Algorytmy i struktury baz danych wykład

Zagadnienia do kolokwium

1. **Co to jest algorytm?**

   Algorytm to skończony, uporządkowany ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego zadania.

2. **Cechy algorytmu**

   a. Poprawność,
   b. Jednoznaczność,
   c. Skończoność,
   d. Sprawność...?
   e. Określony deterministycznie
   f. Posiada złożoność

3. **W jaki sposób można zapisać algorytm?**

   a. lista kroków,
   b. schemat blokowy,
   c. pseudokod,
   d. język naturalny,
   e. język programowania.

4. **Jakie elementy algorytmu oceniamy (jaki algorytm będzie idealnym)?**

   Idealnym algorytmem będzie taki, który:

   - Posiada prosty kod,
   - Jest łatwy do zrozumienia,
   - Może być napisany w każdym języku,
   - Wykonuje się szybko,
   - Zajmuje niewiele pamięci,
   - Zawsze daje poprawne wyniki.

5. **Na podstawie jakich parametrów oceniamy algorytm? Co decyduje o wyborze algorytmu?**

- Koszt algorytmu – Wybór miary oceny, kosztu algorytmu zależy od typu problemu i rodzaju rozwiązania. Czy chcemy algorytm szybszy, czy zajmujący mniej pamięci.

Pamięć <–> Czas	
• Liczba zmiennych • Ilość miejsca potrzebna dla danych tymczasowych	• Liczba wyrażeń arytmetycznych i logicznych • Liczba wywołań procedury

Czasowa złożoność obliczeniowa – rozumiemy ilość czasu niezbędnego do rozwiązania problemu w zależności od liczby danych wejściowych. Złożoność czasowa jest zatem pewną funkcją liczby danych wejściowych [n].

- Złożoność czasową wyrażamy w jednostkach czasu [s] albo w liczbie operacji [n].
- Ilość operacji: Liczba wyrażeń arytmetycznych i logicznych, Liczba wywołań procedur

Złożoność pamięciowa – określa z kolei liczbę komórek pamięci, która będzie zajęta przez dane i wyniki pośrednie tworzone w trakcie pracy algorytmu.

Złożoność może być:

- Minimalna – *optymistyczna*, określa zużycie zasobów dla najkorzystniejszego zestawu danych
- Średnia – określa zużycie zasobów dla losowego zestawu danych
- Maksymalna – *pesymistyczna*, określa zużycie zasobów dla niekorzystnego zestawu danych

6. Opisz i narysuj przykład (funkcje zapisu i odczytu) struktury danych:

a. Tablica jednowymiarowa (wektor)

- Nazwa_wektora[ilość_elementow] = {element1, element2, element3, ...}

$$[1\ 2\ ],\ [1\ 2\ ],\ [1\ 2\ 3\ ],\ [1\ 2\ 3\ ]$$

b. Tablica wielowymiarowa (macierz)

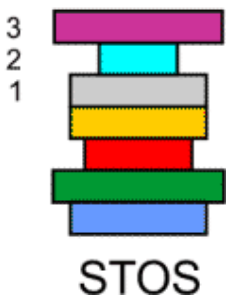
- Nazwa_macierzy[liczba_wierszy][liczba_kolumn]

$$\mathbf{A}_{3\,x\,4} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}$$

c. Stos

Jest to zorganizowana struktura według zasady LIFO (last in first out), stos nie jest adresowany – dostępna jest tylko ostatnio zapisana zmienna.

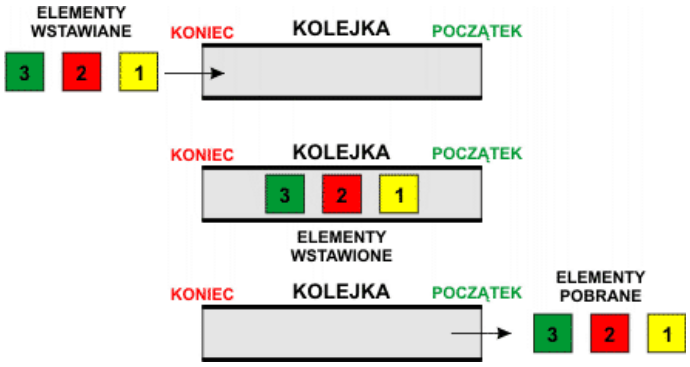
- Dostęp tylko do wierzchołka
- Dostępne dwie podstawowe funkcje
  - PUSH (S, X) – umieszczanie X na stosie
  - POP (S) – zdejmowanie elementu ze stosu
- Każdy element składa się z dwóch pól
  - Dane – dowolna zmienna
  - Wskaźnik – zawiera informacje o miejscu przechowania kolejnego elementu



d. Kolejka

- Działa na zasadzie FIFO (first in first out)
- Działa dokładnie tak samo jak kolejka w sklepie

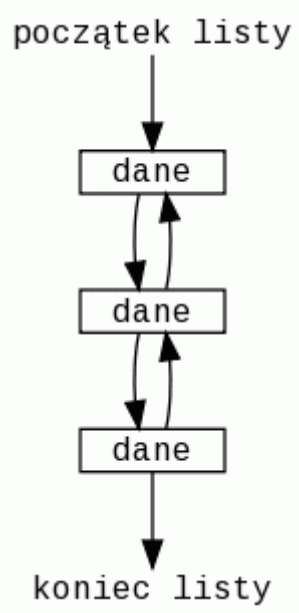
- Pierwszy do kasy idzie klient z przodu kolejki
- Liniowa struktura danych
- Element jest pobierany z początku
- Nowe elementy wstawiamy na koniec



- Sprawdzanie czy kolejka jest pusta – operacja **empty** zwraca *true*, jeśli kolejka nie zawiera żadnego elementu, w przeciwnym razie zwraca *false*.
- Odczyt elementu z początku kolejki – operacja **front** zwraca wskazanie do elementu, który jest pierwszy w kolejce.
- Zapis elementu na koniec kolejki – operacja **push** dopisuje nowy element na koniec elementów przechowywanych w kolejce.
- Usunięcie elementu z kolejki – operacja **pop** usuwa z kolejki pierwszy element.

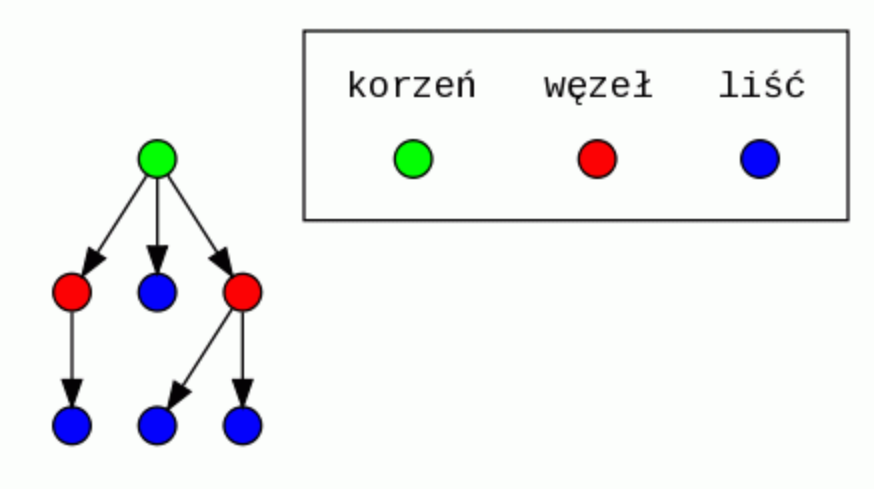
e. Lista

- W tej strukturze dane można dodać gdziekolwiek, ważne jest, aby sąsiednie dane uwzględniły zmianę.
  - Liniowa struktura danych
  - To ciąg elementów, w których każdy pamięta poprzednika i następnika
  - Za pomocą listy dwukierunkowej możemy symulować kolejkę i stos
  - Podstawowe funkcje:
    - Search (L, x) – znajduje wskaźnik do elementu o kluczu x, bądź null
    - Insert (L, x, $w_{(miejsce)}$) – wstawia element x w miejsce w
    - Delete (L, w) – usuwa element z miejsca w
    - Min (L) – najmniejszy klucz
    - Max (L) – największy klucz



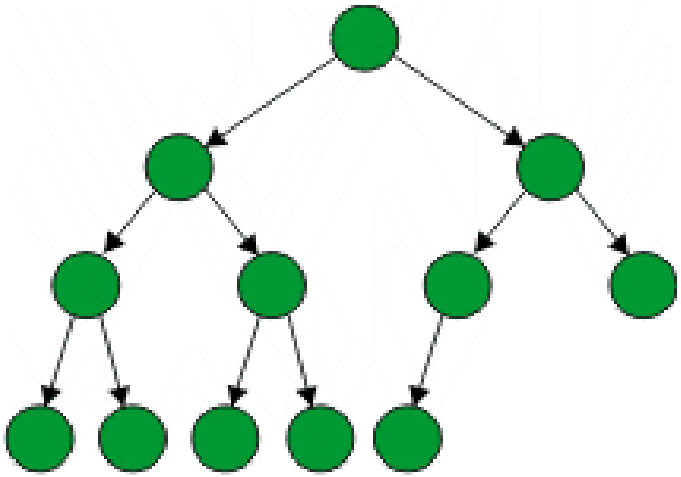
f. Drzewo binarne

Drzewo - jest strukturą danych zbudowaną z elementów, które nazywamy węzłami. Dane przechowuje się w węzłach drzewa. Węzły są ze sobą powiązane w sposób hierarchiczny za pomocą krawędzi, które zwykle przedstawia się za pomocą strzałki określającej hierarchię. Pierwszy węzeł drzewa nazywa się korzeniem. Od niego "wyrastają" pozostałe węzły, które będziemy nazywać synami. Synowie są węzłami podrzędnymi w strukturze hierarchicznej. Synowie tego samego ojca są nazywani braćmi. Węzeł nadrzędny w stosunku do syna nazwiemy ojcem. Jeśli węzeł nie posiada synów, to nazywa się liściem, w przeciwnym razie nazywa się węzłem wewnętrznym.

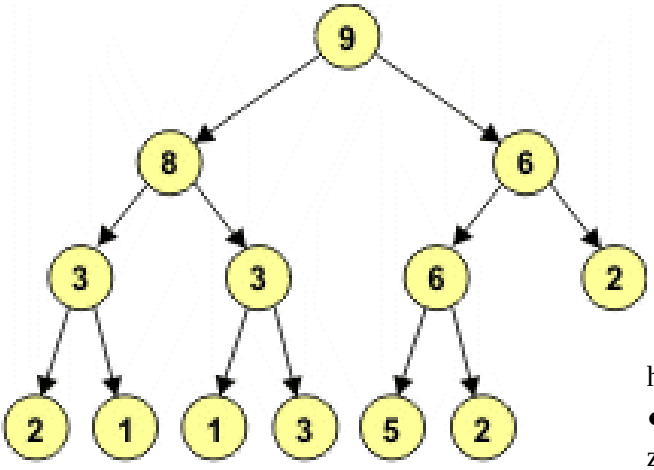


g. Kopiec/Stóg/Sterta

- Jest kompletnym drzewem binarnym, co oznacza, że posada zapełnione wszystkie poziomy za wyjątkiem ostatniego, a ostatni poziom jest zapełniany bez przerw, poczynając od strony lewej do prawej.



- Oprócz wymogu strukturalnego (drzewo binarne musi być kompletne), kopiec posiada dodatkowy warunek: żaden z synów węzła nie jest od niego większy, czyli nie przechowuje w swoim polu data wartości większej od zawartości pola data swojego ojca.



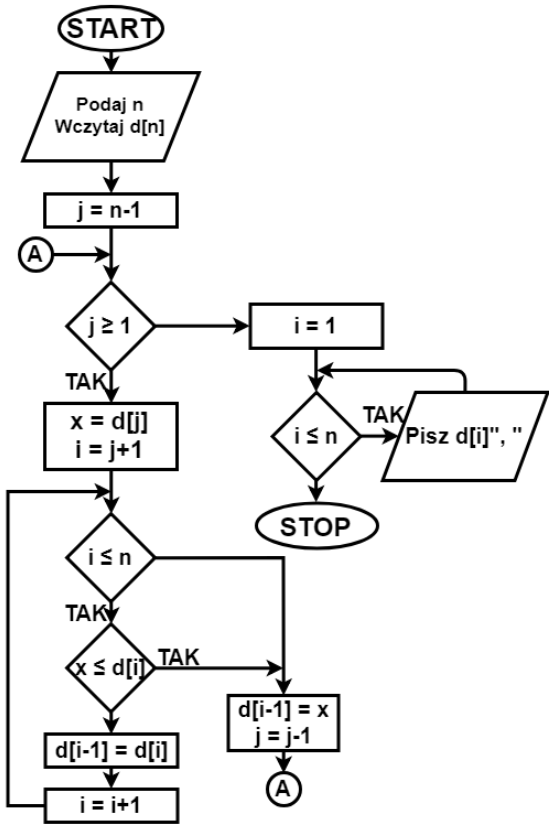
h. Rekord/Struktura

- Zawiera w sobie zmienne dowolnego typu. Pozwala na przechowywanie powiązanych ze sobą logicznie danych. Typowy dla baz danych

7. **Opisz sortowanie przez wstawianie (insertion sort)**

Sortowanie jest podobne do układania kart pobieranych z talii. Bierzemy pierwszą kartę z talii, pobieramy kolejne aż do wyczerpania talii. Dwie pętle: główna (zewnętrzna) symuluje pobieranie kart. Sortująca (wewnętrzna) szuka dla pobranego elementu miejsca w liście. Liczba porównań:

K01:	**Podaj** n
K02:	**Dla** $j = n - 1, n - 2, ..., 1$: **wykonuj** K03...K05
K03:	$x = d[j]; \ i = j + 1;$
K04:	**Dopóki** ($i \leq n$) $\wedge$ ($x >$ d[i]): **wykonuj** d[$i - 1$] = d[i]; $i = i + 1;$
K05:	d[$i - 1$] = x;
K06:	i=1;
K07:	**Dopóki** (i ≤ n) Pisz d[i]", "; i++;
K08:	**Zakończ**



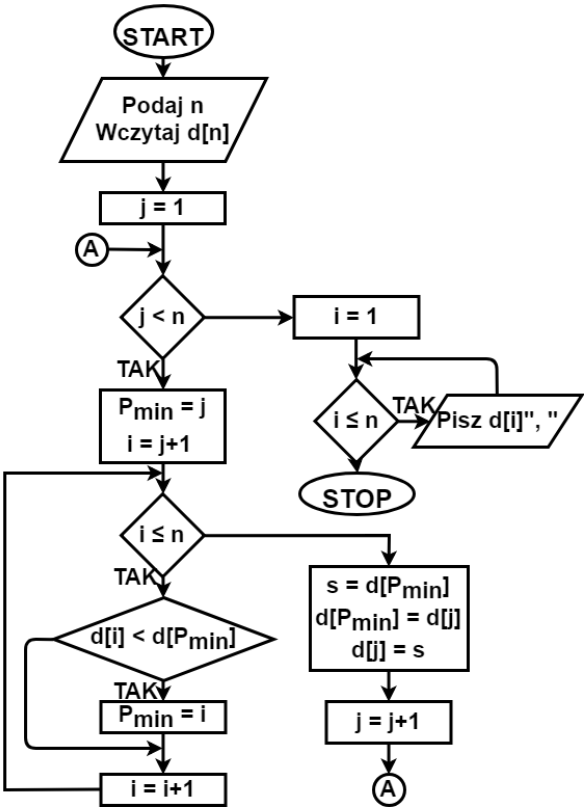
8. **Opisz sortowanie przez wybór (selection sort)**

Szukamy w zbiorze elementu najmniejszego i wymieniamy go z elementem na pierwszej pozycji. Znów wyszukujemy element najmniejszy i zamieniamy go z elementem na drugiej pozycji. Procedurę kontynuujemy dla pozostałych elementów dotąd, aż wszystkie będą posortowane.

K01:	**Podaj** n
K02:	**Dla** j = 1, 2, ..., n - 1: **wykonuj** K03...K05
K03:	$p_{min} = j$
K04:	**Dla** $i = j + 1,\ j + 2, ..., n$: **jeśli** d[$i$] < d[$p_{min}$], **to** $p_{min} = i$

K05:	s = d[p_{min}];
	d[p_{min}] = d[j];
	d[j] = s;

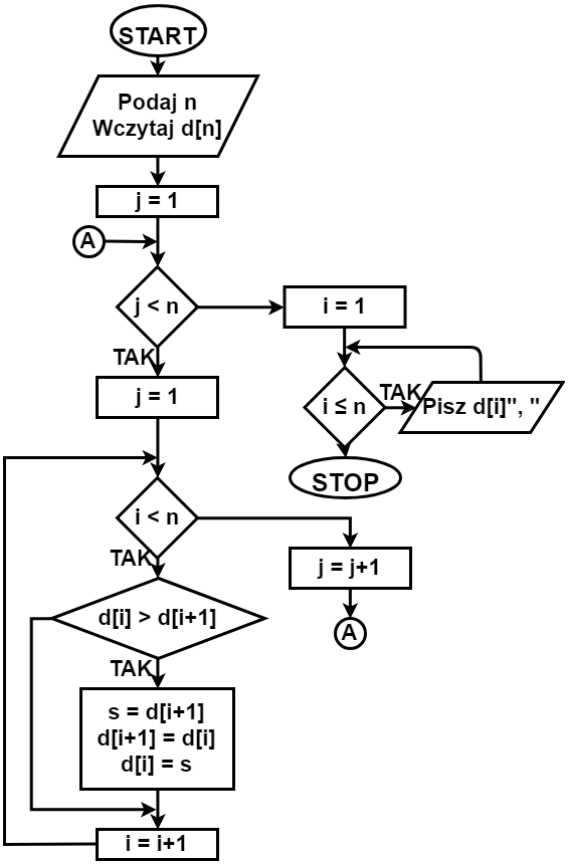
K06:	i = 1;
K07:	**Dopóki** (i ≤ n) Pisz d[i]", "; i++;
K08:	**Zakończ**



9. **Opisz sortowanie przez zamianę na przykładzie sortowania bąbelkowego/ (exchange sort)**

Zasada działania opiera się na cyklicznym porównywaniu par sąsiadujących elementów i zamianie ich kolejności w przypadku niespełnienia kryterium porządkowego zbioru. Operację wykonujemy, aż cały zbiór zostanie posortowany.

K01:	**Podaj** n
K02:	**Dla** j = 1,2,...,n - 1: **wykonuj** K03
K03:	**Dla** i = 1,2,...,n - 1: **jeśli** d[i] > d[i + 1], **to** s = d[i+1]; d[i+1] = d[i]; d[i] = s;
K04:	i = 1;
K05:	**Dopóki** (i ≤ n) Pisz d[i]", "; i++;
K06:	**Zakończ**



10. Opisz sortowanie Shella

Shell zauważył, że sortowanie przez wstawianie ma lepszą wydajność, gdy pracuje na zbiorze względnie posortowanym.

Sortowany zbiór dzielimy na podzbiory, których elementy są odległe od siebie w sortowanym zbiorze o pewien odstęp h. Każdy z tych podzbiorów sortujemy algorytmem przez wstawianie. Następnie odstęp zmniejszamy. Powoduje to powstanie nowych podzbiorów (będzie ich już mniej). Sortowanie powtarzamy i znów zmniejszamy odstęp, aż osiągnie on wartość 1. Wtedy sortujemy już normalnie za pomocą sortowania przez wstawianie.

Dane wejściowe

- n - liczba elementów w sortowanym zbiorze, n N
- d[ ] - zbiór n-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do n.

Dane wyjściowe

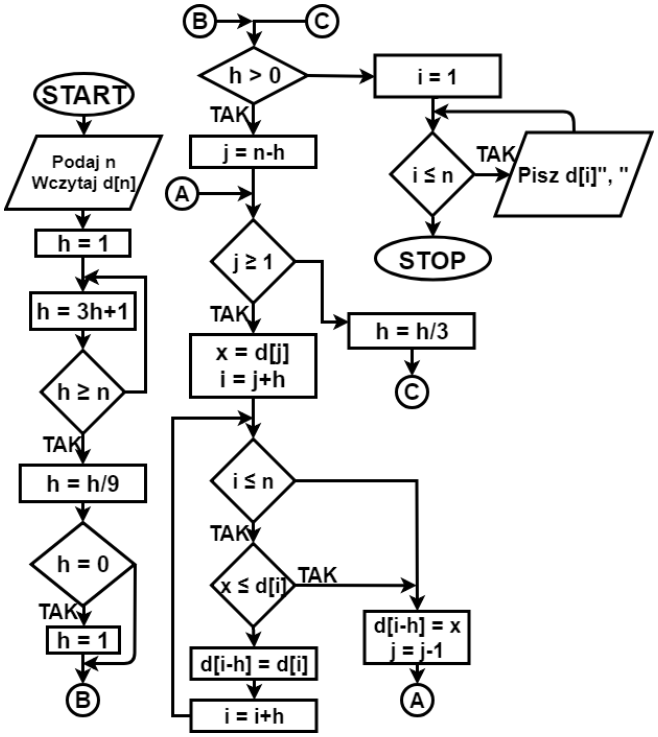
- d[ ] - posortowany zbiór n-elementowy. Elementy zbioru mają indeksy od 1 do n.

Zmienne pomocnicze

- i - indeks elementu listy uporządkowanej, i N
- j - zmienna sterująca pętli, j N
- h - odstęp pomiędzy kolejnymi elementami podzbiorów, h N
- x - zawiera wybrany ze zbioru element

K01:	**Podaj** n
K02:	h = 1
K03:	**Powtarzaj** h = 3h + 1, **aż** $h \geq n$
K04:	h = h/9
K05:	**Jeśli** h = 0, **to** h = 1

K06:	**Dopóki** $h > 0$: **wykonuj** K07...K11
K07:	**Dla** $j = n - h, n - h - 1, ..., 1$: **wykonuj** K08...K10
K08:	$x = d[j]; \quad i = j + h$
K09:	**Dopóki** $(i \leq n)$ i $(x > d[i])$: **wykonuj** $d[i - h] = d[i]; \quad i = i + h$
K10:	$d[i - h] = x$
K11:	$h = h/3$
K12:	**Dopóki** (i ≤ n) Pisz d[i]", "; i++;
K13:	**Zakończ**



11. Opisz Quick Sort

Algorytm sortowania szybkiego opiera się na strategii "dziel i zwyciężaj, którą możemy krótko scharakteryzować w trzech punktach:

- DZIEL - problem główny zostaje podzielony na podproblemy
- ZWYCIĘŻAJ - znajdujemy rozwiązanie podproblemów
- POŁĄCZ - rozwiązania podproblemów zostają połączone w rozwiązanie problemu głównego

Idea sortowania szybkiego jest następująca:

DZIEL : najpierw sortowany zbiór dzielimy na dwie części w taki sposób, aby wszystkie elementy leżące w pierwszej części (zwanej lewą partycją) były mniejsze lub równe od wszystkich elementów drugiej części zbioru (zwanej prawą partycją).

ZWYCIĘŻAJ : każdą z partycji sortujemy rekurencyjnie tym samym algorytmem.

POŁĄCZ : połączenie tych dwóch partycji w jeden zbiór daje w wyniku zbiór posortowany.
Do utworzenia partycji musimy ze zbioru wybrać jeden z elementów, który nazwiemy piwotem. W lewej partycji znajdą się wszystkie elementy nie większe od piwotu, a w prawej partycji umieścimy wszystkie elementy niemniejsze od piwotu. Położenie elementów równych nie wpływa na proces sortowania, zatem mogą one występować w obu partycjach.

Tworzenie partycji:

piwot = d[(lewy + prawy) **div** 2]

piwot - element podziałowy
d[ ] - dzielony zbiór
lewy - indeks pierwszego elementu
prawy - indeks ostatniego elementu

Dzielenie na partycje polega na umieszczeniu dwóch wskaźników na początku zbioru - i oraz j. Wskaźnik i przebiega przez zbiór poszukując wartości mniejszych od piwotu. Po znalezieniu takiej wartości jest ona wymieniana z elementem na pozycji j. Po tej operacji wskaźnik j jest przesuwany na następną pozycję. Wskaźnik j zapamiętuje pozycję, na którą trafi następny element oraz na końcu wskazuje miejsce, gdzie znajdzie się piwot. W trakcie podziału piwot jest bezpiecznie przechowywany na ostatniej pozycji w zbiorze.

Specyfikacja problemu:

Sortuj_szybko(lewy, prawy)

Dane wejściowe:

d[]- Zbiór zawierający elementy do posortowania. Zakres indeksów elementów jest dowolny.

lewy- indeks pierwszego elementu w zbiorze, $lewy \in C$

prawy- indeks ostatniego elementu w zbiorze, $prawy \in C$

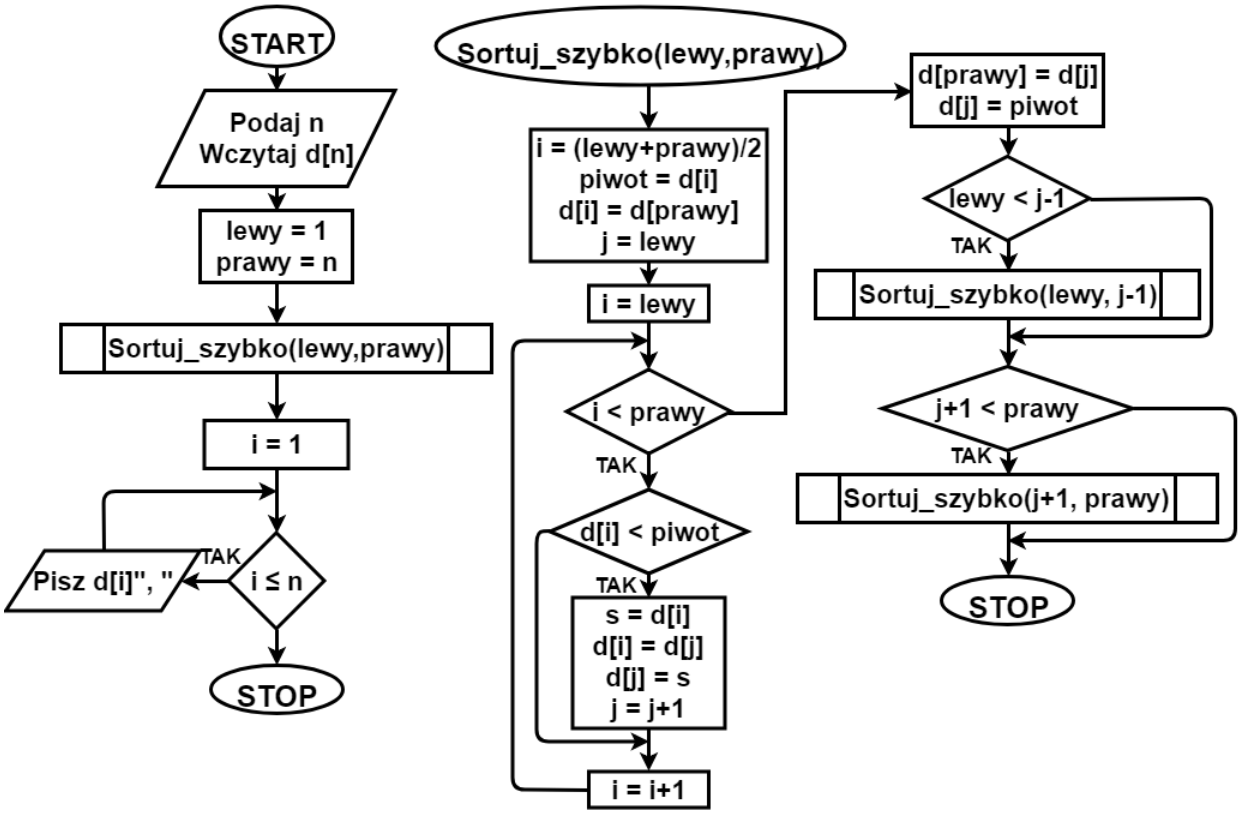
Dane wyjściowe:

d[]- Zbiór zawierający elementy posortowane rosnąco

Zmienne pomocnicze:

piwot- element podziałowy

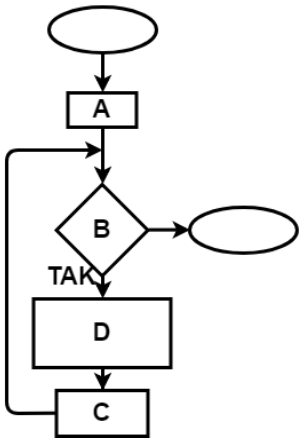
i, j- indeksy, $i, j \in C$



K01:	**Podaj** n
K02:	i = (lewy + prawy)/2
K03:	*piwot* = d[*i*]; d[*i*] = d[*prawy*]; *j* = *lewy*;
K04:	**Dla** *i* = *lewy*, *lewy* + 1, ..., *prawy* - 1: **wykonuj** K04...K05
K05:	**Jeśli** d[*i*] ≥ *piwot*, **to wykonaj kolejny obieg pętli** K03
K06:	s = d[*i*]; d[*i*] = d[j]; d[j] = s; *j* = *j* + 1;
K07:	d[*prawy*] = d[*j*]; d[*j*] = *piwot*
K08:	**Jeśli** *lewy* < *j* - 1, **to Sortuj_szybko**(*lewy*, *j* - 1)
K09:	**Jeśli** *j* + 1 < *prawy*, **to Sortuj_szybko**(*j* + 1, *prawy*)
K10:	**Dopóki** (i ≤ n) Pisz d[i]", "; i++;
K11:	**Zakończ**

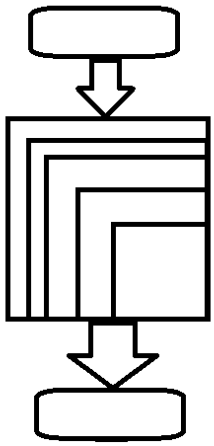
12. Algorytmy iteracyjne

Algorytm iteracyjny - algorytm, który uzyskuje wynik przez iterację, czyli powtarzanie danej operacji początkowo określoną liczbę razy lub aż do spełnienia określonego warunku.



13. Algorytmy rekurencyjne

Algorytm rekurencyjny - polega na tym, że obiekt składa się częściowo z samego siebie (sam się wywołuje).



14. Algorytmy na grafach

Graf to struktura (G=(V, E)) składająca się z węzłów (wierzchołków, oznaczonych przez V) wzajemnie połączonych za pomocą krawędzi (oznaczonych przez E)

Grafy dzielimy na skierowane i nieskierowane.

- Krawędź jest incydentna, jeżeli łączy się z wierzchołkiem.
- Pętla własna to krawędź łącząca wierzchołek z samym sobą.
- Stopień wierzchołka w grafie nieskierowanym to liczba incydentnych z nim krawędzi.
- Ścieżką lub drogą w grafie skierowanym nazywamy sekwencję krawędzi taką, że koniec jednej krawędzi jest początkiem następnej
- Długością ścieżki nazywamy liczbę należących do niej krawędzi.

15. Problemy trudne: Wieże Hanoi

Wieże Hanoi są łamigłówką zbudowaną z deseczki, na której znajdują się trzy patyczki w równych odstępach. Na pierwszym patyczku nasunięte są drewniane krążki o coraz mniejszych średnicach.2

Celem jest przesunięcie wszystkich krążków z patyczka 1 na patyczek 2 według następujących reguł:

- Naraz można przenosić tylko po jednym krążku,

- Krążek zdjęty z patyczka musi być nasunięty na jeden z pozostałych dwóch patyczków,
- Krążek można nasunąć na patyczek, jeśli jest on pusty lub zawiera krążek większy

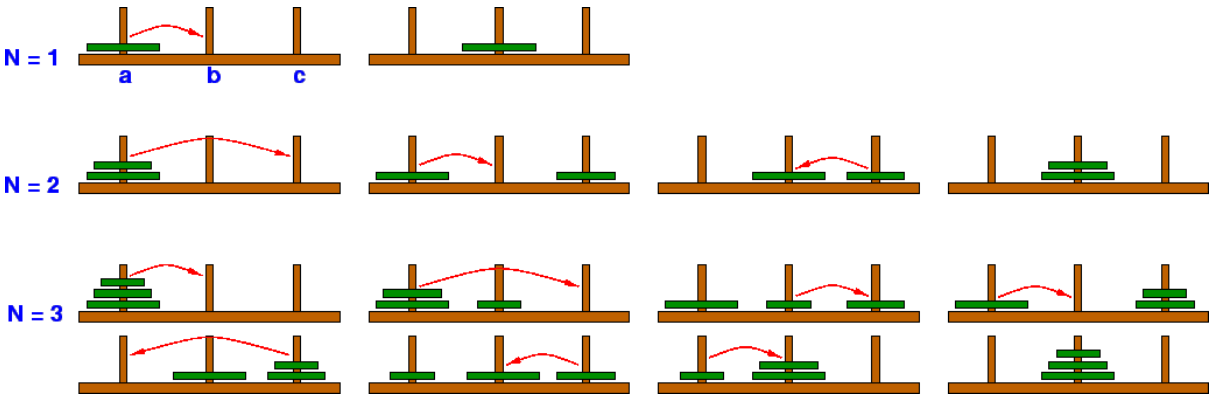
Algorytm rekurencyjny:

- Przenieś (rekurencyjnie) n-1 krążków ze słupka A na słupek B posługując się słupkiem C,
- Przenieś jeden krążek ze słupka A na słupek C,
- Przenieś (rekurencyjnie) n-1 krążków ze słupka B na słupek C posługując się słupkiem A.

Algorytm iteracyjny:

- Przenieś najmniejszy krążek na kolejny* słupek
- Wykonaj jedyny możliwy do wykonania ruch, nie zmieniając położenia krążka najmniejszego
- Powtarzaj punkty 1 i 2, aż do odpowiedniego ułożenia wszystkich krążków.

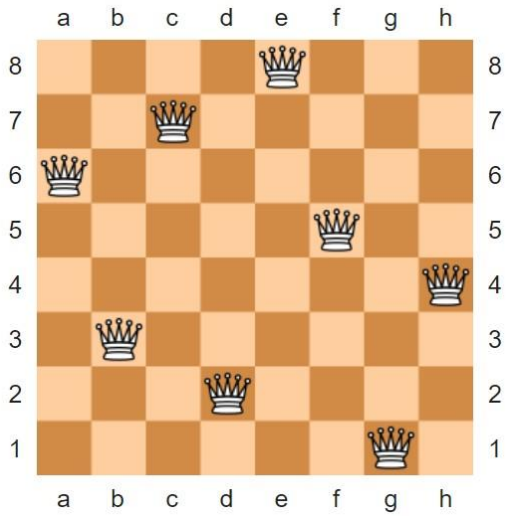
* - Kolejny słupek wyznaczamy w zależności od liczby krążków, jeżeli liczba krążków jest parzysta, kolejnym słupkiem jest tej po prawej stronie (jeżeli dojdziemy do słupka C, w następnym ruchu używamy słupka A), jeżeli nieparzysta, ten po lewej (odwrotnie).



16. Problem 8 hetmanów

Problem 8 hetmanów polega na ustawieniu na szachownicy tychże figur tak, aby żadna z nich nie mogła zabić innej. Królowa, zwana hetmanem, ustawiona na szachownicy atakuje wszystkie pola w poziomie, w pionie i po skosie.

Przykład rozwiązania:



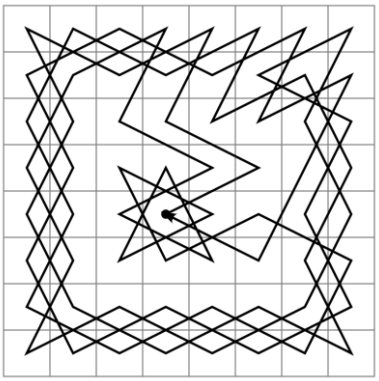
Aby rozwiązać to zadanie można posłużyć się poniższym algorytmem:

1) Stawiamy hetmana w pierwszym wierszu w pierwszej kolumnie.
2) Przechodzimy do drugiego wiersza i szukamy pierwszej, wolnej kolumny

3) Podobnie postępujemy z kolejnymi wierszami.
4) W momencie, gdy któregoś hetmana nie da się postawić, (w każdej z 8 kolumn hetman będzie atakowany) to cofamy się o jeden wiersz i przesuwamy hetmana na kolejne wolne pole. Jeśli takie nie istnieje, to znowu się cofamy itd. (Ta cecha sprawia, że algorytm ustawiający hetmanów na szachownicy jest przykładem rekurencji z powrotami).
5) Gdy udało się szczęśliwie postawić 8 hetmana oznacza to, że znaleźliśmy rozwiązanie.
6) Jeśli okaże się, że hetmana z pierwszego wiersza nie ma już, gdzie postawić, to znaczy, że wszystkie możliwe kombinacje zostały już sprawdzone.

17. Problem skoczka szachowego

Ruszając z wyznaczonego pola na szachownicy, poruszając się zgodnie z regułą ruchu skoczka należy odwiedzić wszystkie pola szachownicy tak aby każde pole zostało odwiedzone dokładnie jeden raz. Odwiedzając kolejne pola skoczek numeruje je.



Zamknięta ścieżka dla planszy 8x8

Skoczek z bieżącego pola (które zawsze na początku numeruje) próbuje przejść na kolejne pole. Jeśli może to uczynić to skacze – wtedy też wywoływana jest rekurencja (gdyż mamy do czynienia z takim samym problemem jednak o powiększonym zbiorze pól odwiedzonych). Jeśli natomiast nie jest dopuszczalny dany wariant ruchu konik usiłuje wykonać kolejny wariant. Jeśli żaden z dopuszczalnych wariantów ruchu nie może zostać wykonany konik wie, że ta droga nie prowadzi do celu, dlatego też wycofuje się z ostatnio wykonanego ruchu. Dodatkowo po powrocie z rekurencyjnego wywołania funkcji sprawdzamy, czy znaleziono już rozwiązanie - wyjście z funkcji w takim wypadku powoduje, że kończymy obliczenia po znalezieniu pierwszego rozwiązania, jeżeli usuniemy ten warunek wówczas szukać będziemy wszystkich możliwych rozwiązań problemu.

18. Metody zapisywania grafów w pamięci

Lista sąsiedztwa – polega na przypisaniu każdemu wierzchołkowi grafu listy sąsiadujących z nim wierzchołków. Kolejność wierzchołków na liście związanej z danym wierzchołkiem może być dowolna. Zapotrzebowanie na pamięć list sąsiedztwa jest proporcjonalna do sumy liczby wierzchołków i liczby krawędzi. Sprawdzenie czy istnieje krawędź **(u, v)** wymaga przeglądania listy sąsiedztwa wierzchołka **u** w poszukiwaniu wierzchołka **v.**

- 1: 2,3,5
- 2: 1,3,4
- 3: 1,2,4
- 4: 2,3,5
- 5: 4,3

Macierz sąsiedztwa – jest tablicą kwadratową o rozmiarze V*V, którą wypełnia się **0**, jeśli dwa wierzchołki nie są połączone krawędzią lub **1**, jeśli dwa wierzchołki są połączone. Taka reprezentacja jest wygodna w obliczeniach, łatwo jest sprawić czy pomiędzy dwoma wierzchołkami istnieje krawędź. Podstawową wadą jest wysoka złożoność pamięciowa.

	1	**2**	**3**	**4**	**5**
1	0	1	1	0	1
2	1	0	1	1	1
3	1	1	0	1	0
4	0	0	0	1	1
5	1	0	0	1	1

Lista krawędzi – lista, na której przechowywane są wszystkie krawędzie występujące w grafie. W przypadku **grafu nieskierowanego krawędź z u do v należy zapisać dwukrotnie**: u-v oraz v-u. Jest to najprostsza reprezentacja jednak mało efektywna, jako że wykonanie jakiejkolwiek operacji wymaga przeszukania całej listy.

- 1-2
- 2-4
- 2-5
- 3-1
- 3-2
- 4-3
- 5-1

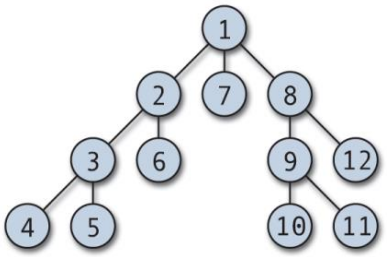
Macierz incydencji – jest tablicą o rozmiarze V*E. Składa się z E kolumn i V wierszy, jeśli krawędź wychodzi z danego wierzchołka to w odpowiedniej kolumnie wypisuje się **(-1)**, jeśli do niego wchodzi **(+1)**, jeśli wierzchołek nie należy do krawędzi **(0)**, jeśli jest to pętla własna **(2)**. Jest to reprezentacja o bardzo dużej złożoności pamięciowej.

	1	**2**	**3**	**4**	**5**
1	2	1	1	0	1
2	1	0	0	1	-1
3	1	-1	0	1	0
4	0	0	2	1	1

5	1	0	0	1	1

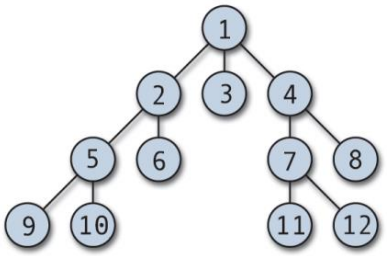
19. Metody przeszukiwania grafu

Przeszukiwanie grafu – (inaczej przechodzenie grafu) to odwiedzenie w sposób usystematyzowany wszystkich wierzchołków grafu w celu zebrania potrzebnych informacji. Powszechnie stosuje się dwie podstawowe metody przeszukiwania grafów: przeszukiwanie wszerz (BFS); przeszukiwanie w głąb (DFS).



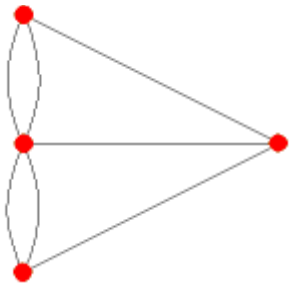
Przeszukiwanie w głąb – depth first serach (DFS). Nazwa algorytmu wywodzi się stąd, że przechodzi on wybraną ścieżką aż do jej całkowitego wyczerpania. W kolejnych wywołaniach rekurencyjnych przeszukiwanie zagłębia się coraz bardziej, jak to możliwe.

Metoda polega na przypisaniu każdemu wierzchołkowi **statusu nieodwiedzony**, wybraniu pewnego wierzchołka jako startowego, nadaniu mu status odwiedzony, a następnie na rekurencyjnym zastosowaniu tej metody do wszystkich następników wierzchołka. Jeśli wszystkie wierzchołki grafu zostaną w tej sposób odwiedzone, przeszukiwanie zostaje zakończone, a jeżeli w grafie pozostaną wierzchołki **nieodwiedzone**, wybiera się którykolwiek z nich jako startowy i powtarza od niego przeszukiwanie.



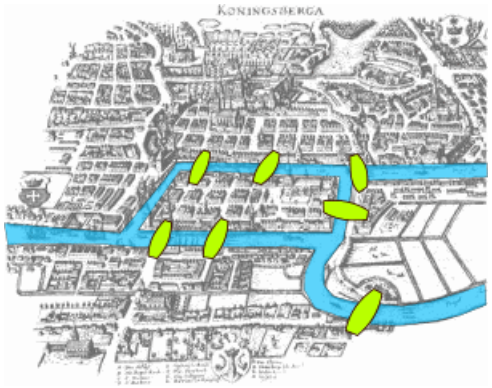
Przeszukiwanie wszerz – breadth first serach (BFS). Procedura porusza się wszerz grafu, aby odwiedzić wszystkich nieodwiedzonych sąsiadów.

20. Problemy trudne: ścieżki Hamiltona (ścieżki Eulera)



Graf Eulera – to taki graf, w którym można skonstruować cykl Eulera, czyli taki cykl, który przechodzi przez każdą jego krawędź dokładnie jeden raz i wraca do punktu wyjściowego.
Aby możliwe było zbudowanie takiej ścieżki, każdy wierzchołek grafu z wyjątkiem najwyżej dwóch, musi mieć parzysty stopień (łączyć się z parzystą liczbą krawędzi).

Czym sie różnia Eulera i Hamiltona



Mapa królewieckich mostów z czasów Eulera. Dodatkowo wyróżniona jest rzeka i mosty.

Różnice między Ścieżkami Eulera, a Hamiltona.

Ścieżka Eulera jest ścieżką, która przecina każdą krawędź dokładnie raz bez powtarzania, jeśli kończy się na początkowym wierzchołku, to jest to cykl Eulera.

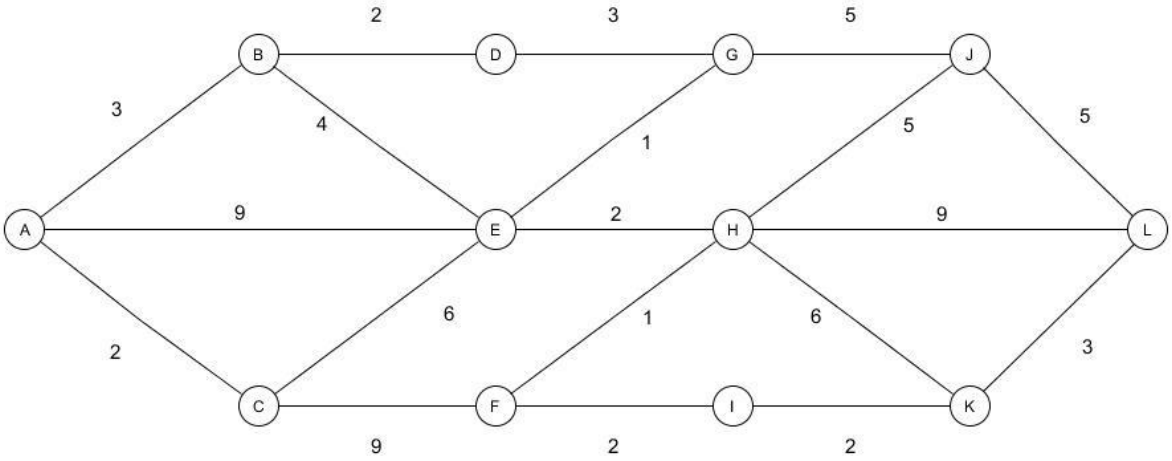
Ścieżka Hamiltona przechodzi przez każdy wierzchołek (nie każdą krawędź), dokładnie jeden raz, jeśli kończy się na początkowym wierzchołku, to jest to cykl Hamiltona.

W ścieżce Eulera możesz przejść przez wierzchołek więcej niż jeden raz.

Na ścieżce Hamiltona nie można przejść przez wszystkie krawędzie.

21. Algorytm Dijkstry

Algorytm ten pozwala znaleźć koszty dojścia od wierzchołka startowego v do każdego innego wierzchołka w grafie (o ile istnieje odpowiednia ścieżka). Dodatkowo wyznacza on poszczególne ścieżki.



Przypuśćmy, że dany graf to „mapa", na której litery od A do L oznaczają miasta połączone drogami. Jaka jest długość najkrótszej drogi z miasta A(źródła) do miasta L(ujścia) jeśli długości dróg są takie jak oznaczone na mapie (liczby na grafie nie muszą oznaczać długości dróg, ale mogą wskazywać czas podróży lub koszt przejazdu).
Można zauważyć, że łatwo znaleźć ograniczenia górne rozwiązania naszego zadania, biorąc dowolną drogę z A do L i obliczając jej długość np.:
A -> B -> D -> G ->J -> L ma całkowitą długość równą 18.

Czyli najkrótsza droga nie może przekroczyć 18. W tego typu zadaniach naszą „mapę" możemy traktować jako graf spójny, w którym każdej krawędzi przypisano pewną liczbę ujemną. Taki graf nazywamy **GRAFEM Z WAGAMI**, a liczbę przypisaną do krawędzi **e** nazywamy **WAGĄ** tej krawędzi i oznaczamy symbolem **w(e)**. Czyli nasze zadanie polega na poszukiwaniu drogi z A do L mającej najmniejszą całkowitą wagę.

Zauważymy, że jeżeli mamy dany graf z wagami, w którym waga każdej krawędzi wynosi 1 to zadanie sprowadza się do znalezienia liczby krawędzi w najkrótszej drodze z A do L. Istnieje wiele metod rozwiązania tego zadania.

Jedna z nich polega na zrobieniu modelu mapy z powiązanych ze sobą kawałków sznurka, których długości są proporcjonalne do długości dróg. Aby znaleźć najkrótszą drogę, wystarczy chwycić węzełki odpowiadające miastom A i L i naciągnąć sznurki

Druga – „bańki mydlane"

Istnieje również bardziej matematyczny sposób na rozwiązanie powyższego problemu. Polega on na tym, by przesuwać się wzdłuż grafu z lewa na prawo, przypisując każdemu wierzchołkowi V liczbę I(V) wskazującą najkrótszą odległość od wierzchołka A do wierzchołka V. Tzn., że jeśli dojdziemy do wierzchołka takiego jak K na rysunku 8.1, to przyjmiemy jako I(K) albo I(H)+6 albo I(I)+2 w zależności od tego, która z tych dwóch liczb jest mniejsza.

Aby wykorzystać ten algorytm, przypiszemy najpierw wierzchołkowi A liczbę 0 oraz wierzchołkom B, E i C tymczasowo przypiszemy liczby I(A)+3, I(A)+9 oraz I(A)+2, to znaczy liczby 3,9 i 2. Weźmiemy teraz najmniejsza z tych liczb i przyjmiemy I(C)=2. Wierzchołkowi C przypisaliśmy na stałe liczbę 2.

Patrzymy teraz na wierzchołki sąsiadujące z C.

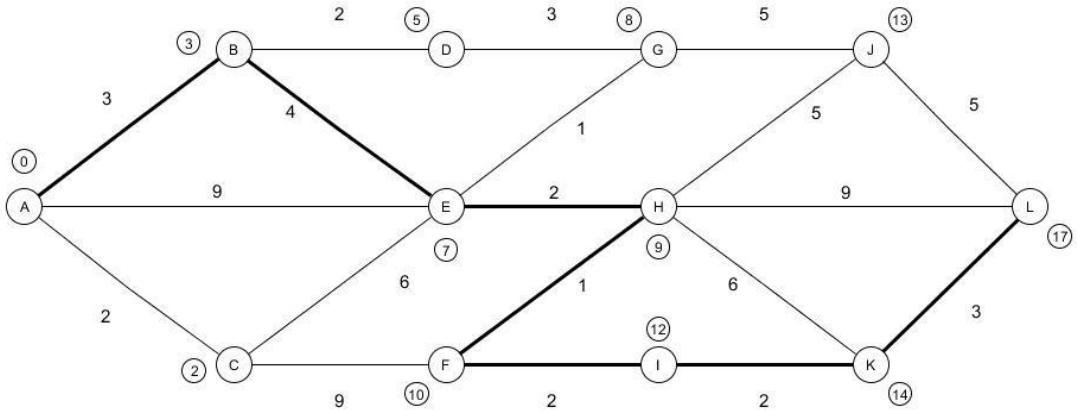
Przypisujemy tymczasowo wierzchołkowi F liczbę I(C)+9 = 11 oraz możemy obniżyć liczbę przypisaną tymczasowo wierzchołkowi E do poziomu I(C)+6 = 8. Najmniejszą liczbą przyporządkowaną tymczasowo jest teraz 3 (i jest ona przypisana wierzchołkowi B), więc przyjmujemy I(B)=3 i uznajemy, że wartość została przypisana wierzchołkowi B na stałe.

Teraz patrzymy na wierzchołki sąsiadujące z B.

Wierzchołkowi D przypisujemy tymczasowo liczbę I(B)+2 = 5 i obniżamy liczbę tymczasowo przypisaną wierzchołkowi E do wartości I(B)+4 = 7. Najmniejszą liczbą przypisaną tymczasowo jest teraz 5 (wierzchołkowi D). A więc przyjmujemy I(D)= 5 i tę liczbę przypisujemy D na stałe.

Kontynuując to postępowanie otrzymamy liczby przypisane na stałe wierzchołkiem: I(E)=7, I(G)=8, I(H)=9, I(F)=10, I(I)=12, I(J)=13, I(K)=14, I(L)=17.

Wynika stąd, że najkrótsza droga z A do L ma długość 17. Jest ona pokazana na rysunku poniżej, na którym w kółeczkach pokazane są liczby przypisane wierzchołkom na stałe:



Słowny zapis zasady algorytmu Dijkstry:

1) Przypisujemy wierzchołkowi **s** (źródłu) stałą cechę równą 0, a wszystkim pozostałym cechy tymczasowe równe ∞.
2) Każdemu bezpośredniemu następnikowi **v** wierzchołka **s** przypisujemy tymczasową cechę równą wadze łuku/krawędzi (s, v).
3) Wybieramy wierzchołek **x**, który ma najmniejszą cechę tymczasową (jest wierzchołkiem najbliższym wierzchołka s), a następnie jego cechę.
4) Przeglądamy wszystkie bezpośrednie następniki wierzchołka **x** i zmniejszamy ich tymczasowe, o ile są równe ∞, bądź droga z **s** do któregokolwiek z nich prowadząca przez **x** jest krótsza od drogi omijającej **x**.

5) Znajdujemy wierzchołek o najmniejszej cesze tymczasowej np. **y** i cechę tę zmieniamy na stałą. Następnie powtarzamy czynności opisane w punkcie 4 względem wierzchołka **y** itd.

Algorytm kończy działanie, gdy cecha wierzchołka t (ujścia) zostanie ustalona.

22. Problemy trudne: Problem komiwojażera

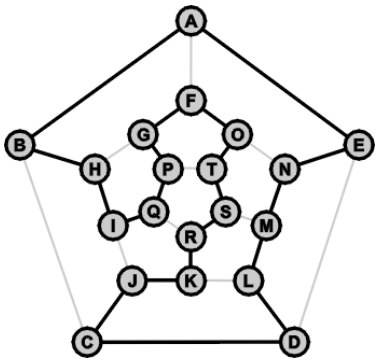
Problem: W danym grafie znaleźć ścieżkę o najmniejszej sumie wag krawędzi, która przechodzi dokładnie jeden raz przez każdy wierzchołek i wraca do wierzchołka startowego.

Wyobraźmy sobie komiwojażera, który podróżuje od miasta do miasta, sprzedając swoje produkty lub zawierając różne oferty handlowe. Wyrusza z miasta rodzinnego, po czym jego trasa przebiega dokładnie jeden raz przez każde z miast. Na koniec komiwojażer powraca do swojego miasta rodzinnego. Z oczywistych powodów chce, aby trasa podróży była najkrótszą ze wszystkich możliwych tras. W ten sposób powstaje problem wędrującego komiwojażera.

W terminologii grafów miasta są wierzchołkami grafu, a trasy pomiędzy nimi to krawędzie z wagami. Waga krawędzi może odpowiadać odległości pomiędzy miastami połączonymi tą krawędzią, czasowi podróży lub kosztom przejazdu - zależy, co chcemy w podróżny komiwojażera zminimalizować. Trasa komiwojażera jest cyklem przechodzącym przez każdy wierzchołek grafu dokładnie jeden raz - jest to zatem cykl Hamiltona.

Cykl Hamiltona - jest cyklem, który odwiedza każdy wierzchołek grafu dokładnie jeden raz. Znalezienie cyklu lub ścieżki Hamiltona w grafie jest bardzo trudne obliczeniowo. Problem ten zalicza się to tzw. problemów NP zupełnych, co oznacza, że dla dużej liczby wierzchołków jest on praktycznie nierozwiązywalny w sensownym czasie.

Cykl Hamiltona:
A > B > H > I > Q > P > G > F > O > T > S > R > K > J > C > D > L > M > N > E > A



23. Problemy trudne: Problem trwałego małżeństwa (kojarzenie małżeństw)

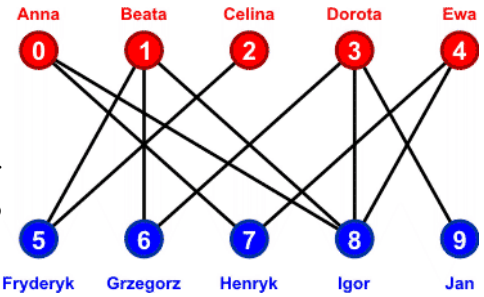
Problem: Dobrać w pary małżeńskie n panien i n kawalerów, gdzie każda z panien jest skłonna wyjść za mąż za jednego z k kawalerów.

Niech wierzchołkami naszego grafu będą panny i kawalerowie. Krawędzie natomiast niech określają preferencje poszczególnych panien, czyli prowadzą do kawalerów, za których panny są skłonne wyjść za mąż. Ponieważ związki są **heteroseksualne**, od razu możemy stwierdzić, iż powstaje

graf dwudzielny – jedną klasą wierzchołków są panny, a drugą kawalerowie. Krawędzie łączą tylko panny z kawalerami – nie ma połączeń pomiędzy pannami ani pomiędzy kawalerami.

Preferencje każdej z panien są następujące:

Anna	→ Henryk, Igor	0 → 7, 8
Beata	→ Fryderyk, Grzegorz, Igor	1 → 5, 6, 8
Celina	→ Fryderyk	2 → 5
Dorota	→ Grzegorz, Igor, Jan	3 → 6, 8, 9
Ewa	→ Henryk, Igor	4 → 7, 8



Problem małżeństw jest rozwiązywalny, jeśli każdy podzbiór k panien, akceptuje jako przyszłych mężów co najmniej k kawalerów, gdzie $k \leq n$.

Okazuje się, że jest to warunek konieczny i wystarczający. Warunek ten oznacza, iż dla każdej grupy panien liczba kawalerów jest wystarczająca do skojarzenia wszystkich małżeństw. Problemem pozostaje znalezienie takiego skojarzenia.

Przejdź przez kolejne wierzchołki grafu. Jeśli wierzchołek jest nieskojarzoną panną, to spróbuj utworzyć ścieżkę rozszerzającą prowadzącą do pierwszego napotkanego wolnego wierzchołka kawalera. Ścieżka rozszerzająca jest ścieżką naprzemienną, która zawiera na przemian krawędzie wolne i skojarzone. Jeśli znajdziemy ścieżkę rozszerzającą, to wszystkie krawędzie wolne zmieniamy na skojarzone, a wszystkie krawędzie skojarzone zamieniamy na wolne.

24. Rozstrzygalność – nierozstrzygalność problemu

Powiemy, że problem decyzyjny jest rozstrzygalny, jeśli istnieje algorytm, który dla dowolnych danych po skończonej liczbie kroków daje rozwiązanie problemu. W przeciwnym razie, tzn., jeśli taki algorytm nie istnieje, mówimy, że problem jest nierozstrzygalny.

25. Problem NP

Problemami klasy NP nazywamy problemy decyzyjne w których sprawdzenie poprawności określonego rozwiązania wymaga złożoności obliczeniowej co najmniej wielomianowej.

26. Problem P

Problemami klasy P nazywamy problemy decyzyjne w których znalezienie rozwiązania wymaga złożoności obliczeniowej wielomianowej. Problemy P należą do zbioru problemów NP. Z tego wynika, że:

- sprawdzenie poprawności rozwiązania danego problemu jest możliwe w czasie wielomianowym;
- znalezienie rozwiązania danego problemu jest możliwe również w czasie wielomianowym.

Różnica pomiędzy problemami P i NP polega na tym, że w przypadku **P** znalezienie rozwiązania ma mieć złożoność wielomianową, podczas gdy dla **NP** sprawdzenie podanego z zewnątrz rozwiązania ma mieć taką złożoność.

27. Problem NP-zupełny

Problemami klasy NP-zupełnej nazywamy problemy decyzyjne w których znalezienie rozwiązania nie jest możliwe w czasie wielomianowym. Problemy NP-zupełne należą do zbioru problemów NP jak również do zbioru problemów NP-trudnych. Z tego wynika, że:

- sprawdzenie poprawności rozwiązania danego problemu jest możliwe w czasie wielomianowym;
- znalezienie rozwiązania danego problemu nie jest możliwe w czasie wielomianowym.

28. Generowanie Permutacji

➢ Permutacją zbioru n-elementowego nazywamy każdy n-wyrazowy ciąg utworzony ze wszystkich elementów tego zbioru.

➢ Permutacja spełnia następujące warunki:
- Każda permutacja obejmuje wszystkie dane elementy,
- Istotna jest tylko kolejność elementów permutacji

➢ Z permutacjami zbioru mamy do czynienia wówczas, gdy porządkujemy element tego zbioru. Permutacja to każde ustawienie wszystkich elementów zbioru w dowolnej kolejności.

➢ Z trzech danych elementów a, b, c, można utworzyć następujące permutacje:

{a, b, c}, {a, c, b}, {b, a, c}, {b, c, a}, {c, a, b}, {c, b, a}

Liczba permutacji zbioru złożonego z n elementów jest równa N!

Pn = n!

➢ Permutacja z powtórzeniami rozpatruje przypadki, gdy ilość powtórzeń danego elementu jest ściśle określona.

➢ Permutacją z powtórzeniami zbioru n-elementowego, nazywamy każdy ciąg n-wyrazowy utworzony z elementów tego zbioru, wśród których pewne elementy powtarzają się odpowiednio n_1, n_2, ..., n_k razy.

➢ Jeśli spośród elementów a, b i c, element a weźmiemy dwa razy, element b jeden raz i element c jeden raz, możemy utworzyć permutacje z powtórzeniami:

{a, a, b, c}, {a, a, c, b}, {c, a, a, b} itd...

➢ Ciąg permutacji zbioru {1, 2, ..., n} jest uporządkowany leksykograficznie, jeżeli permutacje te traktowane jako reprezentacje liczb naturalnych występują w porządku rosnącym, np. 123, 132, 213, 231, 312, 321.

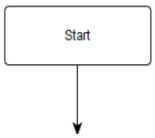
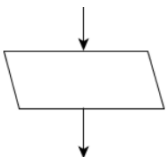
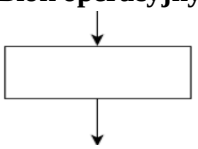
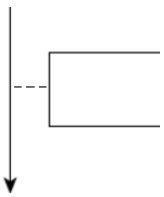
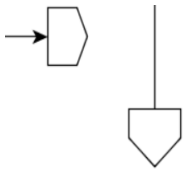
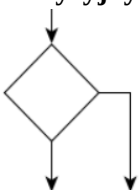
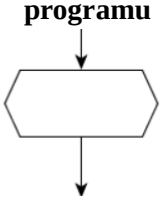
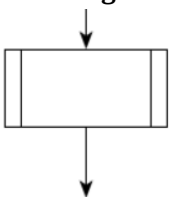
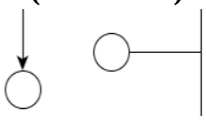
Algorytm generowania permutacji

- Pierwsza permutacja to $a_i = i$, dla $1 <= i <= n$.
- Aby wypisać następną (po (a1, ..., an)) permutację:
  - Znajdujemy największe j spełniające warunek $a_j < a_j + 1$
  - Jeżeli takiego j nie ma, to bieżąca permutacja jest ostatnia,
  - Jeżeli takie j istnieje, to zamieniamy a_j z najmniejszym a_k takim, że $a_k > a_j$ oraz $k > j$, a następnie odwracamy porządek elementów $a_j + 1, ..., a_n$.
- Algorytm wypisuje permutacje w porządku rosnącym, jeżeli potraktujemy permutacje jako liczby zapisane z bazą n + 1, a liczby 1, ..., n jako cyfry w tym systemie.

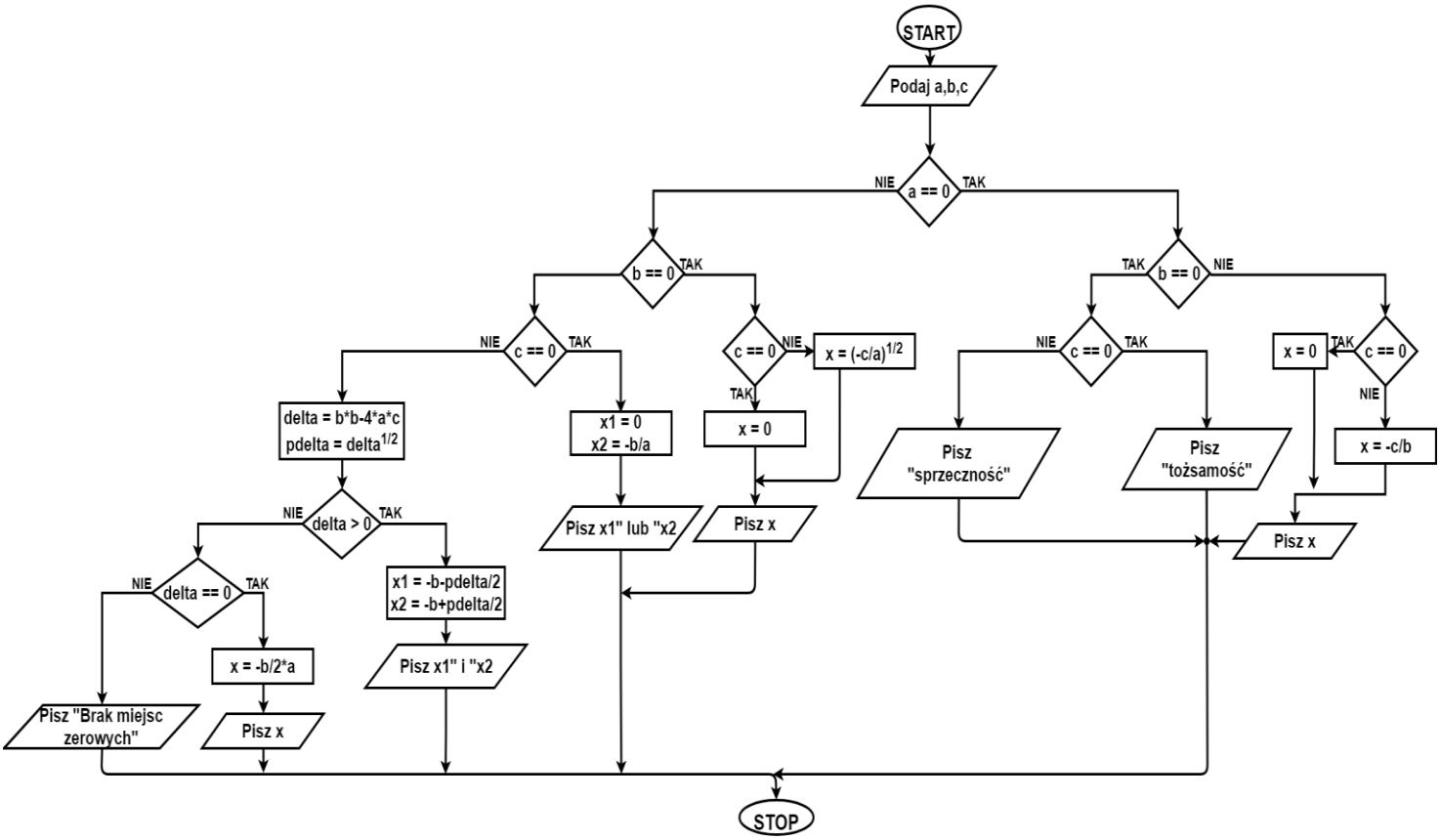
Przykład:

Bieżącą permutacją jest (436521).
Algorytm znajduje j = 2 i aj = 3. Wtedy ta permutacja jest ostatnią (największą) permutacją spośród permutacji zaczynających się od (43...), bo od pozycji trzeciej mamy ciąg malejący (... 6521) i jest to największy ciąg jaki można tu tworzyć z elementów 1, 2, 5, 6. Teraz powinny nastąpić permutacje zaczynające się od (45...) (czwórki na pierwszym miejscu nie zmieniamy, a trójka na drugim miejscu powinna być zamieniona przez następną spośród liczb stojących za nią, czyli przez 5). Pierwszą taką permutacją jest ta, w której pozostałe elementy rosną, czyli (451236).

Schematy Blokowe

Blok graniczny 	**Blok WE/WY** 	**Blok operacyjny** 	**Blok komentarza** 	**Łącznik zewnętrzny (na odrębnych stronach)** 
Blok decyzyjny 	**Blok wywołania programu** 	**Blok fragmentu** 	**Łącznik wewnętrzny (ta sama strona)** 	

Równanie Kwadratowe


START
Podaj a,b,c
a == 0
NIE
TAK
b == 0
TAK
b == 0
NIE
c == 0
NIE
TAK
c == 0
NIE
x = (-c/a)^1/2
TAK
c == 0
TAK
x = 0
NIE
delta = b*b-4*a*c
pdelta = delta^1/2
x1 = 0
x2 = -b/a
x = 0
Pisz
"sprzeczność"
Pisz
"tożsamość"
x = 0
c == 0
NIE
x = -c/b
NIE
delta > 0
TAK
Pisz x1" lub "x2
Pisz x
Pisz x
NIE
delta == 0
TAK
x1 = -b-pdelta/2
x2 = -b+pdelta/2
x = -b/2*a
Pisz x1" i "x2
Pisz "Brak miejsc
zerowych"
Pisz x
STOP