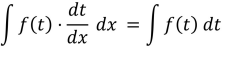
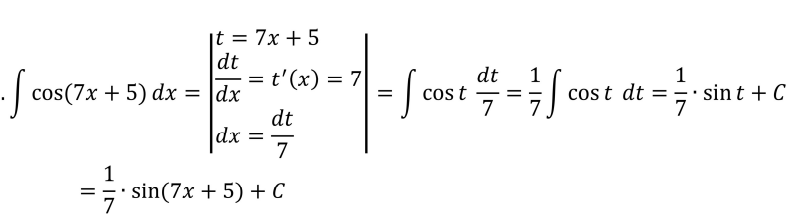
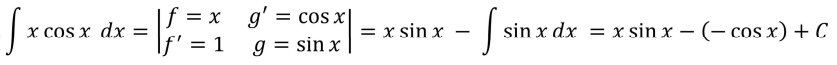
## Całka nieoznaczona, oznaczona, zastosowanie i techniki obliczania.

Pochodna – miara szybkości zmian wartości funkcji względem zmian jej argumentów

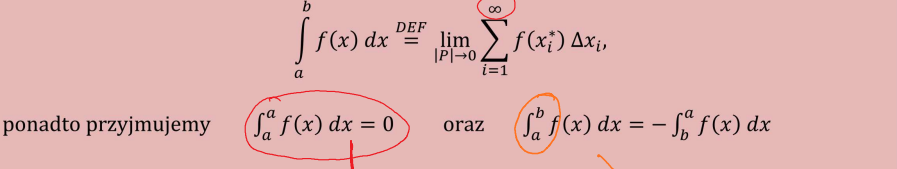
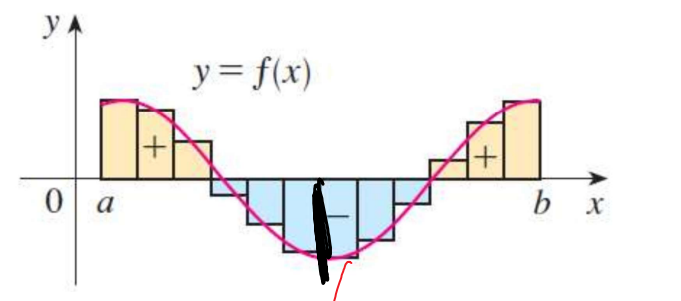
**Całka nieoznaczona** (antypochodna) – oznaczana literą F, to funkcja pierwotna funkcji f w przedziale I, jeśli  
   
Pochodna stałej to zero: (C)’ = 0  
Zapis całki dla funkcji f’(x):

**Metody całkowania:**

1. **Przez podstawianie** – jeżeli funkcja f: I -> R jest ciągła na przedziale I oraz funkcja t: J -> I ma *ciągłą pochodną* t’(x) na przedziale J, to:  
   ****  
   Ostateczny wzór na całkowanie przez podstawienie:  
   ****  
   Przykład:  
   ****  
   Jak to się robi:
   * Całka musi się składać z 2 połączony ze sobą funkcji
   * Wybieram funkcję wewnętrzną i liczę z niej pochodną
   * Dt/dx to ta pochodna – chcę wyznaczyć dx, żeby móc podstawić by mieć całkę od zmiennej t
   * Jak to zrobię, to liczę całkę normalnie (z karty wzorów)
   * Na koniec podstawiam pod t tę funkcję, która wstawiłem na początku
2. **Przez części** – Jeśli funkcja f i g mają ciągłe pochodne, to:  
   ****  
   Używana, gdy mamy iloczyn dwóch różnych funkcji – jedną z nich bierzemy jako f(x), a drugą jako g’(x)  
   Przykład:  
   ****  
   Jak to robić:
   * Gdy mamy iloczyn 2 funkcji – jako jedną bierzemy f, a jako drugą g’
   * Jako f warto brać coś, co da się sprowadzić do stałej liczby, jako g’ brać coś, dla czego łatwo obliczyć całkę
   * Zapisujemy to w takiej macierzy i zapisujemy „iloczyn na ukos minus całka z iloczynu dolnego wiersza”
   * Resztę liczymy z karty wzorów (czasem, jak np. f to x^n, to trzeba powtórzyć dzielenie na części kilka razy – aż sprowadzimy x^n pochodnymi do jeden)

**ALPTW** – skrót pomagający ustalić priorytet przy wyborze funkcji, jaką podstawiamy pod f w całkowaniu przez części:

* A – arcus
* L – logarytm
* P – potęga (np. x^2…, sam x to też funkcja potęgowa: x^1)
* T – trygonometryczna (czyli sinus, cosinus itp.)
* W – wykładnicza (coś do potęgi x)

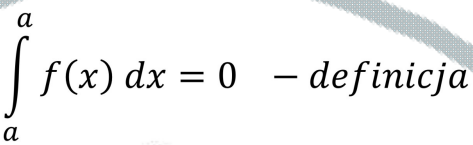
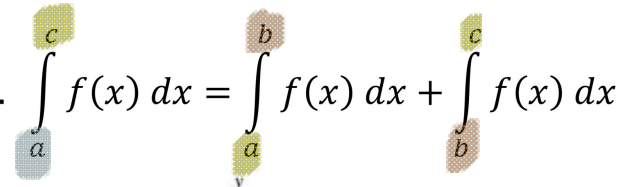
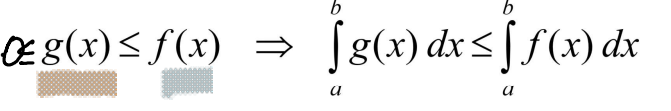
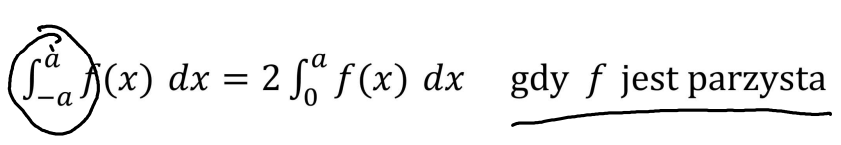
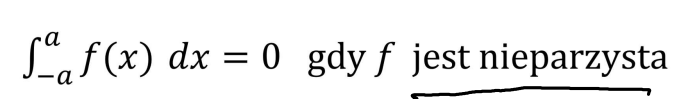
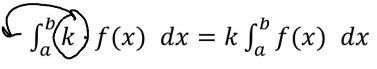
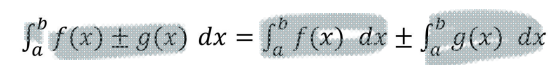
**Całka oznaczona** (całka Riemanna)– intuicyjnie rozumiana jako **pole pod krzywą** y = f(x) > 0 (dodatnia) w ustalonym przedziale [a,b]  
  
Wyjaśnienie – jednym ze sposobów na liczenie pola powierzchni pod krzywą jest tzw. **suma całkowita**. Polega ona na podzieleniu przedziału na n równych części o rozmiarze P. W każdym odcinku podziału ustalamy punkt pośredni i-tego odcinka (gdzieś w środku tego odcinka, może być dokładnie na środku, któryś z krańców odcinka albo jeszcze inaczej) i rysujemy prostokąty o podstawie o długości odcinków i wysokości równej f(). Jak prostokąt wychodzi w dół, to liczymy tak, jakby miał ujemne pole:  


Całka oznaczona, to taka suma tych pól, dla których te odcinki są nieskończenie małe (dążą do zera, dlatego we wzorze jest limes |P|->0)

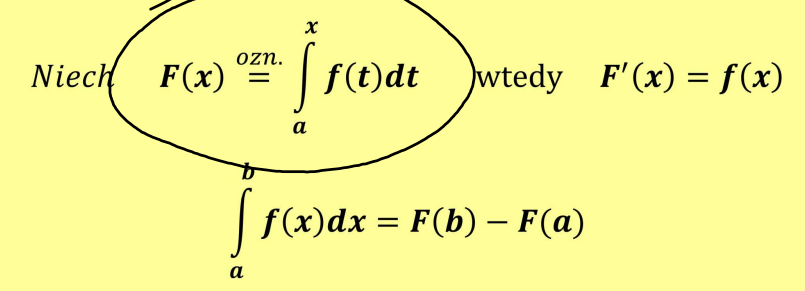
Dla przedziału [a,a] – całka jest równa 0  
Dla przedziału [b,a] – całka równa się minus całka [a,b]

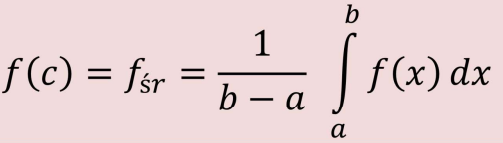
Funkcja jest **całkowalna** – istnieje dla niej całka oznaczona

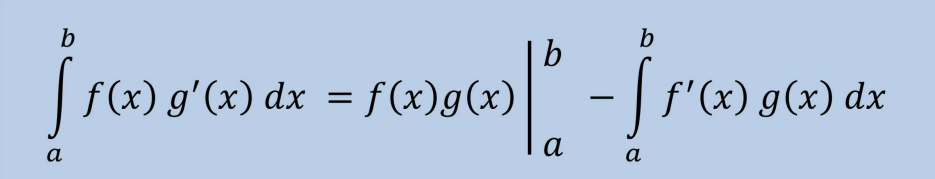
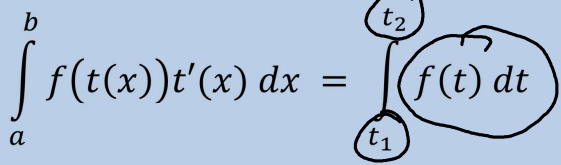
Własności całki oznaczonej:

* **Warunek wystarczającej całkowalności funkcji** – jeżeli funkcja f jest ograniczona na [a,b] i ma na tym przedziale skończoną liczbę punktów nieciągłości, to jest na nim całkowalna (nie może mieć dziury większej niż 1 punkt)
* 
*  (całkę oznaczoną da się rozbić na kilka mniejszych)
*  (jeżeli funkcja jest większa na całym przedziale od innej funkcji, to jej całka też będzie większa)
*  (funkcja parzysta – jest symetryczna po obu stronach osi Y w układzie współrzędnych: f(x)=f(-x))
* (funkcja nieparzysta – jest odwrotna po jednej stronie osi y względem drugiej strony: f(x)=-f(-x)
*  (jak w całce nieoznaczonej, stałe można wyciągnąć przed całkę)
*  (jak w całce nieoznaczonej, sumę/różnicę można rozbić na dwie całki)

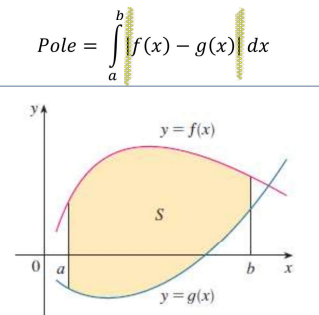
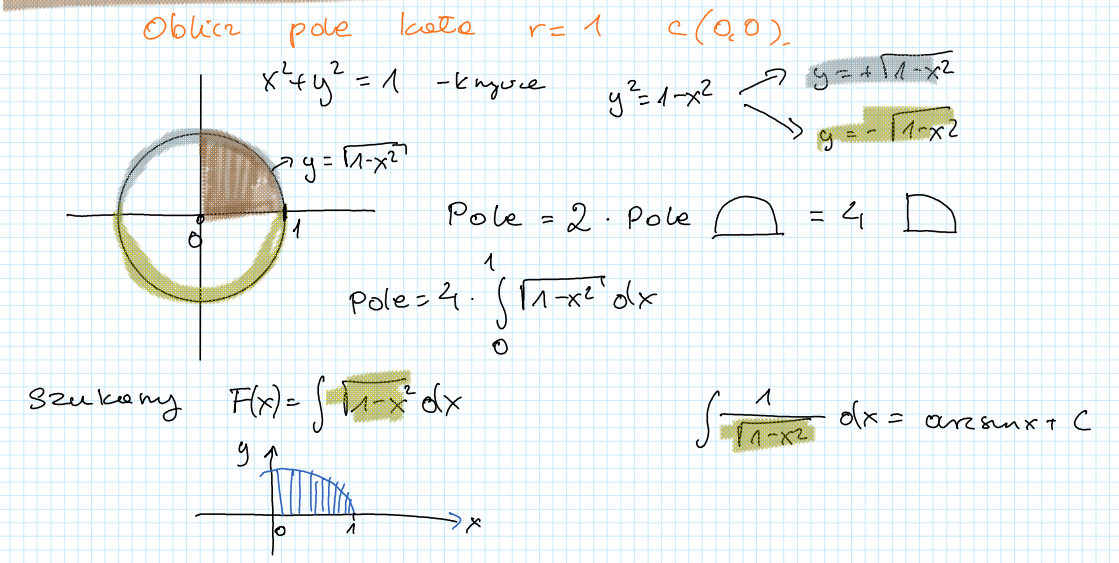
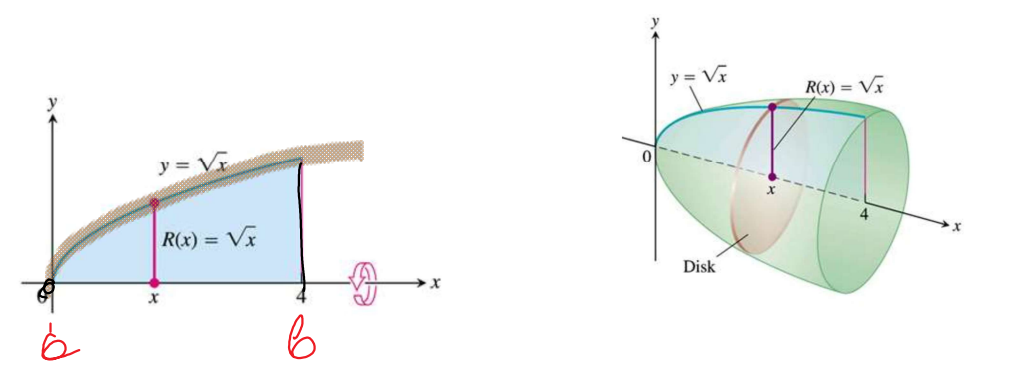
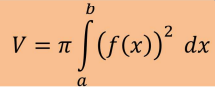
**Główne twierdzenie rachunku całkowego**:

Jeśli funkcja f jest ciągła na przedziale [a,b], to:  
  
(Żeby obliczyć całkę oznaczoną, wystarczy podstawić b oraz a pod całkę nieoznaczoną i odjąć je od siebie)

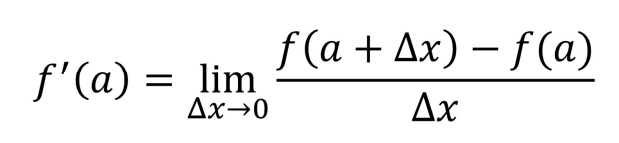
Wartość średnia funkcji na przedziale [a,b]:   
Jest ona równa wysokości prostokąta o podstawie długości przedziału (b-a), którego pole jest równe polu obszaru pod funkcją f(x) w tym przedziale

Całkowanie przez części dla całek oznaczonych:   
f(x)g(x) = f(b)g(b)-f(a)g(a)  
Całkowanie przez podstawianie dla całek oznaczonych:  
  
t\_1=t(a),t\_2=t(b)

**Zastosowanie całek:**

1. Do obliczania **pola trapezu krzywoliniowego**  
   (możemy tak też obliczyć np. pole flagi)
2. Do obliczania **pola koła**:  
     
   (bierzemy ćwiartkę, liczymy dla niej całkę i mnożymy przez 4)
3. Do obliczania **objętości bryły** (liczymy pole powierzchni podstawy i mnożymy przez wysokość bryły):  
     
   Dla brył obrotowych – ich przekrój to koło, pole koła to P = πR^2, dla każdego przekroju R=f(x), pole podstawy to A(x)= π(f(x))^2, objętość bryły to:  
    (odcinek [a,b] to wysokość bryły)  
   Żeby to działało, **wszystkie ściany bryły muszą się rozchodzić równomiernie z jednego punktu** (w każdym kierunku tak samo)

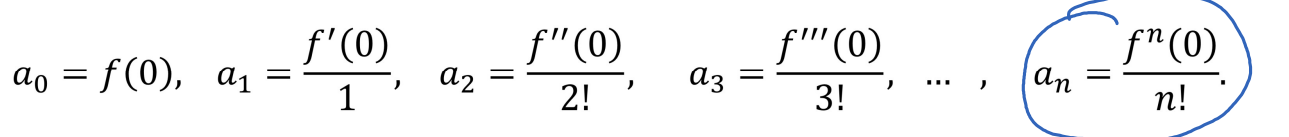
## 2. Wielomian i szereg Taylora funkcji rzeczywistej.

Wzór definiujący pochodną:  


Różniczka funkcji – dla funkcji f w punkcie a, to liniowa funkcja df zmiennej Δx = x-a określona wzorem: . To część liniowa przyrostu funkcji, czy przyrost ‘y’-ów na prostej stycznej do f

Niech funkcja f będzie wielomianem n stopnia, . W punkcie = 0 możemy wyznaczyć współczynniki tego wielomianu postępując następująco:

1. Obliczamy wartość funkcji w = 0 : f(0) =
2. Obliczamy wartość pierwszej pochodnej funkcji w = 0: f’(0) = a\_1
3. Obliczamy wartość drugiej pochodnej funkcji w x\_0=0: f’’(0)=2a\_2
4. To samo dla trzeciej pochodnej: f’’’(0)=3\*2a\_3
5. Liczymy tak do n-tej pochodnej funkcji:

Stałe przy iksach mają wartości zgodnie z zależnością:  


**Wielomian Taylora** – wielomian dla danej funkcji f, o rzędzie równym n o następującym wzorze:

Jest to wersja wielomianu w punkcie a=0. Wielomian Taylora w tym konkretnym punkcie to **wielomian Maclaurina**. Dla dowolnego innego a, wzór to:

Jeżeli funkcja f ma:

* Ciągłą pochodną rzędu n na przedziale [a,x]
* Pochodną właściwą f^(n+1) na przedziale (a,x)

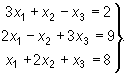
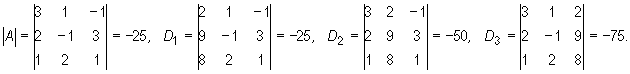
To istnieje punkt c [a,x] taki, że

– tzw. **reszta Lagrange’a** (jest o 1 stopień wyższa niż liczony wielomian Taylora, wielomian Taylora jedynie przybliża daną zmienną, dodanie reszty Lagrange’a dopiero daje nam dokładnie szukaną wartość – wyraża ona błąd przybliżenia)

**Wzór Taylora** = wielomian Taylora + reszta Lagrange’a

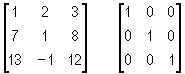
## Układy równań liniowych: różne metody rozwiazywania, liczba rozwiązań.

Metody rozwiązywania:

1. **Przy pomocy wzorów Cramera:**
   * Układ Cramera – taki układ równań, którego wyznacznik jest różny od 0 (tzn. macierz A układu jest nieosobliwa)
   * **Układ Cramera ma dokładnie jedno rozwiązanie** dane wzorami:  
     
   * Przykład dla układu równań:  
       
       
     Liczymy 4 wyznaczniki macierzy (|A| - kolumny w macierzy to stałe przy iksach, |D1,2,…n| - jedną z kolumn (tą reprezentującą x\_n) zamieniamy na kolumnę po znaku równości)
2. Metoda macierzowa:
   * Macierz układu równań A – macierz, w której kolumnami są liczby przy iksach
   * Kolumna niewiadomych układu równań X=
   * Kolumna wyrazów wolnych B – macierz taka jak X, tylko w niej są liczby stojące bez iksa po znaku równości
   * Dla układów Cramera, wyznacznik |A| != 0, to znaczy że istnieje macierz – **mnożymy obie strony równania macierzowego przez tą macierz**
   * Wychodzi z tego równanie (kolejność mnożenia jest ważna – w macierzach mnożenie nie jest przemienne):
   * Obliczenie tego równania daje nam wartości iksów
   * Wzory Cramera są uproszczeniem tego równania
3. **Algorytm kolumn jednostkowych:**
   * Przekształcamy kolumny w macierzy rozszerzonej tak, aby po każdym kroku uzyskać kolumnę z jedną jedynką i resztą zer. Jedynki te muszą znajdować się w różnych wierszach
   * **Macierz rozszerzona** – macierz układu równań A z dołożoną po prawej kolumną w postaci macierzy wyrazów wolnych B
   * W przeciwieństwie do metody Gaussa (o niej w zagadnieniu o własnościach macierzy), nie musimy przestawiać wierszy ani tworzyć macierz odwrotną
   * Wybieramy wiersz, jakim zerujemy daną kolumnę, potem innym wierszem zerujemy kolejną itd. – jak już w każdym wierszu będzie tylko jedna liczba różna od zera, to dzielimy ją tak, by była ona równa jeden
4. **Metoda eliminacji Gaussa** – zamieniamy macierz na macierz górnotrójkątną, a następnie na jednostkową przy pomocy operacji:
   * Zamiany wierszy między sobą
   * Mnożenie wierszy przez dowolną liczbę różną od zera
   * Dodanie do jednego wiersza innego wiersza pomnożonego przez dowolną liczbę
   * Wykreślenie wiersza złożonego z samych zer/proporcjonalnego do innego wiersza
   * Ustawiamy na górze wiersz, dla którego łatwo będzie zrobić jedynkę w pierwszej kolumnie -> mnożymy go tak, by pod tą jedynką po odjęciu były same zera -> robimy to samo dla drugiego wiersza i drugiej kolumny itd. aż dotrzemy do ostatniego wiersza/kolumny -> idąc od dołu odejmujemy te wiersze tak, żeby powstała macierz jednostkowa

Liczba rozwiązań układu równań liniowych:

1. Układ m równań liniowych z n niewiadomymi ma rozwiązania, jeśli rząd r macierzy głównej jest równy rzędowi macierzy rozszerzonej (macierz główna=miacierz A, macierz rozszerzona=A z doczepioną na końcu B)  
   Rząd macierzy – maksymalna liczba liniowo niezależnych kolumn w macierzy (kolumny jest liniowo niezależna, jeżeli nie istnieje w macierzy kombinacja liniowa innych kolumn, która by ją wyzerowała)
2. **Układ oznaczony** – istnieje jedno rozwiązanie układu, rząd macierzy głównej=rząd macierzy rozszerzonej=liczba niewiadomych
3. **Układ nieoznaczony** – nieskończenie wiele rozwiązań, bo niektóre niewiadome nie mają wpływu na wynik. Jest tak wtedy, gdy rząd obu macierzy jest mniejszy od liczby niewiadomych
4. **Układ sprzeczny** – brak rozwiązań, rząd macierzy głównej < rząd macierzy rozszerzonej (w jednym wierszu A są same zera, a w tym samym wierszu dla B jest coś, co zerem nie jest i mamy np. 0=5 <- sprzeczność)

-> rząd macierzy po lewej to 2 (bo: od trzeciej kolumny odejmiemy pierwszą – jest równa drugiej – odjęcie drugiej od trzeciej wyzeruje kolumnę trzecią). Rząd macierzy po prawej jest równy 3

Żeby policzyć rząd macierzy - szukamy w macierzy największej podmacierzy, dla której wyznacznik jest różny od zera (jeśli sama ta kwadratowa macierz ma niezerowy wyznacznik, to jej rozmiar jest równy rzędowi, jak nie, to wykreślamy jeden wiersz i kolumnę (dowolnie) i sprawdzamy, czy nowa mniejsza macierz ma niezerowy wyznacznik itd.)

**Układ jednorodny** – taki, w którym w macierzy B (wyrazów wolnych) są same zera

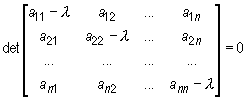
## 4. Wartości własne macierzy i ich zastosowanie w informatyce.

**Endomorfizm liniowy** – przekształcenie liniowe przestrzeni V zgodnie ze wzorem f: V->V (przejście w taką samą przestrzeń)  
Macierz endomorfizmu jest macierzą kwadratową  
v V – **wektor własny** endomorfizmu f, jeśli v != 0 oraz istnieje λ R takie, że f(v) = λ\*v  
W tym równaniu, λ – **wartość własna** endomorfizmu f

Przykład endomorfizmu:

Opisane wzorem: f([x,y,z])=[2x+2y+z,x+3y+z,7x-2z]  
Można to zapisać macierzą:  
Dla f([1,1,1]) = [5,5,5] = 5\*[1,1,1] -> wektor własny to [1,1,1], a wartość własna to 5

W celu znalezienia wartości własnej macierzy, należy rozwiązać równanie:

  
(na skos odejmujemy szukaną wartość własną)

Na przykład:



Z tego wychodzi, że wartościami własnymi są 2 oraz 8

Gdy podstawimy którąś z tych wartości, pod lambdę, to po uzyskaniu nowej macierzy możemy ustalić wektory własne. Przykład dla lambda=2:  
 -> z tego wynika, że wektorem własnym może być np. [1,1] (bo wtedy wychodzi, że 0=0 -> musimy móc doprowadzić układ równań do tej postaci)

Zastosowanie wartości własnych w informatyce:

* Rankingowanie stron internetowych w wyszukiwarce Google (algorytm PageRank – linki między stronami i wagi tych linków są zapisane w macierzach, wartość własna przyspiesza liczenie tych wag)
* PCA (principal component analysis – do zmniejszania wymiarowości danych używanych przy uczeniu maszynowym, przy zmniejszaniu wymiarów, tworzona jest macierz kowariancji zmiennych (powinienem to opisać w zagadnieniu 9 lub 10) po znalezieniu jej wektora własnego, wektor ten wyznacza kierunek hiperpłaszczyzny, na którą rzutujemy atrybuty danych)
* Algorytm partycji grafów (dzielenie grafów na kilka mniejszych, które są ze sobą połączone pojedynczymi przejściami, druga najmniejsza wartość własna w macierzy przejść daje nam dolną granicę optymalnego przecięcia grafu)

Zastosowanie macierzy w informatyce:

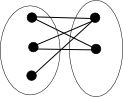
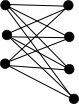
* Grafika komputerowa (zapis obrazu, transformacje obrazu takie jak przesunięcia/powiększenie/obrót/pochylenie to obliczenia na macierzach)
* Uczenie maszynowe (w sieciach neuronowych – obliczanie wartości NET (czyli wektor wejść\*wektor wag) jest robione przy pomocy macierzy
* Reprezentacja danych (tablice to tak naprawdę macierze)

## 5. Grafy i ich typy, metody reprezentacji grafów.

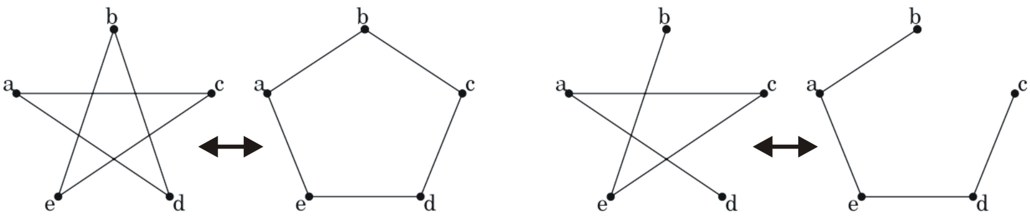
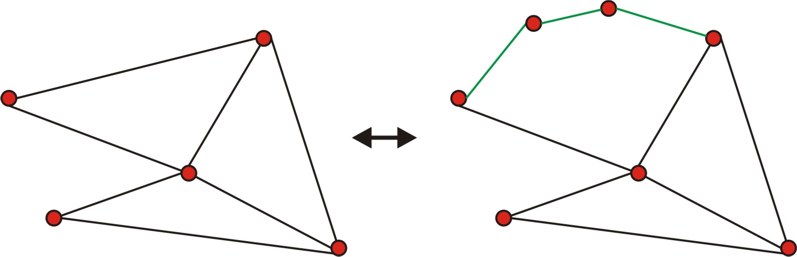
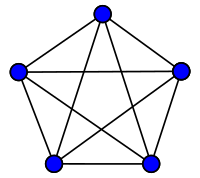
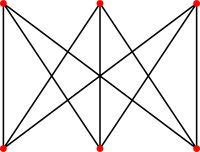
**Graf** ­– struktura danych w postaci uporządkowanej pary zbiorów **G=(V,E)**:

* V – zbiór wierzchołków grafu
* E – zbiór krawędzi grafu
* Każda krawędź e = {v,w} to *nieuporządkowana* para wierzchołków ze zbioru V, zwanych końcami krawędzi e (krawędź e łączy wierzchołki v oraz w [jest z nimi incydentna], v oraz w są sąsiednie w grafie

Typy grafów:

1. **Nieskierowany** – reprezentują symetryczną relację binarną na zbiorze wierzchołków (połączenia między wierzchołkami działają w obie strony)
2. **Skierowany** – taki, w którym kierunek przechodzenia po krawędziach jest ustalony (krawędź e wychodzi z v i wchodzi do w) – krawędzie skierowane nazywamy **łukami**
3. **Zerowy** – taki, dla którego zbiory V i E są puste
4. **Prosty** – nie ma pętli ani krawędzi wielokrotnych (pętla – krawędź postaci (v,v), w grafie skierowanym krawędzie (v,w) oraz (w,v) to dwie różne krawędzie -> nie liczy się jako krawędź wielokrotna).  
   Uogólnienia grafu prostego:
   1. **Multigraf** – graf bez pętli, ale z krawędziami wielokrotnymi (pomiędzy tą samą parą wierzchołków)
   2. **Hipergraf** – zawiera hiper-krawędzie (takie, które mogą łączyć więcej niż 2 wierzchołki – reprezentują relacje o wyższej -arności niż 2)
5. **Pusty** (N\_n) – same wierzchołki, bez krawędzi
6. **Pełny** (K\_n) – dopełnienie pustego: każdy wierzchołek jest połączony z każdym (wszystkie możliwe krawędzie – **graf zawierający tylko jeden wierzchołek jest pełny**)
7. **Dwudzielny** – zbiór wierzchołków da się podzielić na dwa rozłączne podzbiory tak, że ewentualne krawędzie występują tylko między tymi zbiorami a nie wewnątrz nich  (Dwudzielny też może być pełny: )
8. **Ścieżkowy** P\_n (chyba chodzi o to, że taki rozłożony z końcem i początkiem, nie mogę tego znaleźć w Internecie), **cykliczny** C\_n (układa się w drogę tak, że pierwszy wybrany wierzchołek jest też ostatnim), **kołowy** W\_i (jak cykliczny, ale ma dodatkowo wierzchołek połączony ze wszystkimi innymi)

Inny podział:

1. **Etykietowane** – wierzchołki mają etykiety
2. **Ważone** – krawędzie mają wagi (np. w sieciach komputerowych)
3. **Spójne** – gdy istnieje ścieżka w grafie łącząca dwa dowolne wierzchołki (może mieć po drodze inne wierzchołki)
4. **Niespójne** – gdy nie da się wyznaczyć takiej ścieżki dla każdej pary wierzchołków
5. **Eulerowskie** – takie, w których da się wyznaczyć ścieżkę przechodzącą dokładnie raz przez każdą krawędź i wracającą do punktu wyjściowego (**cykl Eulera**)
6. **Hamiltonowskie** – taki, w którym możemy wyznaczyć ścieżkę przechodzącą przez każdy wierzchołek tylko jeden raz (**ścieżka Hamiltona** – każdy graf pełny o min 3 wierzchołka jest takim grafem)
7. **N-Regularne** – każdy wierzchołek jest tego samego stopnia (z każdego wierzchołka wychodzi dokładnie n krawędzi – grafy puste to grafy 0-regularne)
8. **Izomorficzne** – takie, w których istnieje **bijekcja** (każdemu elementowi jednego grafu da się przypisać dokładnie jeden element drugiego) – to tak naprawdę pary tych samych grafów, tylko z inaczej rozłożonymi krawędziami :
9. **Homeomorficzne** – gdy jeden graf z drugiego można otrzymać zastępując wybrane krawędzie prostymi łańcuchami krawędzi i wierzchołków (lub na odwrót):  
   
10. **Planarne** – takie, które można narysować na płaszczyźnie tak, by krawędzie grafu nie przecinały się ze sobą. Minimalne grafy, które nie są planarne to K\_5 i K\_3,3:  
       
    **Twierdzenie Kuratowskiego** – graf skończony jest planarny, jeśli nie zawiera podgrafu homeomorficznego z żadnym z tych grafów

**Reprezentacja grafów:**

1. **Macierz sąsiedztwa** – Kwadratowa macierz o rozmiarze równym liczbie wierzchołków, wiersze i kolumny reprezentują wierzchołki, jeśli A[i,j]=1, to między wierzchołkami i oraz j jest krawędź, jak A[i,j]=0, to nie. Jak A[i,i]=2, to na wierzchołku i jest pętla. Dla grafów nieskierowanych macierz jest symetryczna (nie zmienia się po transponowaniu). Dla grafów skierowanych, transponowanie macierzy odwraca kierunki krawędzi  
   Złożoność pamięciowa: **O(V^2)** (V – liczba wierzchołków)
2. **Macierz incydencji** – Wiersze to wierzchołki, kolumny to krawędzie. I[v,e]=1 -> v jest incydentny z e (e jest krawędzią wchodzącą/wychodzącą z v). Jak nie, to I[v,e]=0  
   Złożoność pamięciowa: **O(V\*E)** (E – liczba krawędzi)
3. **Listy sąsiedztwa (incydencji)** – reprezentacja składająca się z tylu list, ile jest wierzchołków. Każda lista zaczyna się od etykiety danego wierzchołka, po czym jest lista wierzchołków sąsiednich (dla skierowanych: lista wierzchołków, do których można dojść z danego wierzchołka)  
   Złożoność pamięciowa: **O(V+E)**
4. **Lista krawędzi** – lista zawierająca wszystkie krawędzie w grafie (np. {1-2},{2-4},{2-5})  
   Złożoność pamięciowa: **O(E)**
5. **Obiektowa** – wierzchołki i krawędzie są obiektami zaimplementowanymi w języku obiektowym. Wierzchołki mają dowiązania do sąsiadów i krawędzi incydentnych
6. **„Gd0”** – lista w formacie binarnym, zawierająca ciągi identyfikatorów całkowitoliczbowych w postaci: identyfikator\_wierzchołka, stopień wierzchołka, identyfikatory\_sąsiadów

## 6. Relacje binarne, własności i metody reprezentacji.

**relacja** – dowolny podzbiór iloczynu kartezjańskiego na zbiorach A1 x … x An. Ze wszystkich możliwych n-tek (par, trójek, czwórek itp.) wybieramy te, które nas z jakiegoś powodu interesują.

**Relacje binarne (dwuargumentowe)** – podzbiory kwadratu kartezjańskiego A x A (oznaczane też jak A^2). Przykłady:

* Relacja niemniejszości – w zbiorze liczb naturalnych
* Relacja niemniejszości – w zbiorze liczb rzeczywistych
* Relacja przystawania modulo n – w zbiorze liczb naturalnych

**Dziedzina relacji** – zbiór elementów, które występują jako pierwszy składnik pary w relacji dwuelementowej  
**przeciwdziedzina** – zbiór elementów, które występują jako drugi składnik pary

**Relacja pusta** – między żadnymi elementami nie zachodzi relacja  
**Relacja pełna** – między wszystkimi elementami zachodzi relacja (pełny iloczyn kartezjański)  
**relacja odwrotna** – relacja, w której dziedzina i przeciwdziedzina są zamienione  
**złożenie relacji** – gdy (a1,a2) są w relacji oraz (a2,a3) są w relacji, to złożeniem tych relacji jest relacja (a1,a3)  
**funkcja** – taka relacja, w której, jeżeli elementy an oraz an’ są w relacji z elementami a1, … a{n-1}, to an=an’ (w sensie, że dla dowolnego iksa może być tylko jeden igrek)  
**funkcja całkowita** – funkcja, w której dla każdego zbioru a1, … a{n-1} istnieje an będące z nimi w relacji

**Cechy (własności) relacji**:

1. R jest ***zwrotna*** - (każdy element jest sam ze sobą w relacji, np. częściowy porządek [mniejszy-równy/większy-równy])
2. R jest ***przeciwzwrotna***- (żaden element nie jest sam ze sobą w relacji, np. prostopadłość linii)
3. R jest ***symetryczna*** - (jak a,b są w relacji R, to są też w relacji odwrotnej, np. rodzeństwo)
4. R jest ***asymetryczna*** - (jak a,b są w relacji, to w relacji odwrotnej nie są, np. relacja „większy” w zbiorze liczb rzeczywistych, relacja potomstwa)
5. R jest ***antysymetryczna*** - (jest symetryczna tylko w ramach tego samego elementu, np. relacja równości, relacja porządku w alfabecie łacińskim)
6. R jest ***przechodnia*** - (np. podzielność, zawieranie się zbiorów)
7. R jest ***spójna*** - (każda para elementów jest ze sobą w relacji w jakąś stronę, albo są tym samym elementem, np. < na zbiorze liczb naturalnych)

**Relacja pusta** – taka, która nie zachodzi między żadnymi elementami. Jest jednocześnie: symetryczna, asymetryczna, antysymetryczna, przeciwzwrotna i przechodnia (jeżeli jej dziedziną jest zbiór pusty, to jest też spójna i zwrotna)

**Relacja równoważności** – zwrotna+symetryczna+przechodnia (np. równość[identyczność] elementu zbioru, relacja RODZEŃSTWO dla identycznych rodziców, bycie w tej samej grupie ćwiczeniowej z MAD. Relacja BRAT nie jest ani zwrotna ani symetryczna ani przechodnia

**Relacja porządku (częściowego)** – zwrotna+antysymetryczna+przechodnia (np. <=)  
**Relacja porządku liniowego** – dodatkowo jest spójna

**Łańcuch** – ciąg a1,a2,a3,… taki, że każde dwa kolejne elementy są ze sobą w relacji: (a1,a2),(a2,a3)…  
**Relacja dobrze ufundowana** – taka, dla której każdy łańcuch jest skończony (mniejsze/większe nie są dobrze ufundowane dla liczb rzeczywistych, ale mniejsze jest dobrze uporządkowane dla liczb naturalnych, bo łańcuch kończy się na zerze)

**Domknięcie zwrotne relacji** – wzbogacenie relacji o zwrotność (na grafie – dorysowanie pętelek na punktach) oznaczane - Rz  
**Domknięcie symetryczne** – wzbogacenie relacji o symetryczność (na grafie skierowanym – dodanie grotu na drugim końcu każdej strzałki) oznaczane - Rs  
**Domknięcie przechodnie** – wzbogacenie relacji o przechodniość (na grafie – dodanie skrótów między każdą parą punktów) oznaczane – R+

R\* - domknięcie zwrotne + przechodnie

**Podział zbioru A** – każdy zbiór rozłącznych parami podzbiorów, które w sumie dają cały zbiór A  
**Zasada abstrakcji** – między podziałami zbioru a relacjami równoważności istnieje wzajemna odpowiedniość – każdy podział wyznacza unikalną relację równoważności, a każda relacja równoważności wiąże się z pewnym podziałem (np. relacja modn dzieli zbiór liczb naturalnych na n podzbiorów [nazywanych **klasami abstrakcji**] w zależności od tego, ile reszty zostaje po podzieleniu przez nie liczby n [od 0 do n-1])

**Metody reprezentacji relacji**:

* Graf (elementy to punkty, krawędzie określają relacje)
* Roster method – zapisanie relacji jako pary/trójki itd. w klamerkach {(9,3),(9,-3)…}
* Set-builder notation (notacja budowy zbiorów) – tak, jak to było w przykładach relacji binarnych, czyli np.

## 7. Zasada indukcji matematycznej.

W skrócie, jeżeli:

1. Istnieje taka liczba naturalna że T() jest zdaniem prawdziwym
2. Dla każdej liczby naturalnej n prawdziwa jest implikacja T(n)=>T(n+1)

To T(n) jest zdaniem prawdziwym dla każdej liczby naturalnej

**Zasada indukcji prostej** – zasada wnioskowania o prawdziwości wszystkich zdań z podanego ciągu zdań P(0), P(1), … Najpierw pokazujemy prawdziwość jednego z nich (najczęściej P(0)), a następnie żądamy, by udowodnione zdania zaświadczały o prawdziwości kolejnych, zazwyczaj poprzez wynikanie P(n) -> P(n+1) dla każdego n  
**Zasada indukcji zupełnej** – w celu udowodnienia P(n+1) należy założyć prawdziwość wszystkich zdań poczynając od bazowych aż do P(n) włącznie:  
Dany jest ciąg zdań P(0), P(1), … Jeżeli prawdziwe są zdania P(b), P(b+1), … P(b+k) dla pewnych b,k>=0 oraz jeżeli dla każdego n >= b+k, z prawdziwości zdań P(b),…,P(b+k) wynika prawdziwość zdania P(n+1) to zdanie P(n) jest prawdziwe dla każdego n >= b  
**baza indukcji** – ciąg zdań P(b),…,P(b+k)  
**krok indukcyjny** – wynikanie (P(b),…P(n))->P(n+1)  
Liczba k określa **liczbę zdań bazowych** (jest ich k+1)

Przykład indukcji matematycznej dla równania ciągu Fibonacciego:

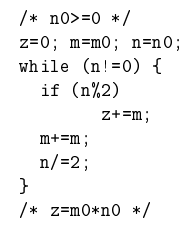
* Z definicji: , po prawej mamy:
* Dla , po prawej mamy
* Zastępujemy ułamki symbolami:
* Zakładamy, że wzór na jest prawidłowy dla wszystkich k=0,…,n-1 gdzie n>1. Trzeba teraz udowodnić, że
* Z definicji ciągu Fibonacciego: , z tego stosujemy krok indukcyjny:
* są pierwiastkami równania , z tego oraz
* Po podstawieniu zgodnie z poprzednim punktem:  
  Co pokazuje, że równanie do udowodnienia jest prawidłowe

**Zasada indukcji noetherowskiej** ­– X to dowolny zbiór, r to relacja dobrze ufundowana określona na X (dobrze ufundowana = nie ma nieskończonych łańcuchów). Rozważamy formułę zdaniową P(x) określoną na elementach zbioru X  
Jeżeli dla każdego x możemy pokazać że z prawdziwości formuły: wynika prawdziwość P(x), to formuła P(x) jest prawdziwa dla każdego x.  
Indukcja zupełna to jest przykład indukcji noetherowskiej (relacja > dla liczb naturalnych jest dobrze ufundowana)

**Zastosowanie indukcji matematycznej w informatyce**:

* Dowodzenie poprawności pętli – tworzymy *specyfikację problemu algorytmicznego* (Parę (In,Out) określającą początkowy warunek pętli i jej wynik końcowy – out ma miejsce tylko, jeżeli pętla nie jest nieskończona) np.  
  mnożenie

Algorytm **całkowicie poprawny** – częściowo poprawny (czyli, że jak dane spełniają In, to jeżeli się zakończy, to wyjście spełnia warunek Out) względem In,Out + dla każdych danych spełniających In się zakończy (częściowo poprawny pozwala na nieskończone pętle)

 - algorytm mnożenia chłopów rosyjskich (oparty o dzielenie całkowite przez 2 i dodawanie. Da się go udowodnić indukcją matematyczną – ma złożoność O(logn))

**Metoda Floyda-Hoare’a** ­(metoda niezmienników pętli) – polega na znalezieniu logicznego warunku N, który będzie spełniony ilekroć będziemy na początku pętli. Powinien on możliwie dokładnie opisywać zależność między zmiennymi w pętli (jeżeli istnieje niezbędnik N, to pętla jest częściowo poprawna)

## 8. Twierdzenie Bayesa.

Treść twierdzenia Bayesa (wzór Bayesa):

* Jeśli tworzą podział przestrzeni S oraz P() > 0, u=1,2,…,k, to dla dowolnego zdarzenia A o dodatnim prawdopodobieństwie oraz dowolnego zdarzenia spośród zdarzeń zachodzi wzór:
* Do tego:
* Interpretacja – jeśli skutek A nastąpił w wyniku zajścia jednej z przyczyn , to prawdopodobieństwo tego, że była przyczyną zajścia A wyraża się wzorem Bayesa

– prawdopodobieństwo **a priori** – prawdopodobieństwo **a posteriori**

**Twierdzenie o prawdopodobieństwie całkowitym**:

* Jeśli zdarzenia tworzą podział przestrzeni zdarzeń elementarnych S (układ zupełny zdarzeń – ) oraz , to prawdopodobieństwo dowolnego zdarzenia A dane jest wzorem:
* – przyczyny
* – skutek

**Zmienna losowa** – Funkcja X określona na przestrzeni zdarzeń elementarnych S o wartościach rzeczywistych . Zmienna losowa X jest dyskretna, jeśli jej zbiór wartości {X(s), s należy do S} jest skończony lub nieskończony przeliczalny  
**Dystrybuanta zmiennej losowej** – Funkcja określona wzorem:  
(Dla max. x, F(x)=1, to jest takie AUC dla prawdopodobieństwa, dystrybuanta to funkcja rzeczywista jednoznacznie wyznaczająca rozkład prawdopodobieństwa)

Własności dystrybuanty:

1. Dla (prawostronna ciągłość dystrybuanty

## 9. Testowanie hipotez statystycznych.

**Hipoteza statystyczna** – przypuszczenie dotyczące nieznanej własności rozkładu prawdopodobieństwa badanej cechy populacji. Może dotyczyć nieznanego parametru rozkładu (np. że średnio coś jest większe/mniejsze) lub stwierdzać, że dana cecha ma pewien określony rozkład (np. rozkład normalny, rozkład Poissona)

**Hipoteza zerowa** – hipoteza, której prawdziwość poddajemy w wątpliwość i która jest testowana celem ewentualnego odrzucenia, oznaczana jako

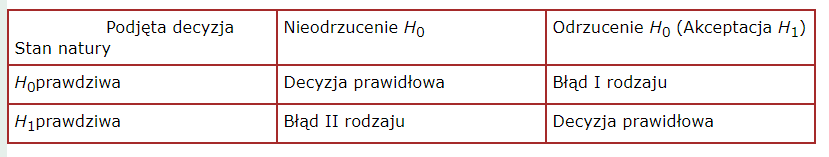
**Hipoteza alternatywna** ­– hipoteza, która będzie przyjęta, jeśli odrzucimy hipotezę zerową, oznaczana jako   
Hipoteza zerowa i alternatywna mają się wzajemnie wykluczać  
Staramy się udowodnić hipotezę alternatywną, co pozwoli na odrzucenie hipotezy zerowej. **Odrzucenie hipotezy alternatywnej nie oznacza, że hipoteza zerowa jest prawdziwa ani fałszywa** (mówimy wtedy, że „nie możemy odrzucić hipotezy zerowej”)

**Błąd I rodzaju** – błąd polegający na przyjęciu gdy hipoteza jest prawdziwa.  
**Poziom istotności testu** – prawdopodobieństwo popełnienia błędu pierwszego rodzaju, oznaczane grecką literą α

**Statystyka testowa** – statystyka, na której podstawie orzekamy prawdziwość , oznaczana dużą literą Z (statystyka to kwantyl danego rozkładu. Z karty wzorów patrzymy na odpowiedni kwantyl, zależny od poziomu istotności, i sprawdzamy, czy nasza statystyka testowa jest od niego większa. To znaczy, że należy ona do zbioru krytycznego). W zależności od typu hipotezy, mamy różne wzory na statystykę testową (mieliśmy do tego kartę wzorów)

**Zbiór krytyczny** – zbiór wartości statystyki testowej Z, dla których odrzucamy na korzyść ,  
oznaczany jako: }

**Błąd II rodzaju** – błąd polegający na przyjęciu , gdy hipoteza jest prawdziwa



Różne testy dotyczące hipotez (mieliśmy na nie karty wzorów):

1. Hipotezy o **wartości średniej** rozkładu normalnego, gdy **znana jest wariancja**
2. Hipotezy o **wartości średniej** rozkładu normalnego, gdy **wariancja nie jest znana**
3. Hipotezy o **wartości średniej** dowolnego rozkładu, gdy **liczebność próby jest większa niż 100**
4. Hipotezy o **wariancji** rozkładu normalnego
5. Hipotezy o **równości średnich jednej cechy w dwóch populacjach** dla rozkładu normalnego ze **znanymi wariancjami**
6. Hipotezy o **równości średnich jednej cechy w dwóch populacjach** dla rozkładu normalnego z **nieznanymi wariancjami**
7. Hipotezy o **równości średnich jednej cechy w dwóch populacjach** dla rozkładu normalnego z **nieznanymi wariancjami ale obydwoma populacjami większymi niż 100**
8. Hipotezy o równości średnich jednej cechy mierzonej przed i po wykonaniu operacji – **metoda zmiennych połączonych** **(znana wariancja)**
9. Hipotezy o równości średnich jednej cechy mierzonej przed i po wykonaniu operacji – **metoda zmiennych połączonych** **(nieznana wariancja)**
10. Hipotezy o **proporcji p** (założenie: npˆ ≥ 5, n(1 − pˆ) ≥ 5))
11. Hipotezy o **różnicy proporcji dwóch populacji**

## 10. Wyznaczanie przedziałów ufności.

**Estymator punktowy** – jest to statystyka dla ustalonej funkcji h, której celem jest oszacowanie parametru rozkładu oznaczanego jako θ (parametr to np. mediana, odchylenie standardowe itp.) np. średnia arytmetyczna to estymator wartości oczekiwanej (średniej oznaczanej literą μ np. w rozkładzie normalnym)

**Estymator nieobciążony** – niezależnie od tego, jaką wartość ma parametr, funkcja będąca estymatorem nieobciążonym zwraca dokładnie tą wartość parametru, jaka jest w rzeczywistości (średnia z prostej próby losowej jest estymatorem nieobciążonym wartości oczekiwanej)

Przedziały ufności pozwalają nam określić zakres wartości, w jakich znajduje się dany parametr na   
1-α% (α określa **poziom ufności**). Tak jak przy testowaniu hipotez, to jak ten przedział wyliczyć zależy od różnych rzeczy – mieliśmy na to kartę wzorów.

Modele obliczania przedziału ufności:

1. Dla wartości oczekiwanej:
   1. Rozkład normalny – znamy odchylenie standardowe
   2. Rozkład normalny – nie znamy odchylenia standardowego
   3. Dowolny rozkład – nie znamy odchylenia standardowego + próbka o wielkości min. 50 obserwacji
   4. Rozkład Bernoulliego – min. 100 obserwacji
2. Dla wariancji:
   1. Rozkład normalny – wartość oczekiwana jest znana
   2. Rozkład normalny – wartość oczekiwana nie jest znana
   3. Rozkład dowolny – wartość oczekiwana nie jest znana + min. 100 obserwacji

## 11. Podstawowe cechy relacyjnych baz danych.

**Dane** ­– wartości, które mogą być przetwarzane (umysłowo lub komputerowo). Coś, co zapisujemy i utrwalamy (np. pod postacią liczb, tekstów, plików graficznych). Są jednym z zasobów firmy – wymagają *zarządzania* i *konserwacji*

**Informacja** – dane + semantyka, czyli klucz do ich interpretacji. Jeżeli danych nie możemy zinterpretować, to są bezużyteczne

**System informacyjny** – system służący do zarządzania danymi

**Baza danych** – część systemu informacyjnego odpowiedzialna za przechowywanie, bezpieczeństwo, zarządzanie danymi i ich udostępnianie. Może oznaczać:

* **System zarządzania danymi (SZBD)** – struktura, bezpieczeństwo i zasady dostępu do danych
* **Kolekcja Danych (DB)**

Wymagania stawiane bazom danych:

1. **Trwałość** – upływ czasu nie może mieć wpływu na stan danych
2. **Zgodność z rzeczywistością** – dane mają wiernie ją opisywać i móc się dostosować do zmian w rzeczywistości
3. **Kontrolowanie replikacji** – kontrola nad **redundancją** (nadmiarową informacją – to samo zapisane na kilka sposobów), co może prowadzić do niejednoznaczności danych
4. **Model danych** – struktura bazy danych musi być zaprojektowana według jednego modelu
5. **Współbieżny dostęp do danych** – wiele operacji na raz przy zachowaniu spójności danych
6. **Ochrona (bezpieczeństwo) danych**
7. **Niezależność danych i wykorzystujących je procesów**

1970 – Edgar Codd – przedstawił podstawowe cechy relacyjnego modelu danych

Najważniejsze założenia modelu relacyjnego:

* Wszystkie dane zapisywane są w postaci **tabel**
* W tabeli znajdują się **kolumny** zawierające dane **określonego typu**
* W **komórce** tabeli znajduje się **pojedyncza, atomowa, niepodzielna wartość**
* Każda tabela ma swój jednoznaczny identyfikator – **klucz główny** (jedna lub więcej kolumn, które jednoznacznie identyfikują dany rekord – wiersz) – wartość klucza głównego musi być **określona (not null) i unikalna**
* Tabele mogą zawierać **klucze obce** – jedna lub więcej kolumn, których wartości występują jako wartości klucza głównego w innej (lub tej samej) tabel i pełnią rolę wskaźników do rekordów drugiej tabeli (do definicji relacji między tabelami)

12 postulatów Edgara Codda (tak naprawdę to 13, bo od 0 do 12):

1. SZBD można uznać za system relacyjny, jeżeli wykorzystuje **wyłącznie rozwiązania zgodne z modelem relacyjnym** (nie mogą być hybrydowe)
2. Postulat informacyjny– na poziomie logicznym, wszystkie dane są reprezentowane **wyłącznie jako wartości w tabelach**
3. Postulat gwarancji dostępu – do każdej danej można się **dostać po nazwie tabeli, nazwie kolumny i wartości klucza głównego**
4. Postulat obiektu NULL – **null to brak wartości** (nieznana wartość), wszystkie obliczenia na NULL dają NULL (nawet NULL=NULL to NIE JEST TRUE)
5. Postulat struktury metadanych – **informacje o obiektach** (nazwy tabel i kolumn, co jest kluczem itp.) **też są zapisane w tabelach** i są dostępne jak każde inne dane
6. Postula pełnego języka danych – w SZBD zaimplementowany jest **pełny język** do definiowania tabel, perspektyw, więzów spójności, operowania danymi, nadawania uprawnień i grupowania operacji w transakcje. Realizowane jest to poprzez język **SQL**, który składa się z 4 podjęzyków:
   1. DDL (data definition language) – do tworzenia/usuwania/modyfikacji obiektów (tabel, perspektyw itp.) – **CREATE, ALTER, DROP VIEW/TABLE**
   2. DML (data manipulation language) – do tworzenia/usuwania/modyfikacji danych (insert into itp.) – **INSERT,UPDATE,DELETE**
   3. DQL (data query language) – do odczytywania danych (ma jedynie polecenie **SELECT**)
   4. DCL – do zarządzania uprawnieniami – **GRANT, DENY, REVOKE**
7. Postulat modyfikowania danych poprzez perspektywy – dane **można modyfikować przez perspektywę**, jeśli jest to **semantycznie sensowne**. Operacje CRUD (create, read, update, delete)
8. Postulat modyfikowania danych na wysokim poziomie abstrakcji – możemy modyfikować dane poprzez **operacje działające na obiektach (tabele, perspektywy)** a nie tylko poprzez przeglądanie w nich wierszy (np. wstawianie kolekcji rekordów, czytanie jednocześnie rekordów z różnych tabel)
9. Postulat fizycznej niezależności danych – **zmiana metody zapisu danych i dostępu do niech nie ma wpływu na aplikację**
10. Postulat logicznej niezależności danych – **zmiany w tabelach zachowujące informacje i poprawne semantycznie nie mają wpływu na aplikację**
11. Postulat niezależności więzów spójności – więzy spójności (warunki, jakie muszą spełniać dane, np. NOT NULL, UNIQUE itp.) są **definiowane w języku bazy danych** (nie muszą być w aplikacji)
12. Postulat niezależności dystrybucyjnej – dane mogą być zapisane w **różnych fizycznie miejscach** (aplikacja ma działać nawet, gdy dane są rozproszone)
13. Postulat zabezpieczenia przed operacjami na niższym poziomie abstrakcji – operacje na niższych poziomach abstrakcji **nie mogą naruszać relacyjnego modelu danych ani więzów spójności**

## 12.  Podstawowe elementy i znaczenie diagramów związków encji oraz zasady  prawidłowego projektowania schematów bazy danych.

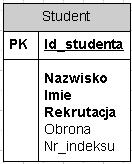
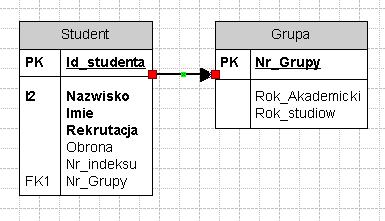
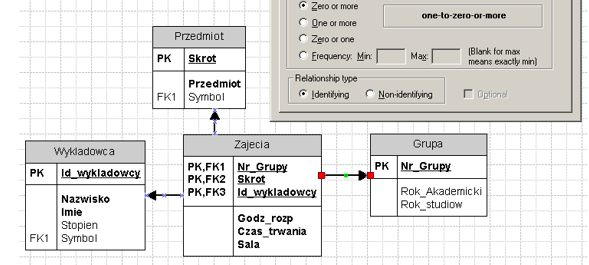
**CASE (Computer Aided System Engineering)** – narzędzia graficzne do projektowania i rysowania diagramów na ekranie komputera, generowania schematu bazy danych itp.

**Diagram związków encji** – diagram służący do przedstawienia modelu danych z pominięciem szczegółów technicznych związaną z implementacją danych w konkretnym systemie.

Diagram związków encji powinien:

* Jednoznacznie określać wymagania użytkowników i pozwolić im sprawdzić, czy analityk systemu dobrze zrozumiał ich intencje i specyfikę działania firmy
* Być istotnie prostszy od schematu bazy danych (ponieważ pomija szczegóły implementacyjne, którymi zajmuje się projektant baz danych)

Elementy diagramu związków encji:

1. **Encja** – obiekt, coś co istnieje, co jest odróżnialne od innych, o czym informację trzeba znać lub przechowywać. Encje reprezentowane są ramką:  
      
   W języku programowania:  
   typ encji – klasa  
   instancja – obiekt danej klasy
2. **Atrybut** – właściwość encji danego typu, opisywana pewną wartością (liczba całkowita, napis, data itp.). Zbiór atrybutów opisuje encję, a zbiór konkretnych wartości atrybutów opisuje instancję encji. **Atrybut nie opisuje związków między encjami** (np. nr miejsca w samolocie należy do encji „samolot” lub „miejsce” – nie jest atrybutem encji „bilet”, chociaż na nim też się pojawia). **Dla pierwszej postaci normalnej:** **każdy atrybut powinien mieć pojedynczą, atomową wartość**
3. **Klucz** – jednoznaczny identyfikator instancji danej encji (jak w bazach relacyjnych, jest główny i alternatywne). **Encje słabe\zależne** – takie, które w kluczu głównym mają klucz obcy
4. **Typy zmiennych (dziedziny atrybutów)** – zbiory wartości, które mogą być przyjmowane przez zmienne zapisywane w kolumnach tabel
5. **Więzy spójności** – w MS Visio są to wyrażenia określające możliwe wartości danego atrybutu, tak jak w relacyjnych bazach danych (na diagramie pojawiają się przy atrybucie w klamerkach)
6. **Indeksy** – atrybut lub ich grupa, względem której wyszukiwane są egzemplarze encji (o indeksach więcej w zagadnieniu 15)
7. **Związek** – uporządkowana lista encji określająca pewną zależność między zbiorami instancji encji. Zapisywana w notacji relacyjnej: Z(E1,…,En) (encje E1,…,En wchodzą w skład związku Z), są różne typy:
   1. **Związek binarny** – łącząca dwie encje   
      Ten przykład to też **związek jednoznaczny** (czyli „jeden-do-wiele” – po kluczu obcym widzimy, że jeden student należy do jednej grupy, ale w jednej grupie może być wielu studentów)
   2. **Nie/identyfikujący** – wartość klucza obcego po stronie stroje jeden nie/wchodzi w skład klucza głównego
   3. **Niejednoznaczny** – związek typu „wiele-do-wielu”, wymaga stworzenia **encji asocjacyjnej**
   4. **Rekurencyjny –** zachodzący między tą samą encją (np. jedna osoba jest szefem innej osoby)
   5. **Jedno-jednoznaczny** – typu „jeden-do-jeden”

Zasady projektowania bazy danych (wzięte z internetu):

1. Określenie celu, któremu ma służyć baza danych
2. Określenie tabel, które są potrzebne w bazie danych
3. Określenie pól, które są potrzebne w tabelach
4. Przypisanie polom jednoznacznych wartości w każdym rekordzie
5. Określenie relacji między tabelami
6. Udoskonalenie projektu
7. Wprowadzenie danych i utworzenie innych obiektów bazy danych
8. Zastosowanie narzędzi analitycznych (np. MS Access)

Zależność funkcyjna – oznaczana jako X->Y. W ramach krotek relacji *r* wartości atrybutów zbioru X determinują jednoznacznie atrybuty zbioru Y (np. dzień urodzenia determinuje znak zodiaku)

Postacie normalne w bazach relacyjnych:

1. Boyce’a-Codda – X->A albo X jest nadkluczem albo (nie ma zależności **nie od klucza**: częściowych – od części klucza ani przechodnich – od czegoś, co kluczem nie jest ani nie jest jego częścią)
2. Pierwsza – na przecięciu wiersza i kolumny jest pojedyncza, atomowa wartość
3. Druga – zależności częściowe nie mogą występować, ale zależności przechodnie już tak
4. Trzecia – to, co w Boyce’a-Codda (czyli brak zależności nie od klucza), ale dla atrybutów kluczowych nie patrzymy na zależności „nie od klucza” (czyli że jeżeli A jest kluczem, to ta postać normalna jest spełniona)
5. Czwarta – nie ma zależności wielowartościowych (czyli, że dany atrybut może determinować tylko jeden inny atrybut)
6. Piąta – nie ma zależności złączeniowej (zależy od wymagań biznesowych. Chodzi o to, żeby nie było zależności przejściowych, np. X->Y, Y->Z, co prowadzi do X->Z – np. dostawca X dostarcza produkty Y, dostawca X pracuje dla firmy Z, a firma Z potrzebuje produktów Y – jeżeli dana firma potrzebuje danego produktu, a dostawca dla niej pracuje, to redundantnym jest umieszczać informację o tym, że dostawca dostarcza produkt potrzebny tej firmie)

## 13.  Mechanizm współbieżności pracy wielu użytkowników w systemie zarządzania bazami danych.

**Transakcja** – ciąg odczytów i zapisów do bazy danych  
Współbieżność jest uzyskiwana poprzez **przeplatywanie ze sobą odczytów i zapisów różnych transakcji** – zajmuje się tym *moduł zarządzania transakcjami*

**Poprawność transakcji** – zgodność transakcji z więzami spójności (np. zapis numeru telefonu – nie może zawierać znaków interpunkcyjnych, liter itp.)

Aksjomaty **ACID** – wymagania stawiane przed SZBD w kwestii współbieżnego wykonywania transakcji:

* **A**tomicity – atomowość, każda transakcja jest niepodzielną operacją z punktu widzenia użytkownika (albo wykonujemy wszystkie kroki, albo żadnego) – operacje COMMIT i ROLLBACK
* **C**onsistency – spójność, po wykonaniu zbioru transakcji, stan bazy powinien być spójny (chyba że nie był przed transakcjami)
* **I**solation – izolacja, transakcje nie powinny przeszkadzać sobie wzajemnie w działaniu. Użytkownik powinien mieć iluzję, że sam korzysta z bazy danych
* **D**urability – trwałość, dane zatwierdzone przez transakcję powinny być dostępne nawet w sytuacji awarii programu, komputera lub nośnika danych

Mechanizmy współdzielenia zasobów bazy danych:

* **Blokady** (zamki) – zakładane na obiekty, ograniczają (lub uniemożliwiają) działanie innych transakcji na zablokowanym obiekcie
* **Dziennik** (log) – zapis wszystkich zmian w bazie danych, pozwalający na wycofywanie lub odtworzenie ich w przypadku awarii
* **Kopia zabezpieczająca** (backup) – wykonywana regularnie, pozwala na przywrócenie spójnego stanu bazy danych
* **Wielowersyjność** – możliwość odczytywania danych zmienianych równocześnie przez inne transakcje w takiej postaci, w jakie istniały w pewnych chwilach w przeszłości

**Plan wykonania transakcji** – ustalenie kolejności wykonywanych akcji odczytu i zapisu na obiektach bazy danych przez współbieżnie działające transakcje. Są różne typy:

* **Plan szeregowy** – wykonanie transakcji po kolei
* **Plany równoważne** – takie, które skutkują tym samym stanem końcowym
* **Plan szeregowalny** – równoważny z pewnym planem szeregowym (zapewnia izolację i spójność, ale nie gwarantuje atomowości ani trwałości)
* **Plan odtwarzalny** – umożliwiający wycofanie każdej transakcji – jest istotny do zapewnienia atomowości i trwałości

Zjawiska niepożądane przy przeplataniu akcji:

* Odczyt niezatwierdzonych danych „**dirty read**” – T1 zmienia dane, ale ich nie zatwierdza, po czym T2 je wczytuje i jest ryzyko, że je zatwierdzi, przez co T1 nie będzie mogła ich wycofać
* **Niepowtarzalny odczyt** – odczyt zatwierdzonych danych, transakcja odczytuje dwukrotnie ten sam obiekt i za każdym razem widzi inne dane – bo w międzyczasie zmieniła i zatwierdziła je inna transakcja (np. T1 sprawdza, czy można zarezerwować miejsce, i zanim je zarezerwuje to T2 zdąży je zająć)
* **Nadpisanie niezatwierdzonych danych** – T1 zmienia dane i nie zatwierdza zmian, a za chwilę T2 zmienia te dane i zatwierdza zmiany (przykład problemu: mamy atrybuty A oraz B, które mają być sobie równe. T1 zamienia je po kolei na 0, a T2 na 1 – jak będziemy przeplatać T1 oraz T2, to w pewnym momencie A oraz B będą różne)

Rodzaje blokad:

* **Współdzielona** (typu S: shared lock) – pozwala na współdzielony dostęp do zasobu, np. kilka transakcji może naraz czytać z tej samej tabeli. Jak transakcje założyły współdzieloną blokadę, to żadna inna transakcja nie może założyć blokady wyłącznej
* **Wyłączna** (typu X: exclusive lock) – dana transakcja ma wyłączne prawo do wprowadzania zmian w obiekcie – w tym czasie nie może istnieć żadna inna blokada

Protokół ścisłego blokowania dwufazowego (Strict 2PL):

1. Każda transakcja musi uzyskać blokadę S przed oczytaniem obiektu oraz blokadę X przed zapisaniem go
2. Jak transakcja trzyma blokadę X, to żadna inna transakcja nie może założyć żadnej innej blokady
3. Jak trzyma blokadę S, to żadna inna nie może założyć blokady X (ale może też założyć S)
4. Gdy transakcja nie może założyć blokady na obiekcie, to może ustawić się w kolejce oczekujących transakcji albo może zostać wycofana
5. **Wszystkie blokady trzymane przez transakcję są zwalniane jednocześnie w chwili gdy transakcja się kończy**

Strict-2PL chroni przed wcześniej wymienionymi zjawiskami niepożądanymi, ale wiąże się z ryzykiem **zakleszczenia** (deadlock) – np. T1 zakłada blokadę na obiekcie A, a T2 na obiekcie B. W kolejnym kroku, T1 chce założyć blokadę na B, a T2 na A – będą czekać na zwolnienie blokad w nieskończoność

Metody radzenia sobie z zakleszczeniami:

* Zapobieganie – ustalanie priorytetów transakcji, jeśli transakcja z niższym priorytetem blokuje tą z tym wyższym, to ta z niższym jest wycofywana
* Wykrywanie – tworzenie **grafów oczekiwań** na zwolnienie blokady (węzły to transakcje, a krawędzie to zakleszczenie), okresowo sprawdzane jest, czy w grafie jest cykl i jeśli tak, to jedna transakcja z cyklu jest wycofywana
* Zapobieganie – metoda **timeout**, transakcja jest wycofywana, jeśli przez jakiś czas jest bezczynna

**Zjawisko przeładowania** ­– jeśli jednocześnie uruchomionych jest zbyt dużo transakcji, to system działa gorzej (wykonuje coraz mniej akcji) – można ograniczyć liczbę transakcji lub rozładowywać zapotrzebowania na najczęściej używane (blokowane) obiekty (tzw. hot spots)

**Problem fantomów** – w momencie, gdy T1 szuka zbioru rekordów spełniających dany warunek, a następnie T2 doda nowy rekord, który spełnia ten warunek, T1 nie weźmie go pod uwagę

Poziomy izolacji transakcji:



* **Read uncommited** – T odczytuje obiekty w dowolnej chwili i nic nie zapisuje (nie wprowadza żadnych zmian)- brak jakichkolwiek blokad
* **Read commited** – blokada X na wykonywanie zmian, utrzymywana do końca działania + blokada S do czytania, która jest zwalniana od razu po odczytaniu (nie trzymana do końca transakcji)
  + transakcja T odczytuje tylko obiekty, których zmiany zostały zatwierdzone.
  + Żadna wartość **zmieniona** przez T nie może być zmieniona przez inną transakcję, dopóki T się nie zakończy**.**
* **Repeatable reads** – blokady zgodnie z protokołem Strict-2PL, ale bez blokad na zbiór wynikowy instrukcji SELECT
  + transakcja T odczytuje tylko obiekty, których zmiany zostały zatwierdzone.
  + Żadna wartość **ODCZYTANA** **LUB ZMIENIONA** przez T nie może być zmieniona przez inną transakcję, dopóki T się nie zakończy**.**
* **Serializable** – blokady zgodnie z protokołem Strict-2PL + blokady typu S na zbiorze rekordów wynikowych instrukcji SELECT
  + transakcja T odczytuje tylko obiekty, których zmiany zostały zatwierdzone.
  + Żadna wartość **ODCZYTANA** **LUB ZMIENIONA** przez T nie może być zmieniona przez inną transakcję, dopóki T się nie zakończy**.**
  + Wyniki instrukcji SELECT wyliczone przez transakcję T nie zmieniają się, dopóki T się nie skończy

**Optymistyczne blokowanie** – transakcja wczytuje dane do lokalnego bufora (bez blokowania ich), wykonuje swoje operacje, po czym sprawdza, czy nie ma konfliktu z odczytami i zapisami już zatwierdzonych transakcji. Jak nie ma, to przepisuje zmiany z buforów, a jak są, to transakcja jest restartowana. Przy przepisywaniu/restartowania transakcji zakładana jest blokada X

## 14.  Podstawowe obiekty, konstrukcje i znaczenie języka SQL.

**Baza danych w terminologii matematycznej to zbiór relacji**

Relacja – podzbiór iloczynu kartezjańskiego zbiorów wartości

**Tabela** – dwuwymiarowa reprezentacja relacji, składająca się z kolumn i wierszy

Cechy tabeli:

* Liczba kolumn jest z góry ustalona
* Z każdą kolumną związana jest *nazwa* oraz *dziedzina* określająca zbiór wartości, jakie mogą wystąpić w kolumnie
* Na przecięciu wiersza i kolumny znajduje się pojedyncza (atomowa) wartość należąca do dziedziny kolumny
* Wiersz reprezentuje jeden rekord informacji
* W modelu relacyjnym kolejność wierszy i kolumn nie ma znaczenia

**Klucz główny** – jednoznaczny identyfikator dla każdego wiersza. Jest zbiorem złożonym z jednej lub więcej kolumn

**Klucz jednoznaczny (alternatywny)** – to samo co klucz główny, ale klucz główny może być tylko jeden, kluczy jednoznacznych może być wiele

**Klucz obcy** – zbiór złożony z jednej lub więcej kolumn, których wartości występują jako wartości ustalonego klucza głównego lub jednoznacznego w tej samej lub innej tabeli. Pełnią rolę wskaźników do wierszy w drugiej tabeli.

Sql składa się z 4 podjęzyków (było o tym wcześniej)

Elementy instrukcji SELECT:

* SELECT – pozwala na wybranie kolumn, które chcemy odszukać w danej tabeli (może też zawierać wartości stałe, wyrażenia arytmetyczne, zapytania sumaryczne itp.)
* FROM – pozwala wybrać źródło danych (tabelę, perspektywę itp.)
* JOIN – do łączenia ze sobą tabel, ma kilka typów (słowa INNER oraz OUTER można pomijać):
  + INNER JOIN – zwraca podzbiór iloczynu kartezjańskiego obu tabel zgodnie z ustalonym warunkiem (definiowanym po ON)
  + LEFT OUTER JOIN – zwraca wszystkie wiersze lewej strony tabeli niezależnie od tego, czy są połączone z jakimiś wierszami drugiej (np. FROM Employee LEFT OUTER JOIN Dept ON Employee.deptID = Dept.DeptID zwróci nawet pracowników, którzy w żadnym departamencie nie są)
  + RIGHT OUTER JOIN – na odwrót – wszystko z tabeli po prawej + to, co udało się złączyć (nie zwróci pracowników bez departamentu, ale departamenty bez pracowników już tak)
  + FULL OUTER JOIN – łączy ze sobą left i right outer join
  + CROSS JOIN – zwraca pełny iloczyn kartezjański
* WHERE – do określania warunków, jakie muszą spełniać wiersze tabeli, by zostać wczytane
* ORDER BY – pozwala sortować wiersze w odpowiedzi instrukcji
* UNION, UNION ALL – sumowanie zbiorów wyników (jak samo UNION, to usuwane są wiersze, które się powtarzają)
* INTERSECT – część wspólna zbiorów wyników
* EXCEPT – różnica zbiorów wyników
* Zapytania sumaryczne – COUNT,AVG,SUM,MAX,MIN – podsumowują wartości w kolumnach
* GROUP BY – dzielą podsumowywane wiersze na grupy – jest to potrzebne, gdy w SELECT mamy jakieś kolumny, które nie są objęte zapytaniem sumarycznym
* SOME,ANY,ALL – SOME/ANY (to to samo) – dla dowolnej wartości ze zbioru (np. 100 >= ANY (SELECT Sal From Emp) będzie true, jeżeli w tej tabeli dowolny pracownik zarabia mniej niż\równo 100). ALL – dla wszystkich wartości ze zbioru (wtedy każdy pracownik musiałby zarabiać =<10)
* EXISTS, NOT EXISTS – do sprawdzania, czy dane zapytanie zwraca jakikolwiek wiersz

Perspektywa – wirtualna tabela, pozwala ograniczać wgląd do tabeli ze względu na np. uprawnienia. Jest definiowana poprzez wynik instrukcji SELECT:

CREATE VIEW nazwa\_pespektywy AS zapytanie;  
Jak w tym zapytaniu mamy WHERE, to dodanie na końcu WITH CHECK OPTION spowoduje, że instrukcje INSERT i UPDATE na tej pespektywie będą tyczyć się tylko wierszy, które ten WHERE spełniają.  
Dodanie na końcu WITH READ ONLY spowoduje, że nie można będzie dokonywać operacji INSERT,ALTER,DELETE

Obiekty programistyczne w SQL:

* Procedury – obiekty programistyczne służące do zapisania w bazie czynności często powtarzalnych, które mogą być używane w aplikacji baz danych (np. utworzenie zamówienia, usunięcie klienta, tworzenie zestawów rachunków). Mają różne typy parametrów:
  + IN – wartość przychodzi z wywołującej jednostki programu i nie ulega zmianie (nie można do tego parametru nic przypisać)
  + OUT – coś jak wyjście funkcji w języku programowania – nic z zewnątrz do tego nie podajemy, ale należy to zwrócić na koniec
  + IN OUT – wartość przychodzi z zewnątrz i możemy ją zmienić w środku  
    CALL – słowo kluczowe do wywołania procedury  
    SELECT INTO – do wczytania wartości z selecta do zmiennych
* Funkcja – podobna do procedury, ale musi zawsze zwracać jakąś wartość. Ponadto, funkcje mogą być wywołane w procedurze, ale procedura w funkcji już nie
* Pakiet – zgrupowane procedury i funkcje. Mogą mieć wspólne kursory/zmienne i stałe/wyjątki
* Kursory – służą do tego, by móc po kolei przeglądać wynik zapytania (coś jak pętla na kolekcji w języku programowania). Są buforem, do którego zapisywane są wiersze wynikowe zapytania  
  (CURSOR nazwa IS SELECT… - tworzy; OPEN nazwa – pozwala z niego korzystać; FETCH nazwa INTO zmienna – wczytuje kolejny wiersz z kolejki; EXIT WHEN nazwa%NOTFOUND – warunek wyjścia z pętli, gdy wiersze się skończą; CLOSE nazwa – zamyka kursor żeby zwolnić zasoby systemowe)
* Wyzwalacze – procedury, które są automatycznie uruchamiane, kiedy wydarzy się jakieś zdarzenie. Są wiązane z tabelą/perspektywą/schematem(kontem użytkownika)/całą bazą danych. Zdarzeniem odpalającym wyzwalacz mogą być instrukcje INSERT,UPDATE,DELETE. Mogą zadziałać przed/po operacji (BEFORE/AFTER – before są dobre do sprawdzania więzów spójności, after są do wykonywania stałych czynności po wykonaniu instrukcji np. aktualizacji budżetu po zwolnieniu/zatrudnieniu/zmianie stawki pracownika. Do tego mają specjalne oznaczenia :OLD oraz :NEW do wczytania wiersza przed/po zmianie.  
  INSERTING,DELETING,UPDATING – pozwalają sprawdzić w wyzwalaczu, jaka operacja go wywołała (W ifach możemy chcieć wtedy robić różne rzeczy)  
  Wyzwalacze na bezie danych operują z kolei na operacjach CREATE,ALTER,DROP,GRANT,REVOKE,SERVERERROR,LOGON,LOGOFF,STARTUP,SHUTDOWN  
  wyzwalacz typu INSTEAD OF – specjalny wyzwalacz dla perspektyw. Jeśli perspektywa jest w rzeczywistości złączeniem kilku tabel, to taki wyzwalacz może pozwolić na realizację INSERT do tej perspektywy jako ciąg operacji INSERT w kilku tabelach.

## 15.  Podstawowe zasady optymalizacji zapytań, w tym rodzaje i znaczenie indeksów w bazie danych.

Zapytania w SQL mają charakter **deklaratywny** – określają **co** ma być wyznaczone w bazie danych, ale **nie jak** ma być znalezione

**Optymalizator zapytań** – moduł SZBD, który rozważa różne sposoby realizacji zapytania, szacuje ich koszt i wybiera możliwie najlepszy z nich

Przetwarzanie zapytania:

1. Zapytaniem zajmuje się najpierw **parser** – dokonuje on analizy składniowej zapytania
2. Po analizie składniowej uruchamiany jest **optymalizator zapytań**. Składa się z 2 modułów:
   1. **Generator planów** – moduł generujący możliwe plany wykonania zapytania
   2. **Estymator kosztu** – moduł obliczający przybliżony koszt wykonania zapytania według danego planu
3. Przy szacowaniu kosztu planu, estymator korzysta z informacji statystycznych zapisanych w **słowniku danych (katalogu systemowym)**. Te informacje to: liczba rekordów w pliku, liczba stron, na których zapisane są rekordy w pliku, liczba różnych wartości w kolumnie, rozkład wartości w kolumnie (histogram)
4. Optymalizator wybiera plan o najniższym koszcie i przekazuje go do **ewaluatora planu** – modułu wykonującego zapytanie
5. Gdy zapytanie zostało wcześniej przeanalizowane i kontekst jego użycia się nie zmienił, system może użyć wyliczony wcześniej plan

Plan wykonania zapytania obejmuje:

* **Drzewa operatorów** SQL
* **Metody dostępu** do każdego wystąpienia tabeli w tym drzewie
* **Metody realizacji** dla każdego wystąpienia operatora relacyjnego w tym drzewie

Pożądane są dwie własności planu wykonania zapytania:

1. **Działanie w miejscu** – niewykorzystywanie tabel tymczasowych (tymczasowe wyniki nie są zapisywane na dysku, a jedynie wskazywane przez kursory przeglądające rekordy w plikach) – plany mające postać drzewa skierowanego w lewo + metody Simple Nested Loops Join oraz Index Nested Loops Join to umożliwiają
2. **Przetwarzanie potokowe** – wynik jednego operatora w planie wykonania zapytania jest przekazywany na wejście kolejnego operatora – przy okazji zapewnia to też działanie w miejscu

**Simple Nested Loops Join** – metoda łączenia rekordów polegająca na przeglądaniu po kolei każdego rekordu z pierwszego pliku, i dla każdego rekordu przejrzenie wszystkich rekordów z drugiego pliku w celu znalezienia par, które dadzą się ze sobą złączyć

**Index Nested Loops** – przeglądamy po kolei każdy rekord z pierwszego pliku, i w drugim pliku szukamy rekordów do połączenia poprzez indeksy

Ogólne strategie optymalizacyjne:

* Dokonuj jak najwcześniej **selekcji zmniejszającej liczbę rozważanych rekordów** (zwłaszcza gdy wynik selekcji trafia do złączenia – najbardziej kosztownej operacji
* Do wykonania selekcji stosuj **indeks**
* Gdy explicite nie są stosowane operatory złączeń, **staraj się wiązać selekcje z iloczynem kartezjańskim w celu zidentyfikowania rodzaju złączenia**
* Do wykonania złączenia stosuj **indeks na tabeli wewnętrznej**
* Wybierz plan działający w miejscu – **bez tymczasowych tabel**, stosuj przetwarzanie potokowe
* Zamiast operatora złączenia **stosuj klaster**, który umożliwia działanie w miejscu
* Ograniczaj się do przechodzenia indeksów a nie tabel (**strategia tylko-indeks**)
* Zastosuj obliczone już wcześniej wyniki **perspektywy zmaterializowanej**
* Wyszukuj **wspólne podwyrażenia** i obliczaj je tylko raz
* Gdy nie jest możliwe działanie w miejscu, dokonuj **jak najwcześniej projekcji** zmniejszając rozmiar zapisywanych tymczasowo rekordów (projekcja – ograniczenie pliku rekordów do wybranych pól, pominięcie kolumn, które w SELECT się nie znajdują)
* **Przetwórz wstępnie** plik rekordów we właściwy sposób (sortowanie, haszowanie)
* **Gromadź statystyki ilościowe** dotyczące tabel, kolumn i indeksów
* **Szacuj koszt każdego planu** i wybieraj plan o najmniejszym koszcie
* **Zapamiętuj plan wykonania zapytania** aby móc ten plan zastosować w tych samych warunkach

**Indeks** – struktura danych na dysku umożliwiająca szybkie wyszukiwanie danych w bazie danych na podstawie wartości klucza wyszukiwania (np. osoby po nazwisku). Ma takie samo znaczenie jak skorowidz w książce

**Klucz wyszukiwania** – wybrane pola rekordu względem których ma się odbyć wyszukiwanie

Rodzaje indeksów:

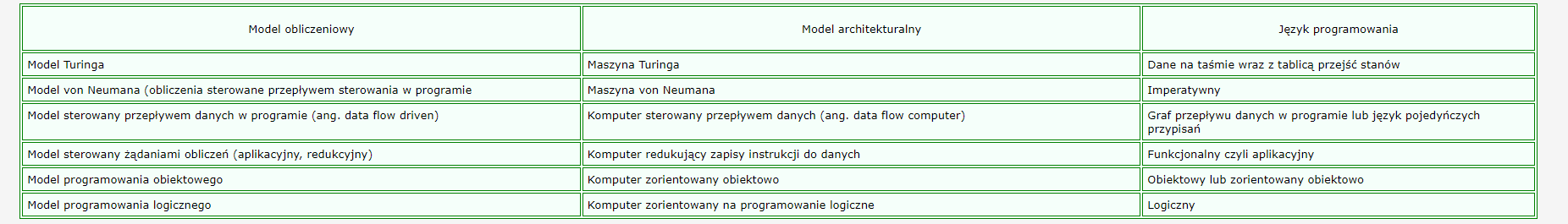
* Wewnętrzny/zewnętrzny – plik indeksu zawiera w sobie plik danych/ plik indeksu i plik danych są rozłączne
* Indeks główny/niegłówny – jego klucz wyszukiwania zawiera klucz główny/nie zawiera klucza głównego
* Indeks jednoznaczny – klucz wyszukiwania zawiera klucz jednoznaczny
* Indeks pogrupowany – indeks **wewnętrzny** i plik danych **posortowany** według wartości klucza indeksu

Metody implementacji indeksu:

* Drzewo ISAM (indexed sequential access method) – statyczne drzewo oparte na wyszukiwaniu binarnym
* Drzewo B+ - dynamiczne drzewo oparte na wyszukiwaniu binarnym
* Indeks haszowany – statyczny plik haszowany

## 16.  Model architekturalny komputera wg. von Neumanna a model obliczeniowy komputera na podstawie maszyny Turinga i ich rola w informatyce.

**Model obliczeniowy** – określa w jaki sposób będą programowane i wykonywane obliczenia w programie. Składa się z modelu architekturalnego systemu i języka programowania



**Model obliczeniowy Turinga**:

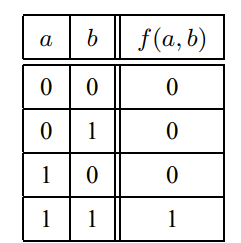
* Pierwszy uniwersalny model obliczeniowy
* Służy porównywaniu własności różnych algorytmów i programów pochodzących z różnych komputerów
* Przez teoretyków informatyki jest używany do dzisiaj
* Składa się z następujących elementów:
  + Skończony alfabet symboli: a,b,c,…,m
  + Skończony zbiór stanów: S0,S1,S2,…,Sk
  + Nieskończona taśma z polami na zapis symboli
  + Głowica czytająco/pisząca na taśmie, która może przesuwać się o jedno pole w zadanym kierunku
  + Diagram przejść między stanami – tablica przejść określająca następny stan, zapis symbolu pod głowicą i kierunek następnego ruchu
  + Działanie mechanizmu sterowania:
    - Maszyna w stanie „i” czyta znak „z” pod głowicą
    - Dla stanu „i” oraz znaku „z” maszyna określa z tabeli przejść:
      * Stan, do którego ma przejść
      * Znak, który ma być wpisany w polu pod głowicą
      * Kierunek ruchu głowicy o 1 miejsce L – w lewo, R – w prawo
    - Głowica wpisuje nowy znak i przesuwa się w zadanym kierunku
  + Instrukcje to: znak, stan, kierunek

**Model obliczeniowy von Neumana (klasyczny)**:

* Obliczenia są wykonywane na danych zapisanych w pamięci i rejestrach komputera
* Kolejność wykonywania instrukcji jest ustalona przez programistę:
  + Poprzez kolejność ich zapisu
  + Poprzez instrukcję przemieszczenia (zmiany) sterowania: skok (go to), wywołanie podprogramu (np. wywołanie funkcji), powrót z podprogramu (return) itp.
* Kolejność instrukcji jest wyznaczana przez **licznik rozkazów (program counter)** – zawiera on adres następnej instrukcji w pamięci po aktualnie wykonywanej
* Jeśli aktualna instrukcja nie zawiera rozkazu zmiany sterowania, to ustawia ona nową zawartość licznika rozkazów – podaje nowy adres następnej instrukcji do wykonania
* Korzysta z **języków imperatywnych** ­– takich języków programowania, w których programista określa bezpośrednio (explicite) i dokładnie jakie obliczenia mają się wykonać i dokładnie w jakiej kolejności
* Komputer w tym modelu składa się z 4 elementów:
  + Pamięci komputera – zawiera dane programu i jego instrukcje (wspólna pamięć na obie te rzeczy)
  + Jednostki sterującej – pobiera dane i instrukcje z pamięci i dba o ich sekwencyjne przetwarzanie
  + Jednostki arytmetyczno/logicznej – wykonuje podstawowe operacje arytmetyczne
  + Urządzeń wejścia/wyjścia – służą do interakcji z operatorem
* Ten model jest też opisany w zagadnieniu 53

## 17.  Logika boolowska i jej zastosowania w warstwie sprzętowej komputerów.

**Funkcja boolowska** – odwzorowanie gdzie B = {0,1} to zbiór wartości funkcji, n to liczba argumentów funkcji. Jest to matematyczny model **układu kombinacyjnego**. Sprowadza się do używania 3 operacji: or,and,not

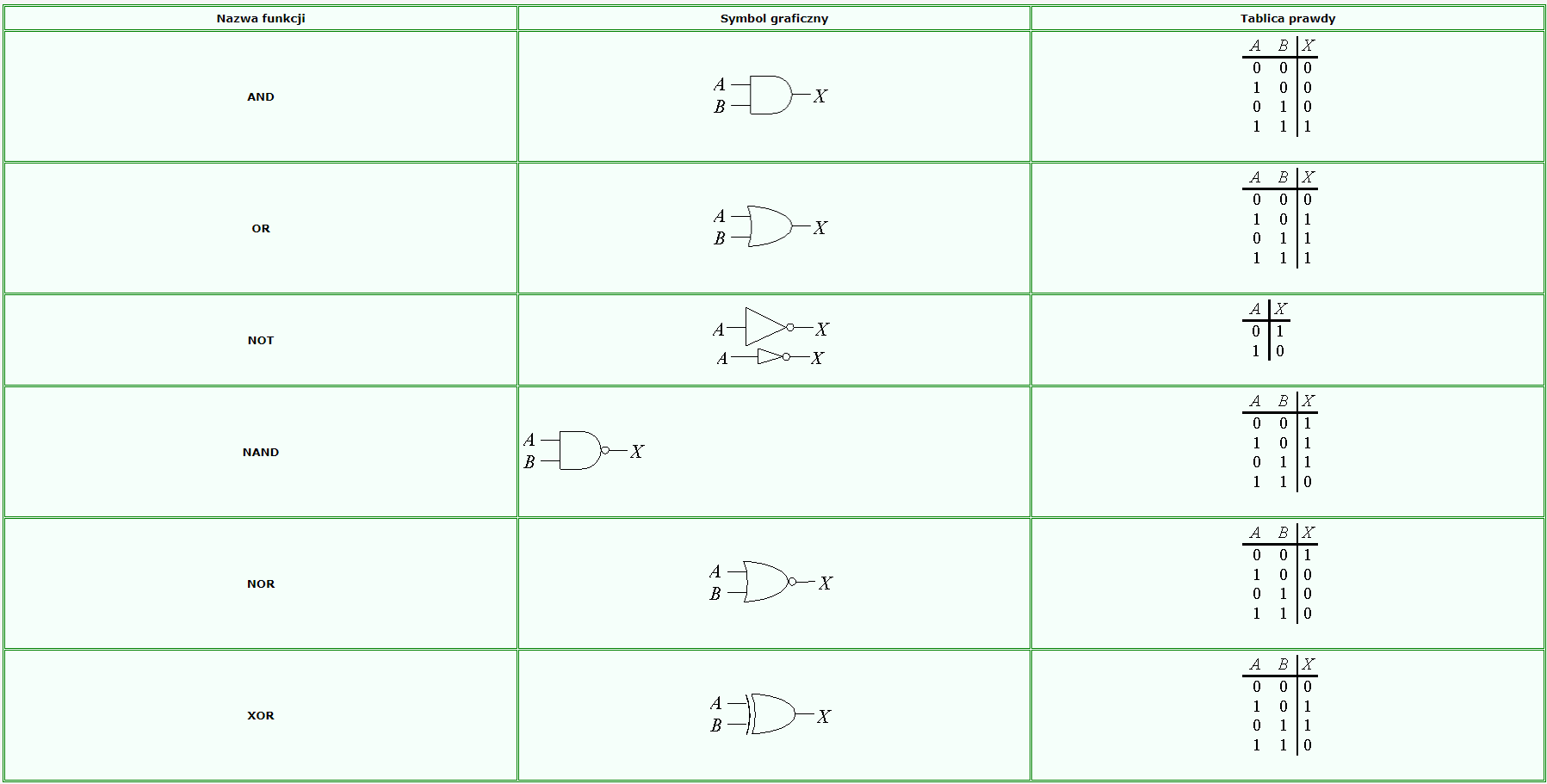
 - **tabela prawdy** dla funkcji logicznej AND (iloczyn)

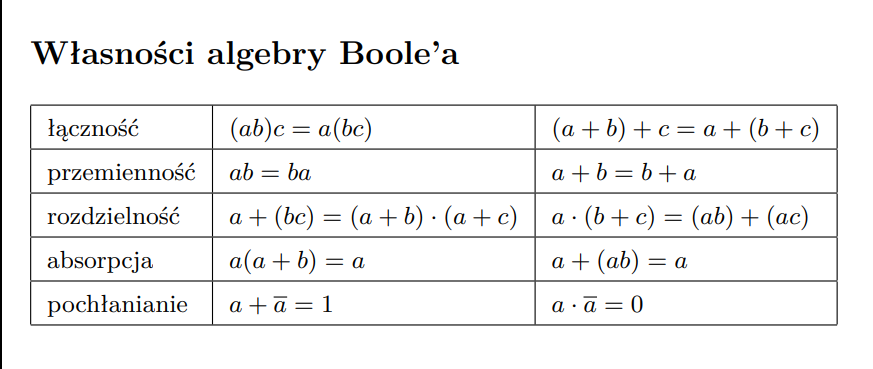
**Sumacyjna postać kanoniczna** (DNF – disjunctive normal form) – funkcja f jest sumą iloczynów Wyrażenie w nawiasie odpowiada jednej jedynce w tabeli. Dla funkcji AND:

**Iloczynowa postać kanoniczna** (Conjunctive Normal Form)– funkcja f jest iloczynem sum Wyrażenie w nawiasie odpowiada jednemu zeru w tabeli. Dla funkcji AND: (W nawiasie jest negacja każdej z tych rzeczy, dlatego jest a LUB b, to odpowiada sytuacji a=0,b=0)

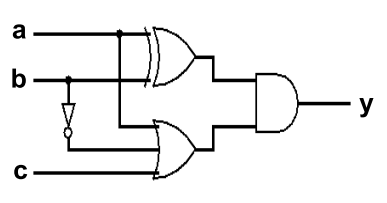
**Zapis dziesiętny** – zapisujemy w nawiasie, jakie liczby w systemie dziesiętnym odpowiadają prawdzie w tabeli prawdy danej funkcji po przekształceniu tych liczb na liczby binarne, np. dla funkcji AND zapis dziesiętny to Sum(3) (bo 11 to true), dla OR: Sum(1,2,3)

Bramki logiczne:





**Schemat logiczny** – schemat opisujący logiczną strukturę funkcji boolowskich przy pomocy bramek logicznych. Przepływ informacji jest od wejścia do wyjścia, y = f(a,b,c)  
Przykładowy schemat logiczny:



Przedstawia on funkcję:

(Tutaj XOR jest rozbity na iloczyny-AND i sumy-OR)

Funkcje boolowskie mogą być sobie **równoważne**, np.  <- oznacza to, że przy realizacji tych funkcji możemy użyć różnych zestawów bramek logicznych

Gdy projektujemy układ kombinacyjny, staramy się **minimalizować jego koszt**, robi się to poprzez:

* Minimalizację liczby bramek
* Redukcję liczby wejść bramek
* Zmniejszenie różnorodności bramek
* Redukcję czasu projektowania układu

**Logika klasyczna** – logika korzystająca z operatorów:

* Koniunkcji -
* Alternatywy -
* Implikacji -
* Negacji -

Przy budowaniu układów jest **nadmiarowa** – część tych funkcji da się zrealizować przy pomocy pozostałych, zmniejszając przy tym różnorodność bramek.

Najmniejsze systemy to:

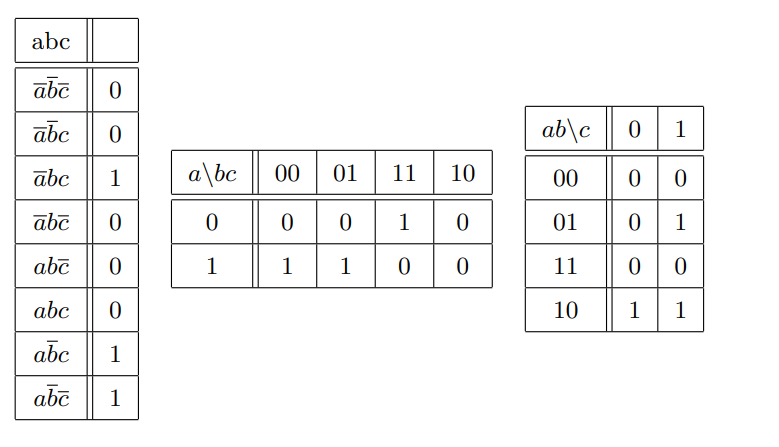
* Implikacyjno-negacyjny – tylko oraz
* Koniunkcyjno-negacyjny – tylko oraz
* Alternatywno-negacyjny - tylko oraz

**Bramki NOR oraz NAND są samowystarczalne** (obie mogą zrealizować dowolną funkcję boolowską)

**Kod Graya** – dwójkowy kod bezwagowy niepozycyjny, który charakteryzuje się tym, że dwa kolejne słowa kodowe różnią się tylko stanem jednego bitu. Jest też **cykliczny** – pierwsze i ostatnie słowo w kodzie też się różnią tylko 1 bitem.  
Zastosowanie kodu Graya – przetworniki analogowo-cyfrowe, czujniki położenia/obrotu (bo tam występują po sobie kolejne wartości), etykietowanie pojedynczych procesorów w sieci będącej hiperkostką

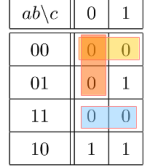
**Mapy Karnaugh’a** – mapy służące do minimalizacji funkcji boolowskiej. Są wypełniane w oparciu o tablicę prawdy. Zmienne w wierszach i kolumnach są uporządkowane zgodnie z kodem Graya, co ułatwia zastosowanie reguły sklejania

Przykładowe mapy Karnaugh’a:



Możemy grupować zera/jedynki w wielokrotności dwójki (pary, czwórki itp. tabela się „zapętla” – górny wiersz łączy się z dolnym, prawa kolumna łączy się z lewą) i zamienić je na iloczynową (dla zer) lub sumacyjną (dla jedynek) postać kanoniczną – pozwala się to pozbyć tej zmiennej, która w danej grupie się zmienia.

Dla tego przykładu:

* Sumacyjna postać kanoniczna (z pozbyciem się pary jedynek) –   
  ( – dla pary jedynek na dole: c się zmienia, a=1,b=0. niczego się nie pozbywamy, pole z jedynką dla którego a=0,b=1,c=1)
* Iloczynowa postać kanoniczna (z pozbyciem się trzech par zer) –   
  - według tej tabelki – pierwszy nawias to żółta para, drugi to czerwona, a trzeci to niebieska (**w nawiasach są negacje**)

## 18.  Zapis binarny liczb całkowitych, zapis zmiennoprzecinkowy liczb rzeczywistych, arytmetyka komputerowa.

**Zapisy binarny liczb całkowitych:**

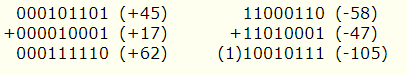
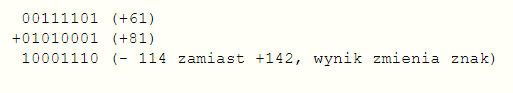
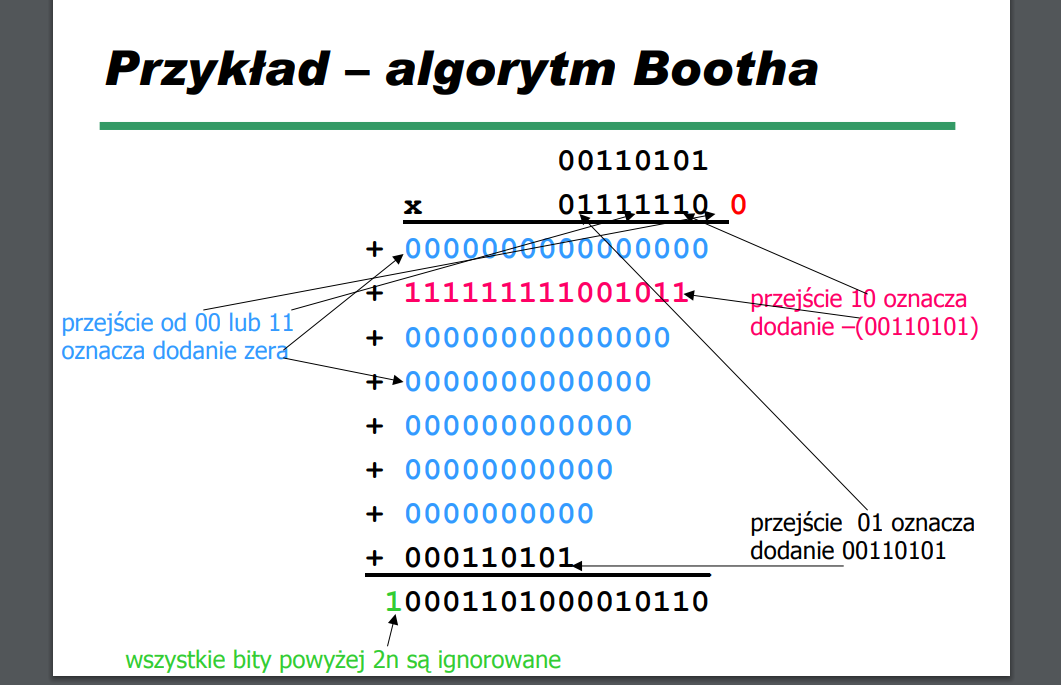
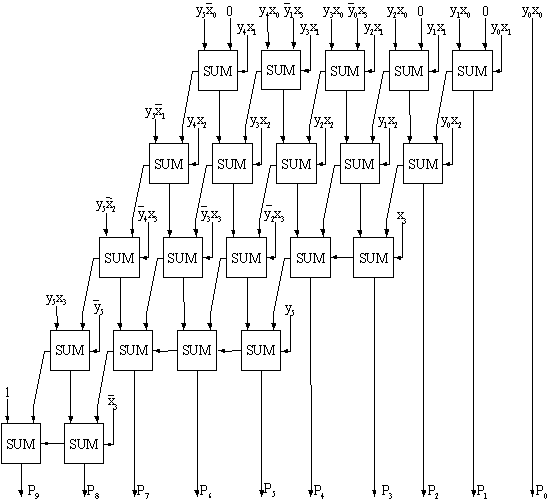
1. **Zapis znak moduł** – najbardziej znaczący bit (najbardziej po lewej) oznacza znak: 0-dodatni, 1-ujemny. Reszta bitów po prostu oznacza liczbę, np.  
   0101 = 5, 1101 = -5  
   Wada: zero nie ma jednoznacznej reprezentacji (100=0,000=0 – są dwa sposoby)  
   Ten zapis jest niewygodny dla realizacji operacji arytmetycznych
2. **Zapis znak moduł odwrotny** – najbardziej znaczący bit oznacza znak (tak jak w poprzednim), ale dla liczb ujemnych, pozostałe bity są „na odwrót”, np.  
   0100=4, 1011=-4, 0000=0, 1111=0  
   Łatwiej się w nim dodaje, niż w poprzednim zapisie. Wciąż mamy problem niejednoznacznej reprezentacji zera. Używany w procesorach motorola.
3. **Uzupełnienie do 2** – najbardziej znaczący bit jest ujemny (jego wartość zależy od długości zapisu liczby), reszta jest dodatnia, np.  
   1000 = -8, 1001 = -7, 1111 -1  
   Ten system jest wygodny do operacji arytmetycznych – zero ma tylko jedną reprezentację  
   Wada – reprezentuje więcej liczb ujemnych niż dodatnich – **ryzyko przepełnienia arytmetycznego**, np.  
   0101=5,0110=6 -> 5+6=0101+0110=1011=-5  
   Żeby odwrócić liczbę w tym systemie, należy:
   1. Zanegować całą liczbę
   2. Dodać (arytmetycznie) liczbę równą 1 (same zera i jedynka na końcu) – dodawanie liczb binarnych można robić normalnie pod kreską  
      (tutaj też może dojść do przepełnienia, np. dla x=-8=1000 – po negacji dalej mamy -8,bo wychodzi 0111+1)
4. **Zapis dziesiętny kodowany dwójkowo** – inaczej BCD (Binary Coded Decimal), każda cyfra liczby jest oddzielnie kodowana binarnie (w blokach po 4 zera/jedynki) + jeden bit znaku na samym początku, np.  
   0 0001 0010 0111 = 127, 1 0001 0010 0111 = -127  
   Zapis ten jest często stosowany w kalkulatorach elektronicznych i niektórych komputerach  
   Liczenie na tych liczbach wymaga specjalnej budowy sumatorów, które dbają o to, by przenosić liczby z 4-cyfrowych bloków, gdy przekroczą one wartość 9 (np. 1100=12 > 9 – to nie może być cyfrą, trzeba odjąć 9 z tego bloku i dodać 1 do następnego)

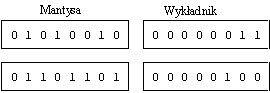
**Format słowa** – budowa słowa z pól bitowych i ich przeznaczenie  
**Pojedyncza precyzja** – liczba jest zapisana w jednym słowie komputera  
**Wielokrotna precyzja** – liczba jest zapisana w co najmniej kilku słowach

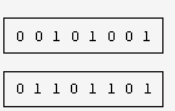
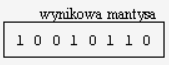
Reprezentacja ułamków (liczb rzeczywistych):

1. **Format stałoprzecinkowy** – mamy jeden bit na znak, pozostałe na liczby. Bity na liczby podzielone są na **część całkowitą** i **część ułamkową**. Miejsce przecinka binarnego jest umowne (nie ma na nie odrębnego bitu).  
   część całkowita – potęgi liczby 2 rosnące w lewą stronę od przecinka  
   część ułamkowa – ujemne potęgi liczby 2 (1/2, 1/4, 1/8…) malejące w prawą stronę od przecinka  
   Przecinek bitowy – między bitami oznaczającymi 1 oraz 0,5  
   Ułamki możemy reprezentować jedynie z pewną dokładnością, zależną od ilości wykorzystanych bitów  
   **skala obliczeń** – w momencie, gdy dla danej liczby są przeprowadzane obliczenia z liczbą reprezentowaną w innym zakresie, przecinek tej liczby jest przesuwany (domyślna skala=0, jak skala jest dodatnia – przecinek przesuwamy w lewo, jak ujemna – przecinek jest przesuwany w prawo), np.  
   00101,010=5.25 w skali 0 = 0,0010101 w skali 5 = 01010100 w skali -4
2. **Format zmiennoprzecinkowy** – liczba jest przedstawiana w postaci iloczynu:  
   A=MpCA – dana liczba, M – **mantysa** liczby A, p – **podstawa** używanego systemu (w systemie binarnym to jest 2), C – **cecha (wykładnik)** użytej potęgi podstawy  
   Cecha i mantysa są przedstawiane osobno w postaci dwóch stałoprzecinkowych liczb binarnych w zadanym zapisie binarnym (znak-moduł, uzupełnień do 2 itp.)  
   Mantysa ma znak i przecinek, może być dodatnia/ujemna oraz całkowita/ułamkowa/mieszana  
   Cecha jest zawsze całkowita, ale też ma znak i może być dodatnia/ujemna. Cecha pełni rolę skali liczbowej (ona przemieszcza przecinek) z formatu stałoprzecinkowego  
   np. dla 3-bitowej cechy (zakres cech to <-4,3>, bo w tym przykładzie jest zapis uzupełnienia do 2) i 4-bitowej mantysy:  
   Cecha=000, Mantysa=0100 -> liczba to ½, Cecha=000, Mantysa=0101 -> liczba to 5/8 (jak cecha jest 0, to tylko jedna cyfra mantysy to część całkowita)  
   Cecha = 010, Mantysa=0100 -> liczba to 2 , Cecha=010, Mantysa=0101 -> liczba to 5/2  
   Cecha = 110, Mantysa=0100 -> liczba to 1/8 )  
   **Postać znormalizowana** – taka, w której moduł mantysy przyjmuje największą możliwą wartość dającą się zapisać w ustalonym formacie stałoprzecinkowym mantysy – ustawiamy możliwie maksymalną mantysę i manipulujemy cechą tak, by uzyskać daną liczbę (dla uzupełnień do 2 mantysa to nie zawsze będą same jedynki, bo niezależnie jaką cechę damy, to jakiejś liczby możemy nie być zdolni reprezentować)

Operacje arytmetyczne na liczbach binarnych:

1. **Sumowanie (odejmowanie) w zapisie znak-**moduł – dodajemy jak pod kreską w systemie dziesiętnym, odejmowanie to tak naprawdę dodanie odwrotności danej liczby  
   Mamy tutaj ryzyko **przepełnienia arytmetycznego** – jak na najbardziej znaczącej pozycji mamy dwie jedynki, to dochodzi do przeniesienia na pozycję znaku – trzeba wtedy przesunąć obie liczby binarnie w prawo (dodać zero na końcu – wtedy znak też jest o jeden bit dalej)
2. **Sumowanie (odejmowanie) w zapisie uzupełnień do 2** – też można to robić pod kreską. Przeniesienie z pozycji znaku jest pomijane (nic się nie stanie, jak wyjdziemy za bit znaku – ignorujemy to)  
   ****  
   tutaj też może dojść do nadmiaru:  
   ****  
   Jak to się stanie, to aktywowana jest **flaga nadmiaru** procesora i sygnalizowany jest błąd do systemu operacyjnego  
   Odejmowanie=dodanie negacji
3. **Mnożenie/dzielenie liczb całkowitych (bez znaku)** – w wykładach tego nie ma, ale można to też robić „pod kreską”
4. **Mnożenie w zapisie uzupełnień do 2** – są 3 główne metody
   1. **Metoda przesuń i dodaj** – to, co mnożymy (mnożną) dodajemy do siebie tyle razy, ile wynosi to, przez czym mnożymy (mnożnik). (Ogólnie to mnożenia na studiach nie mieliśmy)
   2. **Metoda Booth’a/Wallace’a** – w zależności od przejścia w bitach w mnożniku, do wyniku dodajemy/odejmujemy mnożną lub przesuwamy całość w lewo:  
      ****
   3. **Metoda macierzowa** – mamy sumatory elementarne (1-bitowe) poukładane w warstwy. Mamy tyle wierszy, ile bitów mają mnożone liczby. Sumatory są ułożone w następujący sposób:  
      
5. **Dla liczb stałoprzecinkowych** – zależy od zapisu, robi się to tak samo jak dla liczb całkowitych
6. **Arytmetyka zmiennoprzecinkowa** – ma kilka kroków:
   1. Porównanie cech obu liczb
   2. Uzupełnienie mniejszej cechy do wartości większej + zmniejszenie jej mantysy tak, by to była wciąż ta sama liczba (dodanie 1 do cechy = pomnożenie przez 2 -> trzeba podzielić mantysę przez 2 = przesunięcie mantysy o jedno miejsce w prawo)
   3. Dodanie\odjęcie mantys

Przykład dla liczb 41 (5,125\*2^3) oraz 109 (6,8125\*2^4)  


Dla pierwszej liczby – musimy pomnożyć wykładnik (cechę) przez 2 -> dodajemy do niej jedynkę i przesuwamy jej mantysę w prawo (dostajemy 2,5625\*2^4)  
  
Gdy cechy są równe, możemy je „wyciągnąć przed nawias” i normalnie dodać:  
 =   
Do tego mamy jeszcze jeden bit z przodu – bit rozszerzenia (na wypadek gdyby był nadmiar, mamy nowy bit do oznaczenia znaku) – tu nam się nic nie „wylało”, więc ten bit ma wartość 0  
Żeby się pozbyć bitu rozszerzenia – znowu zwiększamy cechę i przesuwamy mantysę w prawo:  
  
Jak przy tym mamy przepełnienie cechy – błąd jest sygnalizowany do użytkownika (tak samo, jakbyśmy chcieli znormalizować wynik poprzez zmniejszenie cechy i przesunięcie mantysy w lewo, gdy cecha już jest równa zero)  
**mnożenie zmiennoprzecinkowe** – cechy do siebie dodajemy, mantysy mnożymy

## 19.  Miary efektywności obliczeniowej procesorów, pojemności pamięci komputerowej oraz wydajności systemów obliczeniowych.

**Mikroprocesor** – procesor zbudowany w technologii układów scalonych, przeważnie w postaci jednego układu scalonego. Układ scalony – każdy element jest robiony w miniaturowej wersji metodą domieszkowania zachodzących na siebie obszarów na powierzchni płytki zrobionej z półprzewodnika (zazwyczaj krzemu)

Dawniej, procesory były budowane w **technologii układów dyskretnych** – czynne elementy elektroniczne (modyfikujące wartości wyjściowe prądów i napięć poprzez sterowanie tymi wielkościami na wejściach – diody, tranzystory) i bierne elementy (np. oporniki, kondensatory) były osobnymi częściami, które były następnie montowane na płytach.

**Parametry mikroprocesorów:**

* Cechy architekturalne:
  + Liczba i cechy bloków wykonawczych – liczba jednostek wykonujących operacje określone w instrukcjach programu w języku wewnętrznym oraz ich cechy funkcjonalne
  + Struktura i parametry pamięci – liczba poziomów pamięci (w tym liczba poziomów pamięci podręcznej). Organizacja zapisywanej informacji, pojemności
  + Cechy i parametry listy rozkazów – typy i format instrukcji języka wewnętrznego, dostępne tryby adresowania, metoda realizacji instrukcji wejścia/wyjścia  
    ***Trzy kolejne określają wydajność obliczeniową mikroprocesora:***
  + **Liczba i rozmiary rejestrów danych**
  + **Liczba i rozmiary rejestrów adresowych**
  + **Szerokość szyn danych i adresów**
  + Cechy układu przerwań
  + Dołączalne koprocesory
* Parametry techniczne:
  + Częstotliwość zegara – decyduje o szybkości wykonywania operacji na komputerze  
    Szybkość obliczeń mierzymy w ***(to są te miary wydajności systemów obliczeniowych)***:  
    **MIPS (Millions of Instructions per Second)** – ilość obliczeń stałoprzecinkowych w milionach na sekundę  
    **MFLOPS (Millions of Floating Point Operations per Second)** – ilość obliczeń zmiennoprzecinkowych
  + Technologia – charakteryzuje się: stopniem scalenia układu, sposobem budowy elementów czynnych i układów logicznych (technologia integracji), liczbą bramek/tranzystorów w jednym układzie scalonym
  + Liczba tranzystorów – podzielona na stopnie integracji:  
    SSI (small scale integration) – max 100  
    MSI (medium…) – max 100  
    LSI (large…) – max 100 000  
    VLSI(very large…) – max 100 milionów  
    ULSI (ultra large…) – więcej niż 100 milionów  
    GSI (gigantic…) – więcej niż 1 miliard
  + Napięcie zasilania
  + Pobór mocy
  + Obudowa

**Procesory typu CISC (Compound Instruction Set Computer)** – komputery o złożonej liczbie rozkazów  
(100-300 rozkazów, złożone operacje, dużo trybów adresowania, mało rejestrów roboczych w procesorze, różne formaty rozkazów wewnętrznych – różnią się długością słowa rozkazowego, podziałem na pola i liczbą argumentów, różne czasy wykonania różnych rozkazów, układ sterowania jest przeważnie mikroprogramowany)

**Procesory typu RISC (Reduced Instruction Set Computer)** – komputery o zredukowanej liście rozkazów (max 128 rozkazów, rozkazy są proste, mało trybów adresowania, dużo rejestrów roboczych, mało formatów rozkazów – jednolita długość słowa rozkazowego, jednolity czas wykonania rozkazów, układ sterowania procesora jest sprzętowy) – TE SĄ POPULARNIEJSZE

**Pojemność pamięci komputerowej** – maksymalna ilość informacji, którą dane urządzenie może przechować

Jednostki pojemności pamięci:

* 1 b (bit) – najmniejsza jednostka (zero lub jeden)
* 1 B (bajt) = 8 b
* 1 kB (kilobajt) = 1024 B (2^10 bajtów)
* 1 MB (megabajt) = 1024 KB (2^20 bajtów)
* 1 GB (gigabajt) = 1024 MB (2^30 bajtów)
* 1 TB (terabajt) = 1024 GB (2^40 bajtów)
* 1 PB (petabajt) = 1024 TB (2^50 bajtów)

Tak mam w notatkach z TAKów, ale na Wikipedii jest jeszcze o tym, że te tradycyjne jednostki to są wielokrotności 1000 (a nie 1024), a ten z wielokrotnościami mają zmienione nazwy tak, że druga sylaba to jest bi (kibibajt, mebibajt, gibibajt itd.)

## 20.  Prawo Moore’a i implikacje z niego wynikające w kontekście rozwoju sprzętu komputerowego.

**Prawo Moore’a** – prawo empiryczne głoszące, że liczba tranzystorów w układzie scalonym rośnie z trendem wykładniczym – Gordon Moore z firmy Intel w 1965r. zauważył, że liczba tranzystorów podwaja się co 18 miesięcy  
Zakłada się, że liczba tranzystorów **podwaja się co 2 lata**

Wzrost liczby tranzystorów wiąże się z:

* Wzroście mocy obliczeniowej komputerów
* Wzroście rozmiaru pamięci operacyjnej
* Wzroście pojemności dysków twardych
* Wzroście przepustowości w sieciach komputerowych
* Spadek cen komputerów, ich rozmiaru i poboru mocy

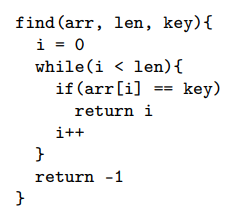
## 21.  Szacowanie złożoności algorytmów, klasy złożoności problemów algorytmicznych.

2 cechy dobrego algorytmu:

* **Poprawność** – zwraca prawidłowy wynik (zgodność z warunkiem końcowym specyfikacji algorytmu)
* **Szybkość** – jest jak najbardziej wydajny (osiąga wyniki przy możliwie minimalnym nakładzie zasobów: czasu i pamięci) – ta cecha wiąże się ze złożonością obliczeniową

Przy wyborze miary szybkości algorytmu należy uwzględnić:

* Niezależność od języka programowania/platformy
* Możliwie dużą niezależność od danych wejściowych

**Operacje dominujące** – te, które proporcjonalnie pokrywają całą pracę algorytmu. Zliczamy je w celu mierzenia szybkości algorytmu. Każda pętla lub odgałęzienie instrukcji warunkowej powinna zawierać operację dominującą. Nie być operacja, która jest wykonywana tylko raz (czyli np. return). Jeden algorytm może zawierać wiele operacji dominujących, np.:  
 operacje dominujące to: porównanie i<len, porównanie arr[i] == key, inkrementacja i++

Żeby wykonać analizę złożoności danego algorytmu należy:

1. Wyznaczyć zbiór operacji dominujących
2. Wyznaczyć funkcję argumentów (danych wejściowych), która stanowi rozmiar danych wejściowych

**Złożoność czasowa algorytmu** – liczba operacji dominujących jakie wykona algorytm jako funkcja rozmiaru danych (w zależności od rozmiaru danych) – im jest niższa, tym szybszy jest algorytm  
Rozmiarem danych może być **długość tablicy**  
W wielu przypadkach, algorytm nie ma jednej funkcji złożoności czasowej, tylko ich **zakres** (np. dla algorytmu wyżej do szukania klucza w tablicy – jak klucz jest na pierwszej pozycji, to pętla wykona się jeden raz, jeżeli jest na końcu/nie ma go w tablicy, to wykona się *len* razy)

Z tego powodu wprowadza się kilka wariantów złożoności czasowej:

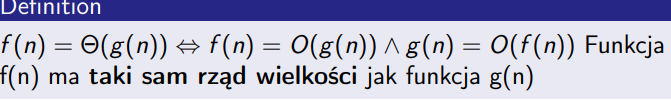
* **Pesymistyczna złożoność czasowa** – oznaczana W() – wyznacza górny kres możliwej liczby operacji dominujących dla ustalonego rozmiaru danych – W(n) =   
  (n – rozmiar danych, – zbiór wszystkich możliwych danych wejściowych o rozmiarze n, t(d) – liczba wykonanych operacji dominujących dla pewnych danych wejściowych o rozmiarze n)  
  Dla szukania klucza w tablicy: W(n) = n
* **Przeciętna złożoność czasowa** – oznaczana A() – wyraża przeciętną liczbę wykonanych operacji dominujących przy założeniu pewnego **modelu losowości danych** wejściowych  
  (n – rozmiar danych, – zmienna losowa oznaczająca faktyczną liczbę wykonanych operacji dominujących dla losowych danych o rozmiarze n, – prawdopodobieństwo, że liczba wykonanych operacji dominujących wyniesie k dla losowych danych o rozmiarze n)  
  Ten typ złożoności wymaga założenia **modelu losowości danych**, czyli to, z jakim prawdopodobieństwem uzyskamy dany zestaw danych dla danej wielkości.  
  Dla szukania klucza w tablicy: A(n)=(n+1)/2 – intuicyjnie, klucz będzie średnio w połowie tablicy

**Pamięciowa złożoność algorytmu** – oznaczana S(n) – liczba jednostek pamięci głównej użyta przez algorytm jako funkcja rozmiaru danych – ta też ma wersję pesymistyczną i przeciętną  
Czasem jest tak, że algorytm zużywa zawsze stałą ilość pamięci: S(n) = const lub S(n) = O(1)

Dlatego, że szybkość jest też zależna od konkretnej implementacji (np. użytego języka programowania i sprzętu), skupiamy się na **charakterze tempa wzrostu**

W celu pominięcia nieistotnych szczegółów (np. stałe: A(n) = 3,45 \* n + 2 -> 3,45 oraz +2 można pominąć, A(n) to funkcja liniowa od n) wykorzystywana jest **notacja asymptotyczna**

**Elementy notacji asymptotycznej**:

* Górne nieostre ograniczenie – <= - duże o – O() - , np. dla nie jest prawidłowe, bo funkcja f ma wyższy rząd wzrostu)
* Taki sam rząd wielkości – duże teta - = – θ() – funkcja ma dokładnie taki sam rząd wielkości jak inna funkcja  
    
  (dla tamtego przykładu )
* Dolne nieostre ograniczenie – >= - wielkie omega Ω()
* Górne ostre – < - małe o o()
* Dolne ostre - > - małe omega ω()

Najczęściej spotykane rzędy wielkości (rosnąco)

1. Stała O(1)
2. Logarytmiczna O(log(n))
3. „Pierwiastkowa” – nie ma nazwy
4. Liniowa O(n)
5. Liniowo-logarytmiczna O(n\*log(n))
6. Kwadratowa
7. Sześcienna – potem kolejne potęgi
8. Pod-wykładnicza
9. Wykładnicza
10. Silniowa
11. „N do N”

**Klasa złożoności** – zbiór problemów obliczeniowych o podobnej złożoności obliczeniowej, wiąże się z pojęciem deterministycznej i niedeterministycznej maszyny Turinga

**Deterministyczna maszyna Turinga (DTM)** – model obliczeń składający się z 5 elementów:

* Q – zbiór stanów sterowania maszyny
* S – alfabet taśmy
* d – funkcja przejścia – d: Q x S -> Q x S x {R,L,N} (na podstawie stanu maszyny i symbolu na danej pozycji decyduje o zmianie aktualnego stanu, zmianie aktualnego symbolu i przemieszczeniu czytnika na taśmie)
* q0 – początkowy stan sterowania
* F – zbiór końcowych stanów sterowania

Ten wariant maszyny jest **ściśle zdefiniowany** w taki sposób, by dało się określić złożoność czasową danego algorytmu – operacją dominującą jest pojedynczy krok maszyny

**Niedeterministyczna maszyna Turinga (NDTM)** – taka sama jak deterministyczna, ale funkcja przejścia d(q,s) może mieć kilka różnych wartości (ten sam stan i znak mogą dać różny efekt) – program ma kilka możliwych ścieżek, wynik obliczeń jest pozytywny jeśli choć jedna z możliwych ścieżek doprowadzi do sukcesu. Tego typu maszyny Turinga nie udało się nikomu zbudować  
Analogią NDTM mogłaby być funkcja forecast() zwracająca 0 lub tak, „żeby było dobrze”

Klasy złożoności algorytmów:

* **Klasa P** – dla danego problemu istnieje DTM rozwiązująca dany problem w czasie wielomianowym względem rozmiaru danych wejściowych
* **Klasa NP** – dla danego problemu istnieje NDTM rozwiązująca danych problem w czasie wielomianowym. Wszystkie problemy klasy P należą również do klasy NP (bo DTM to szczególny przypadek NDTM)

**Problem NP-zupełny** – taki, który należy do klasy NP i do którego da się sprowadzić każdy problem z klasy NP w czasie wielomianowym (problem uniwersalny (najtrudniejszy) w klasie NP). Jak rozwiążemy problem NP-zupełny, to rozwiążemy też wszystkie inne problemy.  
Przykładami problemów NP-zupełnych są: problem plecakowy (ten z ograniczonym plecakiem i elementach o ustalonej wielkości i wartości), problem spełnialności SAT (dla danej formuły logicznej zdań, jak znaleźć takie podstawienie true/false by formuła na wejściu była prawdziwa)

**Problem NP-trudny** – problem pozwalający rozwiązać wszystkie problemy z klasy NP, ale sam niebędący częścią klasy NP. Problemy NP-zupełne zazwyczaj mają postać „czy istnieje…”, a problemy NP-trudne to ich optymalizacyjne wersje - „znajdź najmniejszy…”

## 22.  Najważniejsze algorytmy wyszukiwania i sortowania, przegląd i zastosowania.

**Zasada „dziel i rządź”** – jedna z najskuteczniejszych technik projektowania algorytmów. Dzielimy problem na podproblemy (dane wejściowe na mniejsze części) i wyjaśniamy jak z rozwiązań tych podproblemów otrzymać rozwiązanie oryginalnego problemu. Często wykorzystuje rekurencję

W wyszukiwaniu, **operacją dominującą** jest **porównanie** (między elementem szukanym a elementem w ciągu danych wejściowych), **rozmiarem danych** jest **długość ciągu**

Algorytmy wyszukiwania:

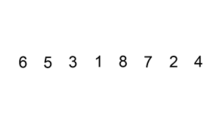
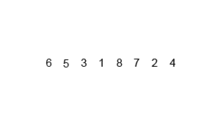
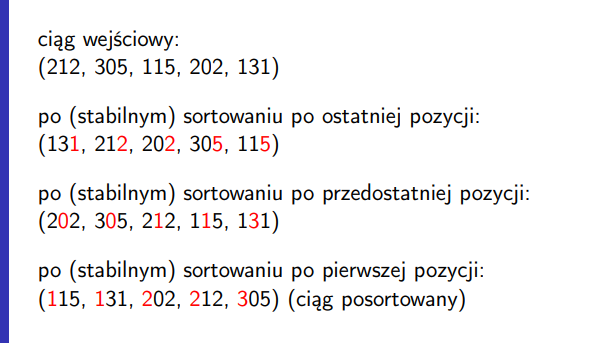
* **Wyszukiwanie sekwencyjne** – najprostszy, przegląda wszystkie indeksy ciągu aż znajdzie szukany element. Ma *liniową pesymistyczną złożoność czasową*– **W(n) = n**  
  Żeby móc wykorzystać szybszy algorytm, ciąg musi być *posortowany*
* **Wyszukiwanie binarne** – wyszukiwanie stosujące zasadę „dziel i rządź” w posortowanym ciągu:
  + Dopóki długość ciągu jest dodatnia:
    - Porównaj klucz ze środkowym elementem ciągu
      * Jak jest równy -> zwróć wynik
      * Jak jest niższy -> ogranicz dalsze poszukiwania tylko do lewego podciągu (odrzuć w ciągu to, co po prawej)
      * Jak jest wyższy -> ogranicz dalsze poszukiwania tylko do prawego ciągu (odrzuć to, co po lewej)
    - wróć do kroku pierwszego
  + Jeśli długość spadła do zera, to nie ma klucza w ciągu  
    **Złożoność czasowa: W(n) = , A(n) =   
    Złożoność pamięciowa: S(n) = 1**
* **Algorytm „turniejowy”** – do znajdowania drugiego najmniejszego elementu w ciągu:
  + W każdej fazie turnieju bieżący zbiór elementów jest dzielony na pary
  + Każda para rozgrywa „grę” – mniejszy element z pary wygrywa i przechodzi dalej
  + Z pozostałych elementów tworzymy nowy zbiór i znowu dzielimy go na pary aż do momentu, gdy zostanie nam jedna para
  + Zwycięzca z ostatniej pary jest najmniejszym elementem – drugi najmniejszy element znajduje się wśród elementów, które przegrały ze zwycięzcą – wśród nich szukamy najmniejszego elementu dowolną metodą  
    Złożoność czasowa: **W(n) =**  (logarytm, bo co „rundę” turnieju odpada nam połowa elementów i przeszukujemy tylko log(n) przegranych; n – 1 bo tyle mamy porównań w pierwszej fazie turnieju (fazie szukania najmniejszego elementu))
* **Algorytm Hoare’a** – do znajdowania k-tego najmniejszego elementu w ciągu:
  + Wykorzystuje pomocniczą procedurę **partition** – bierze dany element m z ciągu i po lewej stronie ustawia nie większe elementy od niego, a po prawej nie mniejsze (nie muszą być one posortowane) np. dla m=5 ciąg ten mógłby wyglądać tak: 1 3 2 5 6 8 7  
    Na wyjściu partition zwraca ostateczny indeks elementu m (dla powyższego przykładu zwróciłby 3)
  + Jeśli partition zwraca indeks równy k, to zawiera on k-ty najmniejszy element ciągu
  + Jeśli i<k -> powtarzamy partition na części podciągu na prawo od i
  + Jeśli i>k -> powtarzamy partition na części podciągu na lewo od i  
    Dla procedury partition: W(n)=n+O(1)
  + Złożoność czasowa: A(n)=

Sortowanie jest wykorzystywane w celu:

* Przyspieszenia wyszukiwania
* Przyspieszenia operacji na danych (np. w bazach danych)
* Wizualizacji danych
* Obliczania pewnych ważnych statystyk (np. kwantyli)

**Stabilny algorytm sortowania** – taki, który ostatecznie nie zamienia elementów o tej samej wartości miejscami, np. . Pozwala to na sortowanie wieloatrybutowych rekordów (np. w pierwszej kolejności posortuj pracowników po pensji, w drugiej po wieku…) bez niszczenia wyników poprzednich sortowań

Algorytmy sortowania:

1. **Selection sort** (sortowanie przez wybór) – znajdujemy minimum, zamieniamy je z pierwszym elementem ciągu, i powtarzamy ten proces z pominięciem pierwszego elementu w podciągu aż zostanie nam tylko jeden element  
   złożoność czasowa:  **= A(n)** (ponieważ niezależnie od ustawienia tablicy, selection sort zawsze wykona tyle samo porównań)
2. **Insertion Sort** (sortowanie przez wstawianie):
   1. Zaczynamy od drugiego elementu w ciągu
   2. W danej iteracji, element *next* (ten, na który aktualnie patrzymy) jest „przepychany” wstecz aż znajdziemy dla niego właściwą pozycję tak, by
   3. Jeżeli chcemy mieć ciąg malejący, to element *next* jest przepychany w prawo aż nie natrafi na element mniejszy od siebie   
        
      Złożoność czasowa: (sytuacja, w której dane są odwrotnie posortowane, ale im bardziej są posortowane, tym mniej porównań trzeba zrobić)
3. **Merge Sort** (sortowanie przez złączanie):
   1. Podziel ciąg na 2 połowy (dziel rekurencyjnie do momentu, gdy masz pary)
   2. Posortuj każdą połówkę oddzielnie (jak są pary, to po prostu porównaj który z dwóch elementów powinien być pierwszy – to wytworzy posortowane czwórki – potraktuj je jak kolejki i bierz najpierw mniejszy element z każdej z nich itd. – to jest funkcja **merge**)
   3. Połącz posortowane połówki w całość (łącz parami)  
        
      Złożoność czasowa funkcji **merge**:  **Ale złożoność pamięciowa S(n) =** (bo tu musimy alokować nowe tablice na połączone ciągi – można to wyeliminować stosując **listy dowiązaniowe** (linked list – każdy element ma wskaźnik na następny/poprzedni)  
      Złożoność czasowa **całego MergeSort** -
4. **QuickSort** (sortowanie szybkie) – algorytm typu „dziel i zwyciężaj” korzystający z funkcji **partition** (tej z algorytmu Hoare’a):
   1. Jeśli aktualny ciąg do posortowania ma długość niewiększą niż 1, zakończ rekursję
   2. Jeśli ciąg jest dłuższy niż 1, to wykonaj funkcję *partition* na ciągu i następnie **rekurencyjnie** wykonaj algorytm QuickSort zarówno na podciągu elementów na lewo od elementu p (tego, który podaliśmy do Partition) i na prawo od tego elementu  
      Złożoność czasowa: (w sytuacji, gdy partition zawsze trafia na któryś z końców ciągu – wtedy co krok sortowany ciąg skraca się tylko o 1 element)
   3. QuickSort NIE JEST STABILNY
   4. n\*log(n) to **najszybsza możliwa prędkość sortowania jeżeli używamy porównań** (jest na to dowód matematyczny)
5. **CountSort** (sortowanie przez zliczanie) – sortowanie używające **adresowania bezpośredniego** zamiast porównań.
   1. Algorytm tworzy 2 tablice pomocnicze:
      1. Counts – do przechowywania liczników wystąpień poszczególnych wartości w ciągu wejściowym. Długość tej tablicy odpowiada maksymalnej liczbie w ciągu wejściowym + 1
      2. Result – do umieszczenia wynikowego posortowanego ciągu
   2. Najpierw przechodzi po całym ciągu wejściowym i w tablicy *counts* zlicza liczbę wystąpień każdego elementu w ciągu (np. jak w ciągu trafi na 1, to zwiększa wartość w tablicy counts o indeksie=1 o jeden)
   3. Potem, elementy tablicy counts są posumowane przyrostowo tak, by było wiadomo ile dokładnie elementów w ciągu jest nie większych od danego elementu (np. counts to najpierw: [0,2,3,1], po sumowaniu to: [0,2,5,6])
   4. Na koniec, przechodzimy po tablicy wejściowej i każdy element wysyłamy do *result* na adres, który dyktuje nam tablica *counts* – **żeby wartości z counts odpowiadały indeksom, należy indeksować od 1** (i zmniejszamy daną wartość w counts o 1, żeby w przypadku ponownego napotkania elementu o tej samej wartości wstawić go o indeks wcześniej) – żeby zachować stabilność, tablicę w tym kroku przechodzimy **od tyłu**
   5. Złożoność czasowa (n – długość tablicy, m – maksymalna wartość w ciągu): Złożoność pamięciowa -
6. **RadixSort** – do sortowania wieloelementowych ciągów o stałej długości (np. wielocyfrowych liczb) – sortuje je najpierw po ostatniej pozycji, potem przedostatniej i tak aż do pierwszej i używa do tego pomocniczego algorytmu sortującego (np. CountSort)  
   ****

## 23.  Charakterystyka i zastosowania podstawowych struktur danych: stos, kolejka, kolejka priorytetowa, struktura Find-Union, słownik.

**Abstrakcyjna struktura danych** – jest zdefiniowana poprzez **zestaw operacji**, które można wykonać na danych, nie wnikając w ich sposób implementacji

**Stos** – jedna z najbardziej podstawowych abstrakcyjnych struktur danych, daje nam 3 operacje:

* push(Element e) – dodaj do stosu element e
* Element pop() – zdejmij i zwróć ze stosu ostatnio dodany element (leżący na górze)
* Element top() – zwróć ostatnio dodany element, ale go nie zdejmuj (operacja niemodyfikująca)

Stos jest też nazywany strukturą LIFO – last in – first out

**Kolejka** – też struktura abstrakcyjna, ma operacje:

* inject(Element e) – dodaj element do kolejki
* Element out() – wyjmij i zwróć najdawniej dodany element (pierwszy w kolejce)
* Element front() – zwróć najdawniej dodany element, ale go nie wyjmuj (operacja niemodyfikująca)

Kolejka to struktura FIFO (first in – first out)

**Kolejka dwustronna** – połączenie kolejki i stosu:

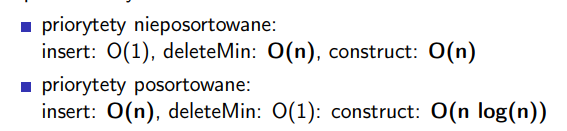
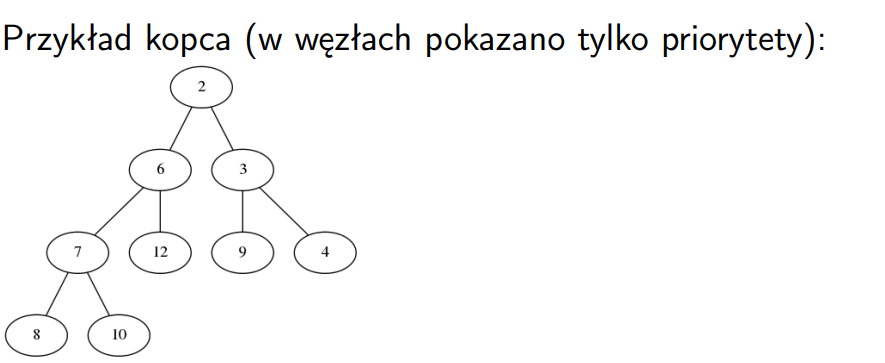
* Element first() – pokaż najstarszy element (pierwszy koniec)
* Element last() – pokaż najnowszy element (drugi koniec)
* pushFront(element) – dodaj element na pierwszy koniec (na start kolejki)
* pushBack(element) – dodaj element na drugi koniec (koniec kolejki)
* popFront() – zwróć i wyjmij najstarszy element
* popBack() – zwróć i wyjmij najnowszy element

**Kolejka priorytetowa** – abstrakcyjna struktura danych do przetwarzania elementów wraz z przypisanymi priorytetami, ma następujące operacje:

* insert(e,p) – dodaj element e o priorytecie p do kolejki
* findMin() – zwróć bez usuwania element najpilniejszy
* delMin() – zwróć z usuwaniem element najpilniejszy

Zakłada się, że im niższa wartość p, tym wyższy jest priorytet.  
Kolejka priorytetowa NIE JEST SPECJALNYM RODZAJEM KOLEJKI (bo ma inny zbiór operacji)

Implementacje kolejki priorytetowej:

* Implementacje „naiwne” – trzymanie elementów w tablicach/listach, priorytety mogą być posortowane lub nie:  
  
* Kopiec binarny – binarne drzewo zupełne (dla każdego węzła, priorytet w tym węźle jest mniejszy/równy priorytetom w węzłach-potomkach) – zupełne, czyli jest wypełniane od góry do dołu, a na każdym poziomie od lewej do prawej  
    
  Dla tej implementacji te operacje robią następujące rzeczy
  + insert(e) – dodaj element w pierwszym od lewej wolnym węźle x ostatniego poziomu, a następnie przywróć porządek kopca począwszy od węzła x w górę (operacja upheap)
  + findMin() – zwróć element e znajdujący się w korzeniu kopca
  + delMin() – usuń element z korzenia, wstaw do korzenia element ostatni kopca (skrajnie prawy na ostatnim poziomie) i przywróć porządek kopca począwszy od korzenia w dół (operacja downheap)

Złożoność czasowa operacji kolejki priorytetowej na kopcu binarnym:

* insert() – O(log(n))
* findMin() – O(1) (bo zawsze zwraca korzeń)
* delMin() – O(log(n))

**Struktura Find-Union** – struktura wykonująca operacje na rodzinie niepustych i rozłącznych zbiorów:

* MakeSet(x) – tworzy nowy jednoelementowy zbiór {x},
* Find – dla danej rodziny i elementu x zwraca zbiór tej rodziny, do którego należy x (dla implementacji drzewa – element zwróci korzeń swojego drzewa) – złożoność **O(log(n))**
* Union – łączy 2 dane zbiory rodziny, tworząc w wyniku nową rodzinę zbiorów (mającą o 1 zbiór mniej) – złożoność **O(nlog(n))**

Do każdego zbioru odwołujemy się poprzez jego **reprezentanta** (wybrany konkretny element danego zbioru) – to właśnie reprezentant jest zwracany jako wynik funkcji Find

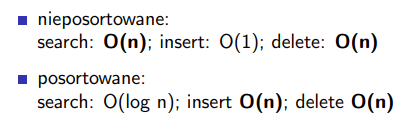
Find-Union jest implementowana przy użyciu drzew:

* Każdy zbiór to osobne drzewo (niekoniecznie binarne)
* Reprezentant zbioru jest korzeniem (który wskazuje sam na siebie)
* Korzeń zna też wysokość drzewa
* Jak wykonujemy operację Union, to korzeń niższego drzewa podpinamy bezpośrednio pod korzeń wyższego

**Słownik** – abstrakcyjna struktura danych służąca do operowania na parach klucz-wartość, ma operacje:

* search(klucz) – zwraca wartość związaną z kluczem (może zwracać specjalną wartość/rzucić wyjątkiem jak nie ma takiego klucza)
* insert(klucz, wartość) – umieszcza nową parę klucz-wartość w słowniku
* delete(klucz) – kasuje parę związaną z kluczem

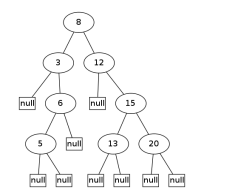
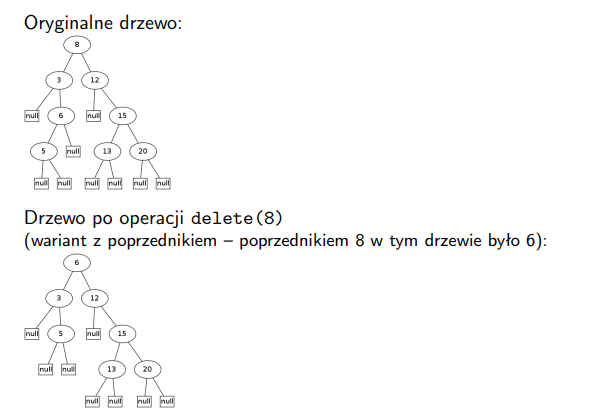
Metody implementacji słowników:

* „naiwna” – przy pomocy tablic lub list – mamy dwie tablice: „klucze” i „wartości”. Złożoności:  
  
* Tablicami mieszającymi (hashtable) – używa funkcji, która przelicza wartość klucza na indeks w tablicy wartości hash: U -> [0,…,m-1] (U – uniwersum wszystkich możliwych kluczy, m – liczba wartości w słowniku)
  + Tablice mieszające wiążą się z ryzykiem **kolizji** (gdy co najmniej 2 różne klucze dają ten sam indeks)   
    Metody radzenia sobie z kolizjami:
    - Metoda mieszania wielokrotnego – jak trafimy na zajęte miejsce, mieszanie jest powtarzane aż do znalezienia pierwszego wolnego miejsca
    - Metoda łańcuchowa – w każdym miejscu tablicy umieszczana jest lista elementów, która może być liniowo przeszukiwana]
  + Wymagania dotyczące funkcji haszującej:
    - Musi być obliczalna bardzo szybko (najlepiej w stałym, niezależnym od liczby kluczy)
    - Musi „równomiernie rozkładać obciążenie” w tablicy – dla klucza wylosowanego z rozkładu jednostajnego z uniwersum U każda wartość z zakresu [0,…,m-1] musi być jednakowo prawdopodobna
  + **Współczynnik obciążenia: α=n/m** (n to liczba par w tablicy, m określa liczbę możliwych wartości)
  + Złożoność czasowa – **O(α)**

**Słownik uporządkowany** – rozszerzenie słownika z operacjami:

* Search(klucz)
* Insert(klucz,wartość)
* Delete(klucz)
* Minimum() – zwraca minimalny klucz w słowniku
* Maximum() – zwraca maksymalny klucz w słowniku
* predecessor(klucz) – zwraca klucz będący bezpośrednim poprzednikiem danego klucza
* successor(klucz) – zwraca klucz będący bezpośrednim następnikiem danego klucza

Metody implementacji słownika uporządkowanego:

* **Drzewo BST** (binary search tree):
  + Każdy węzeł ma klucz z wartością
  + Dla każdego węzła x, klucz w tym węźle jest niemniejszy, niż wszystkie klucze w lewym poddrzewie oraz nie większy niż wszystkie klucze w prawym poddrzewie
  + :
    - Wolne miejsca oznaczone są jako NULL
    - BST nie musi być *zupełne* (wolne miejsca mogą być nie tylko na poziomie liści)
    - Klucz minimalny/maksymalny można znaleźć idąc od korzenia skrajnie w lewo/prawo
  + Jak dany węzeł usuwamy to:
    - Jak nie ma synów – po prostu jest usuwany
    - Ma 1 syna – podpinamy syna do rodzica usuniętego węzła
    - Ma 2 synów – szukamy w podwęzłach bezpośredniego następnika usuwanego węzła i podstawiamy go na miejsce węzła usuniętego (jeśli ten węzeł ma syna, to jest on podpinany pod węzeł wyżej)
    - 
  + Złożoność czasowa wszystkich operacji: **A(n)=O(log(n)), W(n)=O(n)**
* **Drzewo AVL** – drzewo BST z dodatkowym warunkiem – każdy węzeł ma **współczynnik zrównoważenia** (bf – balance factor):
  + Współczynnik zrównoważenia – różnica wysokości lewego i prawego poddrzewa danego węzła
  + Dla każdego węzła, współczynnik zrównoważenia w tym drzewie należy do zbioru {-1,0,1} – **wysokości poddrzew nie mogą się różnić o więcej niż 1**
  + Przy dodawaniu nowych elementów do drzewa, w wypadku przekroczenia wartości współczynnika zrównoważenia, wykonywana jest rotacja
  + Złożoność czasowa **W(n)=O(log(n))**

## 24.  Drzewa binarne i drzewa n-arne w algorytmice. Charakterystyka, sposoby implementacji i zastosowania.

O **drzewach binarnych** ogólnie było w poprzednim zagadnieniu (ich wysokość zawsze zawiera się w **O(log(n))**. Drzewa n-arne były wykorzystane w implementacji struktury Find-Union

**Drzewo** – graf spójny i bez cykli (acykliczny) – każde dwa wierzchołki są połączone dokładnie jedną drogą elementarną, dodanie jakiejkolwiek krawędzi spowoduje powstanie dokładnie jednego nowego cyklu

**Drzewo etykietowane** – takie, którego wierzchołki mają etykiety (na przykład dodatnie liczby naturalne)

**Drzewo ukorzenione** – drzewo z pewnym wyróżnionym wierzchołkiem – korzeniem drzewa. Głębokość (poziom) wierzchołka w takim drzewie to jego odległość od korzenia:

* **Przodek** – dla danego wierzchołka v, jest to wierzchołek który leży na drodze o korzenia do v
* **Potomek** – odwrotnie do przodka (v jest potomkiem danego wierzchołka w, jeśli w leży na drodze od v do korzenia)
* **Rodzic (ojciec)** – przodek wierzchołka v, który z nim sąsiaduje (są połączeni krawędzią)
* **Dziecko (syn)** – potomek wierzchołka v, który z nim sąsiaduje
* **Bliźniaki (bracia)** – wierzchołki, które mają tego samego rodzica
* **Poddrzewo** – dla danego wierzchołka v, jest to podgraf T, którego korzeniem jest v (v ma tyle poddrzew, ile ma dzieci)

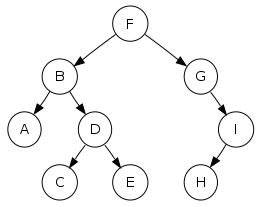
**Drzewa n-arne** - drzewo ukorzenione, gdzie każdy wierzchołek ma co najwyżej n-dzieci

**Las** – graf bez cykli (niekoniecznie spójny – nie z każdego węzła da się dostać do każdego innego)

Algorytmy obchodzenia drzew binarnych:

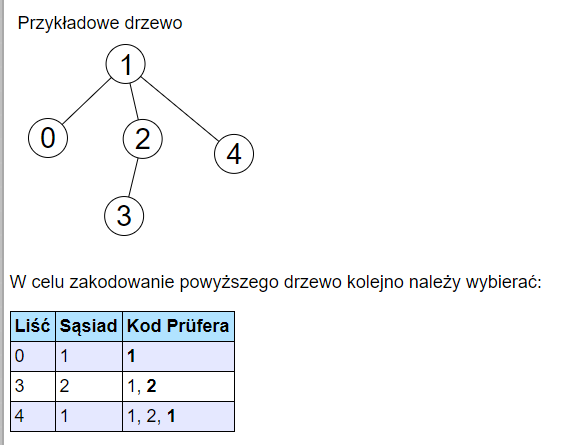
* **Prefiksowa (pre-order)** – najpierw idziemy od korzenia, potem rekurencyjnie najpierw do lewego poddrzewa, a potem do prawego – używane np. do wyznaczenia głębokości drzewa (max odległości od korzenia do liścia)
* **Infiksowa (in-order)** – najpierw zaczynamy od lewego poddrzewa, potem idziemy do korzenia, a na koniec do prawego poddrzewa – **Wartości w drzewie BST są uporządkowane rosnąco w porządku infiksowym** – tę kolejność można wykorzystać do sortowania
* **Postfiksowa (post-order)** – najpierw lewe poddrzewo, potem prawe, a korzeń na końcu – używane gdy musimy najpierw odwołać się do wartości wyliczonych przez synów, aby obliczyć wartość u ojca

Przykład:

  
pre-order: F,B,A,D,C,E,G,I,H  
in-order: A,B,C,D,E,F,G,H,I  
post-order: A,C,E,D,B,H,I,G,F

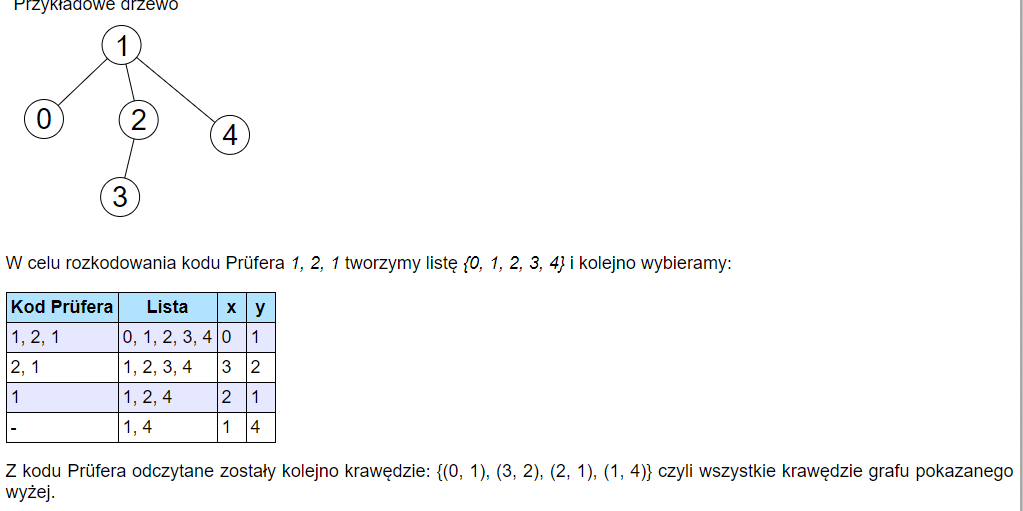
**Kodowanie Prufera** – kodowanie pozwalające zakodować dowolne drzewo etykietowane o n wierzchołkach za pomocą (n-2)-elementowego ciągu liczb z zakresu [1,n]:

* b1 – najmniejsza liczba przypisana liściowi, a1 – jedyny sąsiad tego liścia
* Usuwamy b1 razem z jego krawędzią i zapisujemy a1 jako pierwszy element ciągu – mamy nowe drzewo
* Dla tego nowego drzewa szukamy analogicznie najmniejszego b2 i zapisujemy jego sąsiada a2, b2 zastępujemy a2…
* Postępujemy tak aż do uzyskania dokładnie (n-2)-elementowego ciągu: (a1,…a{n-2})



**Dekodowanie Prufera:**

* Mając ciąg (a1,…,a{n-2}) znajdujemy najmniejszą liczbę b1 w zakresie [1,n], która nie występuje w ciągu
* Łączymy krawędzią a1 i b1
* Usuwamy a1 z ciągu, a b1 pomijamy w kolejnej iteracji
* Powtarzamy to, aż skończą się nam elementy w ciągu
* Na końcu łączymy dwa wierzchołki oznaczone dwoma etykietami nie użytymi dotychczas

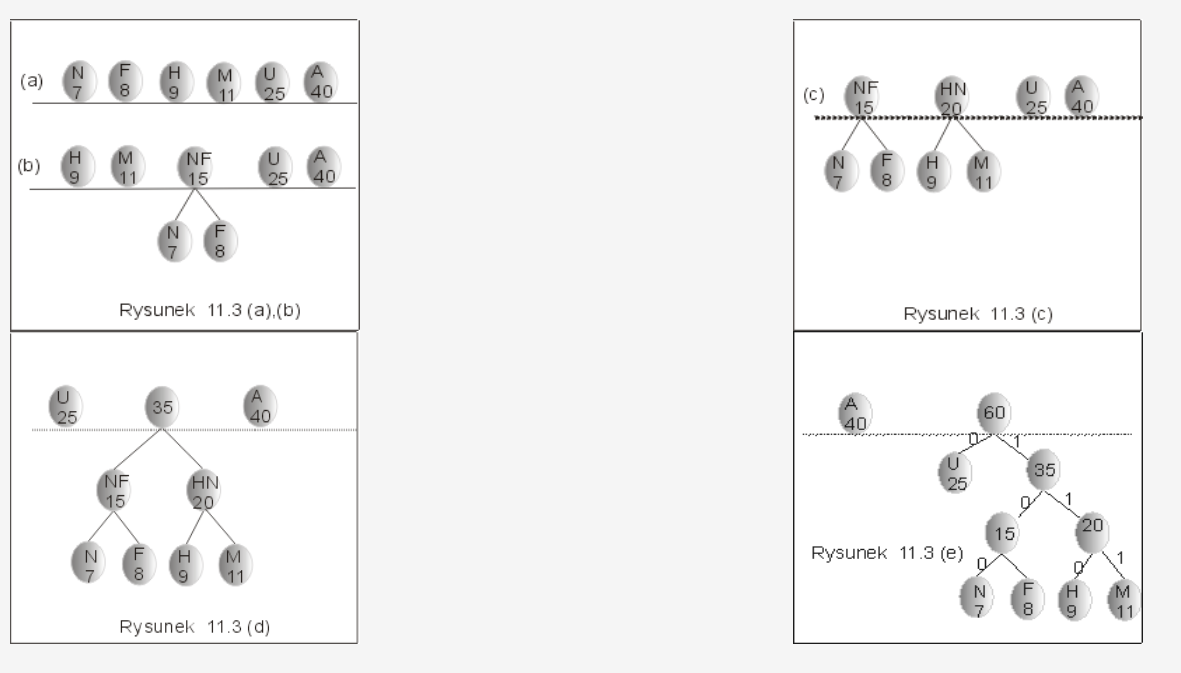
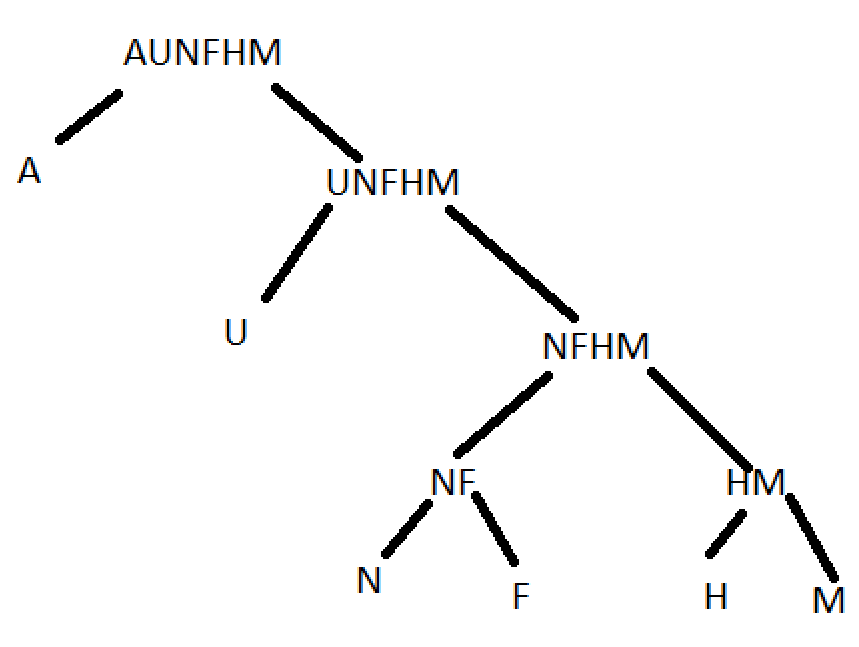
  
(w tym przykładzie b1 jest w zakresie [0,n])

Zastosowania drzew ukorzenionych:

* Informatyka (jako struktury danych – kopce, drzewa BST, AVL, drzewa do kodu Huffmana)
* Uczenie maszynowe (drzewa decyzyjne, drzewa gier)
* Eksploracja danych (kd-drzewa)
* Przetwarzanie języka naturalnego (drzewa parsowania)
* Organizacja informacji (modele hierarchiczne, taksonomie)
* Biologia (drzewa genealogiczne)
* Teoria gier (zasada minimaksowa)

**Algorytm Huffmana** – przykład wykorzystania drzew w celu znalezienia minimalnego kodowania znaków:

* Utwórz kolejkę priorytetową pq zawierającą węzły tworzonego drzewa – priorytet zależy od częstości przypisanej znakom
* Początkowymi elementami są liście drzewa
* Co krok sumujemy dwa najrzadsze węzły i wstawiamy je z powrotem do kolejki (nowy węzeł jest rodzicem dwóch zsumowanych – ten bardziej rzadki z pary jest po lewej)

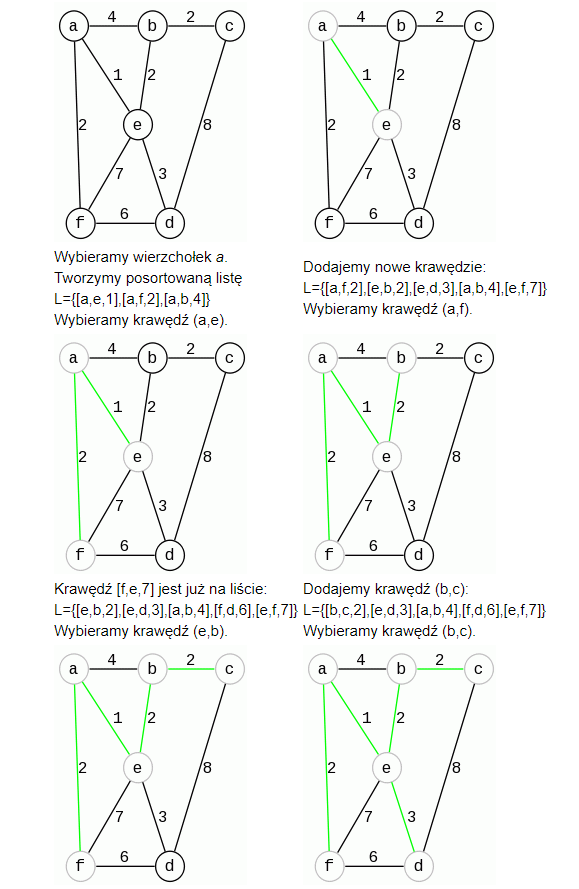
  
Wersja ostateczna:  
   
(Chcąc dostać kod – idziemy od korzenia, jak w lewo to 0, jak w prawo to 1, np. A=0,U=10,N=1100…)

**Drzewo rozpinające** – dla spójnego, nieskierowanego grafu prostego – jest to taki podgraf, który jest drzewem i zawiera wszystkie wierzchołki danego grafu (usuwamy krawędzie tak, by z grafu zrobić drzewo)

**Minimalne drzewo rozpinające** – gdy graf ma wagi na krawędziach – takie, dla którego suma kosztów (wag) krawędzi jest minimalna

Algorytmy do szukania minimalnego drzewa rozpinającego:

* Algorytm Prima:
  + Zaczyna od wierzchołka startowego s i stopniowo powiększa drzewo rozpinające
  + Patrzymy jakie krawędzie wychodzą z wierzchołka s i wybieramy z nich tą o najmniejszej wadze – dodajemy nowy wierzchołek do zbioru drzewa
  + Do zbioru możliwych do dodania krawędzi dodajemy te wychodzące z dodanego wierzchołka i jednocześnie nie łączące się z wierzchołkiem już w drzewie
  + Powtarzamy to aż nie połączymy się ze wszystkimi wierzchołkami

  
złożoność: O((V+E)\*log(V)) (V – liczba wierzchołków, E – liczba krawędzi)

* Algorytm Kruskala:
  + Początkowo mamy pusty zbiór T
  + Rozpatrujemy krawędzie w kolejności niemalejącej (od najmniejszej wagi) i dodajemy do T te, które nie tworzą cyklu z poprzednio dodanymi
  + Ponawiaj aż nie powstanie drzewo rozpinające

Przykład na stronie: <https://eduinf.waw.pl/inf/alg/001_search/0141.php>

Do sprawdzania cykli stosuje się strukturę Find-Union (Na start każdy węzeł jest osobnym korzeniem drzewa, co dodanie krawędzi robimy UNION na węzłach, które łączy dodana krawędź – jeśli węzły już są w tym samym drzewie, to dodanie tej krawędzi spowoduje cykl, więc ją pomijamy)  
Złożoność: O(E\*log(E)) – E to liczba krawędzi

## 25.  Algorytmy rekurencyjne vs algorytmy iteracyjne, porównanie i omówienie podstawowych założeń konstrukcyjnych.

Oba typy algorytmów skupiają się na powtarzaniu operacji

* Algorytm iteracyjny – wykorzystuje instrukcję pętli, np. for/while/do-while
* Algorytm rekurencyjny – moduł (funkcja) wywołuje się ponownie do momentu, gdy warunek zatrzymania nie zostanie spełniony. Każde wywołanie funkcji jest odkładane na stos (tzw. **Struktura rozgałęzień**)

Gdy algorytm się zapętli:

* W algorytmie iteracyjnym – pętla działa w nieskończoność
* W algorytmie rekurencyjnym – nieskończenie wiele wywołań funkcji jest odkładanych na stosie. Zazwyczaj kończy się to **wyjątkiem przepełnienia stosu** (stack overflow exception)

Algorytmy iteracyjne są szybsze – dlatego, że wielokrotne rejestrowanie stosu jest bardziej kosztowną operacją niż iteracja pętli

Algorytmy rekurencyjne są prostsze w implementacji – są najczęściej stosowane do rozwiązywania skomplikowanych problemów, np. **Wieża Hanoi**

Inne różnice między algorytmami rekurencyjnymi a iteracyjnymi:

* Test zakończenia – iteracja kończy się, gdy warunek pętli kończy się niepowodzeniem. Rekurencja kończy się, gdy rozpoznany zostanie podstawowy przypadek (np. zamiast kolejnego wywołania spełniony jest if, który zwraca wartość – tak jest w rekurencyjnej implementacji Ciągu Fibonacciego)
* Nieskończone wywołanie – nieskończona pętla występuje z iteracją, jeśli test kontynuacji pętli nigdy nie staje się fałszywy. Nieskończona rekurencja występuje, jeśli krok rekurencji nie zmniejsza problemu w sposób zbieżny w przypadku podstawowym

Rekurencja ma kilka aspektów:

* Matematyczny – niezbędną częścią definicji rekurencyjnej jest **przypadek bazowy**
* Algorytmiczny – jest definiowana w oparciu o zasadę „dziel i zwyciężaj” (jeden problem dzielimy na mniejsze podproblemy tego samego typu – np. quicksort i mergesort)
* Programistyczny – jest techniką implementacji algorytmów, która polega na tym, że funkcja wywołuje samą siebie (dla mniejszych argumentów)

Największą wadą rekurencji jest duży **koszt pamięciowy** (czasem też czasowy -> rekurencyjny ciąg Fibonacciego ma wykładniczą złożoność).

Dla wieży Hanoi – rozwiązanie rekurencyjne jest jedynym, które można w łatwy sposób sformułować

Założenia wieży Hanoi:

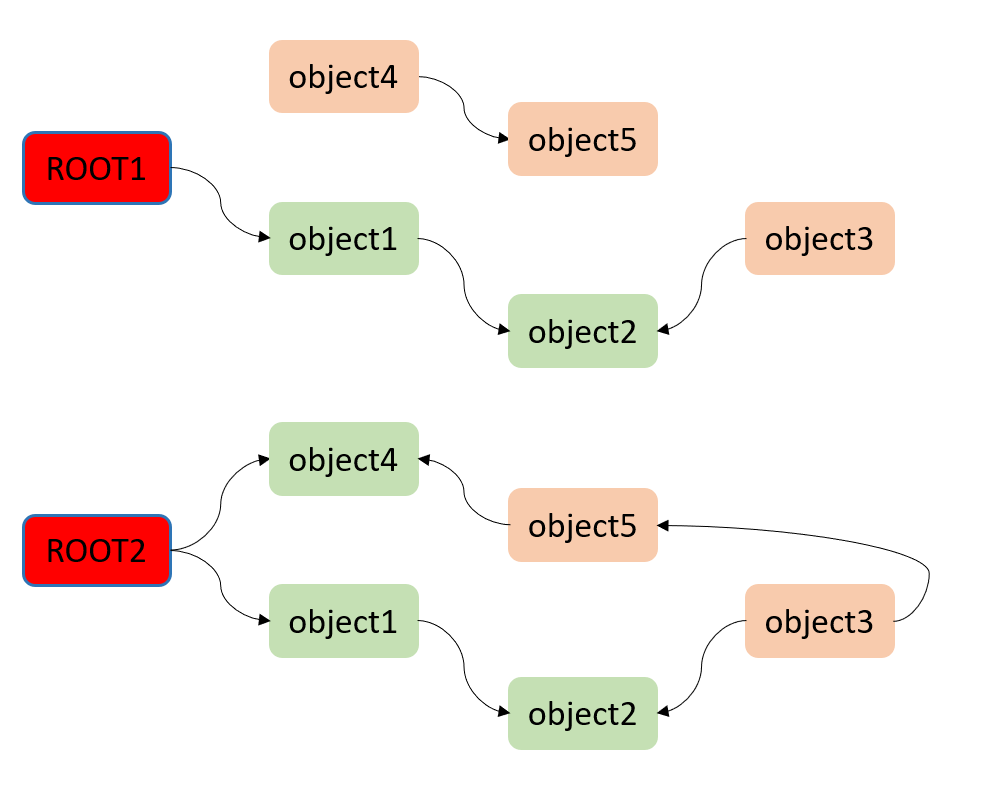
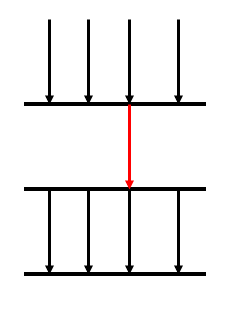
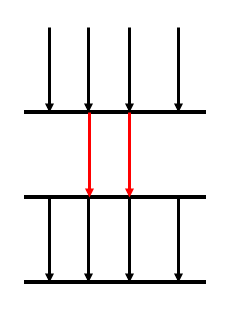
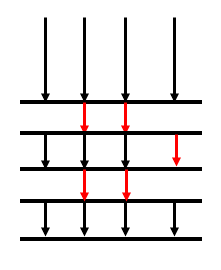
* Mamy n krążków, każdy o innym rozmiarze
* Są one nawleczone na początkowy drążek A od największego do najmniejszego
* Celem jest przeniesienie wszystkich krążków na drążek docelowy C
* W jednym ruchu bierzemy jeden krążek
* Nie wolno kłaść większego krążka na mniejszy
* Mamy do dyspozycji jeszcze jeden drążek pomocniczy B
* Celem jest znalezienie minimalnej liczby ruchów

Implementacja rekurencyjna:

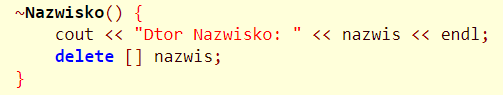
1. Jak mamy 1 krążek, to przenosimy go jednym ruchem z drążka A na drążek C
2. Jeśli na A mamy n>1 krążków, to musimy przenieść n-1 krążków z A na B, dzięki czemu dolny krążek na A możemy przenieść na C
3. Równanie rekurencyjne ma postać: 

## 26. Konstrukcja obiektów i zarządzanie pamięcią operacyjną w Javie i C++.

W javie:

* Tworzenie obiektów przy pomocy konstruktora (Object o = new Object())
* Wykorzystuje **Garbage collector** do zarządzania pamięcią:
  + Jest to osobny program
  + Jego celem jest usuwanie nieużywanych obiektów z pamięci JVM (java virtual machine)
  + Np. jak zechcemy dodać coś do obiektu typu String (albo BigDecimal), to w rzeczywistości tworzymy nowy obiekt – garbage collector dba o to, by stary obiekt usunąć
  + Koncepcja garbage collectora powstała w latach 50. na potrzeby języka LISP
  + Ma 2 typy:
    - Skalarny – dla każdego obiektu licznik odwołań (referencji do niego). W momencie gdy wynosi on 0, obiekt jest usuwany – ma problem bo **nie wykrywa cyklicznych referencji**
    - Wektorowy – rozwiązanie problemu algorytmów skalarnych, obiekty są przedstawiane jako **węzły w grafie** (skierowanym) – jak do danego węzła nie da się dotrzeć z żadnego innego węzła, to jest on usuwany, budowę grafu należy zacząć od obiektów, które na pewno są „żywe” (pełnią one rolę korzenia – by obiekt był żywy, musi być w jakiś sposób połączony z korzeniem)  
      
  + Ma kilka typów:
    - Serial – najprostszy: na czas działania wątku garbage collectora, inne wątki są zawieszane. Dobry do prostych (najlepiej jednowątkowych) programów  
      
    - Parallel – jak serial, ale garbage collector może mieć wiele wątków. Tryb domyślny w Javie:  
      
    - Concurrent Mark Sweep (CMS) – wiele wątków, które są uruchamiane:
      * Podczas oznaczania obiektów, do których istnieją odniesienia w pamięci
      * Jeśli równolegle nastąpi zmiana w pamięci sterty podczas czyszczenia pamięci  
          
        (Wymaga więcej mocy procesora, ale poprawia wydajność aplikacji)
    - G1 – w przypadku dużych obszarów pamięci sterty, algorytm dzieli ją na regiony i zaczyna pracować równolegle

W C++:

* Pamięcią trzeba zarządzać samodzielnie (nie ma garbage collectora)
* Operator **new** – do zapisywania nowego obiektu na stosie lokalnym (jak w javie)
* Oprócz stosu mamy **stertę** (heap) – obszar pamięci wolnej w programie, którą alokujemy zgodnie z potrzebami
* Przy operatorze new musimy podać typ danych i informację o ilości tych danych (na podstawie tego wyliczane jest, ile pamięci sterty należy na daną zmienną zaalokować)
* W momencie, gdy wskaźnik lokalny jest usuwany, **pamięć nie jest zwalniana** – następuje wtedy **wyciek pamięci** (zmienna na stercie istnieje, ale nie mamy do niej żadnego wskaźnika i nie możemy się przez to do niej odwołać)
* Typy proste możemy tworzyć tak samo jak klasy (czyli np. int\* a = new int(18); - trochę tak, jakby int miał jednoargumentowy konstruktor,  
  Możemy też w ten sposób tworzyć dynamiczne stałe: const int\* stala = new const int(1); )
* Żeby tworzyć tablice dynamiczne (takie, których rozmiar nie jest znany w czasie kompilacji), też trzeba użyć wskaźników:  
  int\* pi = new int[wym];
* Operator **delete** – do zwalniania pamięci, można go stosować tylko do adresu, który został zwrócony przez operator new. Można go zastosować tylko raz do danego adresu
* Tablice trzeba usuwać przy pomocy operatora z nawiasami kwadratowymi: delete [] pi;
* Żeby chronić się przed podwójnym zwalnianiem tego samego adresu, po wykonaniu delete warto ustawić wskaźnik na NULL/nullptr/0 (wszystkie te 3 wartości to to samo)
* Do usuwania obiektów klas należy zdefiniować **destruktor** – „odwrotność konstruktora” (ma tyldę przed nazwą): uruchamia delete dla wszystkich pól wskaźnikowych w klasie:  
  

## 27. Rola klas, interfejsów i mixinów w programowaniu na przykładzie języka Java.

**Klasa** – opis takich cech grupy podobnych obiektów, które dla nich niezmienne (np. zestaw atrybutów i metod – usług, które mogą świadczyć)

Definicja klasy określa:

* Zestaw cech (Atrybutów) obiektów klasy – **pola klasy**
* Zestaw operacji, które można wykonywać na obiektach klasy - **metody**
* Specjalne operacje, które pozwalają na inicjowanie obiektów przy ich tworzeniu – **konstruktor**

Cechy konstruktora:

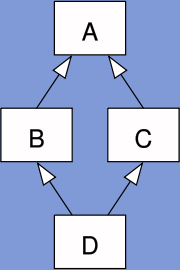
* Zawsze ma taką samą nazwę jak klasa
* Nie ma typu wyniku (NAWET NIE JEST TYPU VOID)
* Ma listę parametrów (która może być pusta – np. hibernate wymaga bezparametrowego konstruktora)

**Referencja –** wartość, która oznacza adres obiektu w pamięci (w Javie, dla typów złożonych, operujemy tylko na referencjach) REFERENCJE NIE SĄ OBIEKTAMI

Klasy w Javie realizują idee:

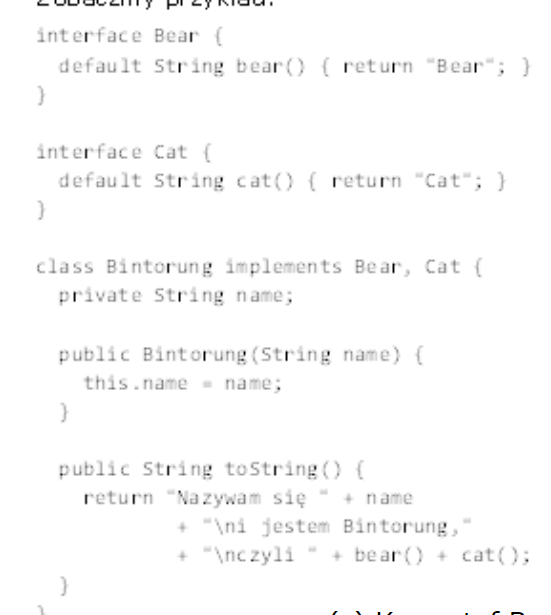
* **Abstrakcji obiektowej –** możemy definiować własne typy danych, ich cechy (atrybuty) i usługi (metody) – do tego, program jest pisany w tym samym języku, co problem, który ma rozwiązać (obiekty są pewną reprezentacją rzeczywistości)
* **Enkapsulacji/Hermetyzacji** (to to samo) – ukrywanie widoczności pól danej klasy dla innych klas w celu ochrony danych (operatory private/public/protected (protected – brak operatora, po prostu nic nie piszemy)
* **Ponownego użycia –** jak mamy w programie wiele danych o tych samych cechach, to zamiast definiować to samo kilkukrotnie, możemy to zawrzeć w klasie i tylko tworzyć nowe obiekty tej klasy

**Interfejs** – zestaw **publicznych (zawsze)** abstrakcyjnych metod (metod bez podanej implementacji) i/lub publicznych statycznych stałych

Interfejsy w Javie mają na celu rozwiązanie problemów związanych z **wielodziedziczeniem** (gdzie klasa dziedziczy po kilku innych klasach) – np. **problem diamentu:**(Zarówno B jak i C dziedziczą konstruktor po A – pola klasy A są inicjalizowane zarówno w konstruktorze B jak i C – problem: którą inicjalizację, od B czy od C, ma wziąć klasa D jeżeli nie przedefiniowuje konstruktora/jakiejkolwiek innej metody?)  
Dzięki temu, że interfejsy skupiają się na definiowaniu „umiejętności” danego obiektu, a nie jej sposobem implementacji, pozwalają one zachować część zalet wielodziedziczenia z pominięciem związanych z nim problemów

**Implementacja interfejsu w klasie** – zdefiniowanie w tej klasie wszystkich metod interfejsu (oznaczane słowem kluczowym *implements*) – chyba, że jest klasą abstrakcyjną (taką, w ramach której nie można tworzyć obiektów), wtedy może implementować interfejs bez definiowania jego metod (klasy dziedziczące po tej klasie będą musiały to zrobić)

Interfejsy poszerzają też **polimorfizm** (mechanizm pozwalający traktować obiekty różnych podtypów w ten sam sposób w związku z tym, że dziedziczą po tej samej klasie – np. wywoływanie toString() dla wszystkich obiektów, ponieważ wszystkie dziedziczą po klasie Object) – polimorfizm pozwala też traktować wszystkie obiekty, które implementują ten sam interfejs w ten sam sposób

**Mixin** – kombinacja metod z różnych klas, która może być dodana do innej klasy, robi się to przy pomocy **metod domyślnych w interfejsach** (słowo kluczowe default)  


## 28. Pojęcie dziedziczenia na przykładzie języków Java i C++.

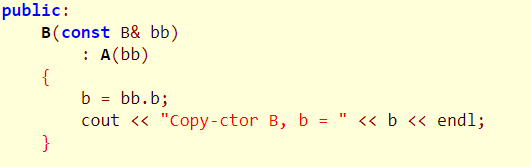
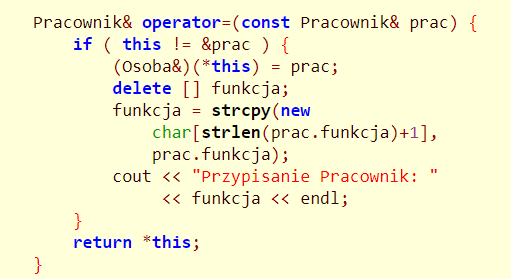
**Dziedziczenie** – mechanizm polegający na przejęciu właściwości i funkcjonalności obiektów innej klasy i ewentualnej modyfikacji tych właściwości i funkcjonalności w taki sposób, by były one bardziej wyspecjalizowane – relacja generalizacja-specjalizacja.  
Dziedziczenie zwiększa **ponowne użycie** kodu

Protected – znaczy, że dla danej klasy i klas pochodnych jest jak public, a dla wszystkiego z zewnątrz jest jak private

W Javie:

* Słowo kluczowe **extends** – do dziedziczenia
* NIE MA WIELODZIEDZICZENIA (można je udawać przy pomocy interfejsów)
* Kod przy tworzeniu obiektu wykonuje się w następującej kolejności:
  + Wywołanie konstruktora klasy pochodnej (tej, która dziedziczy)
  + Jeśli jego pierwsza instrukcja to **super(args)** to wykonywany jest konstruktor nadklasy z argumentami args
  + Jeśli nie ma super, to wywoływany jest bezparametrowy konstruktor nadklasy (trochę tak, jakby zawsze była ukryta instrukcja super())
  + Jak konstruktor nadklasy się wykona, wykonywane są pozostałe instrukcje klasy pochodnej
* Słowo kluczowe **final** przy klasie – po tej klasie nie da się dziedziczyć
* Problem kruchości interfejsu klasy bazowej – jak zmienimy interfejs klasy bazowej, to istnieje ryzyko, że będziemy musieli modyfikować kod wielu klas dziedziczących po tej klasie bazowej
* @Override – pozwala zaznaczyć, że chcemy przysłonić daną metodę

W C++:

* Nie ma wspólnego przodka dla wszystkich klas (takiego jak Object w Javie)
* Zamiast extends używa się dwukropka – class B : public/private/protected A
  + Specyfikatory public/private/protected określają górną granicę dostępności odziedziczonych składowych (to znaczy, że jak podamy private, to wszystkie składowe klasy A, nawet te co były public lub protected, w klasie B będą miały dostępność typu private – można go przysłonić pisząc konkretną dostępność danej składowej wewnątrz klasy)
  + Brak specyfikatora to to samo, jakby napisać private
* Konstruktory i destruktory nie są dziedziczone
* Polimorfizm osiąga się przez **rzutowanie w górę (upcasting)** – wskaźnik nadklasy pokazuje na obiekt podklasy (np. chcemy mieć adres obiektu klasy Shape&, a podajemy adres klasy Triangle):
  + Niejawne rzutowanie w górę zachodzi tylko, jeśli klasa pochodna dziedziczy z klasy bazowej publicznie (słowo public w dziedziczeniu)
  + Jeśli nie, trzeba użyć operatora **static\_cast** lub **dynamic\_cast**   
    
* Tak jak w Javie, najpierw tworzony jest obiekt nadklasy (w Javie było słowo kluczowe super()):
  + Nadklasa musi posiadać konstruktor domyślny (bezargumentowy)
  + Zamiast super jest **lista inicjalizacyjna** – na niej umieszczamy jawnie wywołanie konstruktora dla podobiektu – przy nagłówku konstruktora po dwukropku wywołujemy konstruktor nadklasy:  
    
  + Jak korzystamy z wielodziedziczenia, to konstruktory nadklas są wywoływane zgodnie z kolejnością w liście inicjalizacyjnej
* **Destruktory są wywoływane w kolejności odwrotnej do konstruktorów**
* Dziedziczenie definicji operatora przypisania (w C++ możemy ustalać, co dla danej klasy znaczy +, -, = itd. ) też jest dziedziczone (co może sprawić problemy, jeśli tego nie przedefiniujemy – pola z nowej klasy będą przypisywane w sposób dostarczony przez system – pole po polu, prawdopodobnie nieprawidłowo) – przy przedefiniowaniu operatora w podklasie, można wywołać operator z nadklasy poprzez rzutowanie *this* na nadklasę:   
  
* Jest wielodziedziczenie – class C : public A, B (dziedziczy publicznie z A i prywatnie z B)
* Zamiast pisania, że klasa jest abstrakcyjna, tworzy się **zerowe** **metody wirtualne** (bez definicji:  
  virtual int a())
  + **Zerowe metody wirtualne** – takie, dla których w klasie pochodnej musimy dostarczać implementacji (nie można tworzyć obiektów klasy z tą metodą, jeśli jej nie zaimplementujemy):  
    virtual int f() = 0;
  + **Zwykłe metody wirtualne** (bez = 0 na końcu) – metoda, której wywołanie jest polimorficzne (zakładamy, że będzie przysłoniona w podklasie) – **bez tego polimorfizm nie działa w C++**
* Jak po danej klasie zamierzamy dziedziczyć publicznie, to powinniśmy definiować destruktor jako metodę wirtualną (wtedy polimorficznie możemy usuwać obiekty i każda podklasa będzie musiała w swoim destruktorze usunąć swoją część)

## 29. Istota i zastosowania polimorfizmu na przykładzie języków Java i C++.

Nawet jeśli dany obiekt jest wskazywany przez referencję przeznaczoną dla jego nadklasy, to i tak wywoływana jest metoda tego konkretnego obiektu, np.:

Vehicle v = new Car();  
v.start(); //Nawet jeśli Vehicle ma metodę start, to I tak będzie tu wywołana metoda z klasy Car

**Dynamic/late binding** – dla metod wirtualnych, oznacza to, że **wiązanie odwołań z kodem** następuje **w fazie wykonania**, a nie w fazie kompilacji (w fazie kompilacji nie wiemy, jaki konkretnie obiekt podstawimy pod referencję typu Vehicle – może to być Car, Bike itp.)

**W Javie wszystkie metody są wirtualne (o konkretnej wersji metody decyduje typ dynamiczny – omówiony w części C++ tego zagadnienia) oprócz:**

* Metod statycznych (bo nie dotyczą obiektów)
* Metod ze specyfikatorem **final** (bo nie można jej przedefiniować w podklasie)
* Metod prywatnych

Polimorfizm sprawia, że wszystkie przyszłe podklasy mają już zapewnioną daną funkcjonalność (nie musimy się przejmować tym, że dana klasa nie potrafi czegoś zrobić), i pozwala zdecydować o sposobie realizacji tej funkcjonalności

Przykład polimorfizmu w Javie – metoda toString() – każdy obiekt możemy chcieć wypisać (nigdy nie ma tego problemu, że podanie obiektu do System.out.println() spowoduje błąd kompilacji), i w każdej klasie możemy zdecydować, jak dany obiekt wypisać

W C++:

* Wyróżniamy dwa typy obiektów:
  + **Typ statyczny** – to, co mówi nam wskaźnik (jak w przykładzie z Javy: typem statycznym byłby Vehicle) – znany w trakcie kompilacji
  + **Typ dynamiczny –** rzeczywisty obiekt, na który pokazuje wskaźnik (dla przykładu z Javy byłby to Car) – znany w trakcie wykonania programu
* Metodami wirtualnymi są tylko metody ze słowem kluczowym **virtual –** tylko dla metod wirtualnych znaczenie ma typ dynamiczny obiektu (dla zwykłych metod decydujący jest typ statyczny – **early binding**)
* Różnica wynika z tego, że late binding jest mniej wydajny, a C++ stawia na wydajność – znalezienie odpowiedniej wersji metody w trakcie wykonania programu jest czasochłonne + każdy obiekt klasy polimorficznej posiada adres do tablicy wszystkich wersji metody wirtualnej
* **Konstruktor nigdy nie może być wirtualny** (destruktor zazwyczaj jest)
* Słowo virtual do metody piszemy tylko w klasie bazowej (na szczycie hierarchii dziedziczenia) – w podklasach ta metoda też będzie wirtualna (możemy w nich pisać virtual, ale wpływa to tylko na czytelność kodu)
* **Polimorfizm można wyłączyć** poprzez odwołanie się do wersji metody konkretnie z nadklasy:  
  obiektB->A::fun(); (obiektB będący obiektem klasy B wywołuje wersję metody fun() pochodzącą z klasy A, po której klasa B dziedziczy)
* Na odwrót to nie działa: nie możemy zrobić a.B::fun(), bo klasa A nie wie nic o dziedziczącej po niej klasie B

## 30. Użycie tablic oraz innych struktur danych w Javie i C++. Java Collections Framework.

**Tablice** – zestawy elementów (wartości) tego samego typu, ułożone na określonych pozycjach. Do każdego elementu tablicy mamy bezpośredni dostęp (niewymagający przeglądania innych elementów) poprzez nazwę tablicy i pozycję szukanego elementu. Rozmiar tablicy jest z góry określony i nie może ulec zmianie

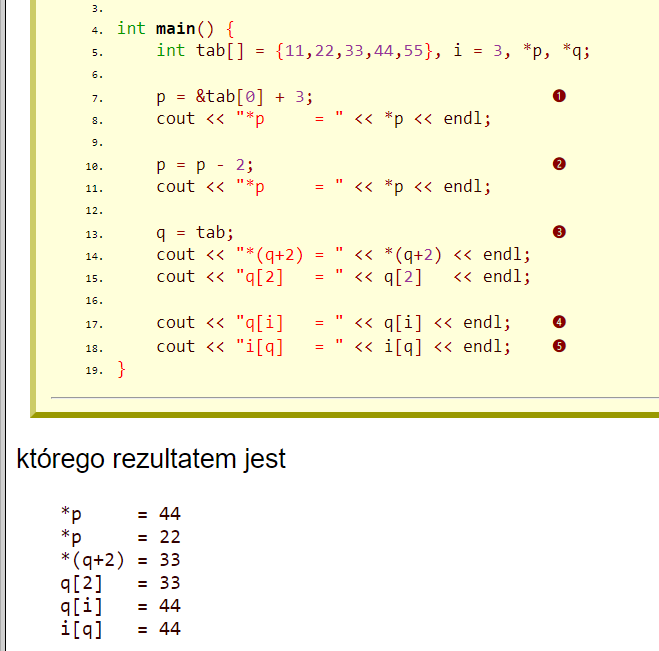
**W Javie tablice są obiektami** – mogą zawierać te same typy proste (np. int, float itd.) lub typy referencyjne (wtedy wszystkie obiekty mają ten sam typ statyczny, ale dynamiczne mogą być różne – możemy mieć tablicę obiektów typu Object, a w środku mieć w jednej tablicy Integer, Thread, int[] itd.)

**Tablica wielowymiarowa** – tablica, która ma w środku inne tablice

Bardziej szczegółowo w Javie:

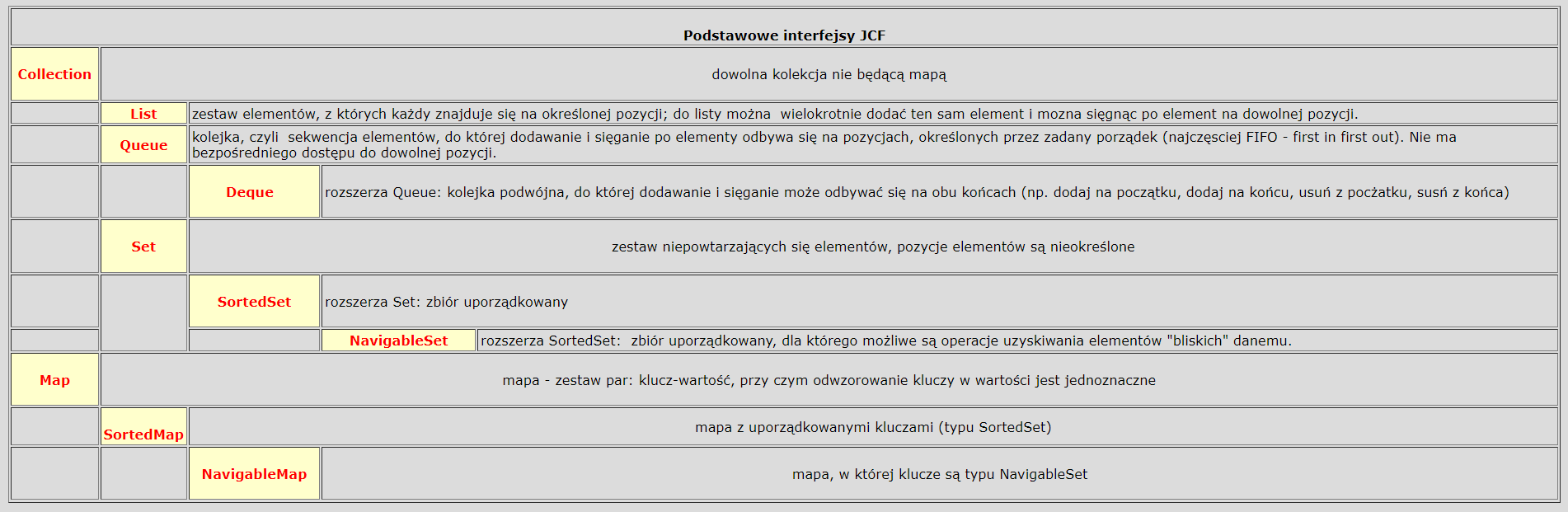
* Rozmiar tablicy nie stanowi składnika deklaracji – nie możemy napisać: int[5] arr; (to 5 w nawiasach nie powinno tam być)
* Tablice są obiektami – ich klasy są tworzone w trakcie kompilacji, mają specjalne nazwy tylko dla potrzeb JVM
* Tablice możemy inicjować poprzez:
  + Listę elementów w nawiasach klamrowych: typ[] tablica = {war1, war2, …, warN};
  + Poprzez słowo new: typ[] tablica = new typ[ile\_elementów]; {wtedy ta tablica jest „pusta” – ma w środku NULL jak używa typu referencyjnego, albo 0/false jak typu prostego)
  + Tablica.length – pole w klasie określającej tablicę, które mówi o jej długości (length w Javie dla tablic to NIE JEST METODA)
  + Tablice heterogeniczne – takie, które mają w sobie obiekty o różnych typach dynamicznych (np. Object[] tab = { "Tekst" , new Para(10,11) }; )

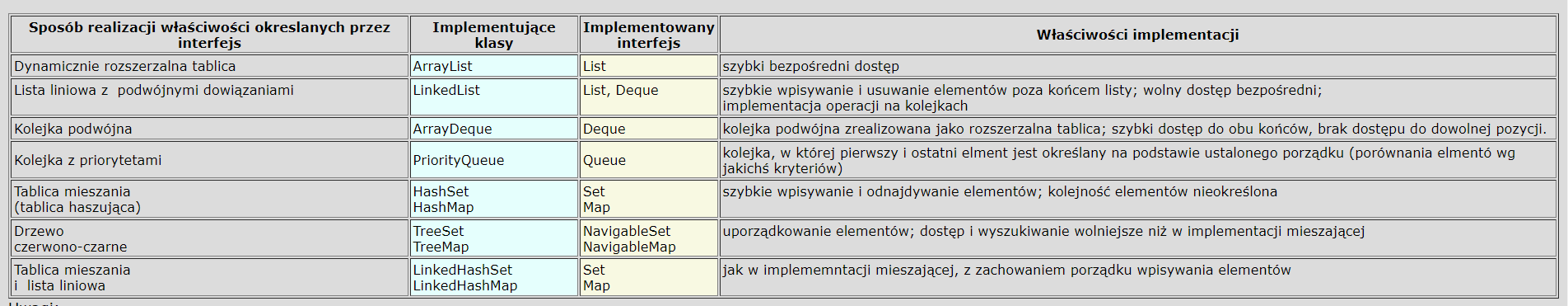
W C++:

* Tablice definiowane **bez wskaźników** są **statyczne** (int tab{1,2,3,4,5}; lub int tab[100]; )– ich rozmiar musi być znany na etapie kompilacji (nie możemy podać zmiennej w rozmiarze tablicy)
* Java ma nawiasy kwadratowe przy typie (int[] tab; ), C++ ma przy nazwie zmiennej (int tab[]);
* Jak nie podamy co ma być w tablicy, to w środku będą „śmieciowe” wartości (nawet częściowo wystarczy: int tab[5]{1,2} – na pozostałych miejscach będą zera; Problem istnieje tylko dla tablic statycznych – jak tworzymy ją słowem kluczowym new, to też ma w środku zera)
* Jak chcemy mieć tablicę dynamiczną, to należy użyć wskaźnika:  
  int\* tab = new int[zmienna];
* Zamiast length trzeba liczyć rozmiar w bajtach przy pomocy słowa kluczowego **sizeof**:  
  sizeof(tab) / sizeof(int) = length z Javy (można też to zapisać w formie niezależnej od typu tablicy: sizeof(tab) / sizeof(tab[0]))
* **Zamiast dostępu do obiektu tablicy, zawsze mamy jedynie wskaźnik do jej pierwszego elementu** (tablica nie zna swojej wielkości – zna pierwszy element i pamięta swój sizeof tylko w zakresie jej stworzenia)
* Przekazując tablicę do funkcji przekazujemy wskaźnik na jej pierwszy element – wewnątrz funkcji wymiar tablicy nie jest znany (sizeof nie zadziała – powinniśmy zawsze posyłać dodatkowo argument o rozmiarze tablicy)  
   - to daje błąd kompilacji – error: ‘tab’ was not declared in this scope (nie możemy mieć tablicy w argumentach metody, powinno być void test(int\* tab, int size))
* &tab[0] to to samo co tab (nazwa tablicy daje jest równoznaczna z adresem jej pierwszego elementu)
* W związku z tym, możemy tworzyć wskaźniki na kolejne elementy tablicy i je przesuwać poprzez dodawanie/odejmowanie od nich liczb:  
  
* W języku C (w C++ też można to stosować), zamiast typu String stosuje się tablice typu char (**C-napisy** – ostatni znak to ‘\0’ nazywany też NUL (przez jedno l) – oznacza on koniec ciągu znaków:  
  char tab1[] = "Kasia";  
  char tab2[] = {'B', 'a', 's', 'i', 'a', '\0'};

**Kolekcje w Javie:**

* Dzielą się głównie na:
  + **Listy** (LinkedList – referencje następny/poprzedni, ArrayList – w tablicy) – uporządkowany ciąg elementów
  + **Kolejki** (ArrayDeque – kolejka dwukierunkowa) – sekwencja elementów, na której operacje wstawiania, pobierania i usuwania są możliwe tylko w określonej kolejności
  + **Zbiory** (HashSet – tablica mieszająca, TreeSet – drzewa czerwono-czarne) – kolekcja bez ustalonego porządku (chyba że to SortedSet), która nie zawiera duplikatów **Wymagają implementacji metod hashCode() oraz equals()** żeby wykryć, czy dodawany obiekt jest duplikatem
  + **Mapy** (HashMap – klucze w HashSet, TreeMap – klucze w TreeSet, LinkedHashMap – pozwala odtworzyć kolejność dodawania elementów, WeekHashMap – współpracuje z Garbage collectorem i usuwa klucze, do których zgubiliśmy referencje) – pary klucz-wartość – wcześniej omawiany słownik (klucze nie mogą się powtarzać, więc są one zawarte w zbiorze)
* Każdy typ kolekcji ma własny interfejs (List, Queue itd.) – tzw. **interfejsy kolekcyjne**
* Java Collection Framework (JCF) składa się z:
  + Interfejsów – do określania typu kolekcji, np. List
  + Implementacji – np. ArrayList, LinkedList
  + Algorytmów – metod do wykonywania operacji obliczeniowych na kolekcjach (sortowanie, wyszukiwanie (metoda contains()), usuwanie elementów, sprawdzanie czy jest pusta isEmpty() itd.) – są polimorficzne
* Odpowiednikiem w C++ jest **STL (Standard Templates Library)**



****

* **Wszystkie klasy kolekcji są sparametryzowane** (możemy podać, jaki typ dana kolekcja ma zawierać)
* Metody korzystające z porównywania elementów (np. contains(..), removeAll(..)) korzystają z metody **equals()** z klasy Object (trzeba ją przysłonić gdy tworzymy własną klasę)
* Możemy je przeglądać dzięki **Iteratorom**  - obiektom implementującym interfejs Iterator, ma on metody:
  + T next() – zwraca kolejny element kolekcji lub rzuca wyjątkiem NoSuchElementException jeśli osiągnie koniec kolekcji
  + void remove() – usuwa element kolekcji, zwrócony przez ostatnie odwołanie do next(), operacja opcjonalna
  + boolean hasNext() – zwraca true, jeśli możliwe jest odwołanie do next() nie rzucające wyjątkiem
* **Jak iterujemy po kolekcji, to nie można ingerować w jej zawartość innymi sposobami niż użycie metody remove() na rzecz iteratora** – inaczej efekt modyfikacji jest nieprzewidywalny (metody add() i remove() danej kolekcji, pochodzące z klasy Collection, spowodują wyjątek ConcurrentModificationException)
* **Interfejs Comparable<T>** - pozwala definiować sposób porządkowania obiektów danej klasy, ma jedną metodę int compare to (T otherObject):
  + Wynik < 0 – this jest przed otherObject
  + Wynik > 0 – this jest po otherObject
  + Wynik = 0 – obiekty są takie same
* Comparator – interfejs do definiowania komparatorów – obiektów do porównywania innych obiektów, z metodą int compare(T o1, T o2) – działa tak jak compareTo z Comparable, ale jest niezależne od this – możemy dla tych samych klas mieć różne komparatory, łączyć je (żeby mieć priorytet sortowania) itd.
* Ma specjalną klasę do działania na kolekcjach – **Collections**, z przykładowymi metodami:
  + static int binarySearch(List<> list, K key) – szuka binarnie elementu key na liście i zwraca jego indeks (wymaga metody compareTo), lista musi być posortowana według naturalnego porządku (metodą sort(List)), jeśli nie znajdzie elementu, to zwraca jakąś wartość < 0 (niekoniecznie -1)
  + static int binarySearch(List<> list, K key, Comparator<> c) – to samo, ale korzysta z komparatora zamiast compareTo (K nie musi implementować wtedy comparable)
  + static void sort(list) – sortuje listę rosnąco
  + static void sort(list, comparator) – to samo, tylko używa komparatora
* **Collections pozwala tworzyć niemodyfikowalne widoki kolekcji** (np. unmodifiableSet), pozwala też tworzyć **synchronizowane widoki** do pracy wielowątkowej
* Ma też klasę **Arrays** do pracy na tablicach, np. z metodą asList(array) tworzącą listę z tablicy (tworzy niemodyfikowalną listę, której rozmiar jest stały: możemy zmieniać jej elementy, ale nie możemy dodawać/usuwać)

## 31. Programowanie współbieżne ? mechanizmy i narzędzia na przykładzie języka Java.

**Proces** – wykonujący się program wraz z dynamicznie przydzielanymi mu przez system zasobami (pamięcią operacyjną, plikami itp.) + ewentualnie innymi kontekstami wykonania programu (np. obiektami tworzonymi przez program)

**Wątek** – sekwencja działań, która może wykonywać się równolegle z innymi sekwencjami działań w kontekście danego procesu

Zmiany wątków w procesorze są wykonywane na 2 sposoby (zależnie od systemu operacyjnego):

1. **Współpracy** (cooperative multitasking) – wątek sam decyduje kiedy oddać czas procesora innym wątkom
2. **Wywłaszczania** (pre-emptive multitasking) – systemowy zarządca wątków przydziela wątkom kwant czasu procesora, po upłynięciu którego odsuwa dany wątek i daje dostęp następnemu

**Tworzenie wątków w Javie** – tworzymy obiekt klasy **Thread** i używamy metody **start()**  
Thread ma też metodę **run()** – do określenia, co ma robić wątek  
Run to metoda interfejsu **Runnable**

Podstawowy sposób tworzenia wątków w Javie:

1. Zdefiniować własną klasę dziedziczącą po Thread
2. Przedefiniować odziedziczoną metodę run()
3. Stworzyć obiekt naszej klasy
4. Wywołać metodę start() na rzecz tego obiektu

Inny sposób tworzenia wątków w Javie – lepsza separowalność kodu + jedyny sposób, gdy nasza klasa nie może dziedziczyć po thread:

1. Zdefiniować klasę implementującą Runnable
2. Dostarczyć definicję metody run()
3. Utworzyć obiekt tej klasy
4. Stworzyć obiekt klasy Thread i podać mu w konstruktorze obiekt naszej klasy – Thread thread = new Thread(x);
5. Wywołać start() na rzecz obiektu thread

Thread ma też metodę **sleep(czas w milisekundach)** – do usypiania wątku (jest on wtedy odsunięty od procesora i czeka określony czas)  
Metoda ta rzuca wyjątkiem **InterruptedException** gdy wątek jest przerywany (jeśli coś zakończy wątek jak jest uśpiony, to możemy coś np. zapisać)

Dlatego, że Runnable ma jedną metodę (**interfejs funkcyjny**), możemy tworzyć wątki **ad hoc** przy pomocy **wewnętrznej klasy anonimowej** lub **wyrażenia lambda**

**Wątek się kończy**, gdy w run() „dobiegniemy do końca” albo przy użyciu słowa **return  
Thread** ma też metodę **stop** – legacy code, którego nie powinno się używać

Synchronizacji wątków możemy uniknąć poprzez:

* Użycie klasy niezmiennych (**immutables**) – jak nic się nie zmienia w obiekcie, to nie ma problemu
* Użycie kodów wielobieżnych (**reentrant**) – kody bezpieczne w użyciu wielowątkowym (np. odwołujący się tylko do zmiennych lokalnych – brak sekcji krytycznych)
* Użycie słowa kluczowego **volatile** – program pamięta w cachu kopię zmiennej. W odwołaniu do niej, zarówno oryginał jak i kopia są uwzględniane

Do synchronizacji wątków:

* Słowo kluczowe **synchronized** przy metodzie– np. synchronized void metoda() – każdy obiekt w Javie posiada rygiel, który chroni przed równoczesnym działaniem wątków na tym samym obiekcie, jak wątek wywołuje metodę ze słowem sychronized na rzecz jakiegoś obiektu, to rygiel tego obiektu jest zamykany (inne wątki muszą czekać aż ten pierwszy wykona tę metodę)
* **Bloki synchronizowane** – np. synchronized (lock) { //kod } – w momencie, gdy wątek wchodzi w ten blok, rygiel obiektu lock jest blokowany. Wszystkie wątki, które chcemy synchronizować, muszą mieć ten sam obiekt lock w tym miejscu

**Koordynacja wątków** – działania mające na celu zapewnienie współdziałania wątków (m. in. w konkretnej kolejności)

Metody do koordynacji wątków (z klasy Object – wywoływane na rzecz this):

* wait() – wątek czeka na wydarzenie, które ma się wydarzyć w innym wątku (jak użyjemy go w sekcji krytycznej, to rygiel jest zwalniany), możemy też podać maksymalny czas czekania. Tak jak sleep, rzuca ona InterruptedException
* notify() – powiadamia konkretny jeden inny wątek o wydarzeniu, co kończy jego czekanie (wątek czeka na danym obiekcie)
* notifyAll() – to samo co notify, ale powiadamia wszystkie czekające wątki (zazwyczaj tego się używa, bo nie wiadomo jaki wątek obudzi samo notify)

**monitor** – fragment kodu programu, którego dostęp zabezpieczany jest przez rygiel – w odróżnieniu od sekcji krytycznej, monitory są powiązane z obiektami i ich stanami

metoda **yield()** – wątek oddaje czas procesora, ale nie zwalnia rygla

Stany wątków:

* New Thread – w momencie stworzenia obiektu wątku
* Runnable – po wywołaniu metody start()
* NotRunnable – w kilku wypadkach:
  + Wątek jest uśpiony metodą slee()
  + Wątek czeka metodą wait()
  + Wątek jest zablokowany na operacji wejścia/wyjścia
  + Obiekt doszedł do bloku/metody synchronized i czeka na otwarcie rygla
* Dead – po skończeniu się metody run()

**Interrupt()** – metoda z klasy Thread, która zmienia status wątku na przerwany (rzuca InterruptedException) – NIE PRZERYWA JEJ, TYLKO USTAWIA FLAGĘ INTERRUPTED NA TRUE  
**isInterrupted()** – metoda zwracająca true, jeśli wywołana została metoda interrupt

**setPriority(**int prior) – metoda z klasy Thread, która ustala priorytet wątku (jest też getPriority()) – sortowanie wątków zależy od platformy, na której uruchamiany jest program

**demon** ­– wątek, który wykonuje prace systemowe lub pomocnicze w tle. Żeby zakończyć program, musimy zakończyć wszystkie zwykłe wątki, ale demony kończą się same gdy wyłączymy program

**Timer** – klasa do zarządzania czasem (z klasy java.util – jest też wersja ze swing). Uruchamia obiekty klasy **TimerTask** (implementującą runnable)i określonym czasie i z określoną częstotliwością (timer.schedule(task,0 – moment startu,2000 – co ile) )

**Executor** – interfejs do zarządzania wątkami (odseparowanie zadań do wykonania od mechanizmów tworzenia i uruchamiania wątków) – ma metodę void execute(Runnable) – do tworzenia i uruchamiania wątków

Domyślne klasy implementujące Executor:

* Executors.newSingleThreadExecutor() – po kolei uruchamia podane mu zadania w jednym wątku
* Executors.newFixedThreadPool() – dysponuje pulą wątków o określonym maksymalnym rozmiarze
* Executors.newCachedThreadPool() – pula o dynamicznym rozmiarze
* Executors.newScheduled() – tworzy i wykonuje wątki w określonym czasie i z określoną częstotliwością

ExecutorService - interfejs rozszerzający Executor, ma metody do zamykania Wykonawcy wątków (tak, że nie będzie się dało mu podać nowych zadań do wykonania):

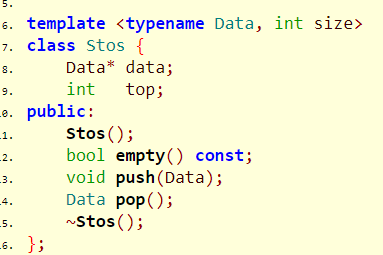
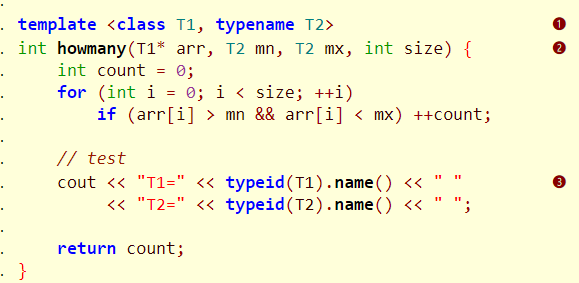
* shutdown() – zamyka wykonawcę, ale pozwala dokończyć się już uruchomionym wątkom
* shutdownNow() – zamyka wykonawcę i przerywa wszystkie uruchomione wątki

## 32. Typy i metody sparametryzowane (generics) w Javie. Szablony  (templates) w C++.

W Javie:

* Gdy chcemy konkretnie określić typ, który może się zmieniać (np. zawartość kolekcji) – mimo zmian, typ jest znany w momencie kompilacji, zachowana jest **mocna kontrola typologiczna**
* **Nie możemy podstawić pod parametr typów prostych** (np. int)
* Pozwala uniknąć konwersji w nawiasach: (Klasa) obiektInnejKlasy – taka konwersja jest mniej czytelna i potrafi powodować błędy
* **Typ sparametryzowany** – taki, który określany dodatkowymi parametrami, dotyczy klas i interfejsów: class Nazwa <parametr1, parametr2…> {//… }
* Instancja sparametryzowanej klasy – tworzona przy deklaracji, gdzie podajemy parametry, np. Para<String,String> p1 = new Para<>(„Jan”,”Kowalski”);
* Po kompilacji szablonu klasy, powstaje tylko jedna współdzielona klasa dla wszystkich instancji (nieważne, czy <String,String>,<String,Integer> itd.) typu sparametryzowanego (w C++ powstaje po klasie na każdą instancję)
* Parametr wewnątrz definicji klasy (np. ArrayList<T> - tu mamy parametr T) może być użyty:
  + Jako pole/zmienna lokalna (T zmienna; )
  + Jako typ parametru/zwracany przez metodę (public T getSth(); )
  + Do jawniej konwersji do typów oznaczonych przez ten typ ( (T) jakiśObiekt; )
  + Wywoływać na rzecz zmiennych tego typu metody z klasy Object (+ z klas będących **górnym ograniczeniem**)
* Ale NIE może być użyty:
  + do tworzenia obiektów (new T() – nie wiadomo jaki konstruktor ma T)
  + do operatora instanceOf()
  + do użycia w kontekście statycznym (nie wiemy, jakie metody może mieć T)
  + do użycia w literałach klasowych
  + do wywołania metod z klas, które nie są górnym ograniczeniem parametru
* **Ograniczenia parametrów** – zawężają nam typy, jakie mogą być podane w miejsce parametru w szablonie:
  + np. class A < T extends Appendable> - T musi być Appendable lub klasą dziedziczącą po Appendable
  + Do tego możemy podać listę interfejsów, jakie T musi implementować:   
    class A < T extends Klasa/Interfejs0 & Interfejs1 &…>
* **Parametr uniwersalny (wildcard)** – chroni przed tym, żeby np. nie podstawić źle pod referencję, np.:  
  ArrayList<Object> list = new ArrayList<Integer>; - niedozwolone, bo moglibyśmy potem dodać do tej listy Integerów jakiś inny obiekt  
  Wildcards mają 3 typy:
  + **Ograniczone z góry** - <? extends X> - wszystkie podtypy X – daje nam **kowariancję**: List<? Extends Number> będzie wtedy nadtypem List<Integer>, List<Float> itd. (przed tym chcieliśmy się chronić w przypadku Object)
  + **Ograniczone z dołu** - < ? super X> - wszystkie nadtypy X – daje nam **kontrawariancję**: możemy wtedy podstawić List<? super Integer> list = new ArrayList<Number>;
  + **Nieograniczone** - <?> - wszystkie typy – daje nam **biwariancję**: możemy podstawić cokolwiek, jest nadtypem każdej wersji super i extends
* Metody też mogą mieć parametry, np. : public static <T extends Comparable<T>> T max(T[] arr) {//…}

**W C++**:

* Podajemy parametry do klas i funkcji (nie ma czegoś takiego jak interfejs)
* Każda instancja ma własną klasę, wygenerowaną w momencie kompilacji
* Definiuje się podobnie do Javy, ale nawiasy <> są przed definicją klasy:  
    
  (Nazwa parametru to Data, zazwyczaj używa się krótszych nazw)
* Możemy podstawiać pod parametr typy proste, np. Klasa<double,int> x;
* W tym przykładzie oprócz parametru typu mamy też do podania wartość określonego typu (int size) – trochę taka stała, ale każda jej wartość generuje nową klasę
* Funkcje też możemy parametryzować:  
  
  + class i typename są synonimami (robią dokładnie to samo) w tym konteście – lepiej używać typename, bo int też możemy podać w parametrze, a nie jest on klasą (może to być mylące)

## 33. Lambda-wyrażenia i interfejsy funkcyjne w języku Java.

**Interfejs funkcyjny** – taki, który zawiera tylko jedną abstrakcyjną metodę (SAM – Single Abstract Method)

Każde Lambda-wyrażenie odpowiada jakiemuś interfejsowi funkcyjnemu i jego typ jest wyznaczany przez rodzaj tego interfejsu (lambda jest dopasowywana do interfejsu przy pomocy **deskryptora funkcji**)

@FunctionalInterface – adnotacja do pilnowania, czy dany interfejs jest funkcyjny

Składnia lambda-wyrażenia:  
(parametry) -> ciało\_lambda  
ciałem może być jedno wyrażenie lub blok instrukcji w nawiasach klamrowych  
jeżeli mamy tylko jeden parametr, to można pominąć nawias, możemy też pomijać typy parametrów jeśli mogą być one określone przez kompilator (jak podajemy lambdę w miejsce jakiegoś argumentu funkcji, który ma implementować określony interfejs, to kompilator się domyśli typu lambdy)

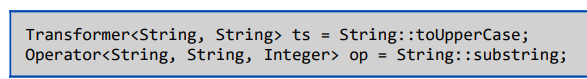
Przykład:  
(int n) -> { return n + 1; }  
Po uproszczeniu:  
n -> n+1

**Lambda nie wprowadza nowego zasięgu widoczności i dostępności (w przeciwieństwie do klasy anonimowej)**:

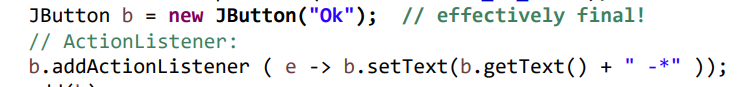
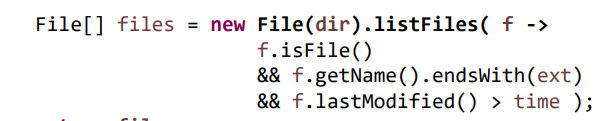
* this wewnątrz lambdy odnosi się do obiektu, w którym lambda się znajduje
* super tak samo jak this
* nie może przesłaniać zmiennych lokalnych z otaczającego jej bloku

Referencje odpowiadające różnym lambda-wyrażeniom:

* Klasa::metodaStatyczna ⬄ x -> Klasa.metodaStatyczna(x)
* Klasa::metoda ⬄ x -> x.metoda(x)
* v::metoda ⬄ x -> v.metoda(x)
* Klasa::new ⬄x-> new Klasa(x)

Przykłady użycia:  


Wbudowane interfejsy funkcyjne:

* Runnable – z metodą run() – do wątków
* ActionListener – do reakcji na zdarzenia przy użyciu GUI (np. na wciśnięcie przycisku  
  
* FileFilter – do wybierania plików z kolekcji:  
  
* Callable – do definicji zadania do wykonania przez wykonawcę (Executor)
* Predicate<T> - boolean test(T v) - do testowania warunków
* BiPredicate<U,V> - boolean test<U u, V b> - do warunków dla 2 obiektów
* Function<S,T> - T apply(S v) – do przemiany typu S w typ T
* Function<U,V,R> - R apply(U u, V v) – do przemiany dwóch obiektów w trzeci obiekt
* UnaryOperator<T> - T apply (T v) – do modyfikacji typu T
* BinaryOperator<T> - T apply (T v1, T v2) – do modyfikacji dwóch obiektów typu T na raz
* Supplier<T> - T get() – do generowania obiektów typu T
* Consumer<T> - void accept(T v) – do wykonania czegoś na T bez zwracania
* BiConsumer<U,V> - void accept(U u, V v) – do wykonania czegoś na dwóch obiektach bez zwracania

Predicate/BiPredicate mają też implementacje metod:

* and(predicate2) – zwraca obiekt implementujący Predicate, który zawiera koniunkcję this i predicate2
* or(predicate2) – zwraca obiekt implementujący Predicate, który zawiera alternatywę this i predicate2

Function/BiFunction maja metody:

* compose(funk) – tworzy lambdę dla interfejsu Function, która najpierw wykona funk, a potem this z argumentem równym wyjściu funk
* andThen(funk) – odwrotna kolejność: najpierw this, potem funk

## 34. Przetwarzanie strumieniowe (środki pakietu java.util.stream).

**Strumień** – sekwencja elementów przesyłanych w potoku (połączeniu) różnych operacji. Same elementy nie są przechowywane w jakichkolwiek konkretnych strukturach danych

**Potok** – ma swoje źródło (może nim być: kolekcja, tablica, plik, napis…), kolejne wykonywane operacje i kończy się redukcją – finalnym efektem w postaci wartości lub zestawu wartości

Rodzaje operacji w potoku:

* **Pośrednie** – dające w wyniku inny strumień (np. operacje map i filter), te dzielą się na:
  + **Bezstanowe** – niewymagające od strumienia wiedzy o wartościach poprzednio przetwarzanych elementów (np. filter, forEach) – skupiają się tylko na aktualnym elemencie
  + **Stanowe** – wymagają od strumienia pamiętania stanów innych od, od akurat przetwarzanego, elementów (np. sortowanie elementów strumienia)
* **Terminalne** – kończące przetwarzanie strumienia (collect, forEach)
* **Skracające** (short-circuit) – takie, które powodują, że wcześniejsze w potoku operacje pośrednie kończą działanie, gdy rezultat operacji skracającej jest jasny. Mogą być zarówno pośrednie jak i terminalne

Cechy operacji strumieniowych:

* **Operacje pośrednie są leniwe** – nie są wykonywane i nie generują żadnych wartości dopóki nie zostanie wywołana jakaś operacja terminalna. Elementy strumienia są przetwarzane tylko w zakresie, jaki jest potrzebny do uzyskania wymaganego wyniku
* **Generatory i iteratory** – specjalne operacje, które generują elementy strumienia ad hoc – takie strumienie mogą być nieskończone
* **Mogą być przetwarzane równolegle**
* Wykorzystują **lambda-wyrażenia**

Operacje pośrednie:

* map(Function f) – do przekształcania elementów przy pomocy funkcji
* flatMap(Function f) – to samo, ale funkcja f musi zwraca strumień (dla każdego obiektu zwraca strumień) – strumienie są na koniec łączone w jeden
* filter(Predicate p) – zwraca tylko te elementy, dla których p daje true
* distinct() – zwraca tylko unikatowe obiekty – operacja stanowa
* sorted(…) – zwraca posortowane elementy – operacja stanowa
* unordered() – zwraca nieuporządkowany strumień
* limit(int n) – zwraca tylko pierwsze n elementów strumienia
* generate(Supplier s) – tworzy strumień przy pomocy elementów zwracanych przez Supplier – warto dać im na którymś etapie operację limit, bo inaczej tworzy obiekty w nieskończoność
* iterate(init, UnaryOperator op) – zwraca iteracyjne tworzony strumień, który jest generowany przez o wobec wartości startowej init
* substream(…) – zwraca część strumienia (np. ostatnie 10 elementów, elementy z zakresu od 5 do 20…)
* parallel() – zwraca strumień do operacji równoległych
* sequential() – zwraca strumień do operacji sekwencyjnych

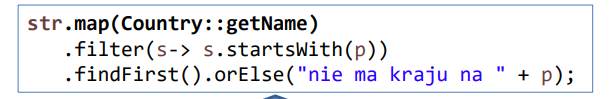
Operacje terminalne:

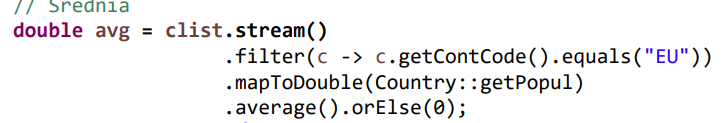
* allMatch(Predicate p) – zwraca true, jeśli dla wszystkich elementów strumienia p zwraca true
* anyMatch(Predicate p) – zwraca true, jeśli dla jakiegokolwiek elementu strumienia p zwraca true
* noneMatch(Predicate p) - zwraca true, jeśli dla żadnego elementu strumienia p nie zwraca true
* findAny() – zwraca dowolny element strumienia jako obiekt typu Optional
* findFirst() – zwraca pierwszy element strumienia jako obiekt typu Optional
* count() – zwraca liczbę elementów strumienia
* forEach(Consumer c) – dla każdego elementu strumienia robi jakąś operację, którą definiujemy w konsumerze
* forEachOrdered(Consumer c) – to samo, ale zachowuje kolejność, jeżeli jakąś ustalimy
* reduce(…) – sprowadza wszystkie elementy strumienia do jakiejś wartości (np. sumy)
* collect(…) – zbiera elementy ze strumienia do modyfikowalnego kontenera (np. listy, mapy, bufora znakowego, innego strumienia)
* max(…)/min(…) – zwraca maksymalny/minimalny element strumienia zgodnie z porównaniem komparatora
* toArray(…) – zbiera elementy strumienia w tablicę

W klasie Files – statyczna metoda lines(Path ścieżka\_do\_pliku) – tworzy strumień linii w pliku (możemy też podać charset, domyślny to utf-8)

O klasie Optional (od metody findFirst() i findAny():

* Jest parametryzowana typem obiektu, którego się spodziewamy
* Ma metodę isPresent() – true, jeśli udało się uzyskać obiekt
* Ma metodę get() – zwraca obiekt, jeżeli isPresent() == true. W przeciwnym wypadku rzuca wyjątkiem
* Ma metodę orElse(…) – możemy w niej podać, co ma być zwrócone jak nie ma obiektu, unikając przy tym wyjątku

O leniwym wykonaniu – mamy przykład:  
  
metoda filter wykona się tylko do momentu, gdy znajdzie pierwszy obiekt, który zaczyna się na literę w zmiennej p (ponieważ potem jest findFirst() – filter nie musi przeglądać całego strumienia, jeżeli znajdzie chociaż jeden element po drodze, który spełnia warunek w lambda-wyrażeniu)

Strumienie liczbowe (IntStream, LongStream …) możemy wygenerować operacją strumieniową mapToTyp (np. mapToInt) – strumienie liczbowe mają specjalne metody redukcji: sum(), min(), max(), average()  


Collect() – tworzy kolekcje ze strumienia, potrzebuje kolektora. Domyślne kolektory z klasy Collectors:

* toList()
* toSet()
* toCollection(Supplier s) – tworzy dowolną kolekcję. Jako s podaje się zazwyczaj col::new jakiejś kolekcji
* groupingBy(…) – tworzy mapę klucz-obiekt strumienia. Klucz jest generowany na podstawie lambda-wyrażenia w argumencie metody. Jak wiele obiektów generuje ten sam klucz, to są one umieszczane w liście pod tym kluczem
* toMap(…) – podajemy 2 metody: jedna generuje klucze, a druga wartości do tych kluczy
* joining(…) – zbiera obiekty do Stringa, w argumencie podajemy separator

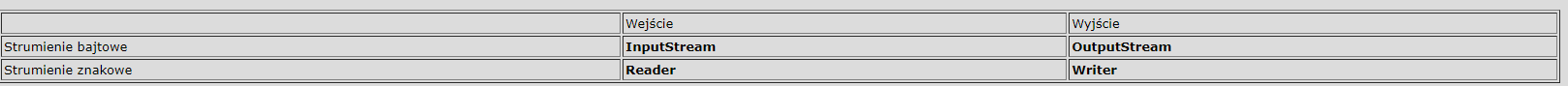
## 35. Narzędzia programowania operacji wejście-wyjścia w języku Java.

Podstawowe pakiety do operacji wejścia-wyjścia:

* java.io – do operowania na strumieniach danych
* java.nio – dodatki: kanały, bufory, selektory. Nie zastępuje .io, ale go rozszerza

hierarchie strumieni:

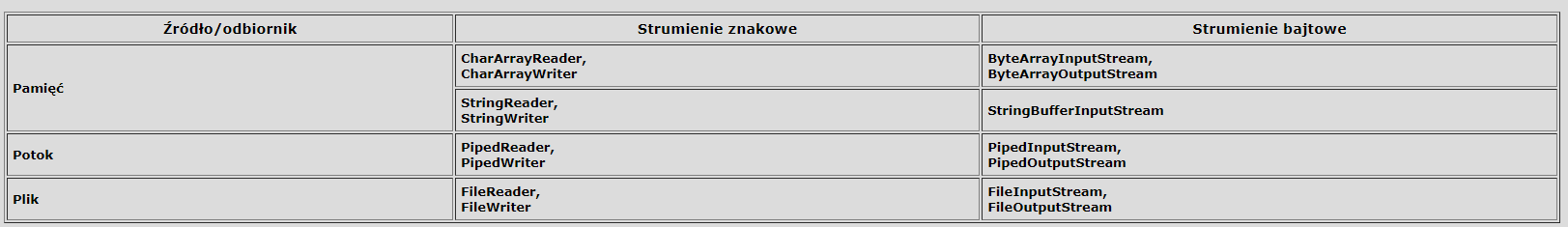
* **wejściowe** i **wyjściowe**
* **bajtowe** i **znakowe** (dla znakowych atomową wartością jest znak Unikodu, który ma 2 bajty)

Klasy reprezentujące te hierarchie:  


Metody z tych klas:

* read(...) – do odczytania bajta/znaku lub jakiejś określonej porcji bajtów/znaków
* write(…) – do zapisu bajta/znaku lub ich tablicy (dla znaków możemy zapisać też String)
* skip(…)/mark(…)/reset() – do pozycjonowania strumienia
* close() – do zamknięcia strumienia – trzeba to robić, gdy kończymy operacje na nim

Strumieniowe klasy przedmiotowe – takie, które skupiają się na zapisie/odczycie ze źródła konkretnego typu (np. pliku):



Klasy przetwarzające – takie, które przekształcają dane w trakcie operacji strumieniowych niezależnie od źródła danych:



Opis tych rodzajów przetwarzania:

* **Buforowanie** – ogranicza liczbę fizyczny odwołań do urządzeń zewnętrznych (warto używać np. czytając duże pliki tekstowe)
* **Filtrowanie** – tu są klasy abstrakcyjne, które definiują interfejsy dla prawdziwych filtrów (czyli kolejnych operacji z tej listy)
* **Konwersja znak-bajty** – InputStreamReader czyta bajty i zamienia je na znaki / OutputStreamReader czyta znaki i zapisuje je jako bajty – warto używać, gdy chcemy zapisać znaki w konkretnym kodowaniu (ważne np. przy pisaniu polskich znaków)
* **Konkatenacja** – łączy wiele strumieni wejściowych i pozwala je traktować jako jeden strumień
* **Serializacja** – do zapisu/odczytu obiektów z/do pliku, żeby móc je wykorzystać np. po wyłączeniu programu lub żeby je przesłać socketem
* **Konwersja danych** – pozwala czytać/pisać dane typów pierwotnych (np. liczby rzeczywiste) w postaci binarnej (ma metody w stylu readInt() i writeInt(int i) do wczytywania/pisania konkretnych typów w postaci binarnej)
* **Zliczanie wierszy** – czyta wiersze i pozwala w każdej chwili sprawdzić, na którym wierszu jesteśmy
* **Podglądanie** – pozwala zobaczyć następny znak/bajt bez „wyciągania” tego znaku/bajta ze strumienia
* **Drukowanie** – ma wygodne metody wyjścia (np. println) – często używane do wprowadzania informacji do innych strumieni

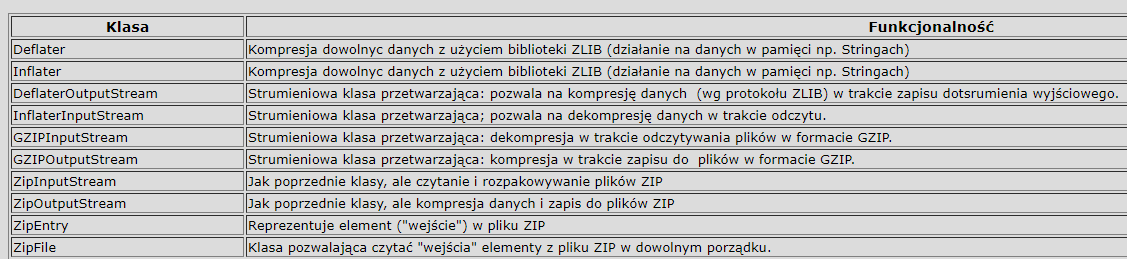
O serializacji:

* Tylko serializowane obiekty mogą być przesyłane strumieniowo
* Żeby móc serializować obiekt, klasa tego obiektu musi implementować interfejs **Serializable**
* Po wczytaniu obiektu serializowanego, rekurencyjnie wczytywane są jego obiekty składowe

**Potoki** – służą do przesyłania danych pomiędzy równolegle działającymi wątkami

**Pliki o dostępie swobodnym** – takie, które pozwalają na jednoczesny odczyt i pisanie (lub sam odczyt) oraz na dostęp do bajtów w dowolnej kolejności, bez potrzeby sekwencyjnego przetwarzania pliku od początku  
Do plików swobodnych mamy klasę **RandomAccessFile** – zawiera one filePointer, który pozwala ustawić pozycję w pliku metodami seek() oraz skip() – RANDOMACCESSFILE NIE JEST STRUMIENIEM

java.util.zip – pakiet do kompresji i dekompresji danych:



Żeby skompresować plik:

* Tworzymy ZipOutputStream, który opakowuje BufferedInputStream, który opakowuje FIleInputStream:  
  new ZipOutputStream(new BufferedInputStream(new FileInputStream("nazwa.zip"));
* Dla każdego pliku, który chcemy skompresować tworzymy obiekt ZipEntry: ZipEntry entry = new ZipEntry(nazwa);
* Podajemy ZipEntry do strumienia ZipOutputStream: zip.putNextEntry(entry);
* Zapisujemy do strumienia zawartość pliku wejściowego
* Zamykamy strumień: zip.closeEntry()

Do rozpakowania:

* Tworzymy ZipInputStream (podobnie jak tworzyliśmy ZipOutputStream)
* W pętli while((entry = zip.getNextEntry() ) != null) wczytujemy elementy archiwum
* Z każdego entry tworzymy strumień, którym wczytujemy plik

java.util.Scanner – pakiet do łatwego i elastycznego rozbioru informacji zawierającej napisy i dane typów prostych (operacje na Stringach, regexy, rozbijanie testu na wiersze, szukanie w tekście)

java.util.Formatter – pakiet do formatowania danych (np. reprezentację liczb/dat w zależności od Locale, kodowanie znaków)

## 36. Zarządzanie projektem budowy oprogramowania: rodzaje działań, dobór metodyki oraz kontekst pozatechniczny.

**Metodyka** – zestaw pojęć, notacji, modeli, języków, technik i sposobów postępowania służących do analizy dziedziny stanowiącej przedmiot projektowanego systemu oraz do projektowania pojęciowego, logicznego i/lub fizycznego

Podział metodyk ze względu na podejście do procedur, dokumentów itp.:

1. Ciężkie – duża formalność, wiele dokumentów, zatwierdzeń, stosowane do dużych projektów:
   1. Prince2
   2. PMBoK/PMI
2. Lekkie (zwinne) – mniej formalne, dokumentacja nie gra dużej roli:
   1. SCRUM
   2. XP
   3. XPrince
   4. Top10

Podział metodyk ze względu na przeznaczenie:

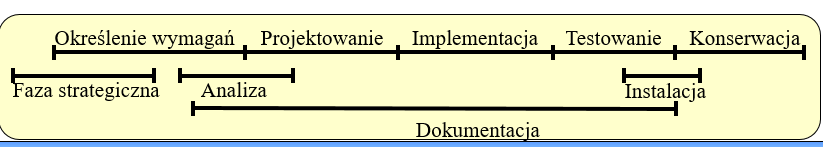
1. Zarządcze – dotyczące organizacji pracy w projekcie (np. rozmiary zespołów, role, hierarchia):
   1. Prince2
   2. PMBoK
   3. AgilePM
2. Wytwórcze – skupiają się na procesie tworzenia (np. jak przeprowadzać testy oprogramowania):
   1. XP (extreme programming)
3. Zarządczo-wytwórcze – łączą cechy zarządczych i wytwórczych:
   1. RUP (rational unified process)
   2. SCRUM

Metodyka ustala:

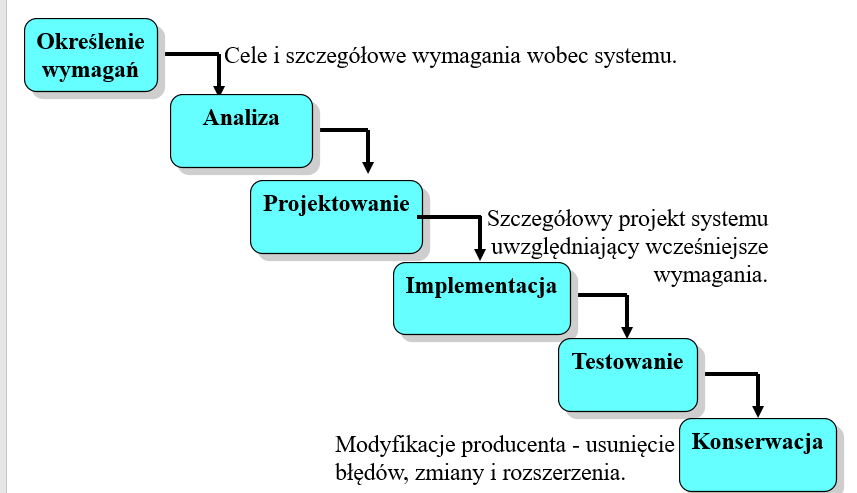
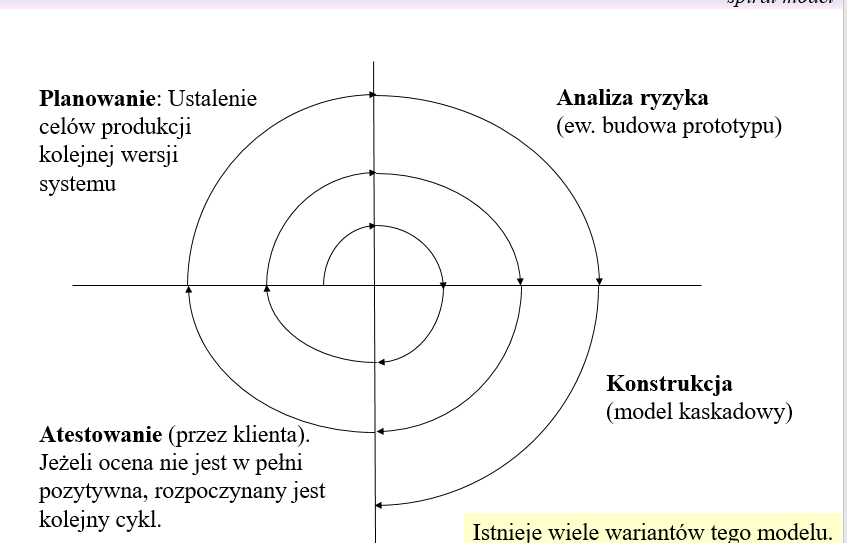
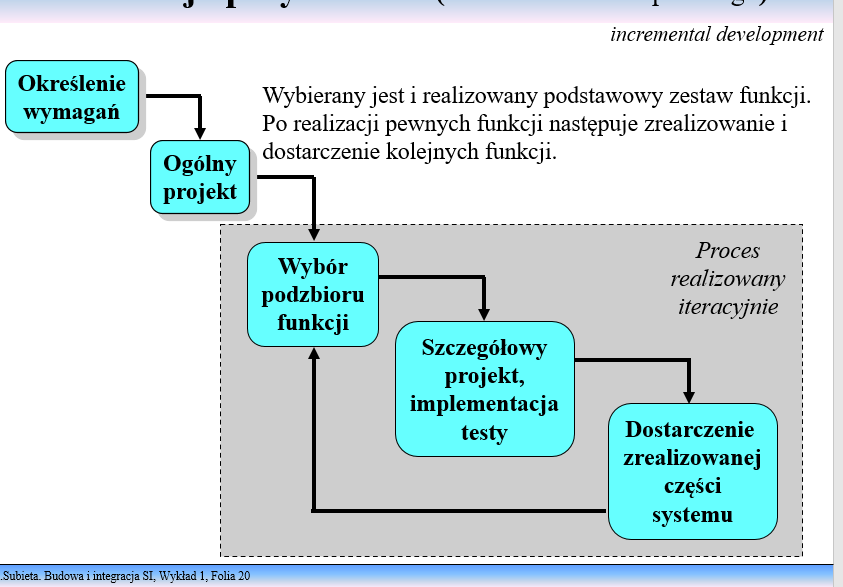
* Fazy projektu
* Role uczestników projektu
* Scenariusze postępowania w każdej z faz
* Reguły przechodzenia między fazami
* Notacje, których należy używać
* Dokumentację powstającą w każdej z faz

Cykl życiowy oprogramowania:

* Faza strategiczna – określenie celów strategicznych, planowanie i definicja projektu
* Określenie wymagań
* Analiza – dziedziny przedsiębiorczości, wymagań systemowych
* Projektowanie – pojęciowe, logiczne
* Implementacja/konstrukcja – rozwijanie, testowanie, dokumentacja
* Instalacja
* Przygotowanie użytkowników, akceptacja, szkolenie
* Działanie, wspomaganie tworzenia aplikacji
* Utrzymanie, konserwacja, pielęgnacja



Konkretne modele cyklu życia oprogramowania:

1. Kaskadowy – wykonujemy każdą fazę po kolei, brak możliwości powrotu do zakończonej fazy:  
     
   + ułatwia planowanie, harmonogramowanie, monitorowanie i rozliczenia finansowe  
   - narzuca kolejność wykonywanych prac, duży koszt błędów, długa przerwa w kontaktach z klientem
2. Model spiralny – kilka faz, które powtarzają się cyklicznie, a projekt rozwija się iteracyjnie:  
     
   Realizacja przyrostowa – inna wersja modelu spiralnego:  
   
3. Prototypowanie – ułatwia uniknąć wysokich kosztów błędów z fazy określenia wymagań, tworzy uproszczoną wersję systemu, którą przedstawiamy klientowi. Po jej zobaczeniu klient doprecyzowuje wymagania, a pełny system jest robiony modelem kaskadowym.  
   Metody prototypowania:
   1. Niepełna realizacja – tylko część funkcji jest implementowana
   2. Języki wysokiego poziomu – Smalltalk, Lisp, Prolog
   3. Wykorzystanie gotowych komponentów
   4. Generatory UI – wnętrze systemu jest „podróbką”
   5. Szybkie programowanie (quick-and-dirty) – np. z pominięciem testowania
4. Montaż z gotowych komponentów – kupujemy gotowe elementy/wykorzystujemy elementy z poprzednich projektów.  
   + Wysoka niezawodność, niskie ryzyko niepowodzenia, utrzymanie standardów  
   - Dodatkowe koszty, ryzyko uzależnienia się od dostawcy elementów, ryzyko braku potrzebnych narzędzi

## 37.    Język UML – charakterystyka oraz sposób wsparcia różnorodnych modeli danych.

UML – Unified Modeling Language – język służący do specyfikowania, konstruowania, obrazowania i dokumentowania składowych systemów oprogramowania

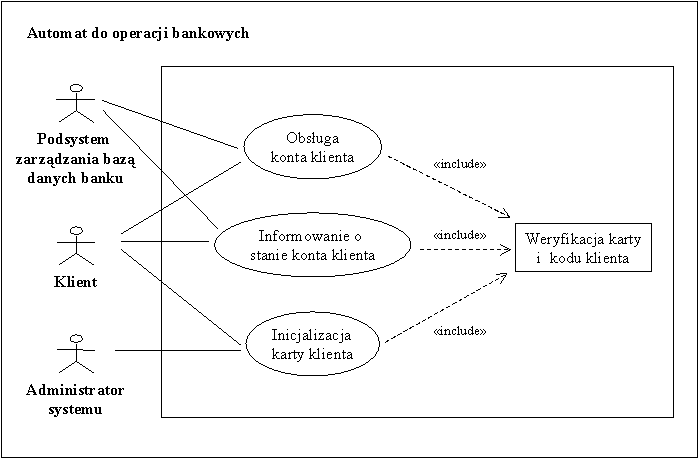
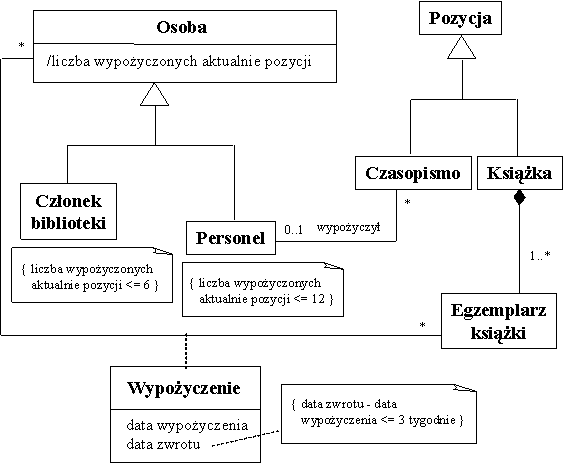
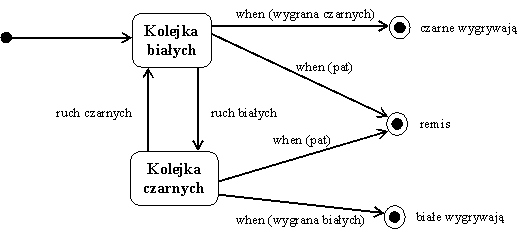
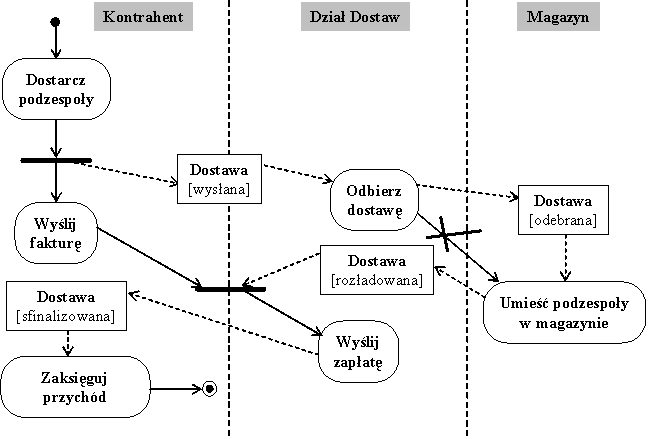
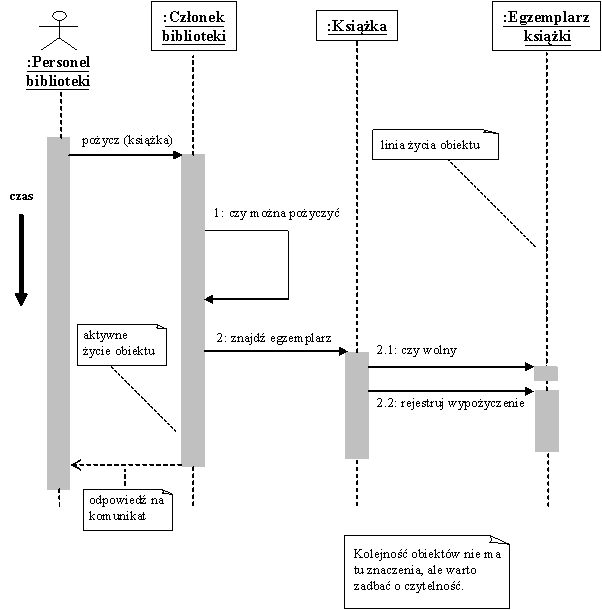
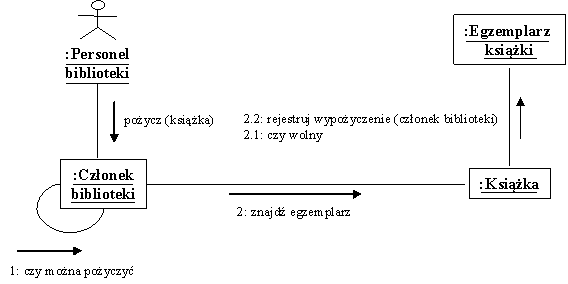
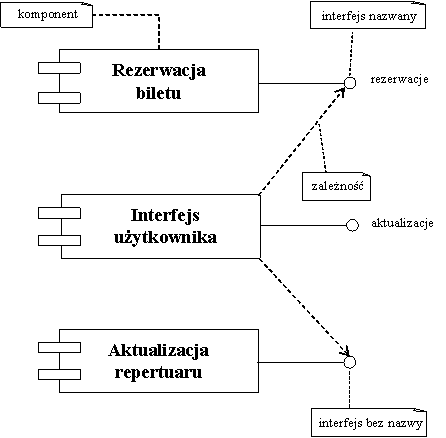
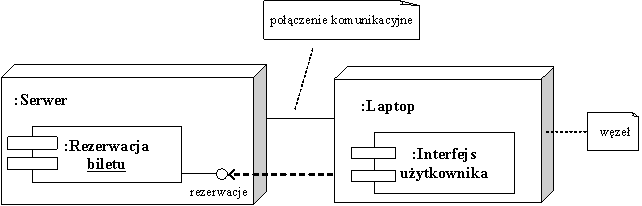
Ma 3 autorów, każdy jest autorem innej metodyki wchodzącej w skład UML:

* OOAD – użyteczna w projektowaniu i określaniu związków ze środowiskiem implementacji, nie wspiera fazy rozpoznania i analizy wymagań użytkowników
* OOSE – do modelowania aspektu użytkowników i cyklu życiowego systemu, ale nie wspiera implementacji
* OMT – do modelowania dziedziny przedmiotowej, ale nie wspiera modelowania użytkowników ani implementacji

UML dodatkowo wprowadził:

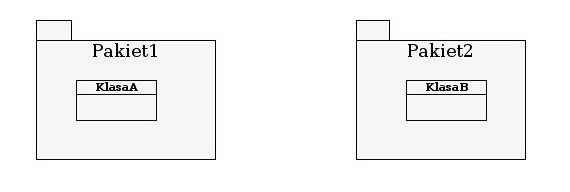
* Prototypowanie
* Komponenty
* Przystosowanie notacji do indywidualnej preferencji projektantów 4 mechanizmy rozszerzalności:
  + prototypy <<extend>> - do meta-klasyfikacji elementów diagramu
  + ograniczenia {>10} – warunki nałożone na element diagramu
  + wartości etykietowane {nowe, rozpatrywane, rozwiązane, anulowane} – nazwane wartości w diagramie
  + komentarze – adnotacje, które nie wnoszą nowej informacji z punktu widzenia analizowanej dziedziny problemowej

Diagramy wchodzące w skład UML:

* Diagram przypadków użycia – do modelowania funkcjonalności systemu z punktu widzenia użytkowników  
  
* Diagram klas – do modelowania struktury danych systemu  
  
* Diagramy dynamiczne – do modelowania zachowań:
  + Diagram stanów  
    
  + Diagram aktywności (ten odpowiednik z BPMN, ma tory pływackie i przekazywanie obiektów)  
    
  + Diagramy interakcji:
    - Diagram sekwencji – te z linią życia obiektów i wysyłaniem komunikatów w czasie  
      
    - Współpracy – łączy ze sobą aktorów i obiekty:  
      
  + Diagramy implementacyjne:
    - Diagram komponentów – do specyfikowania implementacji elementów projektu i zależności między nimi:  
      
    - Diagram wdrożeniowy – do przedstawienia połączeń komponentów sprzętowych:  
      

Modele danych:

* **Model obiektowy** – to, co przedstawiamy na diagramie klas
* **Model przypadków użycia** – na diagramie przypadków użycia
* **Model implementacji** – diagram komponentów + wdrożeniowy
* **Model dynamiczny** – diagram stanów + diagram aktywności + diagram interakcji
* **Model zarządzania** – diagram pakietów

W tamtej liście nie było, ale diagram pakietów to ten z katalogami jeden w drugim, dzielącymi system na elementy (my na wykładzie rysowaliśmy taki diagram dla PJATK, gdzie dziekanat, studium językowe, biblioteka itp. to były osobne pakiety):  


## 38.    Wzorce projektowe oraz ramy programistyczne (frameworks) – charakterystyka, przykłady, zastosowania.

**Wzorce projektowe** – wielokrotnie powtarzające się i pozytywnie zweryfikowane schematy rozwiązań często spotykanych problemów projektowych. Dotyczą architektury całej aplikacji, a nie pojedynczej klasy. Stosowanie ich pozwala na pisanie **lepszych, bardziej efektywnych, skalowalnych, łatwiej modyfikowalnych, mniej narażonych na błędy programów**.

MVC (model-view-controller) – pierwszy wzorzec projektowy (orginalnie w języku smalltalk), podział kodu na części:

* Model – dane związane z komponentami lub stany komponentów
* Widok (view) – wizualna reprezentacja danych lub stanów
* Sterownik (controller) – interakcja użytkownika z widokiem + modyfikacja modelu (danych,stanów)

Wzorce projektowe dzielą się na 3 klasy:

1. **Konstrukcyjne** – tworzenie obiektów, delegowanie procesu tworzenia do innych klas, kontrola nad sposobem tworzenia obiektów:
   1. **Factory** – mamy jedną klasę, która tworzy obiekty innych klas – scentralizowany sposób tworzenia obiektów jest prostszy w modyfikacji
   2. **Factory method** – mamy interfejs funkcyjny Factory do tworzenia obiektów klas implementujących interfejs Product (rozszerzenie factory, bo nie mamy jednej konkretnej implementacji, ale interfejs) – implementujemy te interfejsy do tworzenia fabryk i produktów – pozwala łatwo zmienić sposób tworzenia obiektów bez modyfikacji np. argumentów konstruktora w każdym miejscu
   3. **Abstract factory** – rozszerzenie factory method: Interface AbstactFactory ma wiele metod do tworzenia wielu rodzajów produktów
   4. **Singleton** – zapewnia, że klasa będzie miała tylko jedną instancję (dla danej klasy można stworzyć tylko jeden obiekt) + zapewnia globalny dostęp do tego obiektu (ma metodę getInstance(), która albo tworzy obiekt, albo zwraca już istniejący) – „lazy instantiation”: obiekt jest tworzony tylko wtedy, gdy jest potrzebny, ale mogą być problemy z przetwarzaniem wielowątkowym i serializacją (bo może ona stworzyć kopie singletona)
   5. **Builder** – pozwala tworzyć obiekty etapami krok po kroku. Mamy interfejs Builder, który ma kilka metod: po jednej na etap tworzenia obiektu. Dzięki temu nie musimy wkładać dużej ilości parametrów opcjonalnych do konstruktora. Builder ma też metodę reset(), która wyrzuca wszystko i tworzy nowy obiekt
   6. **Prototype** – do kopiowania już istniejących obiektów bez tworzenia zależności między kodem a klasami obiektów. Mamy interfejs z metodą clone(), która tworzy i zwraca kopię danego obiektu. Dzięki temu nie musimy się martwić np. o pola prywatne
2. **Strukturalne** – zarządzanie strukturą obiektów i strukturami złożonymi z obiektów:
   1. **Adapter** – do współdziałania obiektów o niekompatybilnych interfejsach. Klasa Adapter implementuje interfejs jednego obiektu i opakowuje drugi obiekt (np. Jedna klasa zapisuje dane w formacie XML, a druga może czytać tylko format JSON)
   2. **Bridge** – do dzielenia dużej klasy lub zestawu spokrewnionych klas na dwie hierarchie – abstrakcję (np. GUI) i implementację (np. API). Używane, gdy chcemy przeorganizować dużą klasę, która posiada wiele wariantów tej samej funkcjonalności (np. współpracuje z bazami danych różnego typu)
   3. **Composite** – łączenie część-całość w taki sposób, że części jak i całość mogą być traktowane w ten sam sposób. Używane np. w Java AWT w klasach JMenu i JMenuItem do tworzenia wielopoziomowych menu (zazwyczaj tak tworzy się drzewa)
   4. **Decorator** – do dodawania nowych obowiązków obiektom poprzez opakowywanie ich w inne obiekty – chroni przed psuciem kodu tam, gdzie dany obiekt jest już w danej formie używany + pozwala rozszerzać obiekty tam, gdzie nie możemy użyć dziedziczenia
   5. **Facade** – do dostarczania uproszczonego interfejsu złożonym bibliotekom i frameworkom
   6. **Flyweight** – do klas, które odwołują się do bardzo duże liczby podobnych (takich samych) obiektów, np. ramki wokół komponentów w GUI. Mamy klasę, która tworzy singletony powtarzających się obiektów, dzięki czemu klasy korzystające z tych obiektów współdzielą jedną instancję i oszczędzają pamięć (ten singleton powinien być niezmienny)
   7. **Proxy** – do tworzenia obiektów zastępczych w miejsce innych obiektów (pośrednik do obiektu) – daje kontrolę dostępu i pozwala prowadzić dziennik żądań
3. **Behawioralne** – zachowanie obiektów i komunikacji między nimi:
   1. **Chain of responsibility** – dajemy obiekt jakimś klasom, które po kolei coś robią z tym obiektem i przekazują go dalej – gdy różne rodzaje żądań mają być obsługiwane na różne sposoby, ale dokładne typy tych żądań i ich sekwencje nie są wcześniej znane (dzięki temu nie musimy używać wielkich ciągów else if)
   2. **Command** – pozwala na enkapsulację zlecenia do wykonania jako obiekt klasy implementującej określony interfejs. Uniezależnia logikę działania aplikacji od konkretnych zleceń do wykonania – upraszcza modyfikowalność
   3. **Interpreter** – tłumaczy jedną rzecz na inną (np. w kompilatorze kod źródłowy jest tłumaczony na kod bitowy) – rekurencyjnie interpretuje wyrażenia aż trafi na wyrażenie terminalne – używany np. w java.Pattern
   4. **Iterator** – pozwala przechodzić obiekt po obiekcie niezależnie od użytej struktury danych (listy, stosu, drzewa itd.). – Kolekcje w Javie mają swoje iteratory
   5. **Mediator** – do komunikacji między obiektami, redukuje chaos zależności między obiektami (taki kontroler lotów, który pilnuje, kto kiedy ląduje) – na tym polega Executor w Javie
   6. **Memento** – uzewnętrznia stan obiektu, zazwyczaj żeby móc go przywrócić (np. do poprzedniego stanu – operacja undo, rollback)
   7. **Observer** – jeżeli obiekt „obserwowany” zmienia stan, „obserwator” jest o tym powiadamiany. Pozwala na komunikację między niepowiązanymi obiektami – w Javie można do tego wykorzystać ChangeListener i PropertyChangeListener
   8. **State** – do reprezentacji stanu aplikacji, pozwala zmieniać zachowanie obiektu w zależności od jego stanu (trochę tak, jakby zmieniał klasę) – stan aplikacji jest w jednym obiekcie, a nie mieszance zmiennych w całej aplikacji
   9. **Strategy** – do definiowania rodziny algorytmów, umieszczania ich w osobnych klasach tak, by były wymienialne – pozwala wykorzystywać różne warianty algorytmu w zależności od sytuacji, nawet w trakcie działania programu (też pozwala zmniejszyć użycie else if)
   10. **Template** – definiujemy szkielet algorytmu w klasie bazowej, a podklasy mogą nadpisywać pewne etapy tego algorytmu bez zmiany całej jego struktury (w klasie bazowej mamy metodę templateMethod(), która używa po kolei metod step1(), step2()… - w podklasach przedefiniowujemy tylko metody step())
   11. **Visitor** – definiuje operacje, które są wykonywane na (strukturach lub zestawach) obiektów innych klas, pozwala to wprowadzać nowe operacje bez zmieniania klas tych obiektów. Mamy intefejs Visitor, który definiuje metody operacji na obiektach jakichś klas, a w tych klasach dodajemy metodę accept(visitor), która wykonuje operację visitora na this

**Fabryki** – definiują standardowy sposób tworzenia obiektów w sposób niezależny od ich rodzaju

**Framework** – skupia się na szczegółach technicznych i implementacyjnych. Znajduje się na niższym poziomie abstrakcji niż wzorce (np. JCF – Java Collection Framework)

## 39.    Zapewnienie jakości oraz testowanie oprogramowania – normy, metody, kryteria.

**Testowanie:**

**Weryfikacja** – testowanie zgodności systemu z wymaganiami zdefiniowanymi w fazie określenia wymagań. W skład weryfikacji wchodzą: przeglądy, inspekcje, testowanie, sprawdzanie, audytowanie…

**Atestowanie** (walidacja) – ocena systemu lub komponentu podczas lub na końcu procesu jego rozwoju na podstawie zgodności z wymaganiami – jest to **weryfikacja końcowa**

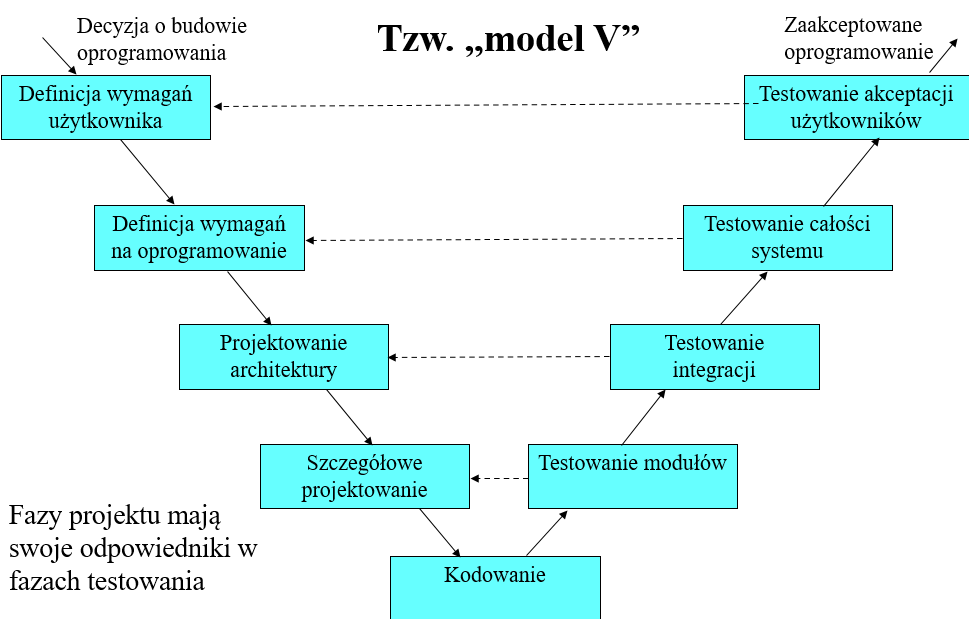
Główne cele testowania:

* Wykrycie i usunięcie błędów w systemie
* Ocena niezawodności systemu

Czynności związane z weryfikacją:

* Przeglądy techniczne oraz inspekcja oprogramowania
* Sprawdzanie zgodności wymagań na oprogramowanie z wymaganiami użytkownika
* Sprawdzanie zgodności komponentów projektu z wymaganiami na oprogramowanie
* Testowanie jednostek oprogramowania (modułów)
* Testowanie integracji oprogramowania, testowanie systemu
* Testowanie akceptacji systemu przez użytkowników

Model V – model przedstawiający związek między fazami projektu a fazami testowania:



**Przegląd** – proces lub spotkanie, podczas którego produkt roboczy lub ich zbiór jest prezentowany personelowi projektu, kierownictwie, użytkownikom, klientom lub innym zainteresowanym strono w celu uzyskania komentarzy, opinii i akceptacji. Może być formalny lub nieformalny

Rodzaje przeglądów formalnych:

* Przegląd techniczny – oceniane są elementy oprogramowania na zgodność postępu prac z przyjętym planem
* Przejście (walkthrough) – ocena dokumentów/modeli/projektów/kodu w celu szybkiego wykrycia defektów lub błędów stylistycznych (jak np. wcięcia w kodzie) i szkolenia
* Audyt – ocena projektu przez osoby niezależne od zespołu projektowego w celu potwierdzenia zgodności projektu z wymaganiami, standardami, procedurami, kontraktami itp. – ma dostarczyć obiektywną ocenę. **JEST ZWIĄZANY Z NORMAMI JAKOŚCI**

Skład zespołu oceniającego:

* Kierownik – ustala członków zespołu, organizację przebiegu oceny, spotkania, rozpowszechnianie dokumentów itp.
* Sekretarz
* Inni (np. użytkownicy, kierownik projektu, inżynierowie oprogramowania, bibliotekarz oprogramowania, personel od jakości…)

Polskie Stowarzyszenie Audytu – organizacja, której członkowie są uprawnionymi, licencjonowanymi audytorami

W ramach audytu należy dostarczyć dowody, że projekt:

* Posiada możliwości by osiągnąć sukces (zasoby, kompetencje, metody, standardy)
* Optymalnie wykorzystuje te możliwości
* Rzeczywiście osiąga założone cele (cząstkowe)

Przedmioty audytu:

* Procesy – sprawdzenie, czy wykonywane prace i sposób ich wykonania są prawidłowe
* Produkty - sprawdzenie, czy rezultaty tych prac odpowiadają zakładanym wymaganiom

Perspektywy:

* Technologia – sprawdzenie, czy użyte techniki i znane rozwiązania są prawidłowe i prawidłowo stosowane
* Zarządzanie – sprawdzenie, czy sposoby zarządzania projektem umożliwiają jego sukces

**Inspekcja** – formalna technika oceny, gdzie wymagania są szczegółowo badane przez osoby nie będące autorami, w celu wykrycia błędów/naruszeń standardów/innych problemów. Stosowane dla „elitarnych” systemów

Cechy inspekcji:

* Zaplanowane i przygotowane sesje
* Notowanie błędów i problemów
* Bez udziału kierownictwa (przez techników dla techników)
* Nie ocenia pracowników
* Ma zagwarantowane zasoby
* Błędy są wykorzystywane w poprawie procesu programowego
* Proces inspekcji jest mierzony i poprawiany

Kroki inspekcji:

1. Inicjowanie – zgłoszenie potrzeby inspekcji, ustalenie lidera inspekcji
2. Planowanie – lider wybiera uczestników, listę kontrolną, reguły, tempo kontroli i daty spotkań
3. Spotkanie inicjujące – ustalenie ról, celów, oczekiwań, dystrybucja dokumentu, szkolenie
4. Kontrola indywidualna – uczestnicy sprawdzają dokument zgodnie z kryteriami, regułami i listą kontrolną
5. Spotkanie kontrolne (burza mózgów) – uczestnicy zbierają razem uwagi z kontroli indywidualnej, szukają nowych zagadnień i poprawiają proces inspekcji
6. Poprawa produktu – edytor (najczęściej autor) odnosi się do uwag – przeredagowuje dokument tak, by uniknąć błędnych interpretacji (czasem zgłoszony błąd jest inny niż błąd rzeczywisty)
7. Kontynuacja – lider sprawdza, że obsłużono wszystkie zagadnienia (sprawdza kompletność a nie poprawność)
8. Decyzja o gotowości – lider decyduje, czy produkt jest gotowy do przekazania dalej
9. Rozpowszechnienie dokumentu

Rodzaje testów:

1. **Wykrywanie błędów** ­– mają wykryć jak najwięcej błędów w programie
2. **Testy statystyczne** – wykrywają przyczyny najczęstszych błędów + ocena niezawodności systemu
3. **Testy dynamiczne** – wykonywanie fragmentów programu i porównywanie uzyskanych wyników z wynikami poprawnymi
4. **Testy statyczne** – analiza kodu bez jego uruchamiania (wykonywanie kodu w myśli). Nieefektywne dla większych programów. Metody matematyczne weryfikacji programów też są testami statycznymi

**Błąd** – niepoprawna konstrukcja znajdująca się w programie, która może doprowadzić do niewłaściwego działania  
**Błędne wykonanie** – niepoprawne działanie systemu w trakcie jego pracy

**Typowe fazy testowania systemu**:

* **Testy modułów** – już w fazie implementacji, tuż po skończeniu modułu
* **Testy systemu** – integrowanie modułów i testowanie podsystemów i systemu jako całości
* **Testy akceptacji** – wykonywane przez przyszłego użytkownika: **testy alfa** – jak system jest na zamówienie i dajemy go klientowi do przetestowania; **testy beta** – jak system chcemy wypuścić na rynek i przekazujemy za darmo kilka kopii do przetestowania

Testowaniu podlegają:

* Wydajność systemu
* Interfejsy systemu
* Własności operacyjne systemu
* Zużycie zasobów
* Zabezpieczenia
* Przenaszalność (np. na różne systemy operacyjne)
* Niezawodność (jak często są awarie)
* Odtwarzalność (ile czasu potrzeba na naprawę awarii)
* Bezpieczeństwo (jak duże są skutki awarii)
* Kompletność i jakość funkcji
* Modyfikowalność
* Obciążalność (np. maksymalna liczba użytkowników jednocześnie korzystających z systemu)
* Skalowalność
* Akceptowalność (poziom satysfakcji użytkowników)
* Jakość dokumentacji

Testy na zasadzie białej skrzynki – **testy strukturalne**:

* Logika wewnątrz programu może być sprawdzona
* Stosujemy do tego kody diagnostyczne i debuggery, które pozwalają śledzić np. zawartość zmiennych
* Wymagają przygotowania danych testowych (tak, by przetestować każdą możliwą ścieżkę w programie – wszystkie ify/else ify, maksymalnie dużo obrotów pętli/przeciętnie dużo obrotów pętli/ani jeden obrót pętli)
* Nie pozwalają pokazać brakujących funkcji w programie

Testy na zasadzie czarnej skrzynki – **testy funkcjonalne**:

* Nie znamy logiki wewnętrznej programu (nie widzimy, co się w środku dzieje)
* Obejmuje cały zakres danych wejściowych
* Żeby tych danych nie było nieskończoność (**kombinatoryczna ekspozja**), tworzymy „klasy równoważności” – dzielimy możliwe dane na takie grupy, jakie będą powodować te same błędy (np. liczby ujemne, dodatnie i zero), podział ten zależy od wymagań (warto też sprawdzać wartości graniczne względem tych wymagań)
* Często wymaga testowania kombinacji różnych typów danych – używamy do tego tablic decyzyjnych lub grafów przyczyna-skutek

Miary niezawodności:

* Prawdopodobieństwo błędnego wykonania
* Częstotliwość występowania błędnych wykonań
* Średni czas między błędnymi wykonaniami
* Dostępność (prawdopodobieństwo, że w danej chwili system będzie dostępny do użytkowania)

Niezawodność rośnie **logarytmicznie względem liczby testów**

Przy tworzeniu zestawów danych testowych warto pamiętać, że:

* Możliwość wykonania funkcji jest ważniejsza niż jakość jej wykonania
* Stare funkcje systemu są ważniejsze niż nowo wprowadzone
* Typowe sytuacje są ważniejsze niż sytuacje skrajne

Debuggery pozwalają na:

* Wyświetlenie stanu zmiennych programu
* Wykonywanie programu krok po kroku
* Ustalenie punktów kontrolnych w programie
* Zarządzanie plikiem źródłowym podczas testowania
* Tworzenie dziennika testowania, co ułatwia powtórzenie testowego przebiegu

**Analizatory przykrycia kodu** – pozwalają określić, jakie obszary kodu były wykonane w danym przebiegu testowania – pozwalają wykryć kod martwy jak i ten najczęściej wykonywany

**Programy porównujące** – pozwalają porównać dwa pliki/programy/zbiory danych w celu znalezienia podobieństw i różnic – używane przy porównywaniu wyników testów z wynikami oczekiwanymi

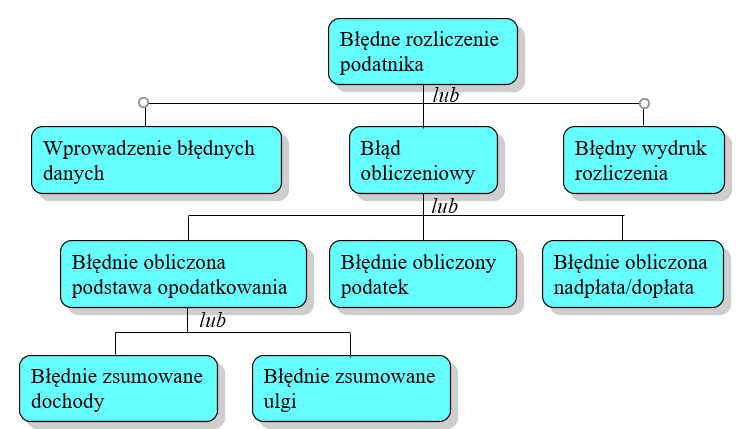
**Technika „posiewania błędów”** – grupa programistyczna celowo dodaje błędy do programu (podobne do błędów już istniejących), a inna grupa przeprowadza testy – służy to testowaniu metod testowania

Techniki testów systemu:

* **Testy wstępujące** – najpierw testujemy pojedyncze moduły i stopniowo wchodzimy na coraz wyższy poziom aż do poziomu całego systemu – nie zawsze możliwe, moduły mogą być od siebie niezależne
* **Testy zstępujące** – najpierw testujemy moduły wyższego poziomu, a te niższego poziomu zastępujemy implementacjami szkieletowymi. Po testach na wyższym poziomie, dokładamy moduły niższego poziomu. Robimy tak, aż zintegrujemy cały system

**Testy obciążeniowe** – badają wydajność i niezawodność systemu, wymuszają obciążenie równe lub większe niż maksymalne

**Testy odporności** – testy tego, co się stanie w przypadku zaniku zasilania, awarii sprzętowej, wprowadzenia niepoprawnych danych lub wydania sekwencji niepoprawnych poleceń

**Drzewo błędów** – w korzeniu ma konkretną niebezpieczną sytuację, w gałęziach ma sytuacje, które mogą prowadzić do korzenia:  


**Jakość:**

**Zapewnienie jakości** – zespół działań zmierzających do wytworzenia u wszystkich zainteresowanych przekonania, że dostarczony projekt realizuje swoje funkcje i odpowiada aktualnym wymaganiom i standardom.

Żeby ocena jakości była bardziej obiektywna, wykorzystuje **pomiary** (np. niezawodności, szybkości)

Ze względu na złożoność projektów, trudność w mierzeniu wszystkich czynników jakości i koszty pomiarów, ocena jakości często korzysta z **metod spekulacyjnych**

Zapewnienie jakości skupia się na **dyscyplinie całości procesu**, a nie na pomiarach efektów procesów

**TQM (zarządzanie przez jakość)** – pomysł założyciela firmy Toyota, „Jakość jest najważniejszym kryterium oceny przydatności produktów dla klienta, a to właśnie klient umożliwia funkcjonowanie wytwórcy tych produktów” – polega na angażowaniu klienta w sterowanie wszystkimi fazami procesu produkcyjnego

Definicje ze standardów ISO 9000:

* **Jakość** – ogół cech i właściwości wyrobu lub usługi decydujący o zdolności wyrobu lub usługi do zaspokojenia stwierdzonych lub przewidywanych potrzeb użytkownika produktu
* **System jakości** – odpowiednio zbudowana struktura organizacyjna z jednoznacznym podziałem odpowiedzialności, określeniem procedur, procesów i zasobów, umożliwiających wdrożenie zarządzania jakością
* **Zarządzanie jakością** – jest związane z aspektem całości funkcji zarządzania organizacji, który jest decydujący w określaniu i wdrażaniu polityki jakości
* **Polityka jakości** – ogół zamierzeń i kierunków działań organizacji dotyczących jakości, w sposób formalny wyrażony przez najwyższe kierownictwo organizacji, będącej systemem jakości
* **Audyt jakości** – systematyczne i niezależne badanie, mające określić, czy działania dotyczące jakości i ich wyniki odpowiadają zaplanowanym ustaleniom, czy te ustalenia są skutecznie realizowane i czy pozwalają na osiągnięcie odpowiedniego poziomu jakości

Model jakości ISO 9126:

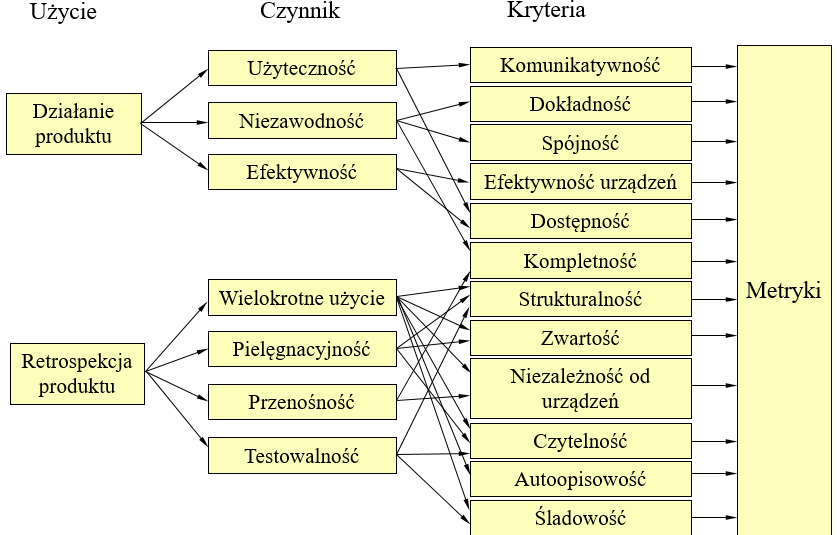
* Funkcjonalność
* Niezawodność
* Użyteczność
* Efektywność
* Pielęgnowalność
* Przenośność

Zasady zarządzania jakością:

* Ukierunkowanie na klienta
* Przywództwo (budowa wizji, identyfikacja wartości)
* Zaangażowanie ludzi
* Podejście procesowe (skupienie na konkretnych krokach procesu i relacjach między krokami)
* Podejście systemowe (skupienie na otoczeniu procesu wytwórczego)
* Ciągłe doskonalenie
* Rzetelna informacja
* Partnerstwo dla jakości (bliskie związki producentów z klientami)

**Zapewnienie Jakości Oprogramowania (ZJO)** – planowany i systematyczny wzorzec wszystkich działań potrzebnych dla dostarczenia adekwatnego potwierdzenie że element lub produkt jest zgodny z ustanowionymi wymaganiami technicznymi – personel ZJO powinien być zaangażowany na jak najwcześniejszym etapie projektu, żeby uniknąć ryzyka utraty jakości

Model jakości oprogramowania:



Proces wytwórczy może być:

* **Niedojrzały** – chaotyczny, dużo improwizacji, nie trzymanie się wymagań, harmonogramu i budżetu, stopniowe okrajanie funkcjonalności, niska jakość, brak obiektywnych kryteriów oceny
* **Dojrzały** – dobrze udokumentowany, obserwowany i ulepszany, role są konkretnie zdefiniowane, korzysta z obiektywnej, ilościowej oceny jakości

5 poziomów dojrzałości procesu wytwórczego:

1. Poziom początkowy (proces chaotyczny)
2. Poziom powtarzalny (wykonujemy te same kroki dla tych samych sytuacji, ale nie są one nigdzie spisane)
3. Poziom zdefiniowany (kroki procesu są spisane i każdy zainteresowany ma do nich dostęp)
4. Poziom zarządzany (proces jest monitorowany, np. pod względem tego, jak często jest wykonywany, ile trwa, które kroki są najbardziej czasochłonne itd.)
5. Poziom optymalizujący (dane z monitorowania procesu są wykorzystywane w celu jego poprawienia)

**Plan zapewnienia jakości oprogramowania (PZJO)** – powinien być sporządzany i modyfikowany przez cały okres życia oprogramowania (pierwsza wersja powinna powstać po określeniu wymagań użytkownika)

Cechy PZJO:

* Zrozumiały, lakoniczny, jasny, spójny i modyfikowalny styl
* Odpowiedniość – wyprodukowany przez komórkę jakości zespołu, który podejmuje się produkcji oprogramowania
* W formie papierowej (czasem też elektronicznej)
* Ma 4 rozdziały:
  + Dla fazy wymagań użytkownika i analizy
  + Dla fazy projektu architektury
  + Dla fazy projektowania i konstrukcji
  + Dla fazy budowy, testowania i instalacji oprogramowania
* Jest tworzony iteracyjnie po każdej fazie projektu

## 40.    Rodzaje, metody specyfikowania oraz rola wymagań w procesie wytwarzania oprogramowania.

**Faza określenia wymagań** – faza projektu, w której klient wspólnie z przedstawicielem producenta konstruuje zbiór wymagań projektu, które będą zgodne z postawionymi celami. Wymaga dużego zaangażowania ze strony klienta, przyszłych użytkowników systemów i ekspertów w dziedzinie

Poziomy ogólności opisu wymagań:

* **Definicja wymagań** – ogólny opis w języku naturalnym, wynik wstępnych rozmów z klientem
* **Specyfikacja wymagań** – ustrukturalizowany zapis wykorzystujący język naturalny + proste, częściowo sformalizowane notacje
* **Specyfikacja oprogramowania** – formalny opis wymagań, podzielenie wymagań na zwięzłe punkty. Podstawa fazy testowania

Dobry opis wymagań powinien:

* Być kompletny i niesprzeczny
* Opisywać zewnętrzne zachowanie systemu, a nie sposób jego realizacji
* Obejmować ograniczenia przy jakich musi pracować system
* Być łatwy w modyfikacji
* Brać pod uwagę możliwe zmiany wymagań wobec systemu
* Opisywać zachowanie systemu w skrajnych lub niepożądanych sytuacjach

Metody rozpoznania wymagań:

* **Wywiady i przeglądy** – lista pytań podzielona na odrębne zagadnienia, powinna pokrywać całość tematu. Wywiad powinien być przeprowadzany na reprezentatywnej grupie użytkowników, i powinien doprowadzić do szerokiej zgody i akceptacji projektu
* **Studium nad istniejącym oprogramowaniem** – ma ustali dobre i złe strony oprogramowania, które nasz projekt zastępuje
* **Studia wymagań systemowych** – gdy nowy system ma być częścią większego systemu
* **Studia osiągalności** – określenie realistycznych celów systemu i metod ich osiągnięcia
* **Prototypowanie** – Zbudowanie prototypu systemu działającego w zmniejszonej skali, z uproszczonym interfejsem

Rodzaje wymagań:

1. **Funkcjonalne** – opisują funkcje wykonywane przez system lub systemy zewnętrzne:
   1. Określenie wszystkich rodzajów użytkowników w systemie
   2. Określenie rodzajów użytkowników, którzy są niezbędni do działania systemu (obsługa, wprowadzenie danych, administracja)
   3. Dla każdego rodzaju użytkownika – określenie funkcji oraz sposobów korzystania z systemu
   4. Określenie systemów zewnętrznych
   5. Ustalenie struktur organizacyjnych, przepisów prawnych, statutów, zarządzeń, instrukcji itd. które określają funkcje systemu
2. **Niefunkcjonalne** – opisują ograniczenia, przy których system ma realizować swoje funkcje:
   1. **Wymagania dotyczące produktu** – np. możliwość operowania systemem wyłącznie przy pomocy klawiatury
   2. **Wymagania dotyczące procesu** – np. realizacja harmonogramów ma być zgodna z jakimś dokumentem
   3. **Wymagania zewnętrzne** – np. system musi współpracować z jakąś konkretną bazą danych bez zmiany jej struktury

**Inny podział:**

1. **Wymagania strategiczne (udziałowców)** – najszerszy kontekst, np. zwiększenie zysków
2. **Wymagania użytkownika** – wymagania, które podaje użytkownik
3. **Podfunkcje** – nie są podawane wprost przez użytkownika, ale wynikają z jego wymagań (np. możliwość logowania się do systemu)

Czynniki uwzględniane przy konstruowaniu wymagań niefunkcjonalnych:

* Możliwości systemu
* Objętość – ile użytkowników jednocześnie, ile czujników, ile danych
* Szybkość – jak długo może trwać operacja/sekwencja operacji
* Dokładność – precyzja pomiarów/przetwarzania
* Ograniczenia – na interfejsy, jakość, czas, sprzęt…
* Interfejsy komunikacyjne – sieć, protokoły komunikacji
* Interfejs sprzętowych – jaki sprzęt jest potrzebny, jego rozmiar/waga, wydajność (szybkość, RAM, dyski), wilgotność/temperatura/ciśnienie otoczenia
* Interfejsy oprogramowania – zgodność z innymi programami, system operacyjny, języki programowania
* Interakcja człowiek-maszyna – jaki rodzaj sprzętu (mysz, klawiatura), język/forma komunikatów
* Adaptowalność – organizacja reakcji na zmiany wymagań
* Bezpieczeństwo – poufność, prywatność, integralność, odporność na ataki
* Odporność na awarie
* Standardy – formaty plików, czcionki, polonizacja, standardy procesów i produktów
* Zasoby – wymagania finansowe, ludzkie i materiałowe
* Skala czasowa – czas wykonania, szkolenia, wdrażania…

**Metody specyfikacji wymagań**:

* **Język naturalny** – najczęściej stosowany, ale ma problemy z niejednoznacznością i nadmierną elastycznością (możliwością wyrażenia tego samego na wiele sposobów)
* **Formalizm matematyczny** – tylko dla specyficznych celów
* **Język naturalny strukturalny** – ograniczone słownictwo i składnia. Tematy i zagadnienia w punktach i podpunktach
* **Tablice, formularze** – zazwyczaj dwuwymiarowe tablice, pozwalają kojarzyć ze sobą różne aspekty (np. typ użytkownika z rodzajem usługi\_
* **Diagramy blokowe** – forma graficzna przedstawiająca cykl przetwarzania
* **Diagramy kontekstowe** – cały system jako jeden blok + jego powiązania z otoczeniem, wejściem i wyjściem
* **Diagramy przypadków użycia** – do przedstawienia aktorów i funkcji systemu

## 41. Usługi i protokoły warstwy aplikacji na przykładzie protokołu HTTP.

Warstwy w modelu OSI (od góry):

1. Warstwa aplikacji – najbardziej i najbliższe potrzebom użytkownika zadania: dyski sieciowe, udostępnianie drukarek/treści. Popularny protokół tej warstwy – **HTTP**
2. Warstwa transportowa – zapewnia niezawodną komunikację między komputerami w sieci: wykrywa zagubione pakiety i żąda retransmisji oraz identyfikuje poprawną kolejność pakietów. Tutaj pojawia się pojęcie **gniazda (socket) – protokół TCP i UDP**
3. Warstwa sieci – pozwala znajdować ścieżkę od hosta źródłowego do docelowego w sieci. Najpopularniejszy protokół – **IP**
4. Warstwa łącza danych – kontroluje ilość i jakość danych w zależności od chwilowych parametrów warstwy fizycznej
5. Warstwa fizyczna – przesyła sygnały przy pomocy fizycznego medium (powietrze, przewody miedziane, światłowody). Implementowana przy pomocy **karty sieciowej**

Protokół w warstwie aplikacji określa:

* Rodzaje komunikatów – np. komunikaty żądania i odpowiedzi
* Składnię komunikatów – jakie pola zawiera komunikat i jak te pola są oddzielane
* Znaczenie informacji w polach komunikatu
* Zasady określające kiedy i jak procesy wymieniają komunikaty

Protokoły publiczne warstwy aplikacji:

* Określane w **Request For Comments (RFC)**
* Pozwalają na współpracę różnych systemów
* np. HTTP (port 80), SMTP (port 25)

KaZaA – przykład protokołu prywatnego

HTTP – hypertext transfer protocol:

* Oparty o model klient\serwer:
  + Klient – przeglądarka, która żąda, otrzymuje i wyświetla obiekty WWW
  + Serwer – wysyła obiekty w odpowiedzi na żądania
* Obiektami WWW mogą być: plik HTML, obraz JPEG, aplet Java, plik audio itp.
* Strona WWW składa się z bazowego pliku HTML, który zawiera odnośniki do innych obiektów
* Każdy obiekt posiada adres **URL (Uniform Resource Locator)**
* Wykorzystuje protokół **TCP** (niezawodne połączenie)
* Jest **bezstanowy** – nie utrzymuje informacji o poprzednich połączeniach HTTP
* Ma 2 wersje:
  + HTTP/1.0 – nietrwałe HTTP: po wysłaniu jednego obiektu połączenie TCP jest zamykane
  + HTTP/1.1 – trwałe HTTP: w ramach jednego połączenia TCP można przesłać wiele obiektów

**RTT (Round Trip Time)** – czas potrzebny na przesłanie małego pakietu od nadawcy do odbiorcy i z powrotem. Dla nietrwałego RTT potrzebne są 2 RTT na każdy obiekt (jedno na inicjację połączenia, drugie na przesłanie żądania obiektu)

W HTTP mamy 2 typy komunikatów:

1. Żądanie – składa się z:
   * Linii żądania - GET /katalog/strona.html HTTP/1.1 (polecenie, adres, wersja HTTP)
   * Nagłówków - np. Host: www.uczelnia.edu.pl, User-agent: Mozilla/4.0
   * Znaku carriage return, line feed (pusta linia) – oznacza koniec nagłówków
   * Danych lub pustej linii
2. Odpowiedzi:
   * Linii statusu - HTTP/1.1 200 OK (wersja HTTP, kod statusu, jego nazwa)
   * Nagłówków – np. Content-Length: 6821, Content-Type: text/html
   * Danych - np. Plik HTMP

Metody żądań HTTP:

* GET – do pobierania danych
* POST – do wysyłania danych
* HEAD – prośba o posłanie odpowiedzi bez żądanego obiektu (danych)
* PUT – do wysyłania pliku, który chcemy umieścić pod danym adresem URL
* DELETE – do usuwania obiektów pod danym adresem URL

Typy kodów odpowiedzi HTTP:

* 1xx – kod informacyjny (np. 101 – Switching protocol, 111 – Connection refused)
* 2xx – kod powodzenia (np. 200 – OK, 201 – Created)
* 3xx – przekierowania (np. 301 – Moved permanently, 302 – Found: przekierowanie tymczasowe, 304 – not modified: serwer nie wysyła obiektu, jeżeli klient ma w schowku aktualną jego wersję)
* 4xx – błąd po stronie klienta (np. 401 – Unauthorized, 404 – Not found)
* 5xx – błąd po stronie serwera (np. 500 – internal serwer error, 501 – not implemented)

**Ciasteczka (cookies)** – mechanizm pozwalający utrzymać stan w protokole HTTP, składa się z 4 elementów:

* Nagłówek „cookie” w odpowiedzi
* Nagłówek „cookie” w żądaniu
* Plik z ciasteczkami na hoście klienta
* Baza danych na serwerze WWW

Ciasteczka pozwalają na:

* Uwierzytelnianie (akt udowodnienia tożsamości użytkownika, NIE JEGO UPRAWNIEŃ)
* Wózki z zakupami
* Rekomendacje
* Stan sesji (np. w banku elektronicznym)

## 42. Usługi warstwy transportu na przykładzie protokołu TCP.

Usługi i protokoły warstwy transportu:

* **Logiczna komunikacja** pomiędzy procesami aplikacji działającymi na różnych hostach
* Protokołytransportowe działają na systemach końcowych:
  + Nadawca – dzieli komunikat aplikacji na **segmenty**, przekazuje segmenty do warstwy sieci
  + Odbiorca – łączy segmenty w komunikat, który przekazuje do warstwy aplikacji

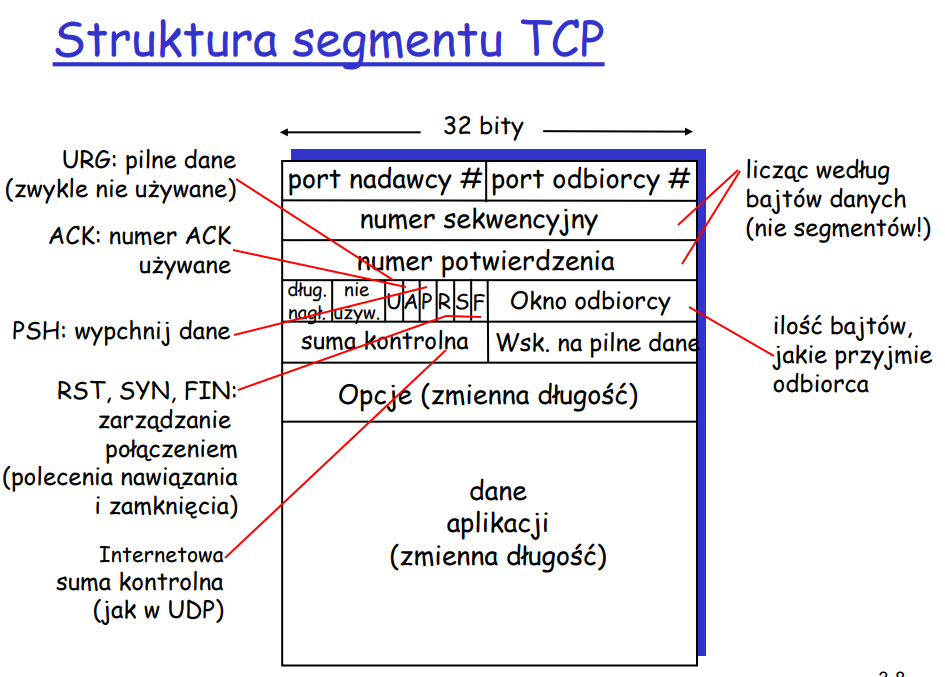
Protokoły warstwy transportu – **TCP** i **UDP:**

**TCP:**

* Niezawodna, uporządkowana komunikacja
* Kontrola przeciążenia
* Kontrola przepływu – żeby nadawca nie „zalał” odbiorcy
* Tworzenie połączenia
* Jeden nadawca-Jeden odbiorca (**koniec-koniec**)
* Bajty są wysyłane strumieniowo (**nie ma granic komunikatów**)
* Wysyłający grupowy
* Bufory u nadawcy i odbiorcy
* Komunikacja „**full duplex**” – dane mogą przepływać w obie strony w ramach tego samego połączenia
* **Wymaga inicjalizacji** połączenia

**UDP (User datagram protocol):**

* Zawodna, nieuporządkowana komunikacja
* Proste rozszerzenie usługi „best-effort” IP – segmenty mogą być zgubione lub dostarczone w zmienionej kolejności
* **Bezpołączeniowy** – nie ma inicjalizacji, każdy segment jest obsługiwany niezależnie od innych
* Często używany **do komunikacji strumieniowej** (bo tolerujemy tam straty, a taka komunikacja jest wrażliwa na opóźnienia)
* Używa **sumy kontrolnej** – zsumowana i zanegowana zawartość segmentu, dodatkowo umieszczana w nagłówku. Odbiorca liczy tę sumę i porównuję ją z tą z nagłówka, żeby sprawdzić, czy segment nie jest błędny



Numer sekwencyjny – numer pierwszego bajtu danych segmentu w strumieniu  
Numer potwierdzenia (ACK) – numer sekwencyjny następnego bajtu oczekiwanego od drugiej strony

Niezawodność komunikacji jest realizowana przez **retransmisji** (ponowne wysłanie segmentu w przypadku zdarzenia timeout lub duplikatu ACK)

Do nawiązywania połączenia w TCP wykonuje się **trzykrotny uścisk dłoni**:

1. Klient wysyła segment SYN (z początkowym numerem sekwencyjnym, bez danych)
2. Serwer odbiera SYN i odpowiada poprzez SYNACK(SYN i ACK)
3. Klient odbiera SYNACK i odpowiada segmentem ACK (w tym segmencie mogą już być dane)

Do zakończenia połączenia:

1. Klient wysyła segment FIN
2. Serwer go odbiera wysyła ACK, zamyka połączenie i wysyła FIN

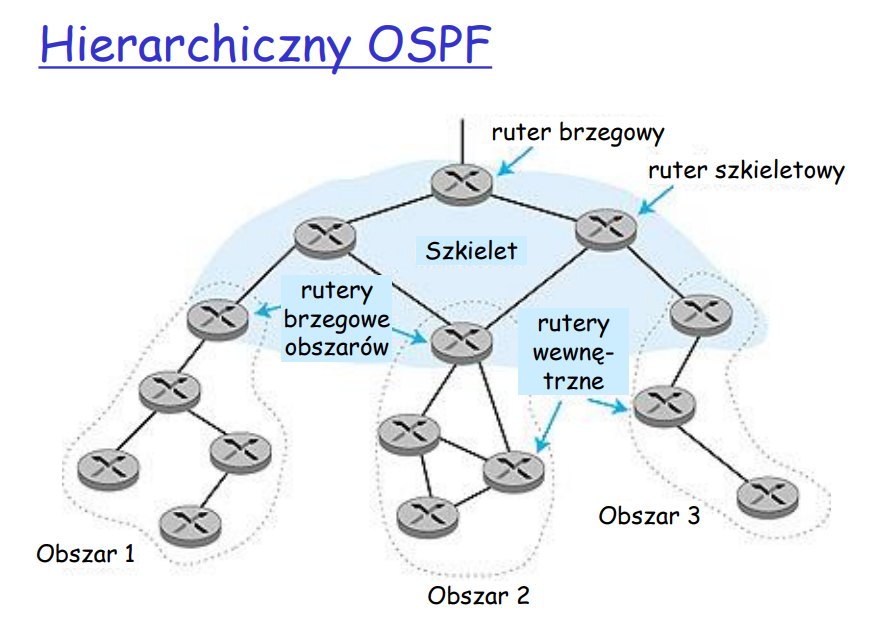
## 43. Protokoły rutingu warstwy sieci na przykładzie protokołu OSPF.

AS – Systemy autonomiczne:

* Elementy globalnego Internetu
* AS z jednym połączeniem – mała organizacja, łącząca się tylko z jednym innym AS
* AS z wieloma połączeniami – duża organizacja, wiele połączeń
* AS tranzytowy – DI poziomu 1 lub 2 łączący wiele systemów autonomicznych

OSPF – Open Shortest Path First:

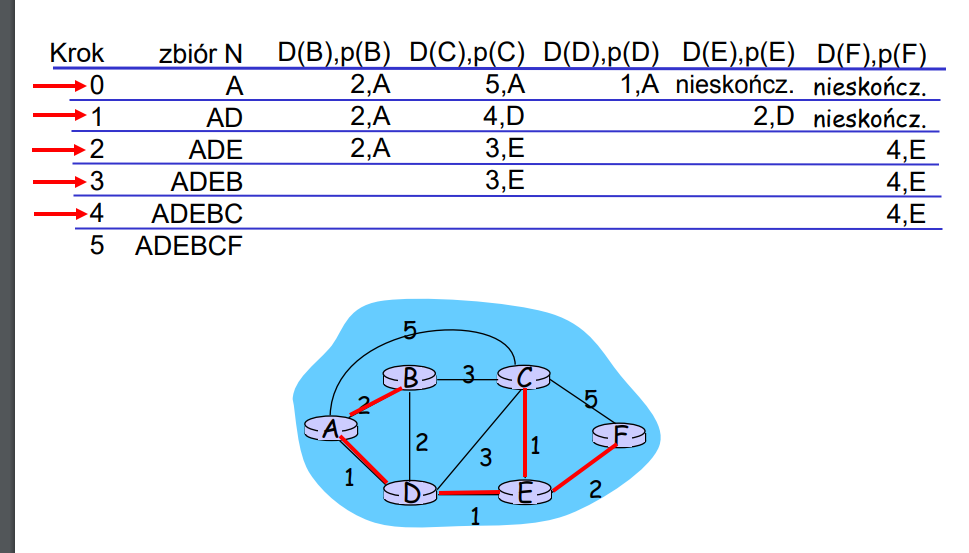
* Open – otwarty: dostępny dla wszystkich za darmo
* Używa algorytmu stanu łącza:
  + Rozsyła pakiety (ogłoszenia) SŁ
  + Mapa topologii w każdym węźle sieci
  + Obliczanie ścieżek przy użyciu algorytmu Dijkstry
* Ogłoszenie OSPF ma jeden wpis dla każdego sąsiadującego rutera
* Ogłoszenia są rozsyłane do całego AS (przez zalew) – komunikaty są rozsyłane bezpośrednio przez IP (a nie TCP/UDP)
* **Ochrona informacji** – każdy komunikat OSPF jest uwierzyteniany
* **Multipath** – przy ustalaniu ścieżki w grafie, może istnieć wiele ścieżek o tym samym koszcie
* Dla każdego łącza możemy mieć **wiele miar kosztu ścieżki** (w zależności od rodzaju usługi)
* Zintegrowany ruting unicast i multicast
* W dużych sieciach stosowany jest **hierarchiczny OSPF**



* **Dwupoziomowa hierarchia** – obszar lokalny i szkielet (stan jest ogłaszany tylko w obszarze lokalnym, każdy węzeł zna tylko kierunek do sieci w innych obszarach, a nie ich całą topologię)
* **Rutery brzegowe obszarów** – „podsumowują” odległości do sieci w swoim obszarze i ogłaszają te informacje innym ruterom brzegowym obszarów
* **Rutery szkieletowe** – realizują ruting OSPF w sieci szkieletowej
* **Rutery brzegowe** – łączą się z innymi AS

Jak działa algorytm Dijkstry:

* N – zbiór węzłów, do których znamy najszybszą ścieżkę
* D(v) – aktualnie znana najkrótsza ścieżka od źródła do węzła „v”
* p(v) – węzeł poprzedzający węzeł „v” na ścieżce od źródła do „v”
* Najpierw patrzymy na źródło – ustalamy ścieżki jego do jego sąsiadów (jak węzeł nie jest sąsiadem, to uznajemy, że ścieżka to nieskończoność)
* Wybieramy najszybszą ścieżkę i dodajemy węzeł do zbioru N
* Na podstawie nowego węzła sprawdzamy, czy pojawiła się nowa najkrótsza ścieżka do jakiegoś węzła



Złożoność alg. Dijkstry: **O(n2)**(istnieją implementacje zmniejszające to do **O(nlogn)**

Inne protokoły warstwy sieci:

* RIP – Routing Information Protocol
* BGP – Border Gateway Protocol

## 44. Usługi warstwy łącza na przykładzie protokołu Ethernet lub protokołów z rodziny 802.11 (WiFi).

Warstwa łącza – warstwa odpowiedzialna za komunikację **ramek** pomiędzy sąsiednimi węzłami przez łącze  
Węzły = hosty, rutery, mosty, switche

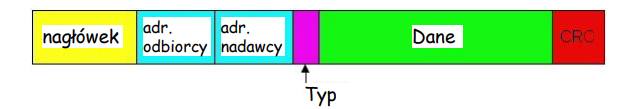
Usługi warstwy łącza:

* Tworzenie ramek, dostęp do łącza:
  + Enkapsuluje pakiet w ramce, dodaje nagłówki i zakończenie
  + Uzyskuje dostęp do łącza, jeśli jest współdzielone
  + **Adresy fizyczne** (co innego niż adresy IP) – używane w nagłówkach ramek do identyfikacji nadawcy i odbiorcy
* Niezawodność komunikacji między sąsiednimi węzłami
* Kontrola przepływu – dopasowanie prędkości nadawania i odbierania przez dwa sąsiednie węzły
* Rozpoznawanie błędów – powodowanych przez tłumienie lub zakłócenia sygnału. Odbiorca rozpoznaje błąd, sygnalizuje nadawcy konieczność retransmisji i wyrzuca błędną ramkę
* Korekcja błędów przez **kody nadmiarowe** – pozwalają na naprawę ramki bez potrzeby retransmisji
* Komunikacja półdupleksowa (obie strony mogą nadawać, ale nie jednocześnie) i w pełni dupleksowa

Ethernet:

* Najpopularniejsza technologia LAN
* Pierwsza powszechnie używana technologia LAN

Pakiety IP (lub pakiety innej warstwy sieci) są enkapsulowane w **ramce Ethernet**:

****

* Nagłówek:
  + 7 bajtów ze wzorem 10101010, a potem jeden bajt z wzorem 10101011
  + Używane do synchronizacji zegarów nadawcy i odbiorcy
* Adresy – po 6 bajtów
* Typ – wskazuje na protokół warstwy wyżej (zwykle IP)
* CRC – sprawdzane u odbiorcy, jeśli jest błąd, to ramka jest wyrzucana

**Ethernet to usługa:**

* Bezpołączeniowa – nie ma sygnalizacji pomiędzy nadającym i odbierającym adapterem
* Zawodna – odbierający adapter nie wysyła ACK ani NAK do nadającego adaptera (ciąg przekazywanych pakietów może mieć luki. Są one wypełniane, jeśli aplikacja używa TCP)

Ethernet używa **CMSA/CD** (Carrier Sense Multiple Access with Collision Detection):

* Adapter nie transmituje jeśli słyszy transmisję innego adaptera, czyli nasłuchiwanie (carrier sense)
* Transmitujący adapter przerywa gdy zauważy, że inny adapter transmituje, czyli wykrywanie kolizji (collision detection)
* Zanim adapter rozpocznie retransmisję, czeka przez losowy okres czasu

Algorytm CMSA/CD w Ethernecie:

1. Adapter otrzymuje pakiet i tworzy ramkę
2. Jeśli adapter nie słyszy transmisji w kanale, zaczyna transmitować ramkę. Jeśli słyszy transmisję, czeka aż kanał zostanie zwolniony i potem transmituje
3. Jeśli adapter wyśle całą ramkę bez wykrycia transmisji, to koniec
4. Jeśli podczas transmisji adapter wykryje inną transmisję, przerywa i **wysyła sygnał zakłócający** (powiadamia nadających o kolizji)
5. Po przerwaniu, adapter rozpoczyna **wykładnicze cofanie** – po m-tej kolizji adapter wybiera losowo K (od 0 do 2^m-1). Adapter czeka K\*521 i wraca do kroku 2

802.11 – protokół do bezprzewodowej komunikacji w sieci LAN:

* Bezprzewodowy host komunikuje się z **punktem dostępowym** (stacja bazowa = access point AP)
* **Basic Service Set (BSS)** – punkt dostępowy + bezprzewodowe hosty w jego zasięgu
* BSS mogą być łączone, żeby stworzyć **system dystrybucji**
* **Sieci Ad Hoc** – hosty porozumiewają się ze sobą bez użycia AP (Pakiet od hosta A do B może być kierowany przez hosty X,Y,Z) – używane do spotkań laptopów w pokoju konferencyjnym i łączeniu urządzeń osobistych

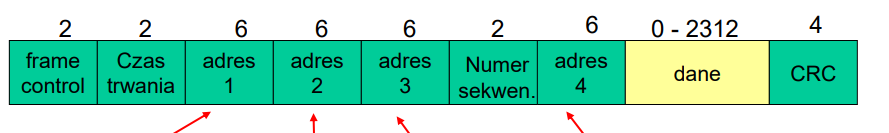
Charakterystyka łącza bezprzewodowego:

* **Słabsza moc sygnału** – sygnał radiowy ulega tłumieniu przy przechodzeniu przez materię
* **Zakłócenia przez inne źródła** – standardowe częstotliwości (np. 2.4 GHz) są współdzielone przez różne urządzenia (np. telefony). Mogą być też zakłócane przez np. silniki
* **Propagacja wielościeżkowa** – sygnał radiowy odbija się od przeszkód i gruntu, dochodząc do celu w różnym czasie, co utrudnia komunikację

Unikanie kolizji – wymiana RTS-CTS:

* RTS – request to send – krótka ramka wysyłana przez nadawcę, która zawiera długość planowanej transmisji
* CTS – clear to send – odpowiedź odbiorcy na ramkę RTS (zawiadamia też „ukryte” węzły o chęci komunikacji, żeby nie doszło do zakłóceń)
* Przez ustalony czas, ukryte węzły nie będą transmitowały

Ramka 802.11



* Adres 1 – adres MAC bezprzewodowego hosta (odbiorcy) lub punktu dostępowego
* Adres 2 – MAC nadawcy lub punktu dostępowego
* Adres 3 – Mac interfejsu rutera, do którego dołączony jest punkt dostępowy
* Adres 4 – Używany tylko w trybie ad-hoc

## 45. Metody ochrony informacji stosowane w bankowości Internetowej.

Elementy ochrony informacji:

* Poufność – tylko nadawca i zamierzony odbiorca powinni „rozumieć” zawartość wiadomości (szyfry)
* Uwierzytelnienie – nadawca i odbiorca chcą wzajemnie potwierdzić swoją tożsamość
* Integralność – nadawca i odbiorca chronią się przed niepostrzeżonym modyfikowaniem wiadomości
* Dostępność – usługi muszą być dostępnie dla użytkowników

Co może zrobić intruz:

* Podsłuchiwać – przechwycić wiadomość
* Dodawać wiadomości do komunikacji
* Podszywać się – **spoofing** (fałszowanie adresu nadawcy w pakiecie danych), **man-in-the-middle** (atak, w którym atakujący pełni rolę pośrednika między obydwiema stronami komunikacji)
* Przechwytywać – przejmować połączenie poprzez zastąpienie jednej ze stron komunikacji
* Zablokować usługę – DoS (Denial of Service)

Wiele ataków skupia się na **wykorzystaniu słabości człowieka** (karteczki z hasłami, „pożyczanie” konta itp.)

**Kryptografia** – nauka o tworzeniu szyfrów  
**Kryptoanaliza** – nauka o łamaniu szyfrów

2 typy kryptografii:

* Z kluczem symetrycznym – nadawca i odbiorca mają ten sam klucz (np. **szyfr zastępujący** – kluczem jest alfabet, z którego zamieniamy – prostsza wersja: szyfr cezara [ten z przesuwaniem, wtedy kluczem jest przesunięcie])
* Z kluczem publicznym – każdy ma 2 klucze – publiczny i prywatny (tajny)

**DES** – data encryption standard (algorytm polegający na użyciu 3 56 bitowych kluczy po kolei na łączonych blokach szyfru o rozmiarze 64 bitów)

**AES** – advanced encryption standard (zastępuje DES, większe bloki i klucze. Brute force trwający 1 sekundę dla DES trwa 149 bilionów dla AES)

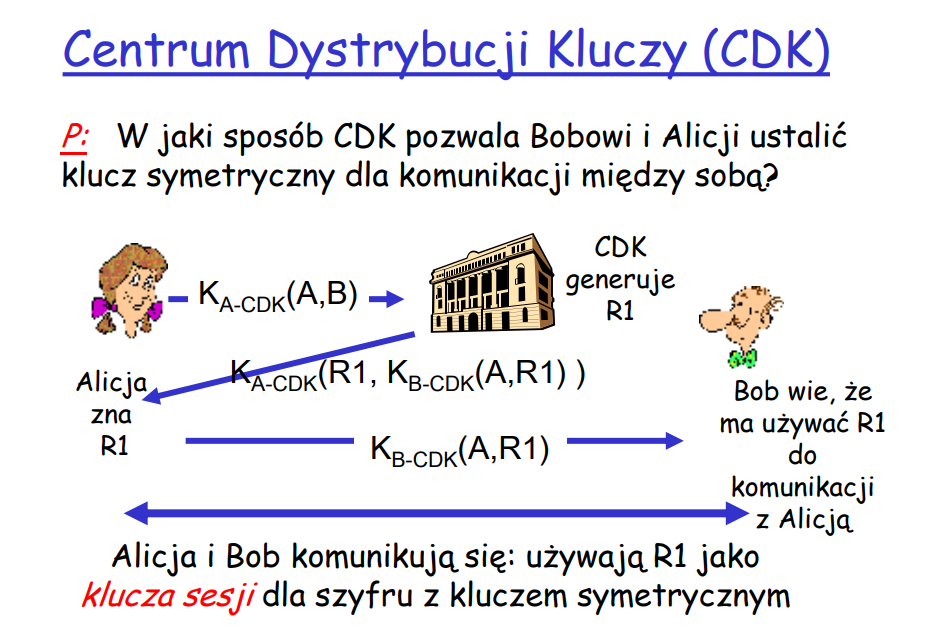
**RSA** – algorytm Rivesta-Shamira-Adlemana – służy do **wyboru kluczy publicznych i prywatnych**:

1. Wybierz 2 duże liczby pierwsze: *p,q*
2. Oblicz *n = pq, z=(p-1)(q-1)*
3. Wybierz *e* – *e<n* i nie ma wspólnych dzielników (oprócz 1) z liczbą *z*
4. Wybierz *d* – e\*d-1 jest podzielne przez z
5. Klucz publiczny to (n,e), klucz prywatny to (n,d)
6. Żeby zaszyfrować ciąg bitów *m – c=m^e mod n* (mod to reszta z dzielenia)
7. Żeby odszyfrować ciąg bitów *c* – m=c^d mod n

**Faktoryzacja** – problem poznania wszystkich liczb pierwszych, których iloczyn jest równy danej liczbie (to np-trudne, dlatego RSA jest skuteczne)

**Podpisy cyfrowe** – technika kryptograficzna analogiczna do podpisów odręcznych. Są wiadomością zaszyfrowaną kluczem prywatnym danej osoby (uwierzytelnienie – odszyfrowanie wiadomości kluczem publicznym)

Długie wiadomości są kosztowne w szyfrowaniu – używa się **funkcji haszującej** do stworzenia skrótu. Podpisem cyfrowym jest podpisany *skrót* wiadomości  
MD5, SHA-1 – najpopularniejsze funkcje haszujące

**CDK** – centrum dystrybucji kluczy – zaufany pośrednik, który pomaga ustalić klucz prywatny dla danej sieci  


**CC** – centrum certyfikatów – do potwierdzania, że otrzymany klucz publiczny należy do danej osoby

Certyfikat zawiera:

* Numer seryjny – niepowtarzalny u nadawcy
* Informacje o właścicielu certyfikatu, algorytmach szyfrowania i skrótu
* Informacje o CC
* Datę ważności
* Podpis cyfrowy CC

**Ściana ogniowa** – izoluje wewnętrzną sieć organizacji od Internetu, pozwalając na niektóre rodzaje komunikacji, a blokując inne  
**Strefa zdemilitaryzowana** – część sieci pomiędzy wewnętrzną siecią a publicznym Internetem, chroniona ścianą ogniową, w której mogą znajdować się serwery proxy

Zalety ściany ogniowej:

* Chroni przed atakami DoS – np. przed zalewem SYN (otwieraniem wielu połączeń TCP)
* Zapobiegają nielegalnym modyfikacjom/dostępowi do danych (np. zastępowanie strony banku własną, fałszywą stroną)
* Chroni przed nieuprawnionym dostępem do sieci z zewnątrz
* Ma 2 typy – w warstwie aplikacji i w warstwie sieci (filtry pakietów – np. względem IP, numeru portu TCP/UDP źródła/celu itp.)

**PGP** (pretty good privacy) – standard szyfrowania poczty elektronicznej  
**SSL** (secure socket layer) – ochrona w warstwie transportu dla TCP, używane w handlu elektronicznym (https) – symetryczny klucz sesji + prywatny klucz dla serwera do odszyfrowywania wiadomości  
**IPsec** – poufność w warstwie sieci (dane szyfrowane w pakieci IP)  
**AH** – authentication header – do uwierzytelniania źródła, integralności ale nie poufności (brak szyfrowania)

Podsumowanie – w banku stosuje się:

* Szyfrowanie transmisji danych
* Proste uwierzytelnianie – np. login i hasło
* Silne uwierzytelnianie – np. token
* Podpis elektroniczny

## 46. Modele barw.

Elementy percepcji barwy:

* Odcień barwy – kolor
* Nasycenie – czy intensywne, czy wygaszone (dodawanie białej farby zmniejsza nasycenie)
* Jasność – (dodawanie czarne farby zmniejsza jasność)

Teoria postrzegania barw Younga-Helmholtza – Wrażenia barwne odbierane są przez 3 rodzaje niezależnych receptorów oka, reagujących na inną długość fali.

Model barw – określony, trójwymiarowy system współrzędnych barw wraz z widzialnym podzbiorem, w którym leżą wszystkie barwy z określonej gamy barw

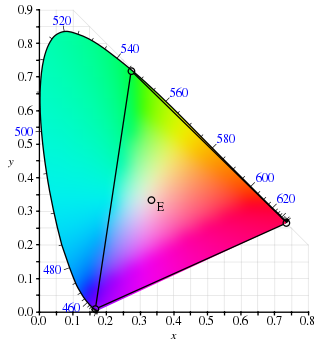
Typy modeli barw:

* Percepcyjne – HSV, HSB, HLS (h – hue (odcień) , s – saturation (nasycenie), v – value, b -brightness, l – lightness (różne rodzaje zapisu jasności, dla value = biały to 1 [max], dla brightness/lightness biały = -1 )
* Addytywne – RGB (wszystkie kolory zsumowane dają biały), RGB jest też dyskretny
* Subtraktywne – CMY (kolory zsumowane dają czarny, CMYK – dodatkowo czarny tusz, żeby nie marnować kolorowych)

Przeliczanie RGB<->CMY:

Do CMYK – K = min(R,G,B), potem odejmujemy K od R, G i B, na koniec liczymy jak powyżej.

CIE XYZ – CIE to „Międzynarodowa Komisja Oświetleniowa”, układ XYZ ma reprezentować wszystkie barwy widzialne. Y mówi o strumieniu światła; X,Z tylko o barwie. Korzysta z fikcyjnych barw podstawowych, reprezentowanych przez XYZ



Ten trójkąt to Gama kolorów (color gamut) – określa, które kolory można wyświetlić na danym urządzeniu (np. jakim dokładnie odcieniem czerwonego jest R w RGB dla jakiegoś monitora)

Inne modele barw:

* YIQ, YUV – modele telewizyjne
* CIE L\*u\*v (dla urządzeń emitujących); CIE L\*a\*b (dla określenia koloru obiektu) – modele percepcyjnie równomierne

sRGB – standarised RGB – Hewlett-Pachard I Microsoft 1996 - standard dla wszystkich urządzeń (monitorów, skanerów itp.)

W CIE LAB:

* L – jasność (luminancja)
* A – barwy na spektrum zielona-czerwona (Magenty)
* B – barwy na spektrum niebieska-żółta

ICC – profil urządzenia, określa jak przeliczać barwy z modelu odniesienia (np. LAB) do modelu urządzenia

Systemy barw – katalogi jak np. w sklepach z farbą:

* Munsell’a – śmieszna, obła bryła, pokazuje na jakie kolory jesteśmy bardziej czuli
* PANTONE

## 47. Techniki cieniowania (shadery).

Model oświetlenia Phonga:

* k\_a, k\_d, k\_s – stałe w przedziale <0,1> określające własności obiektu
* I\_a – światło otoczenia
* I\_p – światło bezpośrednio od źródła światła
* I\_s – światło wynikające z odbicia zwierciadlanego od powierzchni
* α – kąt między rzeczywistym kierunkiem odbicia a kierunkiem obserwacji

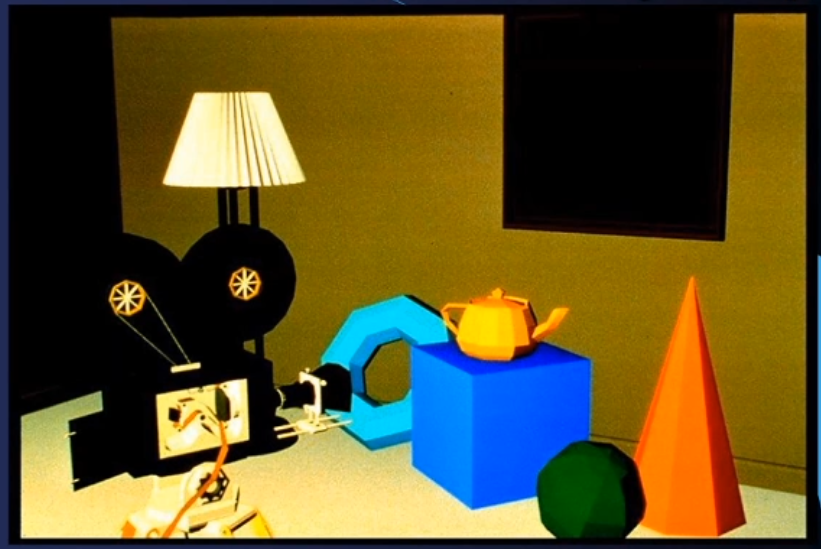
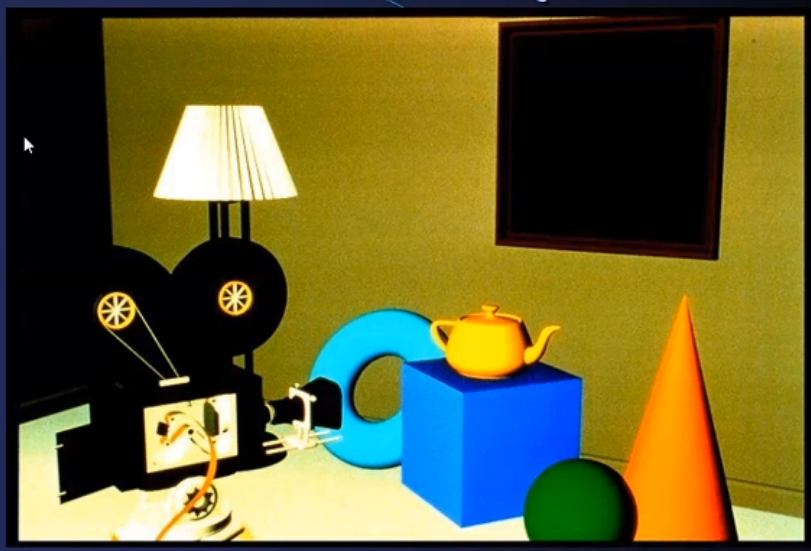
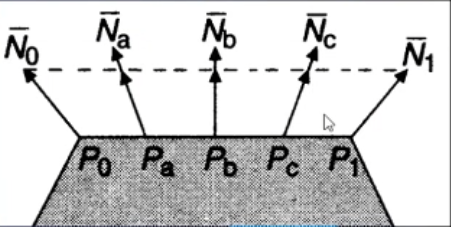
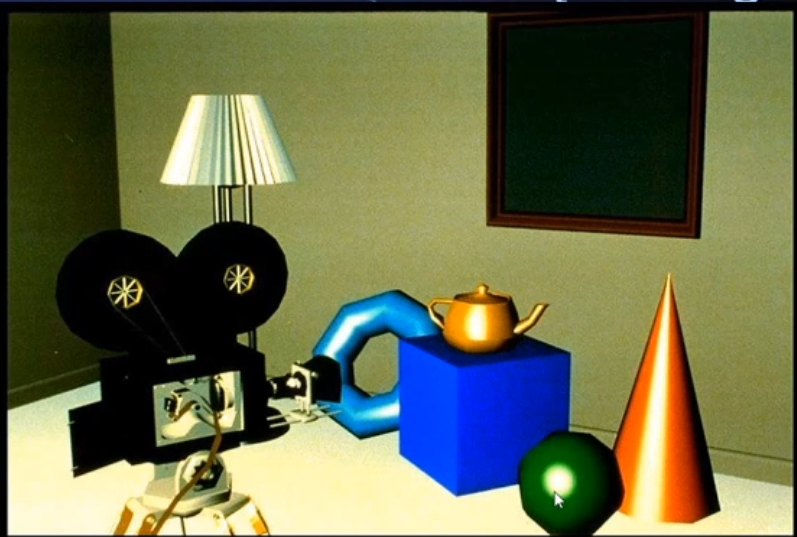
Cieniowanie to NIE model oświetlenia – cieniowanie ma nam mówić, jaki kolor ma dana powierzchnia w danym punkcie przy danym oświetleniu

Cieniowanie siatek wielokątowych:

* Cieniowanie wartością stałą
* Metody z interpolacją: Cieniowanie Gourauda, Cieniowanie Phonga

Bez cieniowania – obiekty mają jednolity kolor, wydają się płaskie

Bardziej szczegółowo:

1. Cieniowanie wartością stałą (cieniowanie płaskie)
   1. Ten sam kolor przypisywany całej ścianie, kolor obiektu zależy od jego umiejscowienia i oświetlenia – określane używając wektorów normalnych ścian (prostopadłych do ściany)  
       
   2. Dobre do obiektów kanciastych, ale nieskuteczne do obiektów gładkich
   3. Efekt Macha – nagła zmiana koloru na krawędzi między ścianami obiektu
2. Metoda Gourauda (metoda polegająca na interpolacji barwy)
   1. Cieniowanie z interpolowaniem jasności (i barwy):
      1. Obliczanie barwy pierwotnej ściany
      2. Obliczanie barwy wierzchołków (poprzez uśrednienie barw pierwotnych ścian, do których wierzchołek należy – tak samo liczony jest wektor normalny w metodzie Phonga)
      3. **Interpolacja liniowa** **barwy** wzdłuż krawędzi
      4. Obliczenie barwy piksela w każdej (poziomej) linii wielokąta jako kombinacja liniowa barwy początku i końca
   2. Wymaga liczenia 3 barw dla każdej ściany (dla siatek wielokątowych, gdzie używa się trójkątów)
   3. Skuteczne dla gładkich obiektów matowych, ale wygładza też obiekty, których wygładzać możemy nie chcieć  
      
3. Model Phonga (cieniowanie z interpolacją wektora normalnego)
   1. cieniowanie z **interpolacją wektora normalnego**:
      1. Wyznaczenie normalnych w wierzchołkach wielokąta
      2. Interpolacja normalnych wzdłuż krawędzi wielokąta
      3. Wyznaczenie normalnej początku i końca segmentu i interpolacja normalnej w poziomych segmentach
      4.  - efekt tego jest taki, że wektory normalne układają się jak na powierzchni wypukłej (np. półkuli)
      5. Mamy efekt odbłysków, ponieważ wektory normalne są skierowane wprost na obserwatora

## 48. Metody kompresji w standardzie MPEG.

MPEG – Moving Picture Experts Group

Polega na wykorzystaniu **częściowego zapisu kolejnych klatek** filmu i **szacowaniu ich wyglądu** na podstawie wybrany klatek

* MPEG1 – 1992 r. – optymalizowany dla prędkości 1410 kb/s do przesyłania **pojedynczych programów** w strumieniu (VideoCD)
* MPEG2 – rozszerzenie, do transmisji **wielu programów** w jednym strumieniu (satelitarna i kablowa TV, HDTV, SuperVideoCD, DVD)
* MPEG4 – zoptymalizowany dla przepustowości 880 kb/s (wideo konferencje, DivX)

Zalety MPEG:

* Skalowalność – do internetu (kb/s) i wysoka jakość (Mb/s)
* Możliwość wyboru rozdzielczości
* Niezależne kodowanie statycznych i ruchomych obrazów
* Rozszerzona korekcja błędów

MPEG wykorzystuje:

* Kompensację ruchu pod kątem nadmiarowości międzyramkowej (nadmiarowość czasowa)
* Kompresję DCT do nadmiarowości wewnątrzramkowej (nadmiarowość przestrzenna)

Przebieg algorytmu:

Sekwencja obrazów -> [KODER KOMPENSACJI RUCHU] -> ramki ze skompresowanym ruchem -> [KODER NA BAZIE DCT] -> współczynnik DCT -> [KODER ENTROPII] -> skompresowana sekwencja obrazów -> [DEKODER ENTROPII] -> współczynnik DCT -> [DEKODER NA BAZIE DCT] -> ramki ze skompresowanym ruchem -> [KODER KOMPENSACJI RUCHU] -> odwrotna sekwencja obrazów

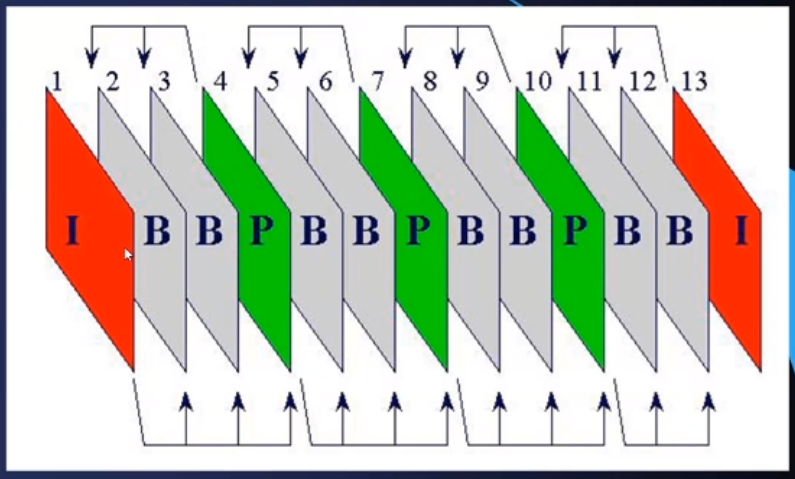
**DCT** – dyskretna transformata cosinusowa (discrete cosine transformation) – rozkłada funkcję na serię kosinusów o różnej częstotliwości

Mamy też VLC (Variable length coding) – usuwa z kompresowanego strumienia powtarzające się ciągi danych i zastępuje je krótszymi

W Kompresji MPEG mamy **3 rodzaje ramek** (klatek) – nazywane GOP (group of pictures):

* Ramki odniesienia I (inter) – niezależne od pozostałych ramek
* Ramki P (predictive) – reprezentują różnicę między bieżącą ramką a ramką odniesienia („przewidywane obrazy”, mniej informacji niż w ramkach B)
* Ramki B (bidirectional) – mają dwie ramki odniesienia – wcześniejsza i późniejsza (koduje różnice między dwoma sąsiednimi klatkami)

Ramki P i B zawierają informacje o ruchu fragmentów ramki odniesienia (fragmenty są w rozdzielczości 16x16 pikseli) – wyliczany jest wektor ruchu dla tych fragmentów, na jego podstawie konstruowana jest klatka P, która jest kodowania JPEG-iem  
Zazwyczaj nie ma dużych różnic pomiędzy następnymi klatkami nagrania, dlatego w MPEG kodujemy tylko zmiany między ustalonymi ramkami odniesienia – jak będą zakłócenia, to przeskakujemy do kolejnej ramki odniesienia gdy łącze wróci do normy.



Najpierw ustalamy I, potem P, na koniec B

MPEG korzysta z JPEG (Joint Photographic Experts Group)

* Używa DCT I procesu kwantyzacji obrazów
* Korzysta z faktu, że szybkie przejścia barw przy krawędziach są niedostrzegalne
* **Jest stratna**. Powoduje artefakty: „aureolki” – pojawiają się wzdłuż krawędzi; „artefakty blokowe” – przy dużej kompresji pojawiają się bloki pikseli o jednakowym kolorze

Schemat kodera JPEG:

[Przejście z RGB->YCbCr – Y: jasność; Cb,Cr: kolor] -> [DCT] -> [Kwantowanie – dzielimy współczynniki transformaty na bloki, wiele wyższych częstotliwości jest zerowana] -> [Kodowanie]

Dekoder – to samo, tylko na odwrót

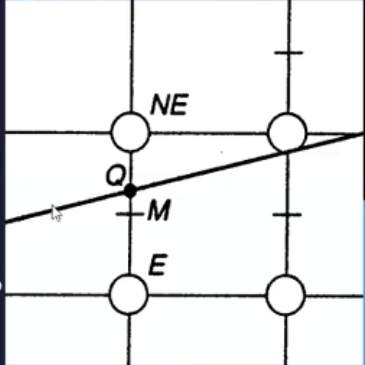
M-JPEG – rozszerzenie JPEG do obrazów ruchomych, uproszczony MPEG: wszystkie klatki są typu I (brak kompensacji ruchu), niższy czas kompresji, ponieważ nie musimy wyliczać klatek pośrednich

## 49. Efekt aliasingu i metody jego zwalczania. Aliasing a częstotliwość  próbkowania.

Aliasing – zakłócenia powstałe w procesie rysowania w przestrzeni dyskretnej (o skończonej rozdzielczości) – jest związany ze skończoną częstotliwością próbkowania (sygnał próbkujemy co jakiś czas, co powoduje powstanie artefaktów)

Przykład aliasingu – rysowanie linii ukośnej na ekranie powoduje pojawienie się „schodków”

Antyaliasing (metody odkłócania):

* Zwiększenie rozdzielczości - nie usuwa aliasingu, ale z pewnej odległości nie będzie on widoczny, dodatkowo ma wady:
  + Zwiększenie rozmiaru pamięci obrazu
  + Zwiększenie czasu rysowania prymitywów
  + Zwiększenie pasma pamięci i pasma monitora
* Metoda z dwoma pikselami w kolumnie:
  + Wiąże się z **algorytmem Bresenhama** – do ustalenia, który piksel podświetlić, gdy linia przechodzi między dwoma pikselami (to jest rozszerzenie tego algorytmu, bo tam wybierało się tylko jeden piksel do kolorowania, a tu koloruje się oba proporcjonalnie)
  + Ustalamy odcień NE i E w oparciu o długości odcinków [NE,Q] oraz [Q,E] – suma odcieni jest stała  
    
  + Metoda ta jest skuteczna wyłącznie, gdy urządzenie wyświetlające dysponuje wieloma odcieniami szarości
* Metoda bezwagowego próbkowania powierzchni:
  + Bardzie dokładna, skuteczna w przypadku linii o grubości >< 1 px (mniejszej i większej)
  + Rysujemy odcinek o określonej szerokości na siatce kwadratowych pikseli
  + Kolor jest określany na podstawie tego, jak duży % powierzchni piksela znajduje się w obrębie odcinka
  + Wymagana jest wielobitowa reprezentacja piksela
  + Podczas rysowania ustawianych jest kilka pikseli
  + 
  + Żeby nie musieć liczyć pola powierzchni, wykonuje się **nadpróbkowanie:** dzielimy każdy piksel na kilka mniejszych (np. 16), patrzymy na które mini-piksele nachodzi linia i % objętych mini-pikseli określa odcień całego piksela
* Metoda wagowego próbkowania powierzchni:
  + Oprócz obejmującego pola powierzchni, wpływ na odcień ma odległość piksela od środka odcinka (wtedy to samo pole może skutkować różną barwą)
  + „okrągłe piksele” – pełne pokrycie (koło opisane na kwadracie – ma większą powierzchnię); brak zależności kierunkowych
  + Wpływ odległości od środka odcinka określa **funkcja wagowa**
  + Efekt jest taki, jak rysowanie stożkowym pisakiem

## 50. Zasady interakcji człowiek-komputer: przedstaw i omów heurystyki Nielsena-Molicha.

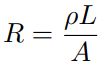
* To zasady interakcji człowiek-maszyna
* Przedstawione w 1990r.
* Są niezależne od zastosowanych rozwiązań technicznych
* Są wynikiem badań statystycznych kształtujących opinię użytkowników w zakresie wygody i satysfakcji z interakcji z oprogramowaniem

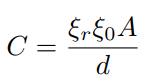
Zasady to:

1. Pokazuj status systemu (paski postępu) – „visibility of system status”
   1. „Always keep user informed”
   2. Przykład – mapy ze znakiem „**you are here**” np. w centrach handlowych
   3. Checkout flow – wieloetapowe procesy np. zakupu – mamy **breadcrumbs menu**, które mówi nam o tym, na jakim kroku jesteśmy i pozwalają na powrót
   4. Phone tap – należy **reagować na interakcję**, chociażby poprzez wibrację telefonu. Pokazujemy w ten sposób, że coś osiągamy, nie tracimy ciągłości interakcji
2. Zachowaj zgodność pomiędzy systemem a rzeczywistością (np. pulpit = biurko, usuwanie plików = opróżnianie kosza) – „match between system and the real world”
   1. Stovetop controls – **kontrolki** na kuchence są rozłożone identycznie **jak palniki**
   2. Car vs. Automobile – trzymamy się **najprostszych słów**, jak użytkownik myśli „car”, to powinniśmy użyć słowa „car”
   3. Shopping cart icon – **ikony** powinny **odpowiadać rzeczywistym obiektom** (jak np. wózek/koszyk sklepowy jako ikona w sklepie internetowym)
3. Daj użytkownikowi pełną kontrolę - „user control & freedom”
   1. Exit sign – przestrzeń cyfrowa potrzebuje **wyjść awaryjnych** tak samo, jak przestrzeń rzeczywista
   2. Undo and redo – możliwość **cofnięcia/ponowienia zmian** daje użytkownikowi kontrolę, ponieważ jego akcje da się łatwo odwrócić
   3. Cancel button – użytkownik powinien **móc porzucić proces**
4. Trzymaj się standardów i zachowaj spójność (np. wstążka jest ta sama we wszystkich aplikacjach ms office) – „Consistency & standards”
   1. „check-in counter” – recepcja w hotelu **wygląda tak samo** wszędzie na świecie
   2. Design system – należy wykorzystywać **wielokrotnie ten sam design elementów**, ponieważ ułatwia to naukę użytkowania (np. używać tych samych kolorów do przycisków)
   3. Notifications – powiadomienia są standaryzowane, a mimo to da się rozróżnić powiadomienia różnych aplikacji (ważna jest spójność wewnętrzna, ale też z **elementami zewnętrznymi** – standardowymi)
5. Zapobiegaj błędom (np. lista z haczykami przy tworzeniu hasła - czy wystarczająco długie, czy są znaki specjalne itp. ; powiadomienie w przeglądarce jak zapomnimy dać załącznik do maila) – „Error prevention”
   1. „Guard rails” – **barierki chroniące w górach**, żeby kierowca nie wypadł
   2. Airline confirmation – **ekran potwierdzenia**, żeby mieć możliwość upewnić się, że wszystko jest ok
   3. Date selection on calendar – dać możliwy zakres dat, **wyszarzyć opcje niedostępne**
6. Pozwalaj wybierać zamiast zmuszać do pamiętania – „Recognition rather than recall”
   1. Lisbon – ludzie prędzej zgadną „Czy Lizbona to stolica Portugalii” niż „Jak nazywa się stolica Portugalii”
   2. Comparison table – **porównujemy rzeczy w tabelach** tak, by użytkownik nie musiał pamiętać obu obiektów przy podejmowaniu decyzji
   3. Search – **wyniki wyszukiwania są podawane razem z zapytaniem**, żeby wiedzieć, dlaczego takie są
7. Zapewnić elastyczność i efektywność (np. dodatkowe search tools w google images dla zaawansowanych użytkowników) – „Flexibility and Efficiency of use”
   1. Shortcuts – pokazujemy **oficjalną drogę, ale zostawiamy skróty** dla bardziej zaawansowanych
   2. Keyboard shortcuts – **skróty klawiszowe** pozwalają na bardziej efektywną pracę
   3. Tap to like – **wiele sposobów** na wykonanie tej samej operacji (np. kliknięcia, gesty…)
8. Dbaj o estetykę i umiar – „Aesthetic and minimalist design”
   1. Ornate vs. Simple teapot – **nadmierne ozdoby szkodzą** użyteczności
   2. Communicate don’t decorate – ozdoby **mogą przysłonić informację**, którą chcemy przekazać
   3. Messy vs. Organized UI – **bałagan** w UI **wydłuża czas interakcji**
9. Zapewnij skuteczną obsługę błędów – „Help users recognize, diagnose and recover from errors”
   1. Wrong way sign – **dawać znaki**, gdy użytkownik robi coś niepoprawnie
   2. Internet connection error – dobra strona z błędem o połączeniu powinna **pokazać co się stało i poinstruować o sposobie naprawy tego problemu**
   3. No search results – oferuj pomoc w wypadku błędu, np. jak mamy brak wyników na zapytanie, to **pokazać jak można to zapytanie poprawić**
10. Zadbaj o pomoc i dokumentację – „Help and documentation”
    1. Airport information center – łatwo rozpoznawalne miejsce z pomocą, rozwiązuje problemy w kontekście i natychmiast
    2. Frequently Asked Questions – dobra strona FAQ **przewiduje problemy** użytkowników i **oferuje pożyteczne informacje**
    3. Information icon – ikonka z tooltipem na wypadek gdyby użytkownik nie rozumiał specyficznego żargonu, pomoc kontekstualna

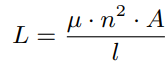
## 51. Implementacje podstawowych elementów pasywnych (rezystorów, kondensatorów i cewek).

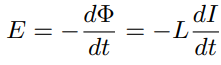
**Element pasywny** – element nie wymagający zasilania do działania w układzie elektrycznym. Dla tych elementów energia na wyjściu nie może być większa niż ta na wejściu

**Rezystor (opornik)** – Rezystancja to stosunek napięcia (V – wolty) na krańcach materiału do prądu (A – ampery) płynącego przez ten materiał. Mierzona w Ω – omach. Rezystancja zależy od rezystywności danego materiału i jego wymiarów.  
Wzór na rezystancję:  
 ρ – rezystywność, L – długość, A – pole powierzchni  
Rezystory są wykonywane z węgla (z dodatkami) i ze stopów chromu i niklu

**Kondensator** – dwie okładziny (przewodniki) odizolowane izolatorem (czymś, przez co prąd nie płynie, może być nawet powietrze)  
Ładunek elektryczny Q (mierzony w C – Kulombach) gromadzi się na okładzinach, co powoduje powstanie napięcia między okładzinami. Napięcie to jest odwrotnie proporcjonalne do pojemności kondensatora C (mierzonej w F – Faradach)  
  
Pojemność kondensatora można policzyć też wzorem:  
 ξ0 – stała *przenikalność elektryczna próżni* (w faradach na metr F/m), ξr – względna przenikalność elektryczna (zależy od izolatora), A – powierzchnia okładzin, d – odległość między okładzinami  
Kondensatory mogą być:

* Ceramiczne
* Foliowe (zwijane)
* Elektrolityczne
* Elektrolityczne tantalowe

**Cewka indukcyjna** – zwinięty przewodnik, który umożliwia skupienie energii pola magnetycznego, które powstaje w wyniku przepływu prądu przez cewkę. Właściwość cewki to **indukcyjność**  
Wzór na indukcyjność:  
 n – liczba zwoi, A – pole powierzchni przekroju, μ – przenikalność magnetyczna, l – długość cewki

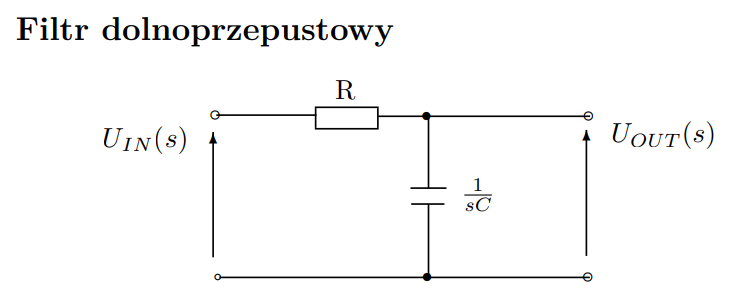
Zmiana pola magnetycznego (strumienia magnetycznego φ) powoduje powstanie (indukcję) napięcia E na końcach cewki. To napięcie to **siła elektromotoryczna SEM**  


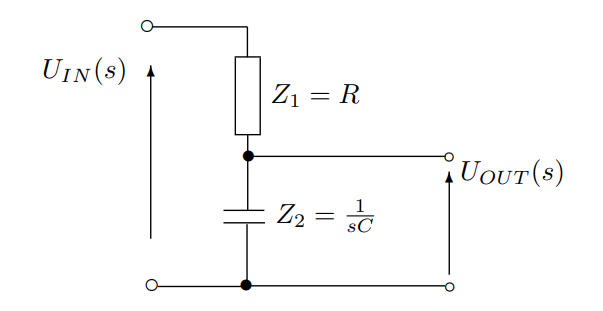
## 52. Filtr dolnoprzepustowy RC. Co to jest częstotliwość graniczna i pasmo przenoszenia filtru.

Transformata Laplace’a – służy do przejścia z dziedziny czasu (t) do dziedziny Laplace’a z liczbami zespolonymi (s)  
Różniczkowanie = mnożenie prze s  
Całkowanie = mnożenie przez 1/s

Transmitancja – funkcja określająca stosunek wyjścia systemów liniowych Y(s) do jego wejścia X(s)

**Częstotliwość graniczna** – częstotliwość, dla której amplituda maleje o ~ 0,707 w stosunku do amplitudy maksymalnej. W skali logarytmicznej to **-3dB**. Gdy osiągnięta zostaje częstotliwość graniczna, mocy sygnału spada o połowę



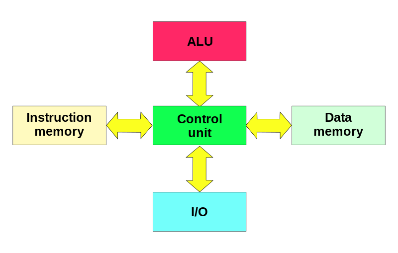
**Filtr dolnoprzepustowy** – filtr, którego 3dB pasmo przenoszenia ograniczone jest górną częstotliwością graniczną  
Przykładem filtra dolnoprzepustowego jest **dzielnik impedancyjny RC**  
RC – rezystor + kondensator  
  
Te dzielniki sprawiają, że napięcia na wejściu i wyjściu są proporcjonalne (proporcje są określone transmitancją). Dla RC, współczynnik proporcjonalności to: (RC=T, to jest tzw. **stała czasowa**)

**Pasmo przenoszenia** – zakres częstotliwości, w którym tłumienie sygnału nie jest większe niż 3dB (do częstotliwości granicznej) – ma znaczenie w nadawaniu i odbieraniu sygnałów analogowych (gramofonowych, liniowych, głośnikowych). Urządzenie jest w stanie przenosić sygnał jedynie w określonym zakresie częstotliwości  
Oprócz tego są filtry górnoprzepustowe (z dolną częstotliwością graniczną) i środkowoprzepustowe (częstotliwość górna + dolna)

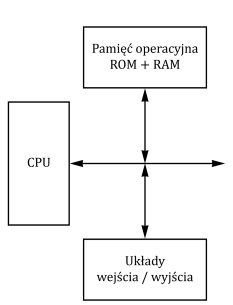
## 53. Architektura harwardzka a architektura von Neumana.

Architektura harwardzka:

* Pamięć danych programu jest **oddzielona** od pamięci rozkazów (u von Neumanna nie)
* **Prostsza** względem architektury von Neumanna, budowa ma większą szybkość działania – wykorzystuje się w procesorach sygnałowych oraz przy dostępie procesora do pamięci cache
* Powszechnie stosowana **w mikrokomputerach jednoukładowych** (program w pamięci ROM – read only memory, dane w RAM – random access memory)



Architektura von Neumanna – 1945



Elementy składowe:

* Procesor z ALU (**arithmentic logic unit – jednostka arytmetyczno-logiczna** – wykonuje obliczenia, operacje logiczne, przesunięcia bitowe itp.), oprócz ALU, w procesorze **jest Układ sterujący** – do dekodowania rozkazów i sterowania ich wykonaniem (pobieranie danych i instrukcji z pamięci i ich sekwencyjne przetwarzanie)
* Pamięć komputera (zawierająca dane i program) – każda komórka pamięci ma unikatowy adres
* Urządzenia wejścia wyjścia

Cechy komputera von Neumanna:

* Skończona lista rozkazów
* Możliwość wprowadzenia programu z zewnątrz i jego przechowywanie w pamięci (tak jak dane)
* Sekwencyjne odczytywanie instrukcji z pamięci i ich wykonywanie

## 54. Sposoby  obsługi zdarzeń w mikrokontrolerze.

Mikrokontroler – komputer zrealizowany w postaci **pojedynczego układu scalonego**, zawierającego:

* Jednostkę centralną (CPU)
* Pamięć RAM
* Pamięć programu
* Rozbudowane układy wejścia-wyjścia

Typowy mikrokontroler zawiera:

* Jednostkę obliczeniową (ALU) – przeważnie 8-bitową
* Pamięć danych (RAM)
* Pamięć programu
* Uniwersalne posty wejścia – część z tych portów może pełnić alternatywne funkcje, wybierane programowo
* Kontrolery transmisji szeregowej lub równoległej (UART, SPI, I2C, USB, CAN itp.)
* Przetworniki analogowo-cyfrowe lub cyfrowo-analogowe
* Timery
* Układ kontroli poprawnej pracy (watchdog)
* Wewnętrzne czujniki wielkości nieelektrycznych (np. temperatury)

Metody obsługi zdarzeń:

* Przerwanie (Interrupt) – zmiana sterowania, niezależnie od aktualnie wykonywanego programu, spowodowane pojawieniem się sygnału przerwania. Po pojawieniu się przerwania, aktualnie wykonywany program jest przerywany i wykonywana jest procedura obsługi przerwania
* Zapytywanie (Polling) – aktywne, okresowe, próbkowanie (sprawdzanie) statusu urządzeń zewnętrznych przez kontroler

Bardziej szczegółowo o zapytywaniach:

* Najczęściej używane w kontekście **obsługi urządzeń wejścia/wyjścia**
* W polling-u – komputer cyklicznie **sprawdza gotowość** urządzenia zewnętrznego
* Znajduje zastosowanie w sytuacjach, gdy komputer łączy się z zewnętrznymi urządzeniami w celu **zebrania (odświeżenia) danych,** współpraca ta odbywa się **w trybie off-line**
* Może być użyty do wymiany informacji, gdy z jakiś względów mikrokontroler i urządzenie zewnętrzne nie mogą rozpocząć komunikacji
* Dobre w systemach **obsługujących tylko jedno zadanie,** procesor nie traci dzięki nim czasu na sprawdzanie gotowości urządzenia
* Dla wielu zadań – mało efektywny w stosunku do przerwań

Rodzaje przerwań:

1. Sprzętowe:
   1. Zewnętrzne – do komunikacji z urządzeniami zewnętrznymi
   2. Wewnętrzne – pochodzące od timera
   3. Wewnętrzne wyjątki (exceptions) – zgłaszane przez procesor dla sygnalizowania sytuacji wyjątkowych (np. dzielenie przez zero)
2. Programowe – z kodu programu wywoływana jest procedura obsługi przerwania (do komunikacji z systemem operacyjnym)

Inny podział:

1. Zegarowe – odmierzanie czasu
2. Od urządzeń zewnętrznych – nieregularne
3. Od układów kontrolujących pracę systemu – NAJWYŻSZY PRIORYTET: zanik zasilania, błąd procesora itp.

Wektor przerwań – **adres początku obsługi przerwania**, w momencie wystąpienia przerwania jest wpisywany do licznika rozkazów – rejestr PC (Program counter). Zawartość rejestru PC jest kładziona na *stos*

Tablica wektorów przerwań – miejsce, w którym zapisane są adresy procedur obsługi przerwań

Przerwania maskowalne – można je blokować i odblokować programowo

Przerwania niemaskowalne – nie można ich zablokować programowo, ich wystąpienie zawsze skutkuje przeskoczeniem do funkcji obsługi przerwania – np. reset

Obsługa przerwania – jeżeli program główny zostanie przerwany, wykonywana jest procedura obsługi przerwania (ciąg rozkazów realizujący pożądaną reakcję na przerwanie) – obsługa przerwania NIE MOŻE WPROWADZAĆ ZMIAN W PROGRAMIE GŁÓWNYM

Kroki procedury obsługi przerwania:

1. Rozpoznanie przyczyny przerwania (może być realizowana sprzętowo)
2. Skasowanie przyczyny przerwania (też może być sprzętowo)
3. Zablokowanie przerwania
4. Składowanie na stosie rejestrów roboczych
5. Właściwa obsługa przerwania
6. Odtworzenie rejestrów roboczych ze stosu
7. Odblokowanie przerwania
8. Powrót do zawieszonego programu

Przerwania mogą mieć priorytety – w mikrokontrolerze AVR wszystkie są równe, w innych: sprzętowy - kontroler przerwań; programowo – wspólna procedura obsługi przerwań

O timerach:

* To liczniki do odmierzania okresów czasu
* Ich częstotliwość jest określana przez podział częstotliwości zegara
* Mogą mieć różne długości – zazwyczaj 8 albo 16 bitów
* Przerwanie od timera jest generowane w momencie przepełnienia licznika

## 55. Popularne interfejsy komunikacyjne w mikrokontrolerze.

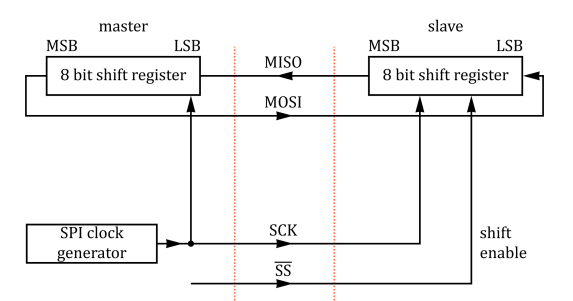
**Port** – interfejs między komputerem a innym komputerem/urządzeniem peryferyjnym. Porty mogą być **sprzętowe** (fragment sprzętu do łączenia się z innymi urządzeniami) lub **programowe** (wirtualne połączenie do wymiany danych, np. w protokołach TCP/UDP)

Porty mogą być **równoległe** (dane są przesyłane jednocześnie kilkoma przewodami. Każdy przewód przewodzi jeden bit informacji) lub **szeregowe** (dane są przesyłane jednym przewodem/parą. Bity są wysyłane po kolei)

Popularne interfejsy/porty:

1. Port LPT – (jedyny na tej liście) port równoległy używany do podłączania urządzeń peryferyjnych: drukarek, skanerów, ploterów
2. RS-232 – interfejs szeregowy do komunikacji z modemem. Wymaga konwersji poziomów napięć
3. I2C – Inter-Integrated Circuit (pośrednik między układami scalonymi), określa 2 najniższe warstwy modelu OSI: fizyczną i łącza danych
4. SPI – podobny do I2C, ale oferuje większą szybkość transmisji danych (do kilku MB/s). **Wbudowany w wielu mikrokontrolerach**
5. D2BUS – Digital Data Bus, do łączenia niewielkiej liczby urządzeń w małym obszarze
6. CAN – Controller Area Network, szeregowa magistrala używana w przemyśle samochodowym (ABS, sterowanie silnika)
7. IEEE 1394 – FireWire, opracowany przez Apple dla komputerów osobistych i cyfrowych urządzeń optycznych
8. USB – Universal Serial Bus – zaawansowany interfejs szeregowy, do komputerów PC, realizuje koncepcję *plug and play*, duża szybkość transmisji (1,5Mb/s minimum, 12 Mb/s max). Może obsługiwać do 127 urządzeń

Bardziej szczegółowo o niektórych interfejsach:  
**SPI:**



* Full-duplex, synchroniczny transfer danych, 7 programowalnych prędkości transmisji
* Możliwość pracy w trybie Master lub Slave
* Koniec transmisji identyfikowany flagą przerwań

**Interfejs szeregowy USART:**

* Full Duplex Operation, transmisja synchroniczna i asynchroniczna
* Możliwość transmisji danych 5-9 bitowych z 1-2 bitami stopu, bit parzystości wykrywany sprzętowo, wykrywanie błędów transmisji
* Przerwania: TX Complete, TX Data Register Empty, RX Complete

**Interfejs szeregowy Two-wire:**

* Obsługa trybów Master i Slave, arbitraż Multi-master
* 7-bitowy adres (128 adresów Slave), w pełni programowalny adres Slave-a w ramach General Call Support
* Prędkość transmisji do 400kHz

## 56. Klasyfikacja systemów operacyjnych.

Systemy dzielą się na:

1. Systemy czasu rzeczywistego – systemy komputerowe, od których wymaga się działania w określonych **ograniczeniach czasowych**, np. reagowania na określone zdarzenia z ograniczonym opóźnieniem (używane np. w samochodach)
2. Systemy rozproszone – złożone z **wielu komputerów** połączonych w sieć, postrzegane przez użytkowników jako jedna, spójna całość
3. Systemy równoległe – **wieloprocesorowe** (mają wiele procesorów/rdzeni), wykonujące obliczenia równolegle – wyższa moc obliczeniowa, większa niezawodność (uszkodzenie jednego rdzenia nie skutkuje przerwaniem działania systemu), wpierają multitasking
4. Systemy wsadowe – w pierwszych komputerach, przechowują w pamięci **kilka zadań w tym samym czasie**, wykonują je po kolei (zgodnie z priorytetem), wymagają operatora
5. Systemy interakcyjne – umożliwiające działanie współbieżne procesów tak, by umożliwić **bezpośredni dialog z użytkownikiem** (z takich korzystamy na komputerach)

Inny podział:

* Ze względu na interfejs:
  + Tekstowe (DOS, UNIX/Linux)
  + Graficzne (Windows, UNIX/Linux, MacOS)
* Ze względu na liczbę zadań:
  + Jednozadaniowe – jeden proces na raz (DOS)
  + Wielozadaniowe – wiele procesów (Windows, Linux, MacOS)
* Ze względu na przeznaczenie:
  + Zastosowanie domowe/biurowe (Windows XP Home)
  + Stacja robocza w sieci komputerowej (Windows 2000/XP Professional, Linux)
  + Serwer w sieci komputerowej (Windows 2000/2003 Server, Novell NetWare)
* Ze względu na architekturę:
  + Monolityczne – najprostsze, jednozadaniowe
  + Warstwowe - hierarchiczna struktura poleceń systemowych, wiele poleceń na raz
  + Klient-serwer – bardzo rozbudowana struktura, serwer nadzoruje podrzędne systemy zainstalowane w poszczególnych komputerach sieci
* Ze względu na typ jądra:
  + Jądro monolityczne – często w systemach UNIX, wszystkie zadania są wykonywane przez jądro
  + Mikrojądro – ma tylko niezbędne elementy (funkcje zarządzania wątkami, komunikacja międzyprocesowa, obsługa przerwań i wyjątków) – np. Amoeba, QNX, BeOS
  + Jądro hybrydowe – kompromis między jądrem monolitycznym a mikrojądrem (ma dodatkowo np. stos sieci

## 57. Problem szeregowania procesów/wątków w systemach operacyjnych.

Proces (inaczej zadanie – task) – to **wykonujący się program**, ma licznik instrukcji, który wskazuje na następny krok do wykonania. Ma też własny obszar przydzielonej mu pamięci operacyjnej

Stany procesu:

* Nowy – proces został właśnie utworzony
* Aktywny – jest właśnie wykonywany przez procesor
* Czekający – czeka na zajście zdarzenie (np. wykonanie operacji wejścia/wyjścia)
* Gotowy – czeka na przydzielenie mu procesora
* Zakończony – zakończył działanie

Procesy są bardzo **złożoną i kosztowną strukturą** (ma swój:  
segment kodu – z instrukcjami do wykonania,   
stos – do wywoływania procedur, przekazywania parametrów i wyników oraz przechowywania zmiennych lokalnych,   
stertę – pamięć dla zmiennych dynamicznych i segment danych – do zmiennych globalnych)

**Wątek** – lżejsza struktura tworzona w ramach procesu, nazywana też procesami lekkimi. Wątki jednego procesu są wykonywane współbieżnie. Mają osobne rejestry i stos wywołań, ale wszystko inne mają wspólne

**Planista** – proces systemowy, który wybiera, któremu procesowi nadać stan „aktywny” w danym momencie, są 2 typy planowania:

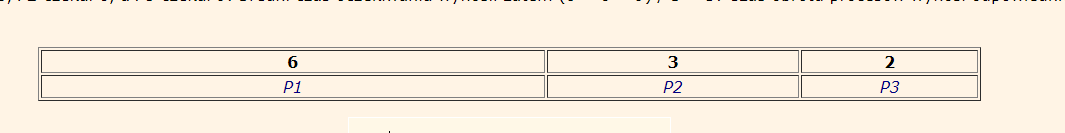
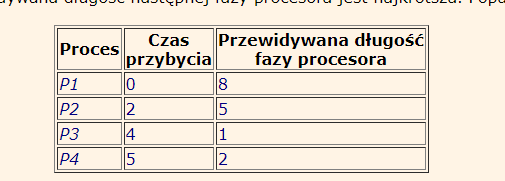
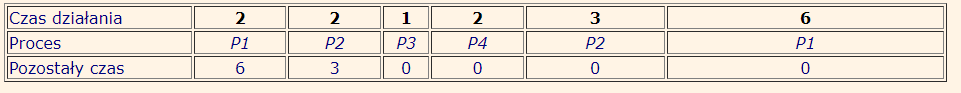
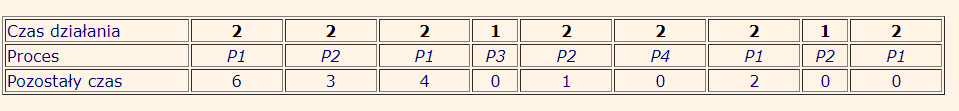
* Niewywłaszczeniowy – nie ma sytuacji, w której planista wstrzymuje proces aktywny
* Wywłaszczeniowy – może wstrzymać proces

**Ekspedytor** – proces egzekwujący wyroki planisty krótkoterminowego. Przekazuje dostęp do procesora procesom. To też zajmuje czas (odpowiednik biurokracji) – ten czas nazywamy *opóźnieniem ekspedycyjnym*

Kryteria jakości planowania procesów:

* Wykorzystanie procesora – nie chcemy go marnować, gdy może wykonywać zadania
* Przepustowość – liczba faz procesora wykonanych na jednostkę czasu – maksymalizujemy
* Czas oczekiwania – czas spędzony przez proces w kolejce gotowych – minimalizujemy
* Czas obrotu – czas wykonania jednej fazy procesora (pobytu procesu w stanie gotowy i aktywny) – minimalizujemy. Czas obrotu = czas oczekiwania (wpływa na to szeregowanie) + czas wykonywania obliczeń (szeregowanie nie ma wpływu)
* Czas reakcji – czas od zajścia zdarzenia, na które proces ma zareagować, do powzięcia pierwszej reakcji – minimalizujemy – ważne w systemach interakcyjnych z podziałem czasu

Strategie szeregowania:

1. FCFS (First-Come, First-Served) – strategia bez wywłaszczania, procesy są wykonywane w tej kolejności, w jakiej pojawiły się w kolejce (co wpływa na czas oczekiwania) – prosta metoda z niskimi narzutami systemowymi  
   
2. SJF (Shortest Job First) – strategia bez wywłaszczania, jak procesor jest wolny, to planista wybiera z kolejki gotowy proces, dla którego długość fazy jest najkrótsza  
     
   Wykonają się w kolejności: P1->P3->P4->P2
3. SRTF (Shortest-Remaining-Time-First) – strategia z wywłaszczaniem, jeżeli w kolejce gotowych procesów pojawi się nowy proces, jeżeli ten proces jest krótszy niż czas pozostały do wykonania aktualnego procesu, aktualny proces jest wstrzymywany i uruchamiany jest ten krótszy  
     
   W chwili 2 przychodzi P2, który jest krótszy niż P1, w związku z czym P1 jest wstrzymywane na rzecz P2  
   TO JEST OPTYMALNA STRATEGIA WZGLĘDEM ŚREDNIEGO CZASU OBROTU I ŚREDNIEGO CZASU OCZEKIWANIA, ALE NIE DA SIĘ JEJ ZAIMPLEMENTOWAĆ (ponieważ przewidzenie długości fazy procesora jest niemożliwe)
4. RR (Round-Robin) – każdy proces po kolei otrzymuje kwant czasu procesora do wykorzystania, przykład gdy ten kwant = 2, dla tabelki z SJF (chyba jest błąd, bo P3 powinien wykonać się o krok wcześniej)  
     
   ZŁA STRATEGIA WZGLĘDEM ŚREDNIEGO CZASU OCZEKIWANIA + DUŻO ZMIAN KONTEKSTU ZNACZY, ŻE EKSPEDYTOR MARNUJE DUŻO CZASU (ale jest to korzystna strategia, gdy ważny jest mały czas reakcji, jak np. przy pracy wielu użytkowników)
5. Planowanie Priorytetowe – procesor jest przydzielany na podstawie priorytetów, SJF i SRTF to są strategie planowania priorytetowego (długość procesu = priorytet)

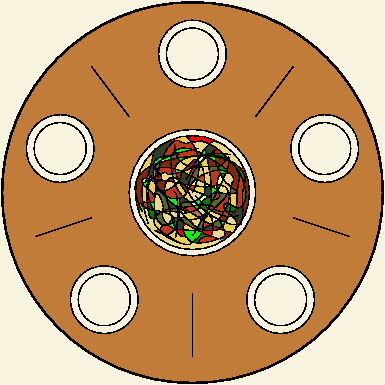
Zagłodzenie procesu – sytuacja, w której proces czeka w nieskończoność na swoją kolej w dostępie do procesora. Pojawia się w planowaniu priorytetowym, gdy ciągle pojawiają się nowe procesy o wyższym priorytecie

## 58. Problem synchronizacji procesów/wątków w programach komputerowych oraz przedstaw jakie wsparcie w tym zakresie oferują systemy komputerowe i operacyjne.

Przyczyny potrzeby synchronizacji procesów:

* Procesy współdzielą pewną strukturę danych – **problem sekcji krytycznej**
* Wyniki działania jednego procesu stanowią dane dla innego procesu – np. **problem producenta-konsumenta**
* Procesy korzystają z pewnej wspólnej puli zasobów, które pobierają i zwalniają wedle potrzeb. Te zasoby też trzeba synchronizować – np. **problem pięciu filozofów**

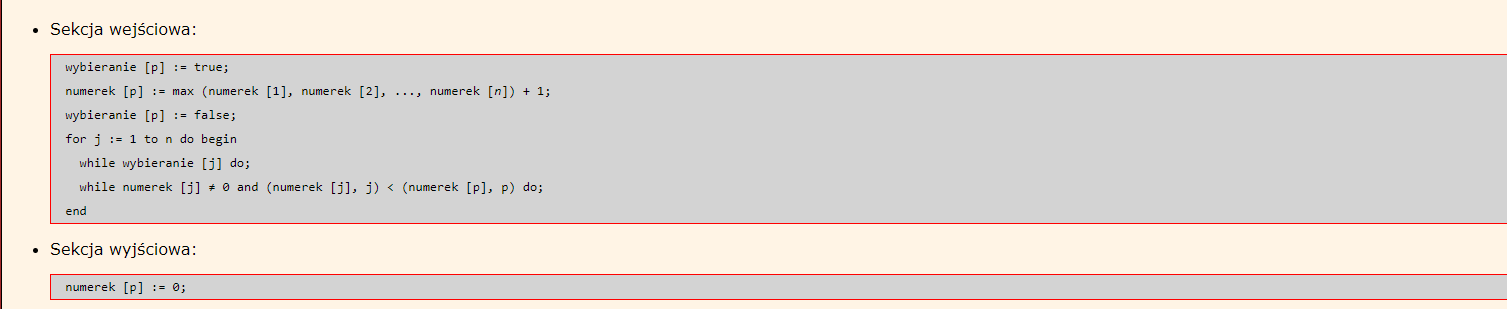
Dokładniej o tych problemach:

* Problem producenta-konsumenta – mamy dwa procesy działające w nieskończonej pętli: producent wytwarza jednostki produktu, a konsument je konsumuje. Produkty są umieszczane w buforze o określonej wielkości. Jeśli bufor jest pusty, konsument musi czekać, a jeśli bufor jest pełny, to producent musi czekać.
  + Przykładem tego problemu jest buforowanie znaków naciskanych na klawiaturze lub wysyłanie wydruków na drukarkę
  + Rozwiązanie problemu – wkładanie i wyjmowanie z bufora powinno tworzyć sekcję krytyczną
* Problem sekcji krytycznej – fragmenty kodu, które nie mogą być wykonywane współbieżnie. Przed wejściem do sekcji krytycznej, każdy proces musi przejść przez sekcję wejściową i wyjść przez sekcję wyjściową. Sekcja wejściowa jest zamknięta, gdy jakiś proces jest już w sekcji krytycznej, i jest otwierana, gdy ten proces przechodzi przez sekcję wyjściową
  + Sekcja krytyczna ma spełniać warunek wzajemnego wykluczania – tylko jeden proces na raz może w niej być
  + Sekcja krytyczna nie powinna też pozostawać pusta, jeżeli jakiś proces chce do niej wejść
  + Procesy oczekujące na wejście do sekcji krytycznej nie mogą być zagłodzone
* Problem czytelników i pisarzy – rozwinięcie problemu producenta-konsumenta:
  + Mamy bibliotekę z czytelnikami i pisarzami
  + Na raz w bibliotece może być wielu czytelników
  + Jak jest w niej pisarz, to nikt inny nie może być
  + Ustawiamy ich w kolejce – jak biblioteka jest pusta, to wchodzi do niej pisarz (o ile jest pierwszy w kolejce). Po wyjściu z niej, czekający po nim czytelnicy wchodzą i proces się zapętla
  + Rozwiązany przy pomocy monitorów
* Problem 5 filozofów – filozofowie rozmyślają albo jedzą przy okrągłym stole, każdy ma swoje miejsce:  
  
* Każdy z nich musi sobie najpierw nałożyć posiłek, a potem wziąć 2 pałeczki obok siebie
* Mają tylko 5 pałeczek i muszą się dzielić tak, by nie było blokady i zagłodzenia – jeżeli przy siadaniu każdy z nich brał by od razu pałeczkę po lewej i czekał na kolegę po prawej, to możemy mieć zakleszczenie (każdy trzyma jedną pałeczkę); Jak biorą 2 naraz, to może być tak, że czekając na to, by obie pałeczki były wolne, jego koledzy po bokach będą mu je na zmianę zabierać
* Rozwiązanie – pozwalamy tylko 4 filozofom na raz siedzieć przy stole, jako pałeczki używamy semaforów binarnych. Liczba talerzy też jest reprezentowana semaforem

Algorytm Dekkera – implementacja sekcji krytycznej dla dwóch procesów współdzielących zmienne. Mamy zmienne:

* K – tablica dwuelementowa, w której każdy element to 1 lub 0
* Czyja\_kolej – zmienna rozstrzygająca, który proces może wejść do sekcji krytycznej

Każdy proces ma swój numer (1 lub 2) pamiętany w zmiennej lokalnej „p”.  
Każdy element tablicy K na początku ma wartość 1. Jak proces chce wejść do sekcji krytycznej, to ustawia swoją komórkę tablicy na 0 aż do wyjścia z sekcji krytycznej. Jeżeli czyja\_kolej=numer\_procesu, to proces wchodzi do sekcji krytycznej, jeśli nie, to czeka i okresowo sprawdza, czy to jego kolej. Przy wychodzeniu z sekcji krytycznej, proces ustawia swój element K na 1, i ustawia czyja\_kolej=numer\_drugiego\_procesu

Algorytm piekarniczy – rozwiązanie problemu sekcji krytycznej dla dowolnej liczby procesów. Działa jak numerki na poczcie-proces chcący wejść do sekcji krytycznej „pobiera numerek” i zapisuje go we wspólnej tablicy. Po wyjściu z sekcji krytycznej, numerek jest „wyrzucany”. Może być tak, że ten sam numerek trafi do np. dwóch procesów. Wtedy, priorytet ma proces o niższym numerze „p”  


Najpierw bierze numerek, potem czeka, aż wszyscy inni wezmą numerek, a potem czeka, aż jego numerek będzie najmniejszy w tablicy

Wada obu tych algorytmów – aktywne czekanie (albo w pętli while, albo poprzez if)

Mechanizmy synchronizacji:

* Semafory – specjalna zmienna do obsługi sekcji krytycznej, którą można zwiększyć o 1 (operacja „v” – przy wyjściu) lub zmniejszyć o 1 (operacja „p” – przy wejściu). Jeżeli semafor jest dodatni, to proces może go zmniejszyć i wejść do sekcji krytycznej, a jak nie jest, to czeka
* Kolejki komunikatów – można do nich wkładać i wyjmować komunikaty, działają na zasadzie FIFO (First in, first out). Można je zaimplementować przy pomocy semaforów
* Monitory – w językach programowania wysokiego poziomu. To rodzaj klasy, której metody stanowią sekcję krytyczną. Na raz, tylko jeden proces może być w monitorze, pozostałe czekają w kolejce FIFO. Chronią przed np. pomyłką w użyciu semaforów. Dodatkowo ma operacje:
  + Wait – wstrzymuje proces w wstawia go na koniec kolejki
  + Signal – wznawia pierwszy proces w kolejce

Sprzętowe mechanizmy synchronizacji – nie powodują aktywnego oczekiwania:

* Blokowanie przerwań – w komputerach jednoprocesorowych
* Instrukcja procesora test-and-set – w komputerach wieloprocesorowych, jest to atomowa funkcja pobierająca wartość zmiennej i ustawiająca ją (nie da się w międzyczasie nic zrobić)

## 59. Mechanizmy obsługi pamięci operacyjnej wykorzystywane w systemach operacyjnych.

Pamięć – duża tablica słów (maszynowych), pozycja słowa w tablicy to *adres*

Jeżeli nie mamy wystarczająco pamięci, by każdy proces miał własną, musimy przeprowadzać **wymianę procesów** – niektóre procesy są wstrzymywane i zapisywane na dysku, a uzyskana z nich pamięć operacyjna jest przydzielana pozostałym procesom, gdy zwolni się wystarczająco dużo pamięci operacyjnej, to procesy zapisane są wznawiane. Wznowione procesy mogą być załadowane w inne miejsce pamięci, niż to w którym były poprzednio – *Kod i zawartość pamięci procesu muszą być przemieszczalne*

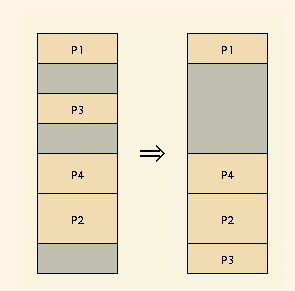
Pamięci fizyczna – cała pamięć operacyjna w komputerze, jaką widzi system operacyjny, operuje na adresach fizycznych

Pamięć logiczna – pamięć przydzielana procesowi przez system operacyjny, pamięć logiczna nie zmienia się gdy jej zawartość jest przenoszona do innego fragmentu pamięci fizycznej, operuje na adresach logicznych

Jednostka zarządzania pamięcią (MMU – memory management unit) – element sprzętowy do realizacji pamięci logicznej:

* Sprawdza, **czy** używane przez proces adresy logiczne **są poprawnymi** adresami
* **Tłumaczy** adresy logiczne na odpowiadające im adresy fizyczne
* Ma 2 elementy programowalne:
  + Rejestr graniczny – zawiera wielkość przydzielonego obszaru pamięci
  + Rejestr przemieszczenia – zawiera początkowy adres fizyczny przydzielonego obszaru

Modele zarządzania pamięcią:

1. Przydział ciągły:
   1. Każdemu procesowi przydzielany jest jeden spójny fragment pamięci fizycznej
   2. Najprostsza metoda, ale marnujemy miejsce
   3. **Fragmentacja** – ma 2 typy:
      1. **Zewnętrzna** – wolna pamięć fizyczna jest poszatkowana i nowy proces nie może się zmieścić. Wolne obszary należy scalić – **kompaktyfikacja** lub **upakowanie**  
         
      2. **Wewnętrzna** – występuje w stronicowaniu (które chroni przed segmentacją zewnętrzną), przy dzieleniu pamięci logicznej na równe strony, często część ostatniej strony danego procesu jest niewykorzystana

Strategie przydziału pamięci:

1. First-fit – najprostsza, przydzielany jest pierwszy wystarczająco duży obszar
2. Best-fit – wybieramy najmniejszy, ale wystarczający obszar
3. Worst-fit – wybieramy zawsze największy obszar, daje to szansę na to, że pozostały obszar będzie wystarczająco duży i kompaktyfikacja nie będzie potrzebna

Pamięć procesu dzieli się na **segmenty:**

* Kod programu
* Zmienne globalne
* Stos
* Sterta

System operacyjny może wydzielić każdy z tych segmentów do osobnego bloku w pamięci (zamiast przydziału ciągłego)  
Wtedy, adres logiczny ma dwie części – numer segmentu + adres w obrębie segmentu. MMU ma 2 tablice zamiast 2 rejestrów – tablicę numerów segmentów i tablicę ich adresów – **2 razy dłuższy dostęp do pamięci logicznej niż do fizycznej**  
Procesy mogą współdzielić segmenty (np. segment kodu) – nie powinny go wtedy modyfikować, bo inaczej potrzeba synchronizacji

**Stronicowanie** – pamięć fizyczna i logiczna są dzielone na kawałki **równej** wielkości (logiczna – na strony, fizyczna – na ramki). Wielkość to potęga 2 z przedziału 512B do 16MB, zazwyczaj to 4kB  
Tablica stron – służy MMU do określenia, czy podawany przez proces numer strony jest poprawny, i jakiej ramce odpowiada dana strona, dla przyspieszenia szukania używa się jeszcze *tablicy asocjacyjnej*, gdzie przechowywane są numery ramek ostatnio używanych stron  
**Stronicowanie wielopoziomowe** – wykorzystanie kolejno kilku mniejszych tablic stron, pozwala to zmniejszyć fragmentację wewnętrzną (w systemach x32 są 2 rzędy tablic, w x86-64 są 4

Odwrotna tablica stron – tablica typu hash-table, która ma tyle elementów, ile ramek w pamięci fizycznej (a nie ile stron w logicznej, które mnożą się w przypadku stronicowania wielopoziomowego)

Segmentacja ze stronicowaniem – segmentacja + stronicowanie, pozwala to chronić się przed fragmentacją zewnętrzną i współdzielenie segmentów jednocześnie

## 60. Istota mechanizmu pamięci wirtualnej - wady i zalety tego rozwiązania.

Pamięć wirtualna – możliwość posługiwania się **znacznie większą pamięcią** niż fizyczna pamięć RAM, oddzielenie pamięci fizycznej od pamięci logicznej dostępnej użytkownikowi.  
Procesy charakteryzują się **lokalnością** – w konkretnej chwili potrzebują tylko części przydzielonej im pamięci, w związku z czym nieużywana pamięć może zostać przeniesiona na pamięć drugorzędną, np. dysk (obszar w pamięci drugorzędnej, jaki wykorzystujemy, to **obszar wymiany**)  
Mechanizm pamięci wirtualnej to *rozszerzenie stronicowania i segmentacji* (stronicowanie/segmentacja na żądanie) – stronicowanie na żądanie jest bardziej elastyczne, dlatego jest częściej stosowane

Mechanizm pamięci wirtualnej umożliwia **współdzielenie przestrzeni adresowej przez wiele procesów i pozwala efektywniej tworzyć procesy**

Stronicowanie na żądanie – najpopularniejszy sposób implementacji pamięci wirtualnej – strona jest sprowadzana z obszaru wymiany do pamięci operacyjnej tylko wtedy, gdy jest potrzebna (*procedura leniwej wymiany*)

Bit poprawności – bit oznaczający, czy dana strona jest w pamięci operacyjnej (jak 1, to tak; jak 0, to nie – może jest w obszarze wymiany, a może adres strony jest niepoprawny)

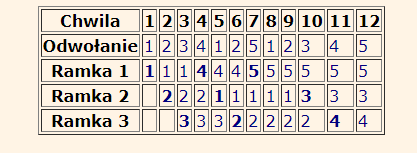
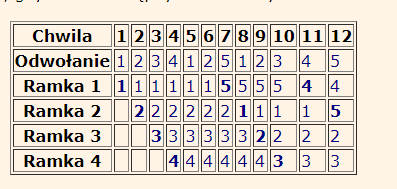
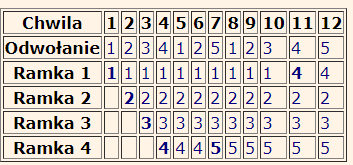
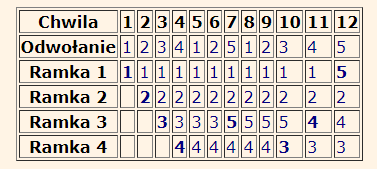
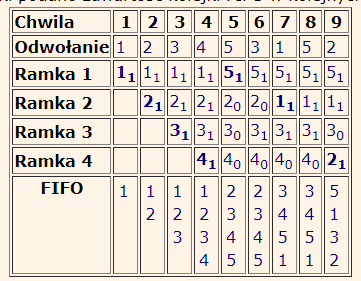
Gdy brakuje strony:

1. Sprawdzana jest wewnętrzna tablica (w bloku kontrolnym procesu), żeby sprawdzić czy odwołanie było poprawne
2. Znajdowana jest wolna ramka w pamięci fizycznej
3. Sprowadzana jest strona z dysku
4. Tablica stron jest aktualizowana (wstawiany jest numer ramki, bit poprawności = 1)
5. Instrukcja zatrzymana z powodu braku strony jest wznawiana

Gdy nie ma wolnej ramki, jakaś nieużywana ramka zostaje przeniesiona na dysk jeśli na dysku nie ma jej wiernej kopii, inaczej jest po prostu usuwana (*zastępowanie stron*)  
Do sprawdzenia, czy kopia na dysku jest wierna, używany jest *bit modyfikacji*

EAT – effective access time (efektywny czas dostępu – miara sprawności pamięci używana przy ocenie sprawności stronicowania na żądanie)

Algorytmy zastępowania stron:

1. **FIFO** – na dysk przenoszona jest ramka, która najdłużej przebywała w pamięci  
     
   Dla tego ciągu odwołań, musimy wymieniać strony 9 razy. Jak mamy 4 ramki:  
     
   Mamy 10 wymian – system działa wolniej pomimo większej pamięci – nazywa się to **anomalią Beladiego**
2. **Algorytm optymalny** – odkładana jest strona, która przez najdłuższy okres nie będzie wykorzystywana:  
     
   Nie da się zaimplementować, bo nie przewidzimy kiedy strona będzie potrzebna
3. **Least Recently Used (LRU) –** odkładana jest strona, która nie była używana przez najdłuższy okres czasu:  
     
   Do implementacji LRU używa się rozwiązań sprzętowych:
   1. **Znaczników czasu** – każda strona ma znacznik z informacją, kiedy była ostatni raz użyta
   2. **Stos** – gdy strona jest użyta, jej numer wędruje na szczyt stosu. Strona na dnie stosu jest wymieniana
4. **Algorytm drugiej szansy** – podobna do FIFO, ale strony mają też *bit odwołania* – patrzymy po kolei na najstarsze strony w kolejce, jeśli bit odwołania=1, to zmieniamy na 0 i idziemy dalej, a jak jest równy 0, to strona jest przenoszona  
   
5. **Algorytmy zliczające** – mają dwa typy, ale oba nie są implementowane, bo ciężko stworzyć sprzętowo odpowiedni licznik, a ich wyniki są daleko od alg. optymalnego:
   1. **Least frequently used (LFU)** – odkładana jest ta strona, do której było najmniej odwołań
   2. **Most frequently used (MFU)** – odkładana jest strona z największą liczbą odwołań

Sposoby przydziału ramek procesom:

* Lokalny – każdy proces ma własną pulę ramek, które mu są w razie potrzeby przydzielane
* Globalny – wspólna pula ramek dla procesów, dzięki temu ramki się nie marnują
* Stały – każdy proces ma na stale przydzielone ileś ramek, a o pozostałe ramki rywalizują (przydział może być równy lub proporcjonalny)
* Priorytetowy – każdy proces ma priorytet decydujący o liczbie przydzielonych ramek

**Szamotanie** – sytuacja, w której proces ma przydzielone mniej ramek, niż potrzebuje. W związku z tym, wymiana stron następuje co chwilę. Procesor wtedy jest słabo wykorzystany, co *planista długoterminowy* **może uznać za sygnał do dodania nowego procesu, pogarszając przy tym sytuację**  
**Zbiór roboczy** – sposób na zapobieganie szamotaniu, to zbiór stron, do których proces odwołał się w ciągu ostatnich „I” instrukcji. Na podstawie rozmiaru zbioru roboczego, możemy przydzielić procesom proporcjonalną ilość ramek