z wierzchołka $s \in V$ nazywamy podgraf $D = (V', E')$ grafu G będący drzewem o korzeniu s taki, że:

1. V' zawiera wszystkie wierzchołki osiągalne z s w G;

2. dla każdego wierzchołka $v \in V'$ ścieżka z s do v w drzewie D jest najkrótszą ścieżką z s do v w grafie E.

Problem (przeszukiwanie grafu wszerz).

- Dane wejściowe: graf $G = (V, E)$ i wierzchołek $s \in V$.

- Wynik: tablice d i π indeksowane zbiorem V, takie że

  1. $d[v]$ jest długością najkrótszej ścieżki z s do v lub ∞ (np. -1) jeśli nie ma takiej ścieżki;
  2. $P[v]$ jest poprzednikiem v na najkrótszej ścieżce z s do v, lub Nil gdy $v = s$ lub v nie jest osiągalny z s.

Podczas przeszukiwania algorytm używa pomocniczej tablicy `kolor`:

$$\texttt{kolor[v]} = \begin{cases} \texttt{bialy} & \text{gdy } v \text{ jest jeszcze nie odwiedzony;} \\ \texttt{szary} & \text{gdy } v \text{ został odwiedzony;} \\ \texttt{czarny} & \text{gdy } v \text{ i jego następniki zostali odwiedzeni.} \end{cases}$$

i `kolejki`:

1. `kolejka<=v` - wstaw v do kolejki na koniec;

2. `v<=kolejka` - wstaw pierwszy element kolejki na v;

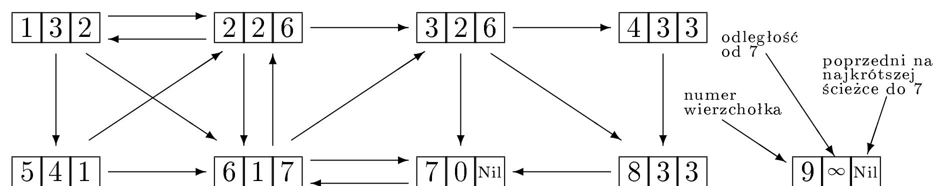
3. `usun_z_kolejki` - usuń pierwszy element z kolejki;

4. `kolejka<>0` - - test pustości kolejki.

Zakładamy, że graf G jest reprezentowany przez listy incydencji. `LI[v]` jest listą końców krawędzi o początku v.

```
procedure BFS(G,s);
  begin
    for v in V-{s} do begin        {inicjalizacja}
      P[v]:=Nil; d[v]:=-1; kolor[v]:=bialy
    end;
    d[s]:=0; kolor[s]:=szary; kolejka<=s;
    while kolejka<>0 do begin      {petla glowna}
      u<=kolejka;
      for v in LI[u] do {przegladamy nastepniki u}
        if kolor[v]=bialy then begin {wlasnie odkrylismy v}
          d[v]:=d[u]+1; P[v]:=u;
          kolor[v]:=szary; kolejka<=v;
        end;
        usun_z_kolejki; kolor[u]:=czarny; {opuszczamy wierzcholek u}
      end;
  end;
```

Przykład. Poniższy rysunek przedstawia efekt działania procedury BSF(G,s). Wierzchołkami grafu są liczby $1, \ldots 9$, krawędzie grafu są 'takie jak widać', wierzchołek s jest wierzchołkiem numer 7. W wierzchołkach zaznaczone są kolejno trzy wartości: numer wierzchołka v, odległość $d[v]$ od wierzchołka $s = 7$, przedostatni wierzchołek $P[v]$ na jednej z najkrótszych ścieżek z 7 do v:



Opis procedury BFS.

1. Inicjalizacja: nadaje wartości początkowe tablicom d i P.

2. Pętla główna (while): jest wykonywana dopóki są jeszcze wierzchołki odkryte, które mają nieodkryte następniki (tzn. szare).

3. Pętla wewnętrzna: przegląda wszystkich sąsiadów pierwszego wierzchołka z kolejki i odwiedza (maluje na szaro) te wierzchołki, które jeszcze nie były odwiedzone (białe).

Czas działania procedury BFS.

1. Inicjalizacja: $O(|V|)$.

2. Każdy wierzchołek jest raz (i nigdy więcej) malowany na biało, przy inicjalizacji. Przy wstawianiu do kolejki malujemy wierzchołek na szaro. Zatem każdy wierzchołek jest co najwyżej raz wstawiany do kolejki i raz z niej zdejmowany. Zatem łączna liczba operacji pętli zewnętrznej, bez pętli wewnętrznej jest $O(|V|)$.

3. Łączna liczba iteracji pętli wewnętrznej jest nie większa od łącznej długości list incydencji grafu G, czyli $|E|$ - gdy graf jest zorientowany i $2|E|$ - gdy graf jest niezorientowany. Zatem czas działania wszystkich iteracji pętli wewnętrznej jest $O(|E|)$.

Stąd otrzymujemy

Fakt 7.2 *Czas działania procedury BFS dla grafu $G = (V, E)$ jest równy $O(|V| + |E|)$.*

Poprawność algorytmu BFS.

Ustalmy graf $G = (V, E)$ i wierzchołek $s \in V$. Niech $\delta(u, v)$ będzie długością najkrótszej ścieżki z u do v w G lub ∞ gdy nie ma takiej ścieżki.

Lemat 7.3 *1. Jeśli $(u, v) \in E$, to $\delta(s, v) \leq \delta(s, u) + 1$.*

2. Po wykonaniu $BFS(G, s)$, $\delta(s, v) \leq d[v]$, dla $v \in V$.

Dowód. Ad. 1. Oczywiste.

Ad 2. Pokażemy przez indukcję ze względu na liczbę wstawień wierzchołków do kolejki, że

$$\delta(s, v) \leq d[v] \qquad \text{dla } v \in V \tag{11}$$

Założenie jest prawdziwe po wstawieniu pierwszego wierzchołka (s) do kolejki. Wtedy mamy

$$\delta(s, s) = d[s], \qquad \delta(s, v) \leq \infty = d[v], \qquad \text{dla } v \in V \setminus \{s\}$$

Krok indukcyjny. Rozważmy teraz biały wierzchołek v odkryty podczas przeszukiwania listy incydencji wierzchołka u. Z założenia indukcyjnego mamy $\delta[s, u] \leq d[u]$. Po odkryciu v wykonujemy podstawienie $d[v] := d[u] + 1$. Wtedy używając punktu 1. mamy:

$$d[v] = d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v).$$

W tym momencie malujemy v na szaro i wobec tego wartość $d[v]$ się już nie zmieni. Stąd teza. Q.E.D.

Poniższy lemat opisuje działanie kolejki w trakcie wykonywania się procedury BFS.

Lemat 7.4 *Przypuśćmy, że w czasie wykonywania się procedury $BFS(G, s)$ kolejka zawiera wierzchołki $< v_1, \ldots, v_r >$ (v_1 pierwszy v_r ostatni). Wtedy*

$$d[v_r] \leq d[v_1] + 1$$

oraz

$$d[v_i] \leq d[v_{i+1}]$$

dla $i = 1, \ldots, r$.

Dowód. Lemat udowodnimy przez indukcję ze względu na liczbę wstawień do i usunięć z kolejki.

Po wstawieniu s do kolejki teza lematu jest prawdziwa.

Jeśli teza lematu jest prawdziwa przed usunięciem wierzchołka z kolejki to tym bardziej jest prawdziwa po usunięci wierzchołka z kolejki.

Jeśli kolejna operacja jest dodaniem wierzchołka v_{r+1} do kolejki $< v_1, \ldots, v_r >$ to dzieje się to w czasie przeszukiwania listy sąsiadów v_1. Zatem podstawiamy $d[v_{r+1}] := d[v_1] + 1$. Wtedy

$$d[v_i] \leq d[v_1] + 1 = d[v_{r+1}]$$

dla $i = 1, \ldots, r$ i oczywiście $d[v_{r+1}] \leq d[v_1] + 1$. Zatem teza jest prawdziwa w dowolnym momencie wykonywania procedury BFS. Q.E.D.

Twierdzenie 7.5 *Procedura $BFS(G, s)$ odwiedza każdy wierzchołek $v \in V$ osiągalny z s. Po jej wykonaniu $d[v] = \delta(s, v)$, dla $v \in V$.*

Ponadto dla dowolnego $v \neq s$ i osiągalnego z s ostatnią krawędzią na jednej z najkrótszych ścieżek z s do v jest krawędź $(P(v), v)$.

Dowód. Niech $v \in V$ będzie nieosiągalny z s. Wtedy z Lematu 7.3(2) $d[v] \geq \delta(s, v) = \infty$. Zatem v nie został odkryty, bo w przeciwnym razie miałby wartość skończoną.

Niech $V_k = \{v \in V : \delta(s, v) = k\}$. W szczególności $\bigcup_{k \in N} V_k$ jest zbiorem wszystkich wierzchołków osiągalnych z s. Dla wierzchołków osiągalnych z s pokażemy tezę twierdzenia przez indukcję po $k \in N$.

Założenie indukcyjne: w czasie wykonywania procedury $BFS(G, s)$, dla dowolnego wierzchołka $v \in V_k$, zdarzy się dokładnie raz taka sytuacja, że v zostanie odkryty oraz:

1. v zostanie pomalowany na szaro;
2. $d[v]$ przyjmie wartość k;
3. jeśli $v \neq s$ to $P[v]$ przyjmie wartość w V_{k-1};
4. v zostanie wstawiony do kolejki.

Dla $k = 0$ mamy $V_0 = \{s\}$. Ten 'jedyny raz' dla s zdarzy się przy inicjalizacji.

Niech $k > 0$, założenie indukcyjne będzie prawdziwe dla wierzchołków z V_{k-1} oraz $v \in V_k$.

Kilka obserwacji:

1. kolejka jest niepusta aż do zakończenia wykonywania algorytmu;
2. po wstawieniu wierzchołka u do kolejki $d[u]$ i $P[u]$ nie zmieniają się;
3. jeśli wierzchołki $v_1, \ldots, v_r$ są kolejno wstawiane do kolejki to $d[v_1] \leq \ldots \leq d[v_r]$.

Ponieważ, z Lematu 7.3, $d[v] \geq k$, to z 3. i założenia indukcyjnego mamy, że o ile v zostanie odkryty, to zostanie odkryty po odkryciu i wstawieniu do kolejki wszystkich wierzchołków z V_{k-1}. Ponieważ $\delta(s, v) = k$, to istnieje $u \in V_{k-1}$ taki, że $(u, v) \in E$. Niech u będzie pierwszym takim wierzchołkiem wstawionym do kolejki. Zatem v zostanie odkryte podczas przeszukiwania sąsiadów u. Wtedy

1. v zostanie pomalowany na szaro;

2. $d[v] := d[u] + 1 = k$;

3. $P[v] := u$;

4. v zostanie wstawiony do kolejki.

Ponieważ v był dowolny to otrzymujemy tezę indukcyjną.

By zobaczyć, że $(P[v], v)$ jest ostatnią krawędzią na najkrótszej ścieżce z s do v wystarczy zauważyć, że $P[v] \in V_{k-1}$.

Q.E.D.

Mając tablice d i P obliczone w procedurze $BFS(G, s)$ drzewo przeszukiwania wszerz $D = (V_P, E_P)$ grafu G z wierzchołka s otrzymujemy w następujący sposób:

$$V_P = \{v \in V : P[v] \neq Nil\} \cup \{s\}$$

oraz

$$E_P = \{(P[v], v) : v \in V_P \setminus \{s\}\}$$

Poniższa procedura rekurencyjna drukuje najkrótszą ścieżkę z s do dowolnego wierzchołka G:

```
procedure sciezka (G,s,v);
 begin
  if v=s then write(s)
  else if P[v]=Nil then write('Nie ma sciezki.')
       else begin
        sciezka(G,s,P[v]);
        write(v)
       end;
 end;
```

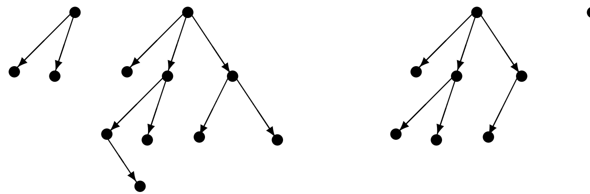
7.4 Przeszukiwanie grafu w głąb (DFS)

Przeszukiwanie grafu w głąb (DFS), jest drugim po BFS, podstawowym sposobem przeszukiwania grafu, i jak zobaczymy później, często bardzo użytecznym przy konstruowaniu innych algorytmów. Idea przeszukiwania w głąb polega na tym by iść jak najgłębiej jest to tylko możliwe. Główna techniczną różnicą algorytmu DFS w stosunku do BFS jest to, że w DFS wkładamy odwiedzane wierzchołki na stos a nie jak w BFS, do kolejki.

Przeszukując graf w głąb odwiedzamy wszystkie wierzchołki a nie tylko te, które są osiągalne z ustalonego wierzchołka. Dlatego w rezultacie przeszukiwania w głąb otrzymujemy las (a nie drzewo) przeszukiwania w głąb. Ponadto przeszukując graf w głąb zapamiętujemy moment, w którym wierzchołki odkrywamy (malujemy na szaro) i opuszczamy (malujemy na czarno).

Las jest to zorientowany graf acykliczny taki, że

1. istnieje co najmniej jeden wierzchołek, który nie jest końcem żadnej krawędzi; wierzchołki takie nazywa się *korzeniami*;
2. istnieje droga do każdego wierzchołka od dokładnie jednego korzenia grafu;
3. każdy wierzchołek, z wyjątkiem korzeni, jest końcem dokładnie jednej krawędzi.



Problem (przeszukiwanie grafu w głąb).

- Dane wejściowe: graf $G = (V, E)$.
- Wynik: tablice d, f i P indeksowane zbiorem V, takie że
  1. $P[v] = u$ oznacza, że v został odkryty w czasie przeszukiwania listy incydencji wierzchołka u;
  2. $d[v]$ jest czasem odkrycia wierzchołka v;
  3. $f[v]$ jest czasem opuszczenia wierzchołka v.

Po wykonaniu procedury DFS dla grafu $G = (V, E)$ krawędzie *lasu* $G_P = (V, E_P)$ *przeszukiwania w głąb* grafu G definiujemy następująco:

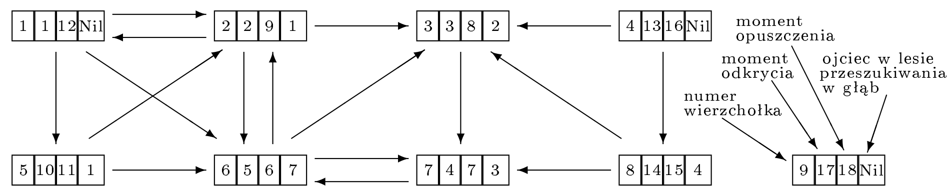
$$E_P = \{(P[u], u) : P[u] \neq Nil\}$$

Procedura DFS podobnie jak BFS używa pomocniczej tablicy `kolor` do barwnego zaznaczania statusu wierzchołków.

```
procedure DFS(G:graf);
 begin
  for v in V do begin {inicjalizacja}
   kolor[v]:=bialy; P[v]:=Nil
  end;
```

```
  time:=0;
  for v in V do {budowa kolejnych drzew lasu przeszukiwania w glab}
   if kolor[v]=bialy then DFS-odwiedz(v)
 end;


procedure DFS-odwiedz(u:wierzcholek);
 begin
  kolor[u]:=szary; time:=time+1; d[u]:=time; {wlasnie odkrylismy u}
  for v in LI(u) do {przegladanie sasiadow u wraz z ich potomkami}
   if kolor[v] = bialy then begin
    P[v]:=u;
    DFS-odwiedz(v);
   end;
  kolor[u]:=czarny; time:=time+1; f[u]:=time
 end;
```

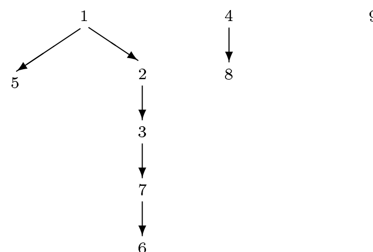
Przykład. Poniższy przykład ilustruje sposób działania procedury `DFS`.



W wierzchołkach zaznaczone są kolejno wartości:

1. numer wierzchołka x;
2. moment odkrycia wierzchołka $d[x]$;
3. moment opuszczenia wierzchołka $f[x]$;
4. ojciec wierzchołka w lesie przeszukiwania w głąb $P[x]$.

Las przeszukiwania w głąb powyższego grafu wygląda tak:



Opis procedury `DFS`.

1. Inicjalizacja: nadaje wartości początkowe tablicom *kolor* i P.
2. Pętla `for` procedury `DFS` przegląda wszystkie wierzchołki grafu G; dla wierzchołków białych woła procedurę `DFS-odwiedz`; każde takie wołanie tworzy jedno drzewo lasu przeszukiwania w głab grafu G;

3. procedura `DFS-odwiedz(u)` wraz z rekurencyjnymi odwołaniami odwiedza wierzchołek u, wszystkich jego sąsiadów, i wszystkich sąsiadów sąsiadów, etc. którzy jeszcze nie zostali odkryci; na koniec procedura opuszcza wierzchołek u;

4. Pętla `for` procedury `DFS-odwiedz(u)` przegląda wszystkich sąsiadów wierzchołka u, dla tych wierzchołków, które jeszcze nie były odwiedzone (białe) woła rekurencyjnie siebie samą.

Czas działania procedury DFS.

1. Procedura `DFS`, nie liczac wołań procedury `DFS-odwiedz`, wykonuje $O(|V|)$ operacji.

2. Procedura `DFS-odwiedz` jest wołana dokładnie jeden raz dla każdego wierzchołka; w momencie wołania `DFS-odwiedz(v)` wierzchołek v musi być biały, jest malowany na szaro i nigdy potem nie jest malowany na biało; w czasie wołania `DFS-odwiedz(v)` pętla `for` jest wykonywana $|LI(v)|$ razy; ponieważ $\sum_{v\in V} |LI(v)| = O(|E|)$, to łączna liczba wykonań pętli `for` jest $O(|E|)$.

Zatem

Fakt 7.6 *Czas działania procedury DFS dla grafu $G = (V, E)$ jest równy $O(|V| + |E|)$.*

Teraz pokażemy kilka własności procedury `DFS`.

Fakt 7.7 (Twierdzenie o nawiasowaniu) *Po wykonaniu procedury DFS dla grafu $G = (V, E)$, dla dowolnych dwóch wierzchołków $u, v \in V$ zachodzi dokładnie jeden z warunków:*

1. *przedziały $< d[v], f[v] >$ i $< d[u], f[u] >$ są rozłączne;*

2. *przedział $< d[v], f[v] >$ jest zawarty w przedziale $< d[u], f[u] >$ oraz v jest potomkiem u w lesie przeszukiwania w głąb;*

3. *przedział $< d[u], f[u] >$ jest zawarty w przedziale $< d[v], f[v] >$ oraz u jest potomkiem v w lesie przeszukiwania w głąb;*

Dowód. Niech $u, v \in V$, $u \neq v$. Możemy założyć, że $d[u] < d[v]$ (przypadek $d[u] > d[v]$ jest analogiczny).

Rozpatrzymy dwa przypadki:

1. $d[v] < f[u]$;

2. $f[u] < d[v]$.

Ad 1. Gdy $d[v] < f[u]$, to w momencie $d[v]$ wierzchołek u jest szary, tzn. v zostaje odkryty gdy przeszukujemy potomków wierzchołka u. Zatem v jest potomkiem u.

Ponadto po momencie $d[v]$ krawędzie wychodzące z v zostaną przeszukane i v zostanie pomalowane na czarno i opuszczony, zanim zostanie opuszczony u. Stąd $f[v] < f[u]$ oraz mamy

$$d[u] < d[v] < f[v] < f[u].$$

Ad 2. Gdy $f[u] < d[v]$, to przedziały $< d[v], f[v] >$ i $< d[u], f[u] >$ są rozłączne. Ponadto żaden z wierzchołków nie zostanie odkryty podczas przeszukiwania potomków drugiego. Q.E.D.

Wniosek 7.8 *Wierzchołek v jest potomkiem wierzchołka u w lesie przeszukiwania w głąb grafu G wtedy i tylko wtedy gdy $d[u] < d[v] < f[v] < f[u]$.*

Fakt 7.9 (Twierdzenie o białej ścieżce) *W lesie przeszukiwania w głąb grafu $G = (V, E)$, v jest potomkiem u wtedy i tylko wtedy gdy w momencie $d[u]$ (tzn. momencie odkrycia u) istnieje ścieżka z u do v przechodząca wyłącznie po białych wierzchołkach.*

Dowód. $\Rightarrow$: Załóżmy, że v jest potomkiem u w lesie przeszukiwania w głąb grafu G. Niech w będzie wierzchołkiem na ścieżce z u do v w lesie przeszukiwania w głąb $G_P(V, E_P)$. Z Wniosku 7.8, $d[u] < d[w]$. Zatem w jest biały w momencie $d[u]$, odkrycia u.

$\Leftarrow$: Przypuśćmy, że w momencie $d[u]$ odkrycia u istnieje biała ścieżka z u do v biegnąca po białych wierzchołkach ale v nie jest potomkiem u w lesie przeszukiwania w głąb grafu G. Możemy przyjąć, że wszystkie poprzednie wierzchołki na tej ścieżce są potomkami u. W przeciwnym przypadku możemy za v przyjąć pierwszy wierzchołek na tej ścieżce, który nie jest potomkiem u. Niech w będzie poprzednim wierzchołkiem przed v na tej ścieżce. Wtedy, z Wniosku 7.8, $f[w] \leq f[u]$ (w może być równy u). Ponieważ v jest biały w momencie $d[u]$ i $(w, v) \in E$, to v musi być odkryty po odkryciu u ale przed opuszczeniem w. Zatem

$$d[u] \leq d[v] < f[w] \leq f[u].$$

Z Faktu 7.7 mamy, że przedział $< d[v], f[v] >$ jest zawarty w $< d[u], f[u] >$. A zatem, z Wniosku 7.8, v jest jednak potomkiem u. Q.E.D.

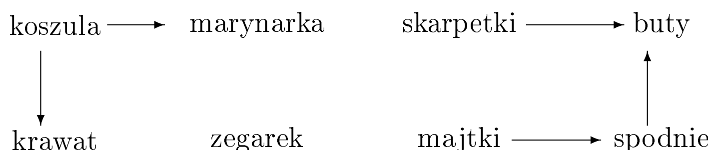
7.5 Sortowanie topologiczne

Problem sortowania topologicznego polega uporządkowaniu liniowym wierzchołków zorientowanego grafu acyklicznego $G = (V, E)$ tak, by wierzchołek u był przed wierzchołkiem v, o ile $(u, v) \in E$.

Problem (sortowanie topologiczne).

- Dane wejściowe: zorientowany graf acykliczny $G = (V, E)$.
- Wynik: lista zawierająca wszystkie wierzchołki z V taka, że jeżeli $(u, v) \in E$ to u jest przed v na tej liście.

Przykład. Wierzchołkami poniższego grafu acyklicznego są części garderoby. Krawędź $x \longrightarrow y$ oznacza, że ubierając się x należy włożyć przed y. Ponieważ na raz możemy włożyć tylko jedną cześć garderoby to by się ubrać musimy (mniej lub bardziej świadomie) zastosować algorytm sortowania topologicznego.



Opis algorytmu sortowania topologicznego:

1. przeszukujemy graf G w głąb procedurą `DFS`;
2. w momencie $f[u]$ wstawiamy u na początek listy;
3. lista, którą otrzymujemy po zakończeniu procedury `DFS` jest żądaną listą.

Niech (V, F) będzie lasem przeszukiwania w głąb zorientowanego grafu $G = (V, E)$. Krawędź $(u, v) \in E$ nazywamy *krawędzią powrotną* względem lasu (V, F) jeśli v jest przodkiem u w (V, F).

Fakt 7.10 *Graf $G = (V, E)$ jest acykliczny wtedy i tylko wtedy gdy nie ma krawędzi powrotnych względem lasu przeszukiwania w głąb (V, F) obliczonego procedurą $DFS(G)$.*

Dowód. $\Rightarrow$. Jeśli (u, v) jest krawędzią powrotną to mamy ścieżkę z v do u w lesie (V, F), która wraz z krawędzią (u, v) tworzy cykl w G. Zatem G nie jest acykliczny.

$\Leftarrow$. Przypuśćmy, że w grafie G jest cykl C. Niech v będzie pierwszym wierzchołkiem z C odkrytym podczas przeszukiwania w głąb grafu G, (u, v) krawędzią cyklu C. Zatem w momencie $d[v]$ istnieje biała ścieżka z v do u (na przykład ta biegnąca po wierzchołkach cyklu C). Z twierdzenia o białej ścieżce u jest potomkiem v w (V, F). Zatem krawędź (u, v) jest powrotna. Q.E.D.

Fakt 7.11 *Algorytm sortowania topologicznego jest poprawny, tzn. zastosowany do grafu acyklicznego $G = (V, E)$ porządkuje wierzchołki grafu zgodnie z krawędziami.*

Dowód. Niech G będzie grafem acyklicznym. Wystarczy pokazać, że jeśli $(u, v) \in E$ to po wykonaniu algorytmu $DFS(G)$ mamy $f(u) > f(v)$. Rozważmy, kolor wierzchołka v w momencie $d[u]$.

Jeśli v jest szary to u jest potomkiem v w lesie (V, F). Czyli (u, v) jest krawędzią powrotną i G nie jest acykliczny wbrew założeniu.

Jeśli v jest czarny to $f[v] < d[u] < f[u]$.

Jeśli v jest biały to zanim u zostanie opuszczony v zostanie odkryty, jego potomkowie przeszukani i v zostanie opuszczony. Zatem znowu $f[v] < f[u]$.

Q.E.D.

7.6 Silnie spójne składowe grafu

Algorytm znajdujący silnie spójne składowe zorientowanego grafu G jest kolejnym zastosowaniem algorytmu przeszukiwania grafu w głąb (DFS).

Niech $G = (V, E)$ będzie grafem zorientowanym. Dla $u, v \in V$, $u \mapsto v$ oznacza, że istnieje ścieżka z u do v w G. Definiujemy relację binarną $\sim$ na zbiorze V tak, że

$$u \sim v \quad \text{wiw gdy} \quad u \mapsto v \text{ i } v \mapsto u$$

dla $u, v \in V$. Oczywiście relacja $\sim$ jest przechodnia. Klasy abstrakcji relacji $\sim$ nazywamy *silnie spójnymi składowymi* grafu G. Graf $G^T = (V, E^T)$ nazywamy grafem transponowanym grafu $G = (V, E)$, jeżeli

$$(u, v) \in E^T \quad \text{iff} \quad (v, u) \in E$$

Uwaga. Składowe silnie spójne grafów G i G^T są równe.

Problem (znajdowanie silnie spójnych składowych).

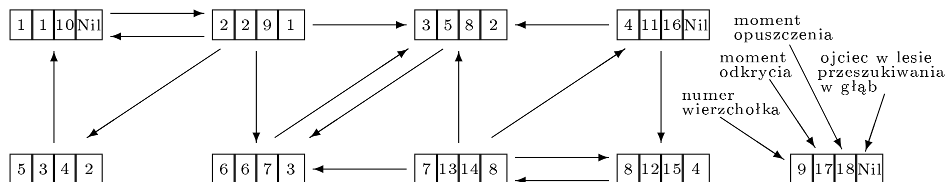
- Dane wejściowe: zorientowany graf $G = (V, E)$.
- Wynik: zbiór silnie spójnych składowych grafu G.

Opis algorytmu znajdowania silnie spójnych składowych:

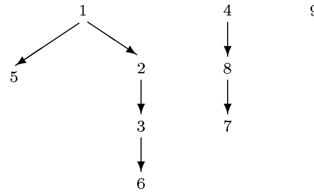
1. przeszukujemy graf G w głąb procedurą `DFS` w celu obliczenia tablicy f;
2. obliczamy graf transponowany G^T;
3. przeszukujemy graf G^T w głąb procedurą `DFS` z tym, że tym razem w w pętli głównej procedury `DFS` przeszukujemy wierzchołki w porządku malejących wartości poprzednio obliczonego f;
4. Wypisujemy wierzchołki drzew lasu przeszukiwania w głąb grafu G^T jako kolejne silnie spójne składowe grafu G.

Przykład. Ten algorytm ma, na pierwszy rzut oka, niewiele wspólnego z problemem, który rozwiązuje. Zanim uzasadnię, że algorytm powyższy poprawnie rozwiązuje problem znajdowania silnie spójnych składowych warto prześledzić przykład, który nieco to uprawdopodabnia. W przykładzie kolejno przeszukujemy grafy G i G^T procedurą DFS z tym, że przeszukując graf G^T, tak jak jest to powiedziane w algorytmie, w procedurze DFS przeszukujemy wierzchołki w kolejności malejących wartości f. Łatwo też sprawdzić, że wierzchołki drzew lasu przeszukiwania w głąb grafu G^T tworzą silnie spójne składowe grafu G (i G^T).

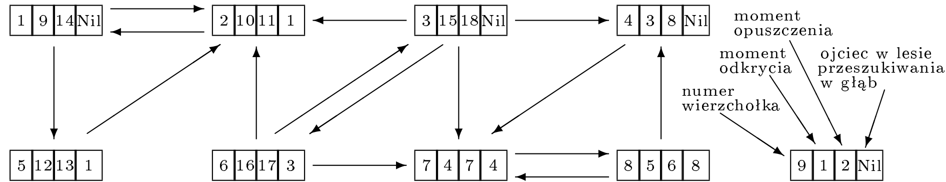
Przeszukanie w głąb grafu G procedurą $DFS(G)$:



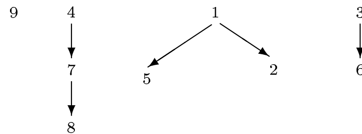
Las przeszukiwania w głąb grafu G:



Przeszukanie w głąb grafu G^T procedurą $DFS(G^T)$:



Las przeszukiwania w głąb grafu G^T:



Poniżej pokażemy, że ten algorytm poprawnie znajduje silnie spójne składowe grafu zorientowanego. Ustalmy do końca tego podrozdziału zorientowany graf $G = (V, E)$.

Lemat 7.12 *Jeżeli dwa wierzchołki należą do tej samej silnie spójnej składowej S grafu G, to każda ścieżka pomiędzy nimi jest całkowicie zawarta w S.*

Dowód. Niech S będzie silnie spójną składową grafu G, $u, v \in S$ oraz $w \in V$ będzie wierzchołkiem na ścieżce z u do v. Wtedy mamy $u \mapsto w$ i $w \mapsto v$ oraz ponieważ $u \sim v$ to $u \mapsto v$ i $v \mapsto u$. Ponieważ relacja $\mapsto$ jest przechodnia to mamy $u \mapsto w$ i $w \mapsto u$ czyli $w \sim u$. A stąd u i w leżą w tej samej silnie spójnej składowej S. Pokazaliśmy zatem, że każdy wierzchołek w na dowolnej ścieżce z u do v leży w silnie spójnej składowej S. Zatem każda ścieżka z u do v biegnie tylko po wierzchołkach z S. Q.E.D.

Lemat 7.13 *Procedura* `DFS` *umieszcza wszystkie wierzchołki tej samej silnie spójnej składowej grafu w jednym drzewie lasu przeszukiwania w głąb.*

Dowód. Niech S będzie silnie spójną składową grafu G, r będzie pierwszym wierzchołkiem z S odkrytym podczas przeszukiwania grafu G, oraz v dowolnym wierzchołkiem z S. Zatem w momencie $d[r]$ odkrycia r wszystkie wierzchołki z S są białe. Ponieważ $r, v \in S$ to istnieje ścieżka z r do v. Z Lematu 7.12 ścieżka ta biegnie tylko po wierzchołkach z S, zatem tylko po białych wierzchołkach. Z Faktu 7.9 (Twierdzenie o białej ścieżce) v jest potomkiem r w drzewie lasu przeszukiwania w głąb grafu G. Z dowolności v, wszystkie wierzchołki z S są potomkami r, zatem składowa S jest umieszczona w jednym drzewie lasu przeszukiwania w głąb grafu G. Q.E.D.

Notacja. Niech f będzie tablicą obliczoną w czasie wykonania procedury `DFS` dla grafu G. Definiujemy funkcję $\varphi : V \to V$ taką, że $\varphi(u) = w$ jeśli z u można dojść do

w i $f[w]$ jest maksymalnym momentem opuszczenia wierzchołka do którego można dojść z u, tzn.

$$u \mapsto w \text{ oraz } f(w) = max\{f(v) : u \mapsto v\}.$$

$\varphi(u)$ nazywamy *praojcem* u.

Lemat 7.14 *Niech* $u, v \in V$.

1. $f(u) \leq f(\varphi(u))$;
2. *jeśli* $u \mapsto v$ *to* $f(\varphi(u)) \geq f(\varphi(v))$;
3. $\varphi(u) = \varphi\varphi(u)$.

Dowód. Ad 1. Ponieważ $u \mapsto u$ oraz $\varphi(u)$ jest takim wierzchołkiem w, że $u \mapsto w$ i $f(w)$ jest maksymalne to $f(u) \leq f(\varphi(u))$.

Ad 2. Mamy

$$f(\varphi(u)) = max\{f(w) : u \mapsto w\}$$

oraz

$$f(\varphi(v)) = max\{f(w) : v \mapsto w\}.$$

Ponieważ $u \mapsto v$ to

$$\{f(w) : u \mapsto w\} \supseteq \{f(w) : v \mapsto w\}.$$

Zatem

$$f(\varphi(u)) = max\{f(w) : u \mapsto w\} \geq \{f(w) : v \mapsto w\} = f(\varphi(v)).$$

Ad 3. Z 1. mamy, że

$$f(\varphi(u)) \leq f(\varphi\varphi(u)).$$

Ponieważ $u \mapsto \varphi(u)$ to z 2. mamy, że

$$f(\varphi(u)) \geq f(\varphi\varphi(u)).$$

Zatem

$$f(\varphi(u)) = f(\varphi\varphi(u)).$$

Ponieważ funkcja f jest różnowartościowa (jako że różne wierzchołki opuszczamy w różnych momentach) to mamy też

$$\varphi(u) = \varphi\varphi(u).$$

Q.E.D.

Lemat 7.15 *Praojciec* $\varphi(u)$ *wierzchołka* $u \in V$ *jest przodkiem* u *w lesie przeszukiwania w głąb grafu* G.

Dowód. Jeśli $u = \varphi(u)$ to Lemat jest prawdziwy. Niech $u \neq \varphi(u)$. Rozważmy kolor $\varphi(u)$ w momencie $d(u)$. Jeśli $\varphi(u)$ jest szary to (świetnie bo) u został odkryty w czasie przeszukiwania potomków $\varphi(u)$. Zatem $\varphi(u)$ jest przodkiem u w lesie przeszukiwania w głąb grafu G. Pokażemy, że $\varphi(u)$ nie może być ani czarny ani biały .

Jeśli $\varphi(u)$ jest czarny to

$$f(\varphi(u)) < d(u) < f(u)$$

i mamy sprzeczność z Lematem 7.14.1.

Pokażemy teraz, że $\varphi(u)$ nie może być biały. Przypuśćmy przeciwnie, że $\varphi(u)$ jest biały w momencie $d(u)$. Mamy dwa przypadki:

1. w momencie $d(u)$ ścieżka z u do $\varphi(u)$ biegnie tylko po białych wierzchołkach;
2. w momencie $d(u)$ na ścieżce z u do $\varphi(u)$ są niebiałe wierzchołki.

W przypadku 1, z Twierdzenia o białej ścieżce (Fakt 7.9) mamy, że $\varphi(u)$ będzie potomkiem u w lesie przeszukiwania w głąb grafu G. Zatem

$$d(u) < d(\varphi(u)) < f(\varphi(u)) < f(u)$$

i otrzymujemy sprzeczność z Lematem 7.14.1.

W przypadku 2, niech t będzie ostatnim niebiałym wierzchołkiem na ścieżce z u do $\varphi(u)$:

$$u \ \ldots \ t \ \underbrace{\ldots \ \varphi(u)}_{\text{białe}}$$

Wierzchołek t jest szary w momencie $d(u)$ ponieważ nie ma krawędzi od czarnych do białych wierzchołków. Zatem w momencie $d(t) < d(u)$ istniała biała ścieżka z t do $\varphi(u)$. Zatem $\varphi(u)$ jest potomkiem t w lesie przeszukiwania w głąb grafu G. A stąd $f(\varphi(u)) < f(t)$. Ponieważ $u \mapsto t$ to otrzymujemy sprzeczność z definicją $\varphi(u)$.

Zatem $\varphi(u)$ nie może być biały w momencie $d(u)$.

Q.E.D.

Wniosek 7.16 *Dla $u \in V$, wierzchołki u i $\varphi(u)$ leżą w tej samej silnie spójnej składowej grafu G.*

Dowód. Z definicji $\varphi(u)$ mamy $u \mapsto \varphi(u)$. Lematu 7.15 $\varphi(u) \mapsto u$. Zatem $u \sim \varphi(u)$.

Q.E.D.

Lemat 7.17 *Niech $u, v \in V$. Wtedy u i v leżą w tej samej silnie spójnej składowej grafu G wtedy i tylko wtedy gdy $\varphi(u) = \varphi(v)$.*

Dowód. $\Leftarrow$ wynika bezpośrednio z Wniosku 7.16.

$\Rightarrow$. Jeśli u i v leżą w tej samej silnie spójnej składowej to $u \mapsto v$ i $v \mapsto u$ oraz Lematu 7.14.2 mamy, że $f(\varphi(u)) = f(\varphi(v))$. Zatem z różnowartościowości f mamy, że $\varphi(u) = \varphi(v)$.

Q.E.D.

Twierdzenie 7.18 *Powyższy algorytm poprawnie znajduje silnie spójnej składowe grafu G.*

Dowód. Dowód przeprowadzimy przez indukcję ze względu dla liczbę drzew w lesie przeszukiwania w głąb grafu G^T.

Pokażemy, że jeśli do tej pory skonstruowane drzewa lasu przeszukiwania w głąb grafu G^T stanowią silnie spójne składowe grafu G^T (i G) to wierzchołki kolejnego drzewa skonstruowanego przez procedurę $DFS(G^T)$ stanowią kolejną silnie spójna składową.

Krok bazowy jest oczywisty, bo na początku nie ma jeszcze żadnych drzew.

Krok indukcyjny. Niech r będzie korzeniem kolejnego drzewa T skonstruowanego przez $DFS(G^T)$.

Jeśli byśmy mieli, że $r \neq \varphi(r)$ to, ponieważ przeszukujemy wierzchołki względem malejących wartości funkcji f, $f(r) < f(\varphi(r))$. Zatem $\varphi(r)$ zostało już umieszczone

w jakimś drzewie, a r jeszcze nie. Ale to jest niemożliwe z Lematu 7.13 i Wniosku 7.16. Zatem $r = \varphi(r)$.

Niech

$$S(r) = \{w \in V : \varphi(w) = r\}$$

będzie silnie spójną składową wierzchołka r. Pokażemy, że v jest wierzchołkiem T wtedy i tylko wtedy gdy $v \in S(r)$.

$\Leftarrow$. Ponieważ wierzchołki v i r leżą w tej samej silnie spójnej składowej to $DFS(G^T)$ umieszcza je na tym samym drzewie. Ponieważ r jest korzeniem T, to v jest wierzchołkiem T.

$\Rightarrow$. Pokażemy, że jeśli $w \in V$ oraz

1. $f(\varphi(w)) > f(r)$ lub
2. $f(\varphi(w)) < f(r)$

to w nie jest umieszczone w drzewie T.

Ad 1. Z założenia indukcyjnego gdy r został wybrany na korzeń T, to w został już wcześniej odkryty i wstawiony do drzewa o korzeniu $\varphi(w)$.

Ad 2. Jeśli w zostanie wstawiony do drzewa T, to $r \mapsto w$ w grafie G^T. Zatem $w \mapsto r$ w grafie G. A to jest sprzeczne z definicją $\varphi(w)$, ponieważ $w \mapsto r$ i $f(\varphi(w)) < f(r)$.

Q.E.D.

7.7 Minimalne drzewo rozpinające

Niech $G = (V, E)$ będzie grafem niezorientowanym. Przez *drzewo* w tym podrozdziale będziemy rozumieli spójny acykliczny graf niezorientowany. Funkcję $W : E \to R_+$ ze zbioru krawędzi grafu G do zbioru dodatnich liczb rzeczywistych nazywamy *funkcją wag*. Graf G, na którym jest określona funkcja wag nazywamy *grafem z wagami*. Dla $(u, v) \in E$ waga $W(u, v)$ może oznaczać koszt połączenia u z v, lub odległość z u do v, i tym podobne wartości.

Problem znajdowania minimalnego drzewa rozpinającego polega na znalezieniu zbioru krawędzi, które łączą wszystkie wierzchołki w 'najtańszy' sposób.

Problem (znajdowanie minimalnego drzewa rozpinającego).

- Dane wejściowe: spójny graf niezorientowany $G = (V, E)$ z funkcja wag W.
- Wynik: spójny (i acykliczny) graf niezorientowany $G' = (V, T)$ taki, że $T \subseteq E$ oraz waga zbioru T
$$W(T) = \sum_{e \in T} W(e)$$
jest minimalna.

Poniższy algorytm używa struktury danych dla rodziny zbiorów rozłącznych. Jest to algorytm zachłanny. Pochodzi on od Kruskala.

Opis algorytmu Kruskala znajdowania minimalnego drzewa rozpinającego:

1. $T := \emptyset$;
2. dla każdego wierzchołka z V tworzymy zbiór $\{v\}$;
3. sortujemy krawędzie grafu G względem niemalejących wag;
4. przeglądamy krawędzie $(u, v) \in E$ w porządku niemalejących wag i jeżeli $find(u) \neq find(v)$ to wykonujemy
   (a) $T := T \cup \{(u, v)\}$;
   (b) $union(u, v)$ (dodajemy zbiory do których należą wierzchołki u i v).

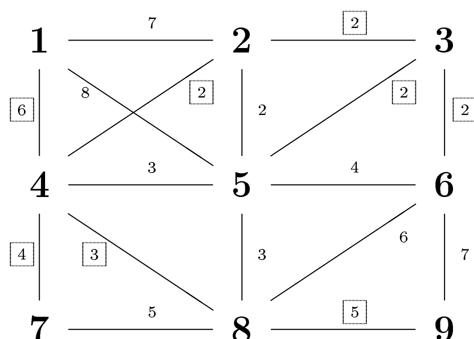
Czas działania algorytmu Kruskala.

1. Na początku inicjalizujemy zbiór T jako zbiór pusty i wykonujemy $|V|$ operacji `make-set`;
2. Następnie sortujemy zbiór krawędzi E w czasie $O(|E| \log(|E|)$;
3. Na koniec dla każdej krawędzi z E wykonujemy dwie operacje `find` i być może po jednej operacji `union` i przypisania;
4. W sumie algorytm wykonuje $O(|E| + |V|)$ operacji `make-set`, `union`, `find`, wśród których jest $|V|$ operacji `make-set`; zatem, z Faktu 6.3 wynika, że te operacje są wykonane w czasie $O(|E| + |V| \log(|V|)$.

Ponieważ graf jest spójny więc $|E| \geq |V| - 1$. Stąd otrzymujemy

Fakt 7.19 *Czas działania algorytmu Kruskala dla spójnego niezorientowanego grafu z wagami $G = (V, E)$ jest równy $O(|E| \log(|E|)$.*

Przykład. W grafie z wagami



krawędzie wybrane przez algorytm mają wagi wzięte w ramkę. Zauważmy, że minimalne drzewo rozpinające zależy od porządku w jakim przeglądamy krawędzie o równych wagach. By zbudować drzewo takie jak zaznaczone na grafie algorytm mógł kolejno dołączać następujące krawędzie:

$$(4, 2), (5, 3), (2, 3), (3, 6), (4, 8), (4, 7), (8, 9), (4, 1).$$

Zanim uzasadnię, że powyższy algorytm poprawnie znajduje minimalne drzewo rozpinające potrzeba kilku definicji.

Niech $G = (V, E)$ będzie grafem niezorientowanym. *Podziałem* zbioru V nazywamy parę $(S, V \setminus S)$ taką, że $S \subseteq V$. Krawędź (u, v) *przecina podział* $(S, V \setminus S)$ jeśli jeden z jej końców jest w S a drugi w $V \setminus S$. Krawędź $(u, v) \in E$ jest krawędzią *lekką* dla podziału, $(S, V \setminus S)$, jeśli (u, v) przecina ten podział oraz ma najmniejszą wagę spośród wszystkich krawędzi przecinających ten podział. Niech $A \subseteq E$ będzie podzbiorem minimalnego drzewa rozpinającego. Podział $(S, V \setminus S)$ *respektuje* zbiór krawędzi A, gdy żadna krawędź z A nie przecina $(S, V \setminus S)$. Krawędź $(u, v) \in E$ jest krawędzią *bezpieczną* dla A, gdy $A \cup \{(u, v)\}$ też jest podzbiorem krawędzi pewnego minimalnego drzewa rozpinającego.

Uwaga. Krawędzie bezpieczne to takie które maja interesujące nas własności a krawędzie lekkie to takie, które są charakteryzowane przez łatwy do sprawdzenia warunek. Poniższy Lemat mówi, że lekkość gwarantuje bezpieczeństwo.

Lemat 7.20 *Niech $G = (V, E)$ będzie spójnym grafem niezorientowanym z funkcją wag $W : E \to R_+$, $A \subseteq E$ podzbiór jakiegoś minimalnego drzewa rozpinającego dla G. Niech podział $(S, V \setminus S)$ respektuje A oraz $(u, v) \in E$ będzie krawędzią lekką dla $(S, V \setminus S)$. Wtedy (u, v) jest krawędzią bezpieczną dla A.*

Dowód. Niech podział $(S, V \setminus S)$ respektuje A oraz $(u, v) \in E$ będzie krawędzią lekką dla $(S, V \setminus S)$ oraz (V, T) będzie minimalnym drzewem rozpinającym takim, że $A \subseteq T$. Musimy pokazać, że istnieje minimalne drzewo rozpinające zawierające $A \cup \{(u, v)\}$.

Jeśli $(u, v) \in T$ to teza jest oczywista. Załóżmy zatem, że $(u, v) \notin T$.

Skonstruujemy drzewo $T' \subseteq E$ takie, że $A \cup \{(u, v)\} \subseteq T'$ oraz (V, T') też jest minimalnym drzewem rozpinającym. Ponieważ (V, T) jest spójny, to istnieje ścieżka z u do v w (V, T). Ponieważ (u, v) przecina $(S, V \setminus S)$, to na tej ścieżce w (V, T)

istnieje co najmniej jedna krawędź $(x, y) \in T$ i przecinająca $(S, V \setminus S)$. Ponieważ (V, T) jest spójny i acykliczny, to $(V, T \setminus \{x, y\})$ ma dokładnie dwie składowe, które łączy krawędź (u, v). Zatem kładąc $T' = (T \setminus \{(x, y)\}) \cup \{(u, v)\}$ otrzymujemy (V, T') jako spójny i acykliczny graf. Musimy jeszcze pokazać, że $W(T) = W(T')$. Z minimalności T mamy, że $W(T) \leq W(T')$. Z drugiej strony

$$W(T') - W(T) = \sum_{e \in T'} W(e) - \sum_{e \in T} W(e) = W(u, v) - W(x, y) \leq 0$$

gdzie ostatnia nierówność wynika stąd, że (u, v) i (x, y) przecinają podział $(S, V \setminus S)$ oraz (u, v) jest lekka dla tego podziału. Zatem $W(T) \geq W(T')$.

Q.E.D.

Wniosek 7.21 *Niech $G = (V, E)$ będzie spójnym grafem niezorientowanym z funkcją wag $W : E \to R_+$, $A \subseteq E$ podzbiór jakiegoś minimalnego drzewa rozpinającego dla G. Niech C będzie składową spójną grafu $G_A = (V, A)$. Jeśli $(u, v) \in E$ jest krawędzią lekką dla $(C, V \setminus C)$ to (u, v) jest krawędzią bezpieczną dla A.*

Dowód. Ponieważ C jest składową spójną grafu G_A, to podział $(C, V \setminus C)$ respektuje A. Zatem z Lematu 7.20 otrzymujemy tezę Wniosku.

Q.E.D.

W ten sposób pokazaliśmy

Twierdzenie 7.22 *Algorytm Kruskala znajdowania minimalnego drzewa rozpinającego jest poprawny.*

Poniżej opiszemy inny algorytm znajdowania minimalnego drzewa rozpinającego, pochodzący od Prima. Jest on też algorytmem zachłannym. Używa on innej ciekawej struktury danych: kolejki priorytetowej. Najpierw opiszemy działanie samej kolejki.

Kolejka priorytetowa

Kolejka priorytetowa jest to struktura danych na której można wykonywać następujące operacje:

1. `buduj-kolejke` - mając dany zbiór V i funkcję priorytetu $d : V \to \mathbf{R}$ tworzy kolejkę Q;

2. `dodaj(Q,v,x)` - dodaje do kolejki Q element v z priorytetem x;

3. `min(Q)` - zwraca element kolejki Q o najniższym priorytecie;

4. `usun(Q)` - usuwa z kolejki Q element o najniższym priorytecie;

5. `zmniejsz-klucz(Q,v,x)` - zmniejsza priorytet wierzchołka v do wartości x i aktualizuj kolejkę;

Kolejkę priorytetową można implementować w tablicy przy pomocy kopca takiego jaki był tworzony przy okazji sortowania przez kopcowanie. Poniższa implementacja kolejki jest przystosowana do tego by wygodnie obsługiwała kolejki wierzchołków używane przez algorytmy Prima i Dijkstry. W szczególności elementy przechowywane w kolejce to liczby ze zbioru $\{1, \ldots, n\}$.

```
d : array[1..n] of real; {priorytety}
element=record
          klucz:real;
          el:integer
        end;
Q : array[1..n] of element; {kolejka}
S : array[1..n] of integer;
{S[i] pozycja elementu i w kolejce: Q[S[i]].el=i;
 S[i]=0 gdy elementu nie ma w kolejce}
koniec :integer; {liczba elementow w kolejce}

 procedure w-dol(Q,i,j); {odpowiada procedurze 'kopcuj'}
   begin
     l:=2*i; p:=2*i+1; w:=i;
     if (l<=j) and (Q[l].klucz<Q[i].klucz) then w:=l;
     if (p<=j) and (Q[p].klucz<Q[w].klucz) then w:=p;
     {teraz w Q[w] jest najmniejszy klucz (priorytet) z
     Q[i],Q[l],Q[p]}
     if w<>i then begin
       zamien(Q[w],Q[i]);
       S[Q[w].el]:=w;
       S[Q[i].el]:=i;
       w-dol(Q,w,j);
     end;
   end;

 procedure w-gore(Q,i);
 {odpowiada procedurze 'kopcuj' ale w odwrotna strone tzn. do gory}
   begin
     while (i>1) and (Q[i div 2].klucz>Q[i].klucz) do begin
       zamien(Q[i div 2],Q[i]);
       S[Q[i div 2].el]:=i div 2;
       S[Q[i].el]:=i;
     end;
   end;

 procedure buduj-kolejke(Q,d);
 {buduje n elementowa kolejke Q z priorytetami z d}
   begin
     for i:=1 to n do begin
       Q[i].klucz:=d[i];
       Q[i].el:=i;
       S[i]:=i;
     end;
     for i:= n div 2 to 1 do
       w-dol(Q,i,n);
     koniec:=n;
   end;

function min(Q); {zwraca element o minimalnym priorytecie}
 begin min:=Q[i].el end;
```

```
function pusta(Q); {test pustosci kolejki}
 begin pusta:=(koniec=0) end;

 procedure usun(Q);
 {usuwa element o minimalnym priorytecie z Q}
   begin
     S[Q[1].el]:=0;
     Q[1]:=Q[koniec];
     S[Q[1].el]:=1;
     koniec:=koniec-1;
     w-dol[Q,1,koniec]
   end;

 procedure dodaj(Q,v,x);
 {dodaje do Q element v o priorytecie x}
   begin
     koniec:=koniec+1;
     Q[koniec].el:=v;
     Q[koniec].klucz:=x;
     S[koniec]:=v;
     w-gore[Q,koniec]
   end;

 procedure zmniejsz_klucz(Q,v,x);
 {zmniejsza klucz elementu v do x i odtwarza
 strukture kolejki priorytetowej w Q}
   begin
     d[S[v]]:=x;
     w-gore[Q,S[v]]
   end;
```

Algorytmu Prima znajdowania minimalnego drzewa rozpinającego

Algorytm Prima inaczej niż algorytm Kruskala wybiera krawędź lekką. W czasie działania algorytmu krawędzie wybrane przez algorytm $\{(v, \pi[v]) \,|\, v \in V - Q - \{r\}\}$ stanowią poddrzewo (tzn. podgraf spójny i acykliczny) grafu G.

```
procedure Prim(G,w,r);
  begin koniec:=n;
    for i:=1 to n do d[i]:=nieskonczonosc;
    {nieskonczonosc = liczba wieksza od wszystkich wag w grafie}
    buduj_kolejke(Q,d,koniec);
    zmniejsz_klucz(Q,r,0); P[r]:=nil;
    while not pusta(Q) do begin
      u:=min(Q); usun(Q);
      for v in LI(u) do
        if (S[v]<> 0) and (w[u,v]<d[v]) then begin
          P[v]:=u;
          zmniejsz_klucz(Q,v,w(u,v));
        end;
    end;
  end;
```

Po wykonaniu procedury `Prim(G,w,r)` graf (V,T) jest minimalnym drzewem rozpinającym , gdzie

$$T = \{(P[v], v)) \,:\, v \in V,\; P[n] \neq nil\}.$$

Czas działania algorytmu Prima.

1. Na początku inicjalizujemy tablicę d w czasie $O(|V|)$;
2. Potem inicjalizujemy kolejkę priorytetową Q w czasie $O(|V|)$
3. Pętla while (bez wewnętrznej pętli for) jest wykonywana $O(|V|)$ razy.
4. Łączna liczba wykonań pętli for jest $O(|E|)$, w tej pętli instrukcja `zmniejsz_klucz` jest wykonywana w czasie $O(\ln |V|)$.

Fakt 7.23 *Czas działania algorytmu Prima dla spójnego niezorientowanego grafu z wagami $G = (V, E)$ jest równy $O(|E| \ln(|V|)$.*

Uwaga. Zauważmy, że mamy $O(|E| \ln |E|) = O(|E| \ln\, |V|)$. Zatem czas działania algorytmu Kruskala i algorytmu Prima (przy naszym sposobie implementacji kolejki priorytetowej) jest taki sam. Istnieje bardziej efektywny sposób implementacji kolejki priorytetowej przy którym algorytm Prima działa w czasie $O(|E| + |V| \ln |V|)$.

7.8 Znajdowanie najkrótszej ścieżki w grafie z wagami

Poniżej opiszemy algorytm Dijkstry znajdujący najkrótsze ścieżki z jednego wierzchołka w grafie z wagami. Jest to nieco podobny problem do przeszukiwania wszerz z tym, że teraz będzie nas interesowała nie najmniejsza liczba krawędzi na ścieżce a najmniejsza łączna waga. Ograniczymy się tylko do wag nieujemnych.

Niech $G = (V, E)$ będzie grafem zorientowanym z funkcja wag $w : E \to R_{\geq 0}$. Niech $p = (v_0, \ldots, v_k)$ będzie ścieżką w G. *Wagą* ścieżki p nazywamy sumę wag krawędzi na tej ścieżce:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Odległością w grafie G z funkcją wag w, nazywamy funkcję przyporządkowującym parze wierzchołków wagę 'najlżejszej ścieżki' pomiędzy tymi wierzchołkami ($P(u, v)$ zbiór ścieżek z u do v):

$$\delta(u,v) = \begin{cases} min\{w(p) \,|\, p \in P(u,v)\} & \text{gdy } P(u,v) \neq \emptyset \\ \infty & \text{w przec. przyp.} \end{cases}$$

Problem (znajdowania najkrótszej ścieżki w grafie z wagami).

- Dane wejściowe: graf zorientowany $G = (V, E)$ z funkcją wag w i wierzchołek $s \in V$.
- Wynik: tablice P i d takie, że
  1. $d[v] = \delta(s, v)$ dla $v \in V$;
  2. $P[v]$ jest przedostatnim wierzchołkiem na jednej z najkrótszych ścieżek z s do v lub Nil o ile takich ścieżek nie ma.

Algorytm Dijkstry oparty jest na technice *relaksacji*. Relaksacja krawędzi (u, v) polega na sprawdzeniu czy przypadkiem idąc po krawędzi (u, v) nie polepszymy oszacowania $d[v]$ na długość najkrótszej ścieżki z s do v. Jeśli tak jest to procedura poprawia oszacowanie $d[v]$ i (kandydata na) przedostatni wierzchołek na najkrótszej ścieżce.

```
procedure Relax(u,v,w);
  begin
    if d[v]>d[u]+w(u,v) then begin
      d[v]:=d[u]+w(u,v)
      P[v]:=u
      zmniejsz_klucz(Q,v,d[v])
     end;
  end;
```

Algorytm Dijkstry oblicza tablice d i P metodą kolejnych przybliżeń. By było łatwiej analizować ten program obliczany jest też zbiór Z do którego należą te wierzchołki v dla których już osiągnięta została równość $d[v] = \delta(s, v)$. Algorytm wykonuje relaksację każdej krawędzi dokładnie jeden raz. W celu wybory właściwej kolejności, w której krawędzie będą relaksowane, algorytm używa kolejki priorytetowej Q w której przechowuje wierzchołki grafu G. Wierzchołek v jest przechowywany w Q z priorytetem $d[v]$ będącym najlepszym oszacowaniem jakie się do tej pory udało

uzyskać na długość drogi z s do v. Gdy wierzchołek u ma najmniejszy priorytet to zostaje usunięty z kolejki, dodany do zbioru Z, a krawędzie z niego wychodzące zostają zrelaksowane.

```
procedure Dijkstra(G,w,s);
  begin koniec:=n;
    for v in V do begin d[v]:=nieskonczonosc; P[v]:=Nil end;
    {nieskonczonosc = liczba wieksza od wszystkich wag w grafie}
    buduj_kolejke(Q,d,koniec);
    zmniejsz_klucz(Q,s,0); P[s]:=nil; Z:={};
    while not pusta(Q) do begin
      u:=min(Q); usun(Q); Z:=Z+{u};
      for v in LI(u) do
        Relax(u,v,w);
    end;
  end;
```

Przykład! Bardzo pouczające jest zobaczyć na kilku przykładach jak w trakcie wykonywania algorytmu zmieniają się wartości w tablicach d, P oraz jak rośnie zbiór S. Ponieważ efekt końcowy nie jest zbyt interesujący, przykład graficzny nie zostanie tu przedstawiony.

Złożoność Algorytmu Dijkstry:

1. Inicjalizacja zabiera $O(|V|)$.
2. Operacja `buduj_kolejke(Q,d,koniec)` też zabiera $O(|V|)$.
3. Operacja kolejki priorytetowej `min(Q)` działa w czasie $O(1)$, operacje `zmniejsz_klucz(Q,v,x)`, `usun(Q)` działają w czasie $O(\ln|V|)$. Procedura `Relax(u,v,w)` też działa w czasie $O(\ln|V|)$ ponieważ może się odwoływać do procedury `zmniejsz_klucz(Q,v,d[v])`.
4. Ponieważ na początku Q zawiera wszystkie elementy V i każdy obrót pętli `while` usuwa dokładnie jeden wierzchołek to pętla `while`, bez wewnętrznej pętli `for` wykonuje $O(|V|)$ obrotów. A zatem $O(|V|\ln(|V|))$ operacji.
5. Łączna liczba obrotów pętli `for` we wszystkich obrotach pętli `while` jest równa sumie długości list incydencji grafu G, czyli $|E|$. Zatem łącznie pętla `for` jest wykonywana w czasie $O(|E|\ln(|V|))$.

Zatem pokazaliśmy

Twierdzenie 7.24 *Algorytm Dijkstry działa w czasie* $O((|E|+|V|)\ln(|V|))$.

Pokażemy teraz, że algorytm Dijkstry jest poprawny.

Lemat 7.25 *Niech* $G=(V,E)$ *będzie grafem zorientowanym z funkcją wag* $w: E \to R_{\geq 0}$, $s \in V$. *Jeśli* $(u,v) \in E$ *to* $\delta(s,v) \leq \delta(s,u) + w(u,v)$.

Dowód: Ścieżka z s do v przez u nie może być krótsza od najkrótszej ścieżki z s do v. Q.E.D.

Lemat 7.26 *Niech* $G = (V, E)$ *będzie grafem zorientowanym z funkcją wag* $w : E \to R_{\geq 0}$, $s \in V$. *W procedurze* `Dijkstra` *po incjalizacji dla dowolnego wierzchołka* $v \in V$ *zachodzi* $d[v] \geq \delta(s, v)$ *i warunek ten zachodzi przez cały czas działania procedury. Ponadto jeśli zostanie osiągnięta równość to potem wartość* $d[v]$ *już się nie zmienia.*

Dowód: Warunek jest oczywiście prawdziwy zaraz po inicjalizacji $d[s] = \delta(s, s) = 0$ oraz $\infty = d[v] \geq \delta(s, v)$ dla $v \in V - \{s\}$.

Przypuśćmy, że v jest pierwszym wierzchołkiem dla którego relaksacja krawędzi (u, v) powoduje $d[v] < \delta(s, v)$. Zatem zaraz po relaksacji (u, v) mamy

$$d[u] + w(u, v) = d[v] < \delta(s, v) \leq \delta(s, u) + w(u, v)$$

Stąd mamy $d[u] < \delta(s, u)$. Ale relaksacja (u, v) nie zmienia wartości $d[u]$. Zatem nierówność $d[u] < \delta(s, u)$ musiała też być prawdziwa przed relaksacją (u, v). Ale to przeczy wyborowi v. Z tej sprzeczności wynika, że nierówność $d[v] \geq \delta(s, v)$ jest zachowana do końca wykonywania procedury `Dijkstra`.

By zobaczyć, że jeśli zostanie osiągnięta równość $d[v] = \delta(s, v)$ to $d[v]$ już nie zmieni swojej wartości zauważmy, że nie może się zmniejszyć bo jak pokazaliśmy $d[v] \geq \delta(s, v)$ i nie może się zwiększyć bo relaksacja nie zwiększa wartości $d[v]$. Q.E.D.

Twierdzenie 7.27 *Po wykonaniu algorytmu Dijkstry dla grafu zorientowanego* $G = (V, E)$ *dla* $v \in V$ *mamy* $d[v] = \delta(s, v)$.

Dowód: Pokażemy, że dla każdego wierzchołka $u \in V$ w momencie wstawiania u do zbioru Z mamy $d[v] = \delta(s, v)$ i że ta równość jest zachowana do końca wykonywania algorytmu.

Przypuśćmy przeciwnie, że u jest pierwszym wierzchołkiem takim, że $d[u] \neq \delta(s, u)$ w momencie wstawienia u do Z. Oczywiście $u \neq s$ ponieważ s był pierwszy wstawiony do Z oraz $d[s] = \delta(s, s) = 0$ w momencie wstawiania s do kolejki. Musi istnieć ścieżka z s do u w G bo inaczej mielibyśmy $d[u] = \delta(s, u) = \infty$, co przeczy $d[u] \neq \delta(s, u)$. Niech p będzie najkrótszą ścieżką z s do u. Zatem w momencie wstawiania wierzchołka u do zbioru Z ścieżka ta łączy wierzchołek s z Z z wierzchołkiem u z $V - Z$. Niech y będzie pierwszym wierzchołkiem na ścieżce p, który nie należy do Z, a x jego poprzednikiem na tej ścieżce.

Pokażemy teraz, że $d[y] = \delta(s, y)$ w momencie wstawienia u do Z. Zauważmy, że $x \in Z$ w momencie wstawiania u do Z. Zatem $d[x] = \delta(s, x)$ ponieważ u jest pierwszym wierzchołkiem niespełniającym tej równości. W momencie wstawienia x do Z krawędź (x, y) została zrelaksowana i $d[y]$ przyjęło wartość $\delta(s, y)$.

Ponieważ y występuje przed u na najkrótszej ścieżce z s do u (i wagi są nieujemne) mamy $\delta(s, y) \leq \delta(s, u)$. Zatem

$$d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u].$$

Ale kiedy u zostało zdjęte z kolejki Q to y też było w tej kolejce bo y nie należało jeszcze do Z. Zatem $d[u] \leq d[y]$ i stąd

$$d[y] = \delta(s, y) = \delta(s, u) = d[u].$$

W szczególności $\delta(s, u) = d[u]$ co jest sprzeczne z wyborem wierzchołka u. Z tej sprzeczności wynika, że każdy wierzchołek u w momencie wstawienia do Z spełnia równość $d[u] = \delta(s, u)$. Q.E.D.

8 Złożoność algorytmów

8.1 Problemy decyzyjne

Problemy łatwo obliczalne

Problem	Złożoność
Szukanie słowa w słowniku	$O(\ln n)$
Minimum w tablicy	$O(n)$
Sortowanie	$O(n \ln n)$
Sortowanie topologiczne	$O(\|V\| + \|E\|)$
Znajdowanie silnie spójnych składowych	$O(\|V\| + \|E\|)$
Pierwszość liczb	$O(n^{12})$ (2002)

Są to problemy wielomianowe. Uważa się, że tylko takie problemy są efektywnie obliczalne na komputerach. Poniżej przytoczę trzy powody dlaczego tak jest.

1. Algorytm o złożoności $O(n^{100})$ nie jest efektywny ale takich złożoności w praktyce 'nie ma'. Zwykle wykładnik jest niewiększy niż 3 lub 4.

2. To czy algorytm działa w czasie wielomianowym czy nie nie zależy od modelu obliczeń (komputera).

3. Problemy wielomianowe mają dobre własności (np. złożenie problemów wielomianowych jest problemem wielomianowym).

Żeby móc łatwiej porównywać problemy musimy je trochę ujednolicić, tzn. popatrzeć na nie trochę abstrakcyjniej.

Abstrakcyjny problem algorytmiczny jest to relacja binarna $Q \subseteq I \times S$ gdzie I - jest zbiorem poprawnych danych wejściowych a S zbiorem możliwych rozwiązań. Jeśli $(i, s) \in Q$ to s jest poprawnym rozwiązaniem problemu dla danych i.

Przykład. Problem znajdowania najkrótszej ścieżki w grafie pomiędzy dwoma wierzchołkami można w abstrakcyjny sposób przedstawić tak:

$$I = \{(G, x, y) : G - \text{graf} \quad , x, y \in V(G)\},$$

S jest zbiorem ścieżek w grafach (skończone ciągi). Wtedy $((G, x, y), s) \in Q_{NS}$ iff s jest najkrótszą ścieżką w G z x do y.

By jeszcze uprościć rozważania ograniczymy się do problemów decyzyjnych.

Abstrakcyjny problem decyzyjny jest to abstrakcyjny problem algorytmiczny $Q \subseteq I \times S$ taki, że $S = \{0, 1\}$ oraz Q jest funkcją.

Przykład. Problem znajdowania najkrótszej ścieżki w grafie Q_{NS} pomiędzy dwoma wierzchołkami można też przedstawić jako problem decyzyjny Q_{DNS}tak: $Q_{DNS}(G, x, y, k) = 1$ wtedy i tylko wtedy gdy najkrótszą ścieżką w G z x do y ma długość co najwyżej k.

Problemy decyzyjne są łatwiejsze ale często abstrakcyjne problemy algorytmiczne można do nich zredukować. Ponadto, jeżeli umiemy szybko rozwiązać problem algorytmiczny to też umiemy szybko rozwiązać odpowiadający mu problem decyzyjny. Na ogół jest też odwrotnie. W szczególności, jeśli problem decyzyjny nie daje się

szybko rozwiązać to problem algorytmiczny odpowiadający mu tym bardziej nie ma szybkiego rozwiązania.

Jak zwykle, zakładamy, że na danych wejściowych I każdego problemy jest określona rozsądna funkcja $|-|: I \longrightarrow N$ przyporządkowująca danym $i \in I$ rozmiar $|i| \in N$.

Problem decyzyjny Q jest *wielomianowo obliczalny* jeśli istnieje algorytm działający w czasie wielomianowym sprawdzający czy $Q(i) = 1$ dla $i \in I$. Klasę takich problemów oznaczamy **P**.

8.2 Algorytmy weryfikujące

Dla wielu problemów łatwo jest sprawdzić, że pewien obiekt jest poprawnym rozwiązaniem choć trudno jest takie rozwiązanie znaleźć. Na przykład, łatwiej jest sprawdzić czy dany ciąg $x_0, \ldots, x_l$ reprezentuje ścieżkę długości niewiększej niż k w grafie G z x do y, niż takiej ścieżki szukać.

Innym przykładem jest problem HAM znajdowania cyklu Hamiltona w grafie. Dla grafu G, $HAM(G) = 1$ (lub $G \in HAM$) gdy istnieje w G cykl Hamiltona tzn. taki cykl, który przechodzi przez każdy wierzchołek dokładnie jeden raz. Łatwo jest sprawdzić czy pewien ciąg wierzchołków wyznacza cykl Hamiltona ale trudno jest taki cykl znaleźć.

Niech I zbiór danych wejściowych, $Q \subseteq I$ problem decyzyjny, W zbiór świadków. Algorytm A o dwóch argumentach *weryfikuje problem* Q jeżeli $i \in Q$ wtedy i tylko wtedy gdy istnieje $w \in W$ taki, że $A(i, w) = 1$.

Problem $Q \subseteq I$ jest *obliczalny w niedeterministycznym wielomianowym czasie* (należy do klasy **NP**) jeżeli istnieje zbiór świadków W, algorytm o dwóch argumentach A działający w wielomianowym czasie, oraz stała $c > 0$ takie, że $Q(i) = 1$ wtedy i tylko wtedy gdy istnieje świadek $w \in W$, taki, że $|w| = O(|i|^c)$ oraz $A(i, w) = 1$.

Problem $Q \subseteq I$ należy do klasy **co-NP**, jeżeli problem $I \setminus Q \subseteq I$ należy do klasy **NP**.

Przykłady. Niech $Graf$ będzie zbiorem grafów skończonych.

1. $HAM \in \mathbf{NP}$.

2. k-kolorowanie. $k - Kolor \subseteq Graf$. $G \in k - Kolor$ jeśli istnieje takie pokolorowanie wierzchołków grafu G, k kolorami, że żadne dwa incydentne ze sobą wierzchołki nie są pokolorowane tym samym kolorem. $k - Kolor \in \mathbf{NP}$.

3. Kliki. $k - Klika \subseteq Graf$. $G \in k - Klika$ jeśli istnieje zbiór k wierzchołków w grafie G, w którym każde dwa wierzchołki są ze sobą incydentne. $k - Klika \in \mathbf{NP}$.

4. Spełnialność. I -zbiór formuł klasycznego rachunku zdań, $SAT \subseteq I$. $\varphi \in SAT$ jeżeli istnieje wartościowanie v zmiennych z φ takie, że $v(\varphi) = 1$. Na przykład $x \vee \neg x$, $x \vee (y \to z) \in SAT$, $x \wedge \neg x \notin SAT$. $SAT \in \mathbf{NP}$.

5. Tautologie. I -zbiór formuł klasycznego rachunku zdań, $TAUT \subseteq I$. $\varphi \in TAUT$ jeżeli dla dowolnego wartościowania v zmiennych z φ mamy $v(\varphi) = 1$. $TAUT \in \mathbf{co} - \mathbf{NP}$.

6. Izomorfizm grafów. $I = Graf \times Graf$. $GIzo \subseteq I$. $(G, G') \in GIzo$ gdy G jest izomorficzny z G'. $GIzo \in \mathbf{NP}$.

7. Pokrycie wierzchołkowe grafu. $k-PW \subseteq Graf$. $G \in k-PW$ gdy istnieje k-elementowy zbiór wierzchołków W taki, że dla dowolnej krawędzi (i,j) w G, $i \in W$ lub $j \in W$. $k-PW \in \mathbf{NP}$.

8. Liczby pierwsze. $PRIME \subseteq N$. $n \in PRIME$ gdy n jest liczbą pierwszą. $PRIME \in P$ (2002).

9. $3-CNF-SAT \in \mathbf{NP}$.

8.3 Redukowalność problemów i problem PNP

Mówimy, że problem decyzyjny $Q \subseteq I$ *redukuje się do problemu decyzyjnego* $Q' \subseteq I'$ (oznaczenie: $Q \leq_P Q'$) jeśli istnieje wielomianowo obliczalna funkcja $f : I \longrightarrow I'$ taka, że

$$i \in Q \quad \text{iff} \quad f(i) \in Q'.$$

Gdy $Q \leq_P Q'$ i Q' jest łatwo obliczalny to Q też i jeżeli Q nie jest łatwo obliczalny to Q' też. Mamy

Fakt 8.1 *Jeżeli $Q \leq_P Q'$ to*

1. *jeżeli $Q' \in \mathbf{P}$ to $Q \in \mathbf{P}$;*

2. *jeżeli $Q \notin \mathbf{P}$ to $Q' \notin \mathbf{P}$;*

Przykład redukcji $k-Kolor \leq_P SAT$.

Niech $G = (V,E)$ będzie grafem, $V = \{1,\ldots,n\}$. Skonstruujemy formułę klasycznego rachunku zdań φ_G rozmiaru wielomianowego w stosunku do rozmiaru G taką, że φ_G jest spełnialna wtedy i tylko wtedy gdy G jest k-kolorowalny.

Formuła φ_G ma zmienne

$$x_{ij} \quad \text{dla} \quad i = 1,\ldots,n, \;\; j = 1,\ldots,k.$$

Intuicyjnie zmienna $x_{i,j}$ ma oznaczać, że i-ty wierzchołek jest pomalowany na j-ty kolor.

Formuła

$$\varphi_1 = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{k} x_{i,j}$$

wyraża fakt, że każdy wierzchołek jest pomalowany na co najmniej jeden z k kolorów.

Formuła

$$\varphi_2 = \bigwedge_{i=1}^{n} \bigvee_{j,j'=1, j\neq j'}^{k} \neg(x_{i,j} \wedge x_{i,j'})$$

wyraża fakt, że każdy wierzchołek jest pomalowany na co najwyżej jeden z k kolorów.

Formuła

$$\varphi_3 = \bigwedge_{(i,i')\in E} \neg \bigvee_{j=1}^{k} (x_{i,j} \wedge x_{i',j})$$

wyraża fakt, że wierzchołki incydentne są pomalowane na różne kolory.

Niech $\varphi_G = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$. Oczywiście formuła φ_G jest rozmiaru wielomianowego w stosunku do rozmiaru grafu G.

Jeśli φ_G jest spełniona przez wartościowanie W to i-ty wierzchołek z V malujemy na kolor j-ty wtedy i tylko wtedy gdy $W(x_{i,j}) = 1$ dla $i = 1, \ldots, n$, $j = 1, \ldots, k$ i w ten sposób otrzymujemy pokolorowanie grafu G na k kolorów.

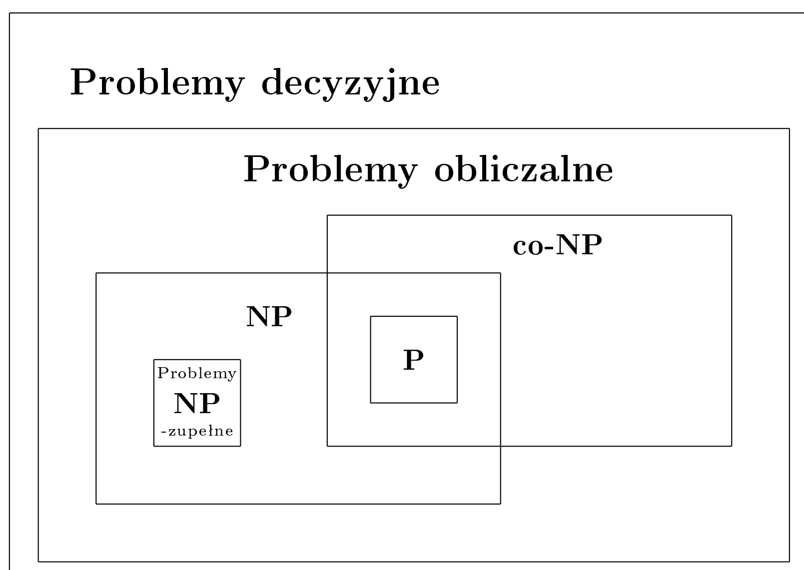
Z drugiej strony, jeśli $C : V \to \{1, \ldots, k\}$ jest pokolorowaniem grafu G na k kolorów to wartościowanie W takie, że $W(x_{i,j} = 1$ wtedy i tylko wtedy gdy $C(i) = j$ dla $i = 1, \ldots, n$, $j = 1, \ldots, k$ spełnia formułę φ_G.

Zatem formuła $\varphi_G = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ ma żądane własności.

Uwaga. Redukcja w drugą stronę pokazująca, że $SAT \leq_P k{-}Kolor$ też istnieje ale jest trudniejsza.

Mówimy, że problem Q jest **NP**-*trudny*, jeśli dla każdego problemu $Q' \in$ **NP**, $Q' \leq_P Q$. Problem Q jest **NP**-*zupełny*, jeśli Q jest **NP**-*trudny* oraz $Q \in$ **NP**.

Poniższy diagram opisuje zależności pomiędzy wprowadzonymi klasami problemów. Przypomnijmy, że problem jest *obliczalny*, jeżeli istnieje (jakikolwiek) algorytm, który go rozwiązuje.



Problem PNP. Czy **P**=**NP**?

Fakt 8.2 *Jeśli jeden problem* **NP**-*zupełny należy do* **P** *to* **P**=**NP**=**co-NP**.

Przykłady problemów **NP**-*zupełnych.*

1. HAM.
2. $k - Kolor$.
3. $k - Klika$.
4. SAT.
5. $GIzo$.
6. $k{-}PW$.
7. $3 - CNF - SAT$.

8.4 Problemy nieobliczalne

Pytanie czy istnieją problemy nieobliczalne ma posmak 'filozoficzny', ponieważ nie dotyczy ono naszej kondycji tu i teraz ale tego co w ogóle da się obliczyć. Innymi słowy dotyka problemu 'granicy naszych możliwości'. Twierdząca odpowiedź na to pytanie pochodząca od K.Gödla wywołała sporo zamieszania, które tu i ówdzie trwa do dziś.

Pokaże, że tak zwany problem stopu jest jednym z problemów nierozstrzygalnych.

Problem stopu.

- Dane wejściowe: ciągi znaków (dowolnej długości) s_1 i s_2.
- Wynik: **true** gdy s_1 jest tekstem procedury (w Pascalu) mającej jako parametr ciąg znaków oraz procedura s_1 zatrzymuje się na danych s_2, **false** w przeciwnym przypadku.

Inaczej mówiąc, by rozwiązać ten problem trzeba skonstruować funkcję `Stop` działającą na parach ciągów znaków sprawdzającą czy pierwszy ciąg zastosowany do drugiego jako procedura (o ile to ma sens) zatrzyma się czy nie:

$$\texttt{Stop[s1,s2]} = \begin{cases} \textbf{true} & \text{gdy } \texttt{s1} \text{ jest tekstem procedury w Pascalu,} \\ & \text{która ma jeden parametr typu} \\ & \text{'ciągu znaków dowolnej dłgości',} \\ & \text{oraz } \texttt{s1} \text{ na danych } \texttt{s2} \text{ zatrzymuje się,} \\ \textbf{false} & \text{w przeciwnym przypadku.} \end{cases}$$

Przypuśćmy, że problem stopu jest obliczalny i że mamy taką procedurę `Stop` jak wyżej. Wtedy możemy skonstruować procedurę `Diag` z jednym parametrem typu ciągu znaków w następujący sposób:

```
procedure Diag (s);
 begin
  if Stop[s,s] then loop
 end;
```

gdzie loop oznacza dowolną pętlę nieskończoną, na przykład

```
while true do i:=i
```

Procedura `Diag` sprawdza czy procedura `s` zatrzymuje się dla parametru aktualnego będącego tekstem tej procedury. Jeśli zatrzymuje się to `Diag` wchodzi w pętlę nieskończoną `loop` a jeśli nie to `Diag` zatrzymuje się.

Niech `'Diag'` oznacza tekst procedury `Diag`. Powstaje pytanie co się stanie gdy wykonamy `Diag['Diag']`? W szczególności, czy procedura się zatrzyma czy nie?

Jeśli `Diag['Diag']` zatrzyma się to procedura `Stop['Diag','Diag']` zwróci wartość **true**. Ale to oznacza, że `Diag['Diag']` wejdzie w nieskończoną pętlę `loop` i się nie zatrzyma. Czyli mamy sprzeczność!

Przypuśćmy teraz, że `Diag['Diag']` nie zatrzyma się. Wtedy procedura `Stop['Diag','Diag']` zwróci wartość **false**. I `Diag['Diag']` po sprawdzeniu warunku nie wejdzie w nieskończoną pętlę `loop` i się zatrzyma. Czyli mamy znowu sprzeczność!

A to oznacza, że takiego programu `Stop` być nie może! Czyli pokazaliśmy, że

Twierdzenie 8.3 *Problem stopu jest nieobliczalny.*

Zauważmy, że problem stopu jest dość skromnym problemem w stosunku do tego co by 'można chcieć'. W szczególności, ciekawszy problem czy dany program rozwiązuje poprawnie dany problem algorytmiczny też nie jest obliczalny.

8.5 Metody przybliżone

Co robić jak nie można rozwiązać problemu efektywnie?

Jeżeli nie możemy rozwiązać problemu efektywnie takim jaki on jest, to trzeba nieco zliberalizować nasze wymagania by 'zyskać na szybkości kosztem jakości'. Na przykład, możemy zmniejszyć nasze oczekiwania dotyczące *dokładności* rozwiązania, i/lub *pewności* rozwiązania. Są trzy metody, które pogarszając jakość rozwiązania zwiększają szybkość działania algorytmów:

1. *Heurystyki*: by znaleźć rozwiązanie stosujemy pewne strategie, które dostatecznie często, choć nie zawsze, działają szybko i dostatecznie często, choć być może nie zawsze, dają poprawne rozwiązania.

2. *Algorytmy aproksymacyjne*: szukamy rozwiązań, które są trochę gorsze od optymalnych ale przy pomocy efektywnych algorytmów.

3. *Algorytmy probabilistyczne*: szukamy rozwiązań optymalnych ale znalezione rozwiązania są poprawne tylko z pewnym, satysfakcjonującym nas, prawdopodobieństwem.

Heurystyki są zwykle bardzo silnie związane z problemem, który rozwiązują i są zwykle trudne do opisania w sposób ogólny. Na przykład istniejące algorytmy szachowe używają różnego rodzaju heurystyk, które są wynikiem wielowiekowych dociekań mistrzów szachowych. Nie dają one pewności, że strategia wybrana przez komputer jest optymalna ale często okazuje się skuteczna.

Poniżej podamy przykład algorytmu aproksymacyjnego i dwóch algorytmów probabilistycznych.

Algorytm aproksymacyjny dla pokrycia wierzchołkowego grafu.

Problem pokrycia wierzchołkowego grafu przedstawiony został wcześniej jako problem decyzyjny. Odpowiedni problem optymalizacyjny wygląda tak:

Problem pokrycia wierzchołkowego grafu.

- Dane wejściowe: graf $G = (V, E)$.
- Wynik: minimalny podzbiór $W \subseteq V$ taki, że dla dowolnej krawędzi $(i, j) \in E$, $i \in W$ lub $j \in W$.

Ten problem jest **NP**-zupełny, zatem nie możemy oczekiwać, że skonstruujemy algorytm znajdujący efektywnie optymalne rozwiązanie. Natomiast istnieje prosty algorytm zachłanny, który znajduje rozwiązanie rozmiaru co najwyżej dwa razy większego od optymalnego rozwiązania:

```
procedure approx-PW(G:graf);
 begin
```

```
W:=[]; F:=E;
while F<>[] do begin
  niech (u,v) pierwszy element z F;
  W:=W+{u,v};
  wyrzuc wszystkie krawedzie z F o koncu u lub v;
end;
```

Po wykonaniu tej procedury W jest zbiorem krawędzi takim, że dla dowolnej krawędzi $(i, j) \in E$, $i \in W$ lub $j \in W$. Ponadto, można pokazać, że W jest co najwyżej dwa razy większy od każdego innego zbioru o tej własności. Procedura `approx-PW(G)` kopiuje zbiór krawędzi E grafu G jako nowy zbiór F. W pętli `while` procedura wybiera kolejną krawędź (u, v) z F i dodaje do konstruowanego zbioru jej końce. Następnie usuwa wszystkie krawędzie z F, których jednym z końców jest wierzchołek u lub v.

Algorytm probabilistyczny weryfikowania pierwszości liczb.

Weryfikacja czy duża liczb naturalna jest pierwsza czy złożona też wydaje się być trudnym problemem. Nie istnieje obecnie, żaden algorytm, który by mógł efektywnie sprawdzić (nawet na najszybszych dostępnych komputerach) czy np. 300-cyfrowa liczba jest pierwsza. Z drugiej strony problem ten ma fundamentalne znaczenie dla kryptografii z kluczem publicznym RSA. Liczby pierwsze i złożone mają różne własności, które można efektywnie sprawdzać. Może istnieć na przykład liczba w, która jest niejako *świadkiem złożoności* liczby n. Poniżej wyliczone są trzy różne sposoby jak w może świadczyć o złożoności liczby n. Mając n i w, obliczenie czy w taki właśnie sposób w świadczy o złożoności n jest efektywne.

1. *Świadek Euklidesa*: Jeżeli $NWD(w, n) > 0$ to oczywiście n jest liczbą złożoną. Problem w tym, że takich świadków w może być bardzo mało. Na przykład, jeżeli n jest iloczynem dwóch 100-cyfrowych liczb pierwszych p i q to takich świadków jest $(p - 1) + (q - 1)$ i szansa na przypadkowe trafienie na jednego z nich jest rzędu $\frac{1}{10^{100}}$, czyli bardzo mała.

2. *Świadek Fermata*: Jeżeli $w < n$ oraz $w^{n-1} \not\equiv_n 1$[5] to, na mocy Małego Twierdzenia Fermata, n jest liczbą złożoną. Niestety są liczby złożone tzw. *liczby Carmichaela*, które w ogóle nie mają świadków złożoności w sensie Fermata. Choć liczby Carmichaela są bardzo rzadkie to jednak test przy użyciu świadków złożoności w sensie Fermata jest uważany za niedoskonały.

3. *Świadek Millera-Rabina*: Jeżeli $1 < w < n$ oraz $w^{n-1} \not\equiv_n 1$ lub $w^2 \equiv_n 1$ (tzn. w jest nietrywialnym pierwiastkiem z 1 modulo n), to n też jest liczbą złożoną. Dokładniej, w jest świadkiem złożoności w sensie Millera-Rabina dla liczby nieparzystej n, jeśli `swiadekMR(n,w)=true`, gdzie `swiadekMR` jest funkcją zdefiniowaną poniżej. Świadków złożoności w sensie Millera-Rabina jest bardzo dużo. W rzeczywistości, o ile n jest nieparzystą liczbą złożoną, to co najmniej połowa liczb pomiędzy 1 a n jest takimi świadkami. Ta mnogość świadków pozwala na skonstruowanie zadowalającego (na razie) algorytmu, weryfikującego czy dana liczba jest pierwsza.

Weryfikację czy dana liczba w jest świadkiem złożoności w sensie Millera-Rabina dla liczby n można wykonać przy pomocy następującej procedury:

[5] $\equiv_n$ oznacza przystawanie modulo n.

```
function swiadekMR(n,w:integer):boolean;
 begin
  niech (b_k,...,b_0) bedzie dwojkowa reprezentacja liczby n-1;
  d:=1; OK:=false; i:=k;
  while (i>=0) and not OK do begin
    x:=d;
    d:=(d*d) mod n;
    if (d=1) and (x<>1) and (x<>n-1) then
      OK:=true; {wykryto nietrywialny pierwiastek z 1 modulo n}
    if b_i=1 then d:=(d*w) mod n;
    i:=i-1;
  end;
   swiadekMR:=OK or (d<>1);
   {wykryto nietrywialny pierwiastek z 1 modulo n
   lub w jest swiadkiem w sensie Fermata}
 end;
```

Procedura `swiadekMR(n,w)` sprawdza czy $w^{n-1} \equiv_n 1$ a trakcie obliczania wartości $w^{n-1} \, mod \, n$ sprawdza przy okazji czy nie ma nietrywialnego (różnego od 1 i -1) pierwiastka z 1.

By zobaczyć, jak jest liczona potęga $w^{n-1} \, mod \, n$, zauważmy, że formuła $d = w^e$ oraz $e = (b_k, \ldots, b_{i+1})_2$[6] jest niezmiennikiem pętli while.

Jeśli dla jednej liczby w liczba n przeszła pozytywnie próbę to z prawdopodobieństwem $\frac{1}{2}$ jest liczbą pierwszą. Ale nic nie stoi na przeszkodzie by taki test powtórzyć wielokrotnie. Możemy wylosować na przykład 100 liczb pomiędzy 1 i n i dla nich przeprowadzić powyższy test. Jeśli liczba n wszystkie te testy przejdzie pozytywnie to prawdopodobieństwo, że n jest liczbą złożona jest $\frac{1}{2^{100}}$ zatem prawdopodobieństwo, że n jest liczba pierwszą jest $1 - \frac{1}{2^{100}}$. Poniższa procedura implementuje tę ideę.

```
procedure test-Millera-Rabina(n:integer);
 begin OK:=false;
   for i:=1 to 100 do begin
     w:=random(1,n-1);
     OK:=OK or swiadekMR(n,w);
   end;
   test-Millera-Rabina:=OK;
 end;
```

Jeśli `test-Millera-Rabina(n)=true` to n jest liczbą złożoną na pewno, a jeśli `test-Millera-Rabina(n)=false` to n jest liczbą pierwszą prawie na pewno.

Algorytm probabilistyczny dla problemu wyboru par.

Technicznym powodem, dla którego liczba świadków złożoności w sensie Millera-Rabina jest duża jest fakt mówiący o tym, że 'nieświadkowie' stanowią właściwą podgrupę grupy Z_n, liczb $\{0, \ldots, n-1\}$ z dodawaniem modulo n. Z elementarnej teorii grup (Twierdzenie Lagrange'a) wiadomo, że liczba elementów podgrupy dzieli liczbę elementów grupy. A zatem podgrupa 'nieświadków' Z_n ma co najwyżej $\frac{n}{2}$ elementów.

[6]Czyli, że $e = 2^i \sum_{j:=i+1}^{k} b_j \cdot 2^j$

Algorytm probabilistyczny dla problemu wyboru par jest oparty na obserwacji dotyczącej mnogości świadków dla innego innego zjawiska. Jeśli mianowicie mamy dwa wielomiany o współczynnikach rzeczywistych (być może wielu zmiennych) $p(x_1, \ldots, x_n)$ i $q(x_1, \ldots, x_n)$, które są różne to są one różne dla bardzo wielu argumentów. Innymi słowy rzadko się zdarza by różne wielomiany przyjmowały te same wartości dla tych samych argumentów. A to powoduje, że by sprawdzić czy są one równe, można wybrać losowo n liczb rzeczywistych $a_1, \ldots, a_n$ i sprawdzić czy liczby rzeczywiste $p(a_1, \ldots, a_n)$ i $q(a_1, \ldots, a_n)$ są równe. Jeśli *nie*, są równe to na pewno wielomiany są różne ale jeśli *tak*, to 'prawie na pewno' są one równe! Algorytm, który opiszę poniżej jest oparty na tej obserwacji.

Problem (decyzyjny) zgodnego wyboru par.

- Dane wejściowe: graf skierowany $G = (V, E)$ taki, że $V = \{1, \ldots, 2n\}$ oraz jeśli $(i, j) \in E$ to $1 \leq i \leq n$ i $n < j \leq 2n$.
- Wynik: TAK, jeżeli istnieje zbiór n krawędzi $E' \subseteq E$ taki, że każdy wierzchołek z $\{1, \ldots, n\}$ jest początkiem a każdy wierzchołek z $\{n+1, \ldots, 2n\}$ jest końcem pewnej krawędzi E'. NIE, w przeciwnym przypadku.

Opis algorytmu.

1. Niech A będzie $(n \times n)$-macierzą taką, której wyrazy są różnymi zmiennymi albo równe 0.

$$a_{i,j} = \begin{cases} x_{i,j} & \text{gdy } (i, n+j \in E) \\ 0 & \text{w przeciwnym przypadku} \end{cases}$$

   dla $i, j = 1, \ldots, n$.

2. Niech $d(< x_{i,j} >_{(i,n+j) \in E})$ będzie wielomianem, który jest wyznacznikiem (formalnym) macierzy A.
3. Wybieramy losowo liczby $r_{i,j}$ dla $(i, n+j) \in E)$.
4. Obliczamy wartość $w = d(< r_{i,j} >_{(i,n+j) \in E})$.
5. Jeżeli $w \neq 0$ to zgodny wybór par istnieje na pewno i odpowiadamy TAK. Jeżeli $w = 0$ to zgodny wybór par nie istnieje 'prawie na pewno' i odpowiadamy NIE.

Powyższy algorytm opiera się na obserwacji, że $d(< x_{i,j} >_{(i,n+j) \in E})$ jest wielomianem tożsamościowo równym 0 wtedy i tylko wtedy gdy nie istnieje zgodny wybór par. Natomiast sprawdzenie czy $d(< x_{i,j} >_{(i,n+j) \in E})$ jest wielomianem tożsamościowo równym 0 ma charakter probabilistyczny. Korzystamy ze wspomnianego wyżej faktu, że jeżeli wielomian $d(< x_{i,j} >_{(i,n+j) \in E})$ nie jest tożsamościowo równy zero, to przypadkowe trafienie na punkt w którym się zeruje jest mało prawdopodobne.

Przypomnijmy, że wyznacznik $(n \times n)$-macierzy można obliczyć ze wzoru:

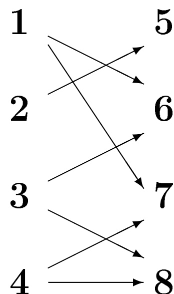
$$detA = \sum_{\sigma \in S_n} sgn(\sigma) \cdot x_{1,\sigma(1)} \cdot \ldots \cdot x_{n,\sigma(n)}$$

gdzie S_n jest zbiorem permutacji zbioru $\{1, \ldots, n\}$ a $sgn(\sigma)$ jest znakiem permutacji σ. Zatem wielomian $d(< x_{i,j} >_{(i,n+j) \in E})$ nie jest tożsamościowo równy zero wtedy i tylko wtedy gdy istnieje permutacja $\sigma \in S_n$ taka, że $a_{i,\sigma(i)} = x_{i,\sigma(i)}$. A to ma miejsce

tylko wtedy gdy zbiór $\{(1, n+\sigma(1)), \ldots, (n, n+\sigma(n))\}$ jest zawarty w E z zatem jest zgodnym wyborem par.

Ponieważ wyznacznik $(n \times n)$-macierzy można obliczyć efektywnie (patrz następny rozdział) to cały algorytm jest efektywny.

Przykład. Niech $G = (V, E)$ będzie następującym grafem:



Wtedy macierz A ma postać:

$$A = \begin{bmatrix} 0 & x_{1,2} & x_{1,3} & 0 \\ x_{2,1} & 0 & 0 & 0 \\ 0 & x_{3,2} & 0 & x_{3,4} \\ 0 & 0 & x_{3,4} & x_{4,4} \end{bmatrix}$$

a jej wyznacznik

$$d(< x_{i,j} >_{(i,n+j)\in E}) = detA = x_{1,2} \cdot x_{2,1} \cdot x_{3,4} \cdot x_{4,3} + x_{1,3} \cdot x_{2,1} \cdot x_{3,2} \cdot x_{4,4}$$

ma dwa niezerowe składniki odpowiadające dwóm różnym zgodnym wyborom par:

$$\{(1,6),(2,5),(3,8),(4,7)\}$$

oraz

$$\{(1,7),(2,5),(3,6),(4,8)\}.$$

Zauważmy jednak, że powyższy algorytm probabilistyczny pozwala tylko na stwierdzenie czy zgodny wybór par istnieje i nie daje możliwości by go efektywnie skonstruować. Dzieje się tak dlatego, że z informacji, że istnieją niezerowe składniki wyrażenia $detA = \sum_{\sigma \in S_n} sgn(\sigma) \cdot x_{1,\sigma(1)} \cdot \ldots \cdot x_{n,\sigma(n)}$ nie możemy wywnioskować, które z nich rzeczywiście są niezerowe.

Indeks

ścieżka, 93

algorytm, 5
    -iczne rozwiązanie, 5
    -iczny problem, 4
    aproksymacyjny, 127
    Euklidesa, 4
    probabilistyczny, 127
    sortowania przez kopcowanie, 48
    weryfikujący, 123
    zachłanny, 45
analiza
    numeryczna, 5
    poprawności, 5
    złożoności, 5

błąd
    bezwzględny, 57
    względny, 57
blok, 13

cecha, 57
cykl, 93
    rozkazów, 9

diagramy składniowe, 11
drzewa
    wysokość -, 97
drzewo, 97
    binarne, 65, 73
    binarnych poszukiwań, 73
    czerwono-czarne, 78
    pod-, 97
    przeszukiwania wszerz, 97

głębokość wierzchołka, 97
graf
    acykliczny, 94
    niezorientowany, 93
    składowe spójne -u, 94
    spójny, 94
    zorientowany, 93

instrukcja
    pętli for, 21
    pętli while, 19
    przypisania, 16
    pusta, 18
    warunkowa, 19
    wejścia-wyjścia, 21
    złożona, 19
iteracja
    ograniczona, 21
    warunkowa, 19

język
    Pascal, 11
    wysokiego poziomu, 11

klasa
    NP, 123
    P, 123
    coNP, 123
Kod Hauffmana, 47
kolejka, 64
    priorytetowa, 115
kopiec, 48
krawędź
    powrotna, 106

las, 102
    przeszukiwania w głąb, 102
liść, 97
lista
    dwukierunkowa, 71
    dwukierunkowa z wartownikiem, 71
    jednokierunkowa, 69
listy
    incydencji grafu, 94

macierz
    incydencji grafu, 94
magistrale komunikacyjne, 9
mantysa, 57
metoda
    'dziel i rządź', 36
    powrotów, 32

nadmiar, 58
niedomiar, 58
niezmiennik pętli, 4, 20, 21
notacja
    $O(f(n))$, 6, 37
    $\Omega(f(n))$, 37
    $\Theta(f(n))$, 37

pamięć, 9
parametr

aktualny, 25
formalny, 25
procedury, 24
poddrzewo, 97
poprzednik, 97
potomek, 97
właściwy, 97
praojciec, 110
problem
hetmanów, 32
-ów redukowalność, 124
abstrakcyjny - algorytmiczny, 122
abstrakcyjny - znajdowania najkrótszej ścieżki w grafie, 122
algorytmiczny, 4
decyzyjny, 122
izomorfizmu grafów, 123
kliki, 123
kodowania znaków, 47
kolorowania grafu, 123
najdłuższego wspólnego podciągu, 41
NP-zupełny, 125
plecakowy
0-1, 47
ułamkowy, 47
PNP, 125
pokrycia wierzchołkowego, 124
przeszukiwania
grafu w głąb, 102
grafu wszerz, 97
SAT, 123
scalania, 36
sortowania, 5
sortowania topologicznego, 106
stopu, 126
TAUT, 123
wież Hanoi, 7
wielomianowo obliczalny, 123
wyszukiwania w słowniku, 7
zgodnego wyboru zajęć, 45
znajdowania
cyklu Hamiltona, 123
minimalnego drzewa rozpinającego, 113
najkrótszej ścieżki w grafie z wagami, 119
silnie spójnych składowych, 108
składowych spójnych, 95
procedura, 22
dispose, 68
new, 68
rekurencyjna, 26
procesor, 9
program, 12
-owanie dynamiczne, 41
częściowo poprawny, 30
poprawny, 30
przodek, 97

redukowalność problemów, 124
rekord, 15
relaksacja, 119
reprezentacja
stałopozycyjna, 54
zmiennopozycyjna, 57
rozmiar danych, 6
rozstrzygalność, 5

semantyka, 11
składnia, 11
składowe, silnie spójne, 108
stała, 14
stopień
wierzchołka, 93
wyjściowy wierzchołka, 93
stos, 26, 63
system
dwójkowy, 53
dziesiętny, 53
szesnastkowy, 54

tablica, 15
typ
łańcuchowy, 14
całkowity, 14
logiczny, 14
okrojony, 15
prosty, 14
rekordowy, 15
rzeczywisty, 14
standardowy, 14
strukturalny, 15
tablicowy, 15
wskaźnikowy, 66
wyliczeniowy, 15
znakowy, 14

układ wejścia-wyjścia, 9

waga
ścieżki, 119

warunek
- końcowy, 30
- początkowy, 30

wierzchołka
- głębokość -, 97

wysokość
- czarna, 78
- drzewa, 97
- wierzchołka, 97

względną dokładnością komputera, 58

złożoność algorytmu, 6

zapis logiczny, 17

zmienna, 13
- dynamiczna, 66
- globalna, 24
- lokalna, 24
- statyczna, 66
- w procedurze, 24
- zasłanianie -, 25

znak
- -moduł, 55
- uzupełnienie do 1, 55
- uzupełnienie do 2, 55