Obrona

Informatyka Stosowana inżynierska obrona

1. Podstawowe układy cyfrowe: bramki logiczne, przełączniki, układy sekwencyjne

Układy cyfrowe to zbiór połączonych elementów elektronicznych, w którym informacje reprezentowane są w sposób binarny. Jeśli chodzi o binarność stanu, to ma on dwa stany: stan wysoki (1 lub H), oznaczający potencjał względem masy bliski napięciu zasilania, i stan niski (0 lub L), potencjał względem masy bliski 0V. W logice stan wysoki reprezentowałby prawdę, a niski fałsz.

Do obliczeń można wykorzystać algebrę boole'a, z oczywistymi przykładami, jak

```
1 ∧ 0 = 0
1 ∨ 0 = 0
```

Zasady:

```
A ∧ A = A
A ∨ A = A
A ∧ (A ∨ B) = A ∧ B
```

lub zapis

```
AB = A ∧ B
A + B = A ∨ B
```

Przykładowe bramki:

- AND = A ∧ B
- OR = A ∨ B
- NOT = !A
- XOR = (A ∨ B) ∧ !(A ∧ B)
- NOR = !(A ∨ B) odwrotność OR
- NAND = !(A ∧ B) odwrotność AND
- XNOR = (A ∧ B) ∨ (!A ∧ !B)

Bramki Nxyz działają jak bramki xyz, ale z negacją wyjścia

Przełączniki to urządzenia wejścia - elementy mechaniczne lub elektromechaniczne.
Niektóre utrzymują swój stan (stabilne - np. przełącznik światła nie zmienia sam stanu), a inne, jak klawisze klawiatury (chwilowe), wracają do domyślnego stanu.

Ze względu na inercyjność, rozróżniamy układy na kombinacyjne i sekwencyjne.
W układach sekwencyjnych wyjścia w nich zależą nie tylko od wejść, ale też od wewnętrznego stanu.
A w kombinacyjnych, wyjścia zależą bezpośrednio od wejść (funkcja wejść na wyjścia).

Przerzutniki - układy sekwencyjne, które są w stanie zapamiętać stan i przekazać go dalej. Czyli nie są prostą funkcją wejść na wyjścia, bo do takiej funkcji jako argument trzeba dołączyć ich aktualny stan. Czyli posiadają wewnętrzny stan, który może zmieniać się w czasie i wpływa na działanie takiego układu (wyjście będzie różne dla tych samych wejść przy różnych stanach)

Przykłady:

Przerzutnik RS (flip flop) - wejścia SR, wyjścia Q i !Q Bardzo prosty i asynchroniczny (czyli nie wymaga obecności zegara)
S ustawia stan układu (Q) na wysoki dla S = 1
R ustawia stan układu na niski dla R = 1
Dla S i R = 0 stan pozostaje taki sam, czyli Q[n] = Q[n-1]
Dla S i R = 1 stan nie jest zdefiniowany (może być losowy przez sprzeczność logiczną układu). Taki stan nazywa się stanem zabronionym
Przerzutnik JK flip flop zachowuje się jak RS, różni się tym, że dla J i K = 1 => Q = !Q

Przerzutniki D (Delay) flip flop i T (Toggle) flip flop - podstawowe przerzutniki synchroniczne. Wymagają sygnału zegara do ustawiania stanu. Dzięki temu ograniczone są zakłócenia ze strony danych wejściowych (debouncing)

- Delay: ustawia Q=D, gdy CLK ma stan opadający lub narastający. Korzysta wewnętrznie z SR flip flop
- Toggle: ustawia Q=!Q, gdy T=1 i CLK ma stan opadający lub narastający

Inne układy sekwencyjne to na przykład:

- Licznik, który liczy liczbę zmian sygnału wejściowego
- Rejestr, który przechowuje wartość bitu

Przykładami układów kombinacyjnych są:

- Komparator, który zwraca, który z sygnałów jest większy, czy też są równe (taki if)
- Multiplekser, który przekazuje jedno z wejść na wyjście w zależności od wejścia sterującego (taki switch)

Hazard to zjawisko, gdzie wynik układu może być chwilowo niepoprawny przez nienatychmiastową naturę prądu. Można walczyć z tym zjawiskiem wykorzystując zegar (układy synchroniczne) lub przez wprowadzenie redundancji elementów.

2. Arytmetyka dwójkowa, funkcje boolowskie, tablice Karnaugh

Arytmetyka dwójkowa dotyczy arytmetyki na liczbach o bazie 2, czyli cyfry to tylko 0 lub 1. Liczby konstruuje się analogicznie do bazy dziesiętnej, gdzie każda ważniejsza cyfra ma bazę razy większą wagę (czyli w przypadku dwójkowego 2 razy większą).

Dodawanie, odejmowanie, dzielenie i mnożenie można rozpisać przy dokładnie tym samym algorytmie jak dla operacji pisemnych w systemie dziesiętnym. Czyli

```
  1
  1 +
---
  2
---
1 0
```

itd.

Arytmetyka dwójkowa jest wykorzystywana w elektronice, i przez to w komputerach, z racji prostego wykrywania stanu 0 a 1 (reprezentowane przez potencjał względem masy zbliżonym do 0 dla 0 i potencjale względem masy zbliżonym do napięcia zasilania dla 1)

Naturalny kod binarny NKB - liczby są zapisywane w formie słów o konkretnej długości, np. 8, co oznacza, że liczbę np. 115 (1110011[2])zapisałoby się jako 01110011, dopełniając po lewej zera, aby słowo miało tą długość. Wtedy długość słowa określa przedział możliwych do reprezentacji liczb w zbiorze N = <0;2^N>

Kod dwójkowo-dziesiętny - reprezentacja osobno każdej cyfry w liczbie dziesiętnej słowami o długości 4 (bo 2^4>9). Niektóre wartości np. 1111 są niemożliwe, przez co występuje redundancja (nadmiarowość). Wg. mnie naturalnie przekłada się to w system 16stkowy, gdzie 10=A, 15=F.

Funkcje boolowskie: funkcje o n argumentach 0 lub 1 i o 1 wyjściu równym 0 lub 1. Układy elektroniczne realizujące funkcje boolowskie to układy kombinacyjne.

Funkcje boolowskie można przedstawić na 4. sposoby:

1. Przykładowa funkcja: F(a, b, c, d) = ab + (c + !d)
2. Iloczyn sum `F(A,B,C,D) = (A+!C)(!A+B+!D)` lub suma iloczynów ```F(A,B,C,D) = (AB!D)+(!A!C). No wiadomo, iloczyn sum jest prawdziwy tylko, jak spełnimy w każdym nawiasie choć jeden, a suma iloczynów jak w choć jednym nawiasie każdy warunek.
3. Tabele prawdy

wartość dziesiętna	a	b	f(a,b)
0	0	0	1
1	0	1	0
2	1	0	–
3	1	1	1

wartość dziesiętna to liczba jakby postawić a jako cyfrę najbardziej znaczącą i b jako najmniej.
4. Zbiory wartości dla F=1 albo dla F=0 Dodatkowo, jeśli jakieś wartości są niepewne, to funkcja jest niezupełna i też możne te wartości podać

Dla określenia wartości argumentów, dla których wartość funkcji wynosi 1 stosuje się zapis:

$$f(x_1, x_2, \dots, x_n) = \sum(d_1, d_2, \dots, d_m) = \bigcup(d_1, d_2, \dots, d_m)$$

Gdzie:
x_i – argumenty funkcji,
d_j – wartości dziesiętne odpowiadające kombinacjom argumentów
m – liczba kombinacji, dla których funkcja równa się 1

Przykład:

$$f(a,b,c) = \sum_{abc}(0,2,3,7) = \sum_{abc}(000,010,011,111) =$$
$$= (\sim a \wedge \sim b \wedge \sim c) \vee (\sim a \wedge b \wedge \sim c) \vee (\sim a \wedge b \wedge c) \vee (a \wedge b \wedge c)$$

Dla określenia wartości argumentów, dla których wartość funkcji wynosi 0 stosuje się zapis:

$$f(x_1, x_2, \dots, x_n) = \prod(d_1, d_2, \dots, d_m) = \bigcap(d_1, d_2, \dots, d_m)$$

Przykład:

$$f(a,b,c) = \prod_{abc}(1,4,5,6) = \prod_{abc}(001,100,101,110) =$$
$$= (a \vee b \vee \sim c) \wedge (\sim a \vee b \vee c) \wedge (\sim a \vee b \vee \sim c) \wedge (\sim a \vee \sim b \vee c)$$

Natomiast gdy funkcja jest niezupełna, wartości argumentów, dla których wartość funkcji jest nieokreślona podajemy w kolejnym nawiasie:

$$f(a,b,c) = \sum_{abc}(1,3,7,(2,4))$$

Tablice Karnaugh można wykorzystać do uproszczenia, czyli minimalizacji funkcji boolowskich. Najlepiej działa, gdy liczba wejść jest niewielka, więc zacznę od przypadku cztero argumentowego. Gdy rozpiszemy tabelę prawdy dla cztero argumentowej funkcji z zachowaniem kodu grey'a (czyli kolumny i wiersze różnią się od sąsiadów wartością tylko 1 argumentu - bez tego te prostokąty byłyby bez sensu), to mamy te 0 i 1. Typowo kolumny to AB, a wiersze CD, bo można grupować w takiej tablicy argumenty w takie ciągi. Zaczynamy od narysowania największego prostokąta/ów, którego każdy bok jest potęgą 2 (1, 2, 4...), i wszystkie komórki wewnątrz są 1 (lub X dla niezdefiniowanych). Każdy krok algorytmu to wzięcie aktualnego N (pole tego kwadratu), znajdowanie kwadratów o takim polu które mają 1 lub X tylko w sobie. Potem dzielimy N przez 2 i ciągle robimy to samo, aż wszystkie 1 będą w prostokącie/prostokątach (mogą być naraz w dwóch, jeśli to optymalne). No i wynik to na logikę można zauważyć, że w takich prostokątach 1/2... argumenty się nie zmieniają i przedstawić funkcję np. jako `Y = B*!C*D + A*B*!D`. Na logikę jak jest 1 w kwadracie gdzie A=0 i C=1 no to (!A*C). Prostokąty mogą przechodzić przez "ściany" tabeli na drugą stronę. Alternatywnie można zrobić to samo, ale szukać 0 to `Y = !(B*D) + !(!A*CD)`
Tablice Karnaugh do max 4-6 zmiennych. Dla więcej niż 4 zmiennych, trzeba brać pod uwagę osie symetrii.

3. Programowanie strukturalne - zasady. Przegląd instrukcji strukturalnych

Programowanie strukturalne to podstawa nowoczesnego programowania. Jej zasady pozwalają pisać kod, który ma jasny przepływ logiczny.
Liniowy przepływ najważniejszy, czyli ogólnie z góry do doły.

Warto wspomnieć, że programowanie strukturalne jest podparadygmatem programowania imperatywnego. Często przedstawia się je jako przekazanie instrukcji komputerowi, co ma zrobić, w kontrze do programowania deklaratywnego, gdzie instrukcja dotyczy tego, co chcemy osiągnąć. Czyli programowanie imperatywne to po prostu ciąg instrukcji, które ma wykonać komputer i zmienia jego stan.

W programowaniu strukturalnym program składa się z bloków, grupujące operacje, które będą wykonane od góry do dołu. Zwyczajowo zmienne definiowane w bloku nie są dostępne poza nim, są za to dostępne w zagnieżdżonych blokach, o ile nie wystąpi mechanizm maskowania (zmienna o tej samej nazwie przesłoni inną). Blok powinien mieć 1 punkt wejścia i 1 punkt wyjścia. Po wykonaniu wszystkich instrukcji wychodzimy z bloku. W większości języków blok oznaczony jest przez curly brackets {} (w Pythonie przez indentację). Bloki można zagnieżdżać w sobie, ale dla czytelności głęboko zagnieżdżone bloki warto wyciągać do funkcji, klas itd.

Główne elementy (są blokami, więc powinny mieć 1 punkt wejścia i wyjścia itd.):

- Instrukcje warunkowe: if; switch
- Iteracja - loop'y (while i for), z jasnymi warunkami zakończenia

W bloku operatorem sekwencji/konkatenacji operacji jest często średnik.

Należy ograniczyć korzystanie z break (w switch oczywiście dalej zezwolone), ograniczenie continue. W ewolucji względem Assemblera, absolutny zakaz goto, który sprawia, że przepływ nie jest liniowy. Chodzi o to, aby przepływ był jasny i ustrukturyzowany, nie tylko dla siebie, ale też dla innych programistów w zespole.

Można wspomnieć o wielu dobrych praktykach, jak

- wydzielanie zagnieżdżonych bloków do funkcji o jasnych nazwach
- dzielenie długich bloków na funkcje
- jasne nazywanie zmiennych i struktur (idealnie, brak potrzeby komentarzy, zamiast tego dobrze nazwane funkcje)
- unikanie zmiennych globalnych
- unikaj nadmiernego powtarzania się
- unikaj złożonych instrukcji warunkowych i iteracyjnych

Jeszcze jedna ciekawostka - instrukcja wiążąca. Np. zamiast

```
var p1 = new Person();
p1.name = "miki";
p1.surname = "fiki";
```

to

```
var p1 = new Person { name = "miki", surname = "fiki" };
```

Dla kolekcji też od razu dane (dla C# to i tak syntactic sugar)

4. Programowanie obiektowe - podstawowe pojęcia, zastosowania

Paradygmat obiektowy to jeden z najpopularniejszych paradygmatów. Jest on intuicyjny, ponieważ może tłumaczyć zjawiska ze świata rzeczywistego na obiekty i klasy, np. każdy człowiek posiada imię, ale konkretny człowiek ma własne imię.

Podstawowe pojęcia:

- Obiekt: zbiór własności tego obiektu oraz metod
- Klasa: instrukcja wykorzystywana do instancjonowania obiektów. Jest to blueprint z polami, metodami
- Pole/atrybut: "zmienne" klasy. Klasa definiuje pola (zazwyczaj z typami), a obiekty typu tej klasy mogą zazwyczaj mieć różne wartości danego pola, oraz mogą je zmieniać w czasie (przy zmianie, klasa jest mutowalna, niezmienne typy to np. record czy struct z C#)
- Metoda: "funkcja" klasy. O ile nie jest oznaczona jako statyczna, to wykonując ją na instancji klasy mamy dostęp do jej pól zazwyczaj przez mechanizm self/this itd. W wielu językach jest to niejawne, w Python trzeba przekazać self do takiej metody. Metoda działa jak zwykła funkcja, może zwracać coś, może mutować pola klasy, może instancjonować itd itd
- Dziedziczenie: klasa może dziedziczyć po innej klasie. Oznacza to, że wszystkie pola, metody klasy dziedziczonej są jakby "kopiowane" do klasy dziedziczącej. Bardzo przydatne, ale w praktyce można potem wpaść na ograniczenia tego i próbę upychania zbyt wielu rzeczy w klasę. Często stosuje się composition (kompozycję), czyli klasa ma inne instancje klas jako pola i w ten sposób rozszerza się jej możliwości (przykład z gier: gracz i wrogowie mają zdrowie i atak. to można by zrobić jako jedną klasę character i dziedziczyć. ale potem krzesła mają zdrowie bo reagują na wybuchy, no i przez dziedziczenie musiałyby w niełady sposób nadpisać pewnie metody ataku. a przy kompozycji to osobne moduły, nie powiązane w sposób ścisły ze sobą)
- Interfejsy: kontrakt zobowiązujący klasę do implementacji metod z interfejsu i sprawiający, że inne części kodu mogą prawidłowo oczekiwać, że klasa potrafi wszystko w kontrakcie
- Klasa abstrakcyjna: klasa oznaczona keywordem `abstract`. Pewnie posiada metody abstrakcyjne z samą sygnaturą metody (nazwa, argumenty, return value). Nie można bezpośrednio instancjonować, dopiero nieabstrakcyjne klasy dziedziczące mogą.
- Przesłanianie metody: nadpisanie działania metody w klasie dziedziczącej. Często wykorzystuje się np. `base().TaMetoda();`, ale nie trzeba
- Przeciążanie metody: zdefiniowanie metody parę razy w klasie, ale każda z różnymi argumentami. Przydatne, gdy klasa akceptuje różne metody wywołania. Częstą praktyką jest to, że wszystkie te metody zwracają wynik z jednej, głównej, wybranej, która przyjmuje argumenty w dogodnej postaci. Np. w Unity instancja obiektu w scenie może mieć pozycję, rotację, rodzica... różnego rodzaju typ layera... każda metoda przekazuje argumenty jakiejś jednej.
- Polimorfizm parametryczny: template/generics np. `List<T>`
- Polimorfizm: jak kot dziedziczy po zwierzęciu to można traktować go tak i tak, czyli wsadzić do listy zwierząt i traktować jak zwierzę, bez zwracania uwagi na konkretny typ.
- Modyfikator dostępu: metody i pola w klasach mogą być public, private, protected. Public wiadomo metoda dostępna dla innych klas, private tylko dla klasy (inna instancja może w kodzie metody korzystać z metody prywatnej innej instancji). Protected to jak private, ale dodatkowo dostępne dla klas dziedziczących.
- Hermetyzacja: modyfikatory dostępu, interfejsy przydają się do implementacji hermetyzacji. Chodzi o to, aby dla innych klas najważniejszy był głównie wynik którego potrzebują od klasy, a nie wewnętrzna implementacja. Czyli klasa powinna udostępniać minimum metod i publicznych pól, aby inne klasy musiały w kontrolowany sposób ją mutować / uzyskiwać z niej wynik.

Programowanie obiektowe można stosować praktycznie wszędzie. Prawie wszystkie większe gry komputerowe są napisane w tym paradygmacie, ale też dużo aplikacji desktopowych, mobilnych, backendów, frontendów...

SOLID:

- single responsibility: funkcja, klasa ma zajmować się tylko jedną rzeczą. Car nie zajmuje się przewidywaniem pogody
- open/closed: otwarte na rozszerzenie, zamknięte na modyfikację. Car rozszerza Vehicle "rozszerzając" metodę accelerate
- liskov substitution: typ obiektu można zmienić na wyższy przy dziedziczeniu i dalej działa (np. traktować Car jako Vehicle). Nie powinno się metod rodziców w dzieciach ustawiać jako throw new Exception. Np. ElectricVehicle: exception na shift() : (
- interface segregation: lepiej mieć więcej interfejsów niż mniej (tutaj podział samochodów na silniki spalinowe, elektryczne). ADT w C#
- dependency inversion: moduły high level nie powinny zależeć od low-level: powinny zależeć od abstrakcji (interfejsy), a abstrakcje nie zależeć od implementacji

5. Podstawowe operacje na zbiorach, funkcjach i relacjach. Rachunek zdań. Rachunek kwantyfikatorów

Zbiór to dobrze określona kolekcja elementów, gdzie jednoznacznie można stwierdzić, czy dany obiekt należy, czy nie do zbioru.

Podstawowe operacje na zbiorach:

- Suma zbiorów A + B czyli zbiór do którego należą wszystkie elementy w zbiorze A i w zbiorze B
- Różnica zbiorów A - B czyli wszystkie elementy zbioru A, poza tymi, które są w zbiorze B (jest też różnica symetryczna taki xor)
- Iloczyn A i B czyli wszystkie elementy będące naraz w zbiorze A i B
- Dopełnienie A czyli wszystkie elementy, które są poza A w jakimś zbiorze dostępnych wartości U (czyli U - A) (U to zbiór wszystkich możliwych wartości)
- Iloczyn kartezjański A x B czyli wszystkie możliwe pary każdy element a R b zebrane w pary uporządkowane.
- Pary uporządkowane to takie, że jeśli (a,b) = (c,d), to a=c i b=d
- Przynależność A e B gdy każdy element A znajduje się również w zbiorze B

Własności działań na zbiorach: Operacje sumy, iloczynu są przemienne

```
A * (B + C) = (A * B) + (A * C)
A + (B * C) = (A + B) * (A + C)
A + (B + C) = (A + B) + C
A * (B * C) = (A * B) * C
A \ B = A * B`
(A * B)` = A` + B`
(A + B)` = A` * B`
```

Potoczna definicja funkcji: Jeśli mamy 2 zbiory X i Y, i stworzymy relację dla każdego X dokładnie jeden Y, to takie przyporządkowanie to funkcja.
Funkcje można składać, np. h(x) = f(g(x)) = (f o g)(x)
Funkcje to relacje, więc można na nich wykonywać operacje mnogościowe, ale nie zawsze wyjdzie z tego funkcja

Relacja to podzbiór iloczynu kartezjańskiego
Relacje mogą mieć wiele własności

- Symetryczna - jeśli x R b, to b R x
- Zwrotna - każde x jest x R x
- Przechodnia - jeśli x R y i y R z, to x R z
- Antysymetryczna - jeśli x R y i y R x => x = y (takie wnioskowanie z symetrii: 2 elementy mogą być ze sobą w relacji dwustronnej, tylko jeśli są takie same)
- Spójna - wszystkie elementy są w parze z wszystkimi innymi (przynajmniej w jedną stronę)

Relacja jest relacją równoważności, gdy jest zwrotna, symetryczna i przechodnia (taki graf Google Maps trochę niekierunkowy)

Rachunek zdań

```
                    implikacja        równoważność (xnor)
ab  a ∧ b  a ∨ b    a => b  b => a    a <=> b (a => b) ∧ (b => a)
00  0      0        1       1         1
10  0      1        0       1         0
01  0      1        1       0         0
11  1      1        1       1         1
```

(implikacji: z prawdy nie może wynikać fałsz)

negacja to wiadomo

zdania są równoważne, gdy mają równe wartości logiczne dla wszystkich możliwości np. a = a * a
tautologia jest zawsze prawdziwa np. p + p` = 1

Mamy kwantyfikatory A; ∀ (ogólny - dla każdego; egzystencjalny - istnieje)

Przykłady:

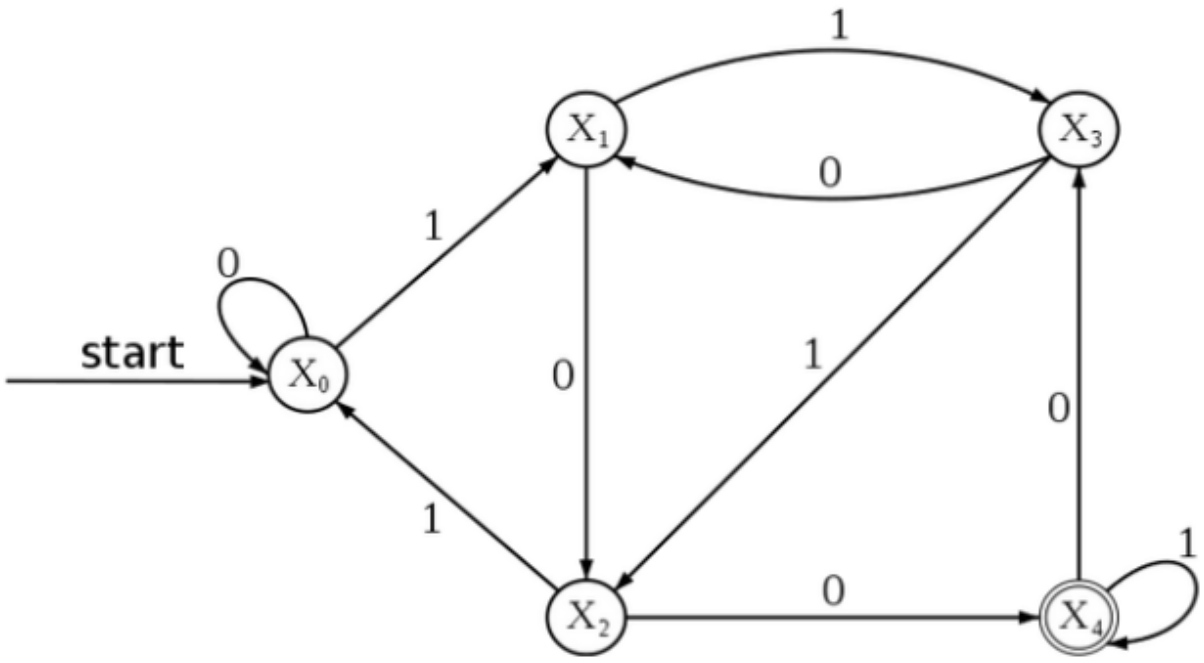
- Ax; x w zbiorze R; x*x >= 0
- ∀x; x w zbiorze R; x = 123.25

Oba to prawda

6. Deterministyczne automaty skończone - definicja, zastosowania

Chodzi o takie fajne rzeczy jak regex czy lekser.

Jest to abstrakcyjna maszyna stanów o skończonej liczbie stanów, która na podstawie aktualnie czytanej litery i aktualnego stanu (na początku pusty) przechodzi do kolejnego stanu. Gdy znajdzie się w stanie oznaczonym jako akceptujący (końcowy), przerywa działanie, klasyfikując czytane słowo/tekst do języka regularnego, do rozpoznawania którego jest zbudowane.



Formalnie, automat skończony to uporządkowana piątka `A = <K, T, M, K0, H>`

- K = niepusty skończony zbiór - stanów
- T = niepusty skończony zbiór - alfabet
- M = relacja przejścia K x T -> K
- K0 e K - stany początkowe
- H e K - stany końcowe/akceptowalne

Lekser jest używany w kompilatorach i interpreterach.
Regex jest wykorzystywany przy wytwarzaniu oprogramowania głównie do walidacji inputu użytkownika, znalezienie odpowiedniego tekstu/tekstów, zamienienia fragmentów tekstu.

Przykłady regexa:

- znalezienie wszystkich słów w tekście (litery między innymi znakami)

```
[a-zA-Z]+
```

- znalezienie wszystkich słów zaczynającego się na "pies"

```
pies[a-zA-Z]*
```

- znalezienie liczb o długości 5

```
\d\d\d\d\d
```

- walidacja maila lvl easy

```
.+@.+\..+
```

7. Przykładowe architektury komputerów: von Neumana, Princeton, Harvard

Komputer składa się z CPU (jednostki centralnej), magistrali (drogi komunikacyjne), pamięci (dysk + RAM). CPU składa się z rejestrów (przechowują szybkie dane do procesora), ALU (matematyka + logiczne operatory) i jednostki sterującej (pobiera instrukcje, dekoduje i steruje ALU i rejestrami).

Ze względu na liczbę strumieni danych a instrukcji powstała Taksonomia Flynna o 4 rodzajach:

- SISD - najprostsze, jeden ciąg instrukcji i jeden danych
- SIMD - np. GPU, te same instrukcje są wywoływane dla różnych porcji danych (ale CPU też może być)
- MISD - dąży do zmniejszenia marginesu błędu przez redundancję obliczeń (bardzo rzadkie, misje kosmiczne)
- MIMD - wiele programów ma różne dane (komputery osobiste)

Podział na architektury:

- Von Neumann i Princeton to dwie różne nazwy na tą samą architekturę komputerów. Charakteryzuje się przechowywaniem instrukcji razem z danymi. Instrukcje mogą łatwo modyfikować inne instrukcje. Problem, jaki występuje, to von Neumann/Princeton bottleneck, czyli ograniczenie wykonywania instrukcji przez czytanie danych (bottleneck). System powinien mieć skończoną i kompletną listę instrukcji.

- Harvardzka: Rozdzielenie instrukcji od danych do osobnych szyn. Łagodzi to wcześniej opisany problem bottleneck.

Większość komputerów i GPU korzysta ze zmodyfikowanej architektury Harvardzkiej, optymalizującej działanie. Logicznie jest to von Neumann, ale fizycznie ścieżki i cache są rozdzielone dla instrukcji i danych.

8. Procesory typu RISC i CISC - charakterystyka, różnice

RISC a CISC to dwa podejścia do projektowania procesorów, różniące się filozofią wobec tego, ile cykli zegara mogą zajmować instrukcje.

CISC (Complex Instruction Set Computing) zakłada, że niektóre operacje mogą trwać kilka do nawet kilkunastu cykli zegara - złożone, specjalistyczne rozkazy. Dodatkowo, instrukcje mogą operować na danych z pamięci, a nie tylko na tych z rejestrów. Jest to swego rodzaju abstrakcja dla programisty. Przez to wszystko dekoder rozkazów jest skomplikowany.

RISC (Reduced Instruction Set Computing) starał się zoptymalizować CISC, po zauważeniu, jak niewielki procent ogólnie operacji faktycznie był wykonywany dłużej niż 1 cykl zegara. Jest to architektura mikroprocesorów. Zmniejsza o rząd wielkość liczbę możliwych operacji do parudziesięciu. Upraszcza to dekoder rozkazów. Każda operacja zajmuje 1 cykl zegara. Dane nie mogą operować bezpośrednio na danych

z pamięci, dlatego stosuje się proces Load, Process, Store. Czyli załadowanie do jednego z wielu rejestrów w takim procesorze (może być ich 30 do ponad setki), przeprowadzenie obliczeń, i zapisanie wyniku.

RISC są szybkie, ale trudniejsze bez abstrakcji CISC dla programisty. Dlatego niektóre procesory (x86) logicznie to CISC, a tak naprawdę mają wewnątrz jednostkę RISC. ARM to rodzaj procesorów, które są RISC, i są znane z małego poboru prądu i szybkości.

Dodatkowe porównanie:

- RISC: 1 zegar i ograniczony tryb adresowania. CISC wykorzystuje wielodostępne tryby adresowania.
- RISC: ustalony format instrukcji 32 bity. CISC: zmienne zakresy od 16-64 bitów na instrukcję.
- RISC: sterowana na stałe bez konieczności pamięci sterującej. CISC kiedyś wymagało pamięci kontrolnej (ROM), ale teraz można tak jak RISC

9 Grafy. Drzewa rozpinające. Cykle Eulera i Hamiltona. Spójność. Algorytmy przechodzenia po grafie

Graf to taka struktura danych, która składa się z wierzchołków, połączonych z innymi wierzchołkami krawędziami. Wierzchołki, krawędzie i same grafy mogą mieć różne własności, w zależności od typu grafu.

Typy grafów:

- etykietowane: wierzchołki są podpisane
- ważone: każda krawędź ma wagę (liczba, zazwyczaj pozytywna, ale może być też ujemna/zerowa, choć niektóre algorytmy przestają wtedy działać)
- skierowane: krawędź ma początkowy wierzchołek i końcowy wierzchołek (mogą być te same, ale jest kierunek)
- nieskierowane: krawędź łączy 2 wierzchołki w 2 strony, można nim przejść w obie strony
- spójne: z każdego wierzchołka można się dostać do każdego innego
- niespójne: istnieją wierzchołki, między którymi nie da się wyznaczyć drogi
- Eulerowskie: ma cykl Eulera. Warunek istnienia: każdy wierzchołek ma stopień parzysty
- Hamiltonowski: ma cykl Hamiltona. Warunek istnienia: NP trudny. Problem komiwojażera, 100 miast = 100!
- regularne: każdy wierzchołek tego samego stopnia, czyli równa ilość krawędzi z każdego wierzchołka

Ważne pojęcia:

- Droga: lista krawędzi od wierzchołka początkowego do końcowego
- Ścieżka: droga co nie powtarza wierzchołków
- Cykl (droga zamknięta): A -> B -> A
- Pętla: A -> A
- Drzewo: graf spójny acykliczny (czyli nie ma żadnego cyklu)
- Drzewo rozpinające: podgraf grafu zawierający wszystkie jego wierzchołki, ale pomijając krawędzie tworzące cykle. Dzięki temu jest niecykliczny i jest spójny. Dany graf może mieć wiele drzew rozpinających. Minimalne drzewo ma minimalną wagę

W drzewach binarnych wierzchołki mają najwyżej (i zazwyczaj dokładnie) dwójkę dzieci. Są one oznaczone jako lewe i prawe. Krawędzie, które nie mają dzieci, to liście (leaf). Korzeniem jest główny rodzic.

Cykle Eulera i Hamiltona:

- Cykl Eulera: można stworzyć drogę, rozpoczynającą i kończąca się w jakimś wierzchołku, i przejść przez wszystkie krawędzie dokładnie raz
- Ścieżka Hamiltona: ścieżka, która przechodzi przez każdy wierzchołek dokładnie raz
- Cykl Hamiltona: ścieżka Hamiltona, ale odwiedza początkowy dodatkowo na końcu

Spójność: chodzi o to, że z każdego wierzchołka można by się dostać do każdego innego.

Algorytmy przechodzenia po wierzchołkach grafu (w akademii zazwyczaj dotyczą drukowania etykiet/indeksów w jakiejś kolejności grafu). Przejście rozpoczyna się od korzenia (lub, dla grafów, wybranego wierzchołka) i w każdej iteracji idziemy do sąsiada. Aby graf nie zapętlił się, notujemy, jakie wierzchołki odwiedziliśmy. 2 rodzaje:

- DFS (depth first search): chciwe, eksplorują ścieżki jak najgłębiej, zanim sprawdzą inne (rekurencyjne). Wykorzystują stos LIFO
    - Dla drzew binarnych są dodatkowe specyfikacje:
        - pre order: zaczyna w głównym wierzchołku i wykonuje dla niego taki sam algorytm jak dla reszty, czyli najpierw operuje na sobie, potem idzie ciągle do lewego dziecka, jak je wyczerpie, to prawe, a potem wraca do rodzica (noo w przypadku korzenia rodzica nie ma, to koniec algorytmu).
        - post order: pre order, ale operacja na wierzchołku dopiero na końcu, najpierw dzieci
        - in order: najpierw wykonuje algorytm dla lewego dziecka, potem wypisuje siebie, potem prawego
- BFS (breadth first search): jakby założenie, że każda ścieżka ma tyle samo potencjału (iteracyjne). Analiza całego poziomu, zanim pójdzie się głębiej. Kolejka
    - często takie drzewa binarne są w postaci idealnie skonstruowanej do takiej iteracji, bo korzeń ma indeks 0, lewe dziecko 1, prawe 2, lewe lewego 3 itd.
    - level order - zwykły bfs

Są inne ale te najważniejsze

Niektóre starają się stworzyć minimalne drzewo rozpinające, czyli dla grafu ważone takie, których suma wag krawędzi jest najmniejsza możliwa (minimalna)

Djikstra to algorytm zachłanny (BFS), mający szybko znaleźć połączenie dwóch wierzchołków o najmniejszej sumarycznej wadze. Działa tylko dla wag nieujemnych

Travelling Salesman problem można wspomnieć (cykl hamiltona!), mrówczano-feromonowe rozwiązanie

VRP to Vehicle Routing Problem - jak zaplanować trasy, aby obsłużyć wszystkich najniższym kosztem. Np. magazyn, klienci, pojazdy o ograniczonej pojemności

10. Pojęcie algorytmu. Algorytmy sortowania. Algorytmy wyszukiwania

Algorytm (definicja nieformalna) to sposób postępowania (przepis), jak rozwiązać zadanie (klasę zadań), podany w postaci skończonego zestawu czynności, ze wskazaniem ich następstwa.

Komputery korzystają z algorytmów, aby przetworzyć przekazane im dane. Każdy poprawny algorytm można przełożyć na zestaw instrukcji dla teoretycznego modelu komputera - maszyny Turinga.

Algorytmy typowo przyjmują parametry wejściowe, modyfikują stan wewnętrzny w komputerze i zwracają wynik. Niektóre programy korzystają z algorytmów, których jedynym zadaniem jest modyfikacja tego stanu wewnętrznego.

Algorytm musi mieć skończoną i kompletną liczbę instrukcji, w języku jednoznacznie zrozumiałym przez komputer. Ludzie komunikują się niejednoznacznie, kod musi być jednoznacznie interpretowany przez komputer. Aby program był obliczalny, jego wykonanie musi być możliwe w akceptowalnym czasie na konkretnej maszynie z określoną mocą obliczeniową i pamięcią. Komputery, z których korzystamy, są skończonym przybliżeniem maszyny Turinga, min. bo mają ograniczoną pamięć. Nowoczesne języki programowania, za to, są Turing complete.

Algorytmy sortowania: często występuje potrzeba uporządkowania obiektów według pewnego klucza/algorytmu. Dla lepszej prezentacji dla człowieka, uporządkowania danych, do innych algorytmów (np. wyszukiwania). Zazwyczaj sortuje się obiekty po jednym z ich atrybutów, często numerycznym. Ale można w wielu językach zdefiniować własną funkcję/lambdę sortującą, której zadaniem jest wzięcie dwóch elementów. Konwencja: zwrócenie 0 jeśli są równe jeśli chodzi o to, po czym sortujemy, x>0 jeśli np. lewy element jest większy, x<0 jeśli prawy. Algorytmy sortowania generalnie wspierają sortowanie rosnąco i malejąco. Algorytmy mogą sortować stabilnie (kolejność oryginalna będzie zachowana dla równych elementów). In place - sortuje bezpośrednio na oryginalnej tablicy.

Przedstawię algorytmy sortowania, posortowane malejąco po ich własności - średniej złożoności czasowej.

- Bubble sort - przechodzi od lewej do prawej, sprawdzając po drodze, czy sąsiedzi są w dobrej kolejności. Jeśli nie, zamienia ich kolejność. Bubble, bo jeśli duża wartość jest na początku, to bąbelkuje do góry.
  Czas: O(n^2) (optymistycznie O(n)). Pamięć: O(1)
- Insertion sort - jak karty w ręce. Bierzemy kartę do posortowanej listy, potem kolejno bierzemy karty, umieszczającym je w odpowiednie miejsce, między jakieś karty.
  Czas: O(n^2). Pamięć: O(1)
- Selection sort: za każdym razem szukamy minimum/maksimum i umieszczamy na końcu/początku posortowanej listy.
  Czas: O(n^2). Pamięć: O(1)
- Quick sort: bierzemy pivota, dajemy wszystkie mniejsze na lewo, wszystkie większe na prawo. Rekurencyjnie.
  Czas: średnia O(n*log n). Pesymistyczna: O(n^2). Pamięć: O(log n), pesymistycznie O(n)
- Heap sort: użycie kolejki priorytetowej przy wykorzystaniu binarnego kopca zupełnego (które mają dostęp łatwy do min i max, szybkie wstawianie i usuwanie elementów - logarytmiczny czas). Najpierw się kopcujesz, potem właściwe sortowanie.
  Czas: O(n*log n). Pamięć: O(n) lub O(1) dla mądrej implementacji
- Merge sort: Ciągle dzielimy ciąg na 2 równe części, aż mamy tylko 1 element (posortowany jest), a potem ciągle w górę i możemy scalać, bo łatwo scalić 2 posortowane ciągi w 1.
  Czas: O(n*log n). Pamięć: O(n)
- Counting sort: bardzo fajny algorytm. Jeśli znamy zakres liczb, to możemy po prostu zrobić listę prewypełnioną zerami o długości max-min (= k). Potem przechodzimy przez każde do posortowania i robimy `sorted[num]++;`. Overkill dla paru liczb, działa na całkowitych liczbach tylko
  Czas: O(n + k). Pamięć: O(n + k)

Algorytmy wyszukiwania:

- liniowy: porównujemy każdy element, aż znajdziemy. O(n)
- logarytmiczny: dzielimy posortowany array na mniejsze połówki, patrząc, gdzie byłby szukany element. O(log n)

Jeszcze inne fajne struktury danych

- hash table - aby znaleźć element, obliczasz hash, i wiesz, że jeśli jest w tabeli, to w liście, która jest pod hashem obliczonym. Generalnie O(1), ale przez to, że kompresuje się różne hashe przy użyciu modulo do 1 listy, może być O(n) (kolizja)
- bst - binarne drzewo przeszukiwań: mamy drzewo, gdzie dla każdego węzła dziecko po lewej jest mniejsze, dziecko po prawej większe. Warto równoważyć drzewa - AVL i Red-Black, to mamy O(log n)
- linked list - lista z obiektów, przechowujących następny i poprzedni element. Bardzo łatwe usuwanie i dodawanie elementów, niezależnie czy w środku itd

11. Podstawy analizy algorytmów. Złożoność obliczeniowa

Algorytmy warto analizować pod kątem ich złożoności obliczeniowej - czasu i pamięci. Wykorzystuje się do tego 3 notacje, oraz pojęcia złożoności średniej, pesymistycznej i optymistycznej. Złożoność średnia liczona jest względem przyjętego rozkładu danych, zazwyczaj jednostajnego. Pesymistyczna zakłada najgorszy możliwy zbiór danych wejściowych. Optymistyczna najlepszy możliwy.

Złożoności przedstawia się jako funkcja od liczby danych wejściowych. Nie jest to dokładna funkcja, ma przedstawić rząd złożoności. Tak więc np. f(n) = (n/100000000000)^3 + 3*n^2 + 92.5 => O(n^3). Można by powiedzieć, że to lim n->nieskończoność. Nie ma również potrzeby podawać bazy logarytmu. Służy więc to głównie porównywaniu algorytmów, ale trzeba mieć na uwadze, że niższa złożoność nie gwarantuje fizycznie szybszego czasu wykonania dla niektórych danych (np. algorytm szybki może wymagać kosztownych operacji matematycznych, przygotowania odpowiedniej struktury danych itd.).

Typowe rzędy złożoności (przykłady czasu):

- O(1): natychmiastowy, np. lista[0]
- O(log n): wyszukiwanie logarytmiczne
- O(n): wyszukiwanie liniowe
- O(n*log n): merge sort (szybkie są, rosną powoli wraz ze wzrostem n)
- O(n^2): bubblesort
- O(n^k): wielomianowa (stałe k)
- O(k^n): wykładnicza (stałe k)
- O(n!): silnia (superwykładnicza - proste rozwiązanie travelling salesman)

Przyjmuje się, że akceptowalna jest <= wielomianowa. Niektóre problemy są jednak prawdopodobnie zbyt skomplikowane, aby opracować takie algorytmy, np. optymalne rozwiązanie travelling salesman. Wtedy innym rozwiązaniem może być szukanie nienajlepszego, ale akceptowalnego rozwiązania (mrówki)

Wykorzystuje się 3 notacje:

- Notacja O: najlepsze górne ograniczenie, czyli jak O(n^2), to algorytm też jest być O(n^3), ale Big O = O(n^2)
- Notacja Omega: asymptotyczne dolne ograniczenie złożoności, czyli lepiej niż O(n) nie będzie, ale może być O(n^2)
- Notacja Theta: ścisłe ograniczenie. Może być tylko, kiedy Omega=O. Wtedy Theta=Omega=O.

12. Warstwowa struktura systemu operacyjnego, pojęcie jądra systemu

System operacyjny to złożony system informatyczny, który zarządzają zasobami sprzętowymi komputera, umożliwia sprawne wykonywanie zadań, tworząc dla nich środowisko i je kontrolując. Pośredniczy między sprzętem a aplikacjami użytkownika.

Główne zadania systemu operacyjnego to:

- Planowanie i przydzielanie czasu procesora poszczególnym zadaniom
- Kontrola i przydział pamięci operacyjnej uruchomionym zadaniom
- Mechanizmy synchronizacji i komunikacji między zadaniami
- Wsparcie systemu plikowego
- Obsługa sprzętu i zapewnienie zadaniom dostępu do niego
- Użycie powłoki do przyjmowania poleceń od użytkownika i wyświetlanie mu informacji zwrotnych

Warstwy systemu operacyjnego:

- Sprzęt: warstwa fizyczna, hardware komputera
- Sterowniki: programy do komunikacji między samym jądrem, a urządzeniami (elementami fizycznymi)
- Jądro systemu operacyjnego (później opisane)
- Powłoka systemowa: program umożliwiający użytkownikowi komunikację z systemem operacyjnym, np. w formie tekstowej (bash, powershell) lub graficznej (gnome, cinammon). Aplikacje komunikują się z jądrem poprzez API i wywołania systemowe.

Jądro systemu realizuje zadania systemu operacyjnego, czyli:

Planista czasu procesora, przełącznik zadań, synchronizacja i komunikacja między zadaniami, obsługa przerwań i urządzeń, obsługa pamięci i jej ochrona.

Są 3 główne architektury jądra:

- jądro monolityczne - ścisłe powiązanie ze sobą wszystkich zadań systemu operacyjnego. Łatwiejsze w stworzeniu, jest szybkie i lepiej zarządza pamięcią, ale mogą też wystąpić problemy przy dużej bazie kodu czy częstszym zatrzymaniem pracy systemu przez np. podłączenie urządzenia, do którego nie ma sterowników albo błąd w sterowniku nadpisze coś gdzie indziej w jądrze i zcrashe'uje komputer. Najważniejszy przykład to Linux
- mikrokernel - podzielenie zadań na wiele różnych serwerów. Zwiększa skomplikowanie, ale zmniejsza to, jak ściśle powiązane są elementy systemu. Rzadki w użyciu do desktopów, częstszy w systemach embedded
- jądro hybrydowe - łączy jądro monolityczne z mikrokernel. Jądro monolityczne dla najważniejszych zadań, które muszą być szybkie, ale delegacja części innych do własnych serwisów. Najważniejszy przykład to Windows, macOS

Są 2 tryby procesora: użytkownika (ograniczony) i jądra (root)

13. Model warstwowy OSI

OSI to teoretyczny model referencyjny do komunikacji internetowej między komputerami. Jest to baza dla innych standardów, np. modelu TCP/IP. Powstał w latach siedemdziesiątych, aby rozwiązać problem ustandaryzowania komunikacji między sprzętami różnych firm.

Składa się z 7 warstw:

- Warstwa fizyczna - zakodowanie danych w postaci bitów. Dodatkowo medium ich transportu, np. kabel RJ45, fale wi-fi.
- Warstwa łącza danych - odbiór i przesyłanie danych, oraz weryfikacji poprawności (CRC) danych. Przypisywany jest adres MAC. Np. ethernet
- Warstwa sieciowa - zarządzanie routowaniem danych do odpowiedniego celu. Dołącza adres IP
- Warstwa transportowa - segmentacja danych oraz przypisanie portu (80 dla HTTP, 443 for HTTPS). 2 metody:
    1. UDP - bez komunikacji zwrotnej, czy otrzymało się pakiet. Przydatne, gdy nie jest krytyczne otrzymać każdy pakiet, np. w niektórych grach czy streamingu
    2. TCP - złożony sposób przekazania informacji, z naciskiem na niezawodne otrzymanie uporządkowanych danych. Kroki: deklaracji chęci otrzymania informacji, gotowości do odebrania, samego przesłania, a na końcu potwierdzenia odbioru. Najczęściej stosowany, gdyż większość komunikacji internetowej jest krytycznie ważna, i trzeba być pewnym, że na każdym kroku nie popełniono błędu
- Warstwa sesji - rozpoczęcie, kończenie, zarządzanie sesją wymiany danych
- Warstwa prezentacji - kompresja i dekompresja danych, zapisanie ich w odpowiednim formacie, kodowaniu i zaszyfrowanie (np. TLS). Deklaracja formatu pliku przesyłanego
- Warstwa aplikacji - np. http vs https vs poczta itd

Swoją drogą, chrome i edge korzystają z tego samego portu 443? Tak, bo tworzą sobie sockety w OS! Stamtąd wysyłają i tam dostają dane

14. Protokoły warstwy łącza danych. Sieć Ethernet. Stos protokołów internetowych TCP/IP

note dla mnie: Datagram (UDP) to jakby segment (TCP). IP pakiet, Ethernet ramka. Dlaczego nie używać adresu MAC zamiast IP? Bo IP jest hierarchiczne i z drugiego końca świata wiadomo, do kogo uderzać po kolei, a MAC to jak nazwa człowieka/adres domu bez miasta i ulicy

Warstwa łącza danych to druga warstwa modelu OSI. Zaimplementowana jest w warstwie dostępu do sieci w modelu TCP/IP. Protokoły w tej warstwie przemieniają pakiety w ramki. Wykorzystywane są różne protokoły warstwy łącza danych, między innymi Ethernet, Wi-fi, PPP. Wykorzystywany jest też ARP do mapowania adresów IP na adresy MAC (ARP request identyfikuje, do kogo dokładnie przesłać wiadomość, ta informacja jest cache'owana do tabelki ARP, trzeba uważać na man in the middle ARP spoofing).

Ethernet to rodzina technologii działającej na warstwie łącza danych oraz fizycznej z modelu OSI. Jest to standard dla sieci lokalnej LAN. Ethernet wykorzystuje i opisuje ramki, schemat okablowania, złącza jak końcówki RJ45 w warstwie fizycznej OSI, aby przekazać dane, np. od routera do komputera.
Ethernet wykorzystuje współcześnie topologię fizyczną gwiazdy (switch w centrum - przełącznik eliminujący kolizje), choć logicznie działa jak magistrala (broadcast)
Ethernet 2 (teraz najczęściej używany) korzysta z ramki z adresem MAC, określającej fizyczny adres urządzenia, typ transmisji i CRC do detekcji błędów.

TCP/IP to model oparty na OSI, upraszczający go do 4 warstw. Stos protokołów internetowych TCP/IP składa się z 4 warstw

- Warstwa dostępu do sieci: połączenie warstwy fizycznej i łącza danych. Przypisanie adresu fizycznego MAC. Przypisanie ramek
- Warstwa internetowa: odpowiednik warstwy sieciowej z OSI. Przypisanie adresu logicznego IP. Protokoły IPV4, IPV6. Podział na pakiety
- Warstwa transportowa: implementacja warstwy transportowej z OSI. Przypisanie portu (np. 80 dla HTTP, 443 dla HTTPS). Tutaj występują największe różnice między TCP a UDP. Podział na segmenty.
    - TCP: protokół zapewniający otrzymanie danych i ewentualną retransmisję w przypadku błędów. Gwarantuje też kolejność. Jednostka: segment
    - UDP: protokół nie gwarantujący otrzymania wszystkich danych, za to znacznie szybszy. Przydatny, gdy program będzie działać w przypadku utraty części danych (np. streamowanie filmu). Jednostka datagram
- Warstwa aplikacji: wykorzystuje protokoły HTTP, SMTP - email, DNS - nazwy domen. Jedyna warstwa, z jaką użytkownik ma bezpośredni kontakt.

15. Protokoły warstwy aplikacji

Warstwa aplikacji to najwyższa warstwa w modelu OSI czy TCP/IP. Jest to jedyna warstwa, z jaką użytkownik ma prawie bezpośrednio do czynienia.

Stosuje się tu wiele protokołów, do przeglądania internetu, maili, zdalnego terminala, zdalnego systemu plików. Oto część najważniejszych:

- HTTP (port 80) - służy do przesyłania hipertekstu (WWW, API). Protokół bezstanowy
    - RESTful: bezstanowość, zasoby pod URL, używanie odpowiednich metod HTTP (GET, POST, DELETE, PUT...)
- HTTPS (port 443) - szyfrowana przez TLS wersja HTTP. Ukrywa url, body, headery requestów, nie szyfrując tylko domeny na początku
- DNS (port 53) - tłumaczy znane ludziom adresy stron np. onet.pl na adres ip tej strony (głównie UDP)
- POP3 - do odbioru maili
- IMAP - POP3 + zarządzanie zdalnymi folderami na serwerze
- SMTP - do przesyłania maili wraz z załącznikami
- FTP - interaktywne przesyłanie plików między serwerami
- NfS - udostępnianie systemów plików (dysków sieciowych)
- TELNET - emulacja zdalnego terminala na innym komputerze, niebezpieczny przez przesyłanie jawnego tekstu
- SSH (port 22) - szyfrowana emulacja zdalnego terminala na innym komputerze
- DHCP (dynamic host configuration protocol) - dynamiczna konfiguracja adresów IP, bramy, adresów serwerów DNS urządzeń

16. Techniki efektywnego programowania - przykłady

Na bazie labów z TEP:

- Pliki należy dzielić na .h z deklaracją struktury klasy (nazwa, metody i podział na publiczne i prywatne, pola) i .cpp z implementacją metod
- Zmienne, klasy i metody należy nazywać deskryptywnie, unikać jedno literowych nazw zmiennych, prócz iteratorów np. (for size_t i = 0;)

- Należy stosować wszędzie jedną konwencję kodowania, dotyczącą się nazywania struktur, zmiennych, ale też pilnowania wcięć (automatyczne formattery kodu)
- Klasa nie powinna zawierać metod niepowiązanych ściśle z nią (np. BigInteger z metodą makeUppercase)
- Tak samo metoda powinna skupić się na 1 rzeczy, a niepowiązane wydzielić do innych metod
- Klasa nie powinna mieszać różnych warstw, np. BigInteger nie powinien jednocześnie implementować BigInteger, jak i metod jej drukowania/pobierania danych od użytkownika (metody publiczne od tego)
- Jak najmniej (najlepiej zero) pól publicznych. Dane takie należy setować konstruktorem i metodami, oraz otrzymywać metodami
- Unikać domyślnych wartości pól, lepiej ustawić je jako domyślne w konstruktorze czy przypisać wartość w metodach
- W C++ do metod przekazywać obiekty przez referencję/wskaźnik, zamiast bezpośrednio, bo wtedy obiekt się kopiują (a jego wskaźniki same się nie skopiują)
- Korzystanie z inteligentnych wskaźników w nowoczesnym C++, aby zautomatyzować destrukcję obiektów, gdy inne obiekty przestaną na ten obiekt wskazywać
- Korzystanie z std::move w celu uniknięcia kopiowania danych (przenosi je, stary obiekt jest w stanie jakby pustym, bezpiecznym do destrukcji)
- Nie zwalniać pamięci dwukrotnie dla tego samego obiektu
- Implementacja i używanie algorytmów o niskiej złożoności
- Zamiana rekurencji na iterację (brak ryzyka stack overflow + szybsze, bo alokowanie ramek trwa i wymaga pamięci)
- Parallelizacja kodu przy użyciu wątków
- Zarządzać pamięcią - zero wycieków - każdy obiekt alokowany dynamicznie trzeba usunąć używając delete, listy delete[] (ale czasem elementy listy trzeba jeszcze wcześniej usunąć osobno)
- Program powinien być odporny na każdy możliwy input użytkownika
- Unikać break (poza switchem) czy continue, bo psują one naturalny przebieg funkcji. Często można to załatwić, korzystając z odpowiedniej rodzaju loop, albo wydzielić do osobnej funkcji i wykorzystywać return. Kontrowersyjna opinia prowadzącego - unikać wyjątków, bo one też utrudniają śledzenie przebiegu.
- Absolutny zakaz goto
- Zaprzyjaźnianie klas - friend tylko wtedy, kiedy jest faktycznie potrzebne
- Jak najmniej mutowalnych zmiennych globalnych - najlepiej 0. Const jak najbardziej jest ok, trzeba go tylko wydzielić do pliku .h z constami albo do odpowiedniej klasy .h. Unikać mutowalne, bo ciężko się to debuguje, powstaje "spaghetti code" i ciężej parallelizować
- Jedna klasa = 1 plik header, chyba, że są mocno powiązane, to można
- Jeśli ten sam blok instrukcji powtarza się, należy go wydzielić do funkcji lub metody. Nawet jeśli się nie powtarza, ale jest to osobna funkcja od reszty, należy to wydzielić i dać deskryptywną nazwę. Moją zasadą jest to, że prawie każdy komentarz można zastąpić wydzieleniem do funkcji czy metody.
- O kod i architekturę należy dbać, a dodanie i utrzymywanie testów sprawi, że refactoring jest znacznie mniej bolesny (często jest się mądrym po pełnej implementacji, i wtedy można wprowadzić abstrakcje)
- Usuwać nieużywane zmienne, klasy, metody - z kontrolą wersji nic nie zginie
- Używać zautomatyzowanych testów na różnych poziomach. Testy jednostkowe do funkcji, integracyjne do połączeń między modułami, funkcjonalne do przypadków użycia, E2E dla skomplikowanych przypadków użycia i pewności, że aplikacja działa po paru akcjach. Każdy test niższego poziomu jest wielokrotnie szybszy, dlatego np. E2E należy ograniczyć do minimum (ale warto mieć choć parę).

17. Zarządzanie pamięcią. Typowe problemy. Wskaźniki

W C++ programista ma bezpośredni dostęp do zarządzania pamięcią, inaczej niż w językach jest C# czy Python, gdzie jest garbage collector, a każdy typ jest opakowany w obiekt.

Są dwa miejsca alokacji. Alokacja dynamiczna przydziela na stercie (heap), a automatyczna na stosie (stack).

Automatyczna alokacja polega na zwykłym zadeklarowaniu zmiennej, np. inta czy nawet klasy, bez użycia keyworda new. Takie zmienne alokowane są w bloku, i kiedy blok zostanie zakończony, dla obiektów z tego stosu wywołany będzie automatycznie destruktor. Jest to więc wygodne dla programisty, ale ogranicza czas życia tylko do tej jednej funkcji.

Dynamiczna alokacja pozwala na tworzenie obiektów długo żyjących oraz przekazywanie im innym metodom, ale jest znacznie trudniejsza z perspektywy programisty. W C++ to programista zarządza poprawną alokacją i dealokacją dynamicznych obiektów. Używa się do tego wskaźników. Do alokacji używa się głównie keyworda `new`, a do destrukcji `delete`, np.

```
int* foo = new int;
*foo = 5;
delete foo;
// bez delete memory leak
```

Tutaj na stosie stworzona zostanie zmienna foo, i dzięki new int zaalokuje sobie odpowiednio dużo miejsca na int (ileś bajtów) i zapamięta adres. Potem weźmie ten adres, interpretując go jako int zapisze tam int 5. Potem po delete foo ze sterty zostaną usunięte dynamicznie alokowane dane. Potem foo zostanie usunięte ze stosu (delete nie usuwa zmiennej, tylko daje znać, że ten adres jest teraz nieużywany i dostępny do alokacji).

W przypadku, gdy programista zapomni dealokować dynamicznie alokowane dane, zostaną one zwolnione przez system operacyjny dopiero po zakończeniu działania całego programu. Tworzy to memory leak, który może z czasem wyczerpać zasoby komputera.

W C++ nowo alokowane zmienne nie czyszczą zaalokowanej sobie pamięci, czyli znajdują już się tam bity. Trzeba na to uważać zwłaszcza przy wskaźnikach, które bez odpowiedniej inicjalizacji mogą wskazywać na adresy niezalokowane przez komputer dla programu (crash) lub zmieniając taką zmienną, zmienić przypadkiem dane dla zupełnie innej zmiennej,

Arraye w C++ działają jak wskaźniki w wielu kontekstach. Jeśli odwołamy się do elementu poza listą, np. elementu 5. dla 3 elementowej listy, operujemy po prostu na pamięci o adresie = adres arraya + 5 (`5 * sizeof(int)` dla listy intów). Powoduje to ten sam error lub błąd, co opisany wyżej.

Arraye mogą być arrayem wskaźników w C++. Tak samo wskaźnik może wskazywać na wskaźnik - trzeba dodać odpowiednią liczbę gwiazdek do kodu. Używając delete, trzeba pamiętać o usunięciu danych z każdego stopnia wskaźnika, inaczej mamy memory leak. Do destrukcji arraya trzeba użyć `delete[]`, ale ta sama uwaga jest aktualna dla arraya arrayi (arraya wskaźników) - trzeba dla każdego elementu delete/delete[] wykonać.

18. Dobór paradygmatów programowania do rozwiązywania problemów informatycznych

Paradygmat określa sposób myślenia, aby rozwiązać problemy jako programista. Najważniejszy podział paradygmatów to podział na paradygmat deklaratywny i imperatywny. W deklaratywnym instrukcje dotyczą tego, jaki wynik chcemy osiągnąć (rezultat), a imperatywne, jakie kroki ma dokładnie wykonać program, aby osiągnąć wynik (kroki).

najważniejsze podparadygmaty paradygmatu deklaratywnego:

- Funkcyjny: brak mutowalnego stanu. Program to wywołanie funkcji, a każdą funkcję można w każdym przypadku zastąpić jej wartością (bo nie ma skutków ubocznych). Funkcje to obywatele pierwszej rangi: funkcje mogą zwracać funkcje, przyjmować je jako argumenty czy mogą być częścią samych struktur danych. Plusy: jasny kod, łatwiejszy w utrzymaniu, brak problemów z współbieżnością. Minusy są takie, że nie nadaje się do każdego problemu (np. gry) i wymaga innego sposobu myślenia. Przykłady języków to Ocaml czy Scala.
- Logiczny: pojęcie ograniczeń, przydatny do matematyki, formalny. Przykłady to np. Prolog, dziś jest mniej używany.
- SQL i jemu podobne również należą do paradygmatu deklaratywnego (ale ani ściśle logiczny ani funkcyjny).

Najważniejsze podparadygmaty paradygmatu imperatywnego:

- Proceduralny: logiczny podział kodu na procedury (funkcje), co zwiększa czytelność i zmniejsza ryzyko błędu.
- Obiektowy: jest zarówno strukturalny, jak i proceduralny. Zapisanie wielu struktur jako obiekty, stworzone z klas. Klasy to instrukcje, jak stworzyć obiekt. Intuicyjny paradygmat, gdyż łatwo zrozumieć, że instancja klasy ma własny stan (wartość atrybutów) i własne metody, które mogą zmieniać jej stan. Bardzo popularny paradygmat.
- Event-driven - wydarzenia (wywołane przez użytkownika lub nie) są obsługiwane przez klasy i funkcje, dzięki czemu powiązania są mniej ścisłe.

Należy pamiętać, że granica między oba paradygmatami się zaciera. Współcześnie w bardzo wielu popularnych językach imperatywnych wprowadzono wiele możliwości z paradygmatu funkcyjnego. Np. w Pythonie, bezproblemowe jest przekazanie funkcji jako argument, tworzenie lambd itd. W językach funkcyjnych również często możliwe jest używanie zmiennych, ale wtedy nie są czysto funkcyjne.

Aby wybrać odpowiedni paradygmat, należy wziąć pod uwagę następujące czynniki:

- doświadczenie zespołu
- wielkość projektu (obiektowy dobry do dużych)
- wymogi współbieżności (funkcyjny)
- wymogi szybkości działania (imperatywny bliżej jest działania komputera)
- wymóg udowodnienia działania (funkcyjny, logiczny)

19. Programowanie funkcyjne a programowanie imperatywne

Paradygmaty w programowaniu dzielą się na 2 główne paradygmaty i ich podparadygmaty.
Jednym z tych 2 paradygmatów jest paradygmat deklaratywny, a drugim imperatywny.

- Programowanie funkcyjne wywodzi się z paradygmatu deklaratywnego, więc jest na innym "poziomie" niż imperatywne.
- Paradygmat imperatywny dzieli się na podparadygmaty, jak strukturalny (bloki kodu, bez goto), aż po proceduralne (podział na procedury) i obiektowe.

Paradygmat deklaratywny skupia się na rezultacie - programista podaje to, czego oczekuje jako wynik i to komputer tworzy listę kroków, które wykona. Programowanie imperatywne skupia się na krokach, jest to więc lista instrukcji, jakie programista podaje komputerowi.

Paradygmat funkcyjny to jeden z podparadygmatów deklaratywnego (inne to np. logiczny). Program to wywołanie 1 funkcji, która:

- nie ma skutków ubocznych (brak stanu, które funkcje modyfikują, output jest deterministyczny z inputu - przezroczystość referencyjna)
- każdą funkcję można zastąpić jej rezultatem
- nie zmienia się wartości zmiennych (immutability), ale można tworzyć nowe
- funkcje to obywatele pierwszej klasy, co oznacza, że funkcje mogą otrzymywać inne funkcje jako argumenty, dynamicznie je tworzyć, zwracać jako wynik

Często wykorzystywany jest mechanizm rekurencji (funkcja wywołuje samą siebie, ale z innymi argumentami). Języki, które są funkcyjne, to np. Ocaml albo Scala. Paradygmat jest mniej popularny niż imperatywne alternatywy, gdyż wymaga innego sposobu myślenia, a także nie każdy program da się w nim zaprojektować. Może być mniej wydajny (duży stos przez rekurencję - tail call optimization pomaga, ciągłe tworzenie nowych zmiennych zajmuje czas), gdyż paradygmat imperatywny jest bliżej tego, jak działają komputery. Dużym plusem jest możliwość udowodnienia działania, jest to dobra metoda do rozwiązań mocno matematycznych.

Paradygmat imperatywny dzieli się na różne podparadygmaty. Wykorzystują zmienne, pętle, modyfikowalny stan. Opiszę proceduralny oraz obiektowy.

- Proceduralny: podział programu na procedury (funkcje), otrzymujące argumenty, zwracające wynik i modyfikujące globalny/lokalny stan. Znacznie zwiększa czytelność i ułatwia współpracę z innymi programistami, umożliwia pisanie większych baz kodu. Jednak trudno utrzymać duże bazy kodu, w których lepszym paradygmatem będzie obiektowy.
- Obiektowy: również wykorzystuje funkcje, ale zawiera je wewnątrz obiektów (metody). Obiekty powstają z klasy, która jest instrukcją, jak wytworzyć daną strukturę danych. Obiekty mają atrybuty (mutowalny stan wewnętrzny) i metody (funkcje, operujące na konkretnej instancji klasy). Paradygmat obiektowy jest intuicyjny, bo w rzeczywistości również mamy obiekty, które mają własności, a także mogą wykonywać czynności / mogą być na nich wykonywane czynności. Dobry do pracy w dużych projektach. Da się w nim zrobić praktycznie każdą aplikację (webową, desktopową, mobilną, grę, złożony systemy...)

Należy wybrać paradygmat funkcyjny, gdy:

- gdy zespół posiada takie doświadczenie
- rozwiązujemy problem stricte matematyczny
- potrzebujemy łatwo wspieranej współbieżności (nie ma stanu, więc nie ma możliwości desynchronizacji wątków)
- potrzebujemy dowodu działania

Należy wybrać paradygmat imperatywny, gdy:

- gdy zespół posiada takie doświadczenie
- tworzymy program z bardzo zmiennym stanem
- projekt ma być duży
- projekt bazuje na modelowaniu rzeczywistych encji i ich interakcji
- potrzebujemy bogatego ekosystemu bibliotek z języków jak C++, C#, Python

Na końcu wspomnę, że granica między paradygmatami się zaciera. Większość imperatywnych języków wspiera funkcyjne elementy (lambdy, przekazywanie funkcji jako argumentów, zwracanie ich...), a w językach funkcyjnych można dodać mutowalny stan, choć wtedy nie są czysto funkcyjne.

20. Abstrakcyjne typy danych i ich realizacja w językach programowania

Wiele języków implementuje podstawowe, przydatne struktury danych w bibliotekach standardowych. Ich implementacja nie jest jawna i ważna dla dewelopera - najważniejszy jest kontrakt ADT, który jest zadeklarowany. Czyli to, jakie są dozwolone metody (np. Get, Add, Pop...) oraz gwarantowane zachowania (np. pierwszy wchodzi pierwszy wychodzi). Różne implementacje mogą używać tego samego kontraktu, aby być lepiej przystosowanym do specyficznych przypadków użycia i optymalizacji - tak więc kontrakt całkowicie ignoruje implementację. W wielu językach można wykorzystać do tego słowa kluczowe `interface` albo `abstract class`, i różne implementacje dziedziczą po klasie abstrakcyjnej lub implementują interfejs.

List, czyli rozszerzalną i modyfikowalną tablicę elementów, można zaimplementować na wiele sposobów. Częstą implementacją listy jest stworzenie pod spodem tablicy, która gdy zostanie wypełniona, jest rozszerzana o ileś elementów, zachowując bufor. Inną implementacją, wspierającą szybkie usuwanie/dodawanie elementów na początku i środku, jest linked list. Zaimplementowana jest w postaci grafu, gdzie węzły są połączone z następnikami (i często z poprzednikami). Dla programisty ważne jest to, iż mimo bardzo różnią się implementacyjnie, może wykorzystać ich wspólny kontrakt/interejs. Np. w C# jest interfejs IList, który jest implementowany przez List, jest też LinkedList, ale implementuje ICollection.

Stack: pierwszy wchodzi, ostatni wychodzi. Czyli element dodany jako ostatni jest pierwszy w kolejce do pozyskania. KLasa Stack

Queue: pierwszy wchodzi, pierwszy wychodzi. Czyli elementy najdłużej będące w strukturze danych są pierwsze w kolejce do pozyskania. W tym przypadku w C# jest PriorityQueue, który nadpisuje domyślne zachowanie - ustala kolejkę wychodzenia na podstawie "priorytetu" ustalonego wcześniej dla obiektu. Implementacje Queue (FIFO), PriorityQueue, ConcurrentQueue (bezpieczna wątkowo).

Hash Map / Dictionary: zawiera unikalne klucze, do których przypisana jest wartość. Gwarantuje szybki dostęp do wartości dla kluczy. W C# ADT to IDictionary, implementowane przez Dictionary, SortedList (klucze posortowane), ConcurrentDictionary (wielowątkowe).

Warto zauważyć, że ADT to teoria, ale język może powstrzymywać dewelopera przed niepoprawnym użyciem struktury danych. Np. LinkedList nie implementuje interfejsu IList, ponieważ indeksowanie jest wolne w pętli (a iterator mógłby być cyrkularny).

21. Algorytmy identyfikacji obiektów statycznych. Analityczne i numeryczne metody optymalizacji

Obiekty statyczne to obiekty o stałej strukturze właściwości, które można zmierzyć. Wyjście zależy wyłącznie od aktualnego wejścia, a nie od czasu czy historii.

Algorytm identyfikacji obiektu statycznego służy do tego, aby na podstawie analizy danych dla pewnych statycznych obiektów wytworzyć model przewidujące pewne właściwości w zależności od parametrów wejściowych - czyli obiekt identyfikacji. Np. mamy obiekty statyczne: zdjęcia kotów zapisane w odpowiednim formacie, wykorzystując ten sam standard RGB, i chcemy przewidzieć rasę kota ze zdjęcia.

Algorytm ten składa się z kroków.

- Pierwszy to określenie obiektu identyfikacji: zrozumienie problemu, własności fizycznych. Wynikiem tego punktu jest określenie kształtu wektora danych wejściowych i wyjściowych, oraz określenie charakteru zakłóceń mogących wpływać na kształt danych. Zrozumienie celu badań
- Określenie klasy modeli: dobranie odpowiedniego modelu (regresja liniowa, model fizyczny, sieć neuronowa) do problemu, na podstawie jego charakterystyk. Warto przeprowadzić analizę zjawisk fizykochemicznych lub analizę wymiarową (analiza sugerująca sposób przekształcenia/korelacje między danymi). Do problemu rozpoznania ras kotów prawdopodobnie najlepiej nadawałaby się sieć neuronowa
- Organizacja eksperymentu - 2 sposoby: bierna i czynna. W czynnej przeprowadzamy dodatkowo planowanie eksperymentu, związane z identyfikacją danych, które mogłyby zakłócić algorytm przez niereprezentatywne własności, np. obrazki o zbyt niskiej jakości, dane, które po analizie numerycznej można określić jako błędne dane - anomalie. Określenie parametrów eksperymentów takich jak długość serii pomiarowej i ustalenie technik eksperymentu - anotatorów, wybór narzędzia pomiarowego
- Opracowanie algorytmu identyfikacji - wybieramy sposób szukania modelu, np. optymalizacja wag sieci neuronowej gradientowo albo użycie algorytmu genetycznego. Także dobranie funkcji i metody kosztu (F1 score, entropia)
- Realizacja algorytmu identyfikacji - napisanie kodu, który implementuje algorytm

Metody optymalizacji dotyczą znalezienia minimum/maksimum funkcji, dla podanych ograniczeń, jednej lub wielu zmiennych. Podam przykłady działania dla szukania minimum, bo do tego są często wykorzystywane (szukanie jak najmniejszej wartości dla funkcji kary/kosztu). Każdą funkcję można też przekształcić, aby szukać w niej maksimum zamiast minimum tym samym sposobem, co dla szukania minimum.

Analityczne metody produkują dokładny wynik, ale wymagają zaawansowanych umiejętności matematycznych dla skomplikowanych funkcji (np. z wieloma parametrami), lub stają się wtedy zupełnie niemożliwe. Polegają na operacji przekształceń funkcji.

Metody analityczne:

- Klasyczna metoda to wyznaczanie ekstremum - znalezienie minimum/maksimum funkcji dla wejść, gdzie pochodna funkcji wynosi 0.
- Metoda mnożników Lagrange - mając ograniczenie równościowe, możemy znaleźć punkty, gdzie gradient funkcji = lambda * gradient ograniczenia. Następnie należy rozwiązać to równania i sprawdzić punkty kandydujące znajdujące się w ograniczeniu
- Metoda Kuhn-Tucker uogalnia metodę Lagrange do ograniczeń nierównościowych.

Numeryczne metody optymalizacji nie produkują zazwyczaj dokładnego wyniku, ale nadają się do prawie każdej funkcji. Działają na operacjach na liczbach zamiast przekształceń funkcji. Zazwyczaj są wykonywane

przez komputer, z racji wymagania wielokrotnego obliczania funkcji w wielu pętlach. Metody można podzielić na te bez ograniczeń i z ograniczeniami.

- Metoda podziału odcinka: dla funkcji jednoargumentowych, odrzucamy iteracyjnie przedziały, gdzie funkcja wydaje się rosnąć
- Gradientowe: obliczamy pochodną, i podążamy w jej kierunku (dla maksimum, dla szukania minimum w kierunku antygradientu). Szeroko wykorzystywane w optymalizacji przez sieci neuronowe funkcji kosztu.
- Algorytmy genetyczne i ewolucyjne: ileś osobników, każdy z własnym "genotypem" czyli zbiorem liczb które wstawiamy do funkcji, iteracyjnie usuwa się słabsze osobniki zastępując je krzyżówką mocniejszych / genetyczną mutacją pojedynczych. Celowo wprowadza się elementy losowe, aby wyjść z minimów lokalnych.

Funkcje kary dodają do funkcji kosztu wartość powiązaną z tym, jak daleko punkt wychodzi poza ograniczony zakres. Funkcja bariery dodaje składnik do funkcji rosnący do nieskończoności wraz z zbliżaniem się do barier (ograniczeń).

22. Specyfika Internetu Rzeczy, obszary zastosowań, rozwiązywanie problemów z adresowaniem dużej liczby urządzeń, ich rozproszeniem i bardzo dużą ilością generowanych danych

Internet rzeczy dotyczy integracji urządzeń elektronicznych (rzeczy) z kategorii embedded, które są połączone w sieć i przesyłają między sobą pomiary oraz reagują na sygnały.

Obszary zastosowań są bardzo szerokie, są to np.:

- Przemysł - kontrola linii produkcyjnej, monitorowanie i zarządzanie procesami produkcyjnymi, wczesne wykrywanie awarii, usprawnianie procesów
- Transport - zarządzanie flotą pojazdów, optymalizacja tras, monitorowanie warunków pogodowych, czujniki w dużych pojazdach jak samoloty/statki
- Zdrowie - monitorowanie stanu pacjenta, zarządzanie systemami opieki zdrowotnej
- Smart home - różne czujniki, sterowanie światłami, wentylacją, temperaturą i warunkami roślin itd.
- Elementy miejskie - sygnalizacja świetlna, tablice z rozkładem jazdy, systemy kolejek

Urządzenia mają mikrokontrolery i każde ma warstwę fizyczną, są podłączone do internetu przez Wi-Fi, kable Ethernet, LPWAN, sieci komórkowe

Z racji dużej liczby urządzeń, występują problemy z wyczerpaniem dostępnych adresów IPv4, gdyby przypisać je statycznie. Jest na to parę rozwiązań:

- Główny serwer ma pulę dostępnych adresów IP. Udostępnia on adres urządzeniu dynamicznie tylko wtedy, kiedy musi się skontaktować przez internet, a po zakończeniu komunikacji adres wraca do puli
- Używanie IPv6 dla znacznie większej puli adresów
- Adresowanie przy użyciu unikalnego adresu fizycznego MAC
- Adresowanie przy użyciu przypisanych nazw za pomocą DNS mapującego nazwę na IP

Dodatkowo, samo adresowanie (przypisanie początkowego adresu IP) w dużych systemach IoT jest wyzwaniem. Do tego przydaje się DHCP (dynamic host configuration protocol)

Z racji, że urządzenia są rozproszone na bardzo różne obszary - w innych państwach i regionach, w obszarach ze słabą/zawodną siecią Wi-Fi, wykorzystuje się szereg rozwiązań

- Wykorzystanie platformy chmurowej dużych firm, jak AWS od Amazon czy Microsoft Azure
- Wykorzystywanie kabli do niezawodnego połączenia, Wi-Fi, Bluetooth, GSM, LTE, w zależności które ma najwięcej sensu i połączenie paru sposobów

Trzeba pamiętać o odpowiednim zaprojektowaniu sieci, aby wspierała dużą przepustowość. Jeśli chodzi o bezpieczeństwo, nie zapominać o autoryzacji i autentykacji urządzeń, szyfrowaniu np. TLS,

Z racji, że urządzenia mogą generować ogromne ilości danych (np. czujniki w samolocie terabajty w skali lotu), wykorzystuje się szereg rozwiązań:

- Wykorzystanie platformy chmurowej jak Microsoft Azure czy AWS, które oferują ogrom miejsca na zapisanie danych
- Wykorzystanie edge computing (przetwarzanie na urządzeniach na granicach sieci), aby filtrować/agregować dane przed przesyłką dalej
- Lokalizacja usług analitycznych fizycznie blisko urządzeń

23. Rozwiązania sprzętowe wspierające komunikację i protokoły komunikacyjne wykorzystywane w sprzęcie wbudowanym i Internecie Rzeczy

Rozwiązania sprzętowe rozróżnia się w zależności od wielkości sieci, w której urządzenia muszą się komunikować. Wyróżnia się 4 grupy:

- PAN (Personal Area Network) i HAN (Home Area Network). Maksymalne odległości od urządzeń w tej grupie to od paru centymetrów (RFID, NFC) do kilkunastu metrów (Bluetooth Low Power, Wi-Fi). Dotyczy urządzeń znajdujących się na ciele i wewnątrz ciała, sensorów i urządzeń w smart home / laboratorium domowym. Dzięki bardzo niskiemu zasięgowi, komunikacja wymaga bardzo mało energii, dlatego urządzenia mogą wyjątkowo długo działać na baterii.
- LAN (Local Area Network) - dotyczy skali biurowca / zakładu pracy. Główne rozwiązania to Wi-Fi, Ethernet, bardzo ewentualnie Bluetooth 5. Wymaga więcej energii niż PAN, dlatego urządzenia są zwykle podłączone do sieci energetycznej.
- WAN (Wide Area Network) - skala obszarów geograficznych - miast, regionów. Zwyczajowo ograniczone do GSM (2G, 3G, 4G, 5G), ale od pewnego czasu udało się stworzyć rozwiązania pobierające znacznie mniej prądu, jak rodzina technologii LPWAN

Protokoły komunikacyjne w warstwie aplikacji:

- AMQP (Advanced Message Queue Protocol) - połączenia punkt-punkt. Exchange, Message Queue i Binding. Skuteczna, niezawodna wymiana wiadomości - TCP
- CoAP (Constrained App Protocol) - ograniczona przepustowość, UDP do lekkich wiadomości klient - serwer HTTP
- DDS (Distributed Data Service) - peer to peer, zarządza wszystkim setupem i łączeniem sieci
- MQTT (Message Queue Telemetry Transmission) - subscribe and publish, protokół wykorzystujący zdarzenia. Bardzo prosty i oszczędny energetycznie. Najbardziej popularny w IoT, idealny do broadcastowania informacji. TCP, może być uspany

Protokoły komunikacyjne w warstwie transportowej:

- TCP - stara się zagwarantować dostarczenie, bezstratny, zajmuje więcej czasu przez handshake's, dzieli dane na uporządkowane segmenty
- UDP - nie oczekuje informacji zwrotnej, czy wiadomość została dostarczona. Bardzo prosty i wydajny

Warstwa sieciowa:

- 6LoWPAN - bezprzewodowa sieć osobista o niskim poborze mocy IPv6. Kompresja i enkapsulacja nagłówków. Niski pobór mocy
- IP - IPv4 lub IPv6

Warstwa łącza danych:

- LPWAN - low power wide area network - 500 metrów do 10 km, zoptymalizowane prądowo

Warstwa fizyczna:

- Bluetooth: szeroko stosowane, wariant niskiego poboru mocy to Bluetooth Low Energy
- NFC - bardzo krótki zasięg, różnego rodzaju karty
- RFID - może zasilać pasywne tagi energią z fali radiowj, komunikując się z nimi
- Ethernet - kabelki

Przykłady hardware to Raspberry PI (cały komputer z Python), Arduino - ESP32 (C)

24. Modele baz danych. Relacyjna baza danych. Normalizacja. Transakcje

Modele baz danych:

- Hierarchiczny: przypomina system plików / drzewo. Dane przechowywane są w dokumentach (rekordach), a każdy dokument ma dokładnie jednego rodzica (prócz korzenia). W przypadku, gdy dokument dostanie drugiego rodzica, dokument jest kopiowany pod tego rodzica, i każdy ma już jednego rodzica tylko. Usunięcie węzła usuwa również wszystkie jego dzieci
- Sieciowy: hierarchiczny, ale umożliwia relację N-N. Informacja w dokumentach oraz przebiegu połączeń sieci
- Obiektowy: bliski paradygmatowi obiektowemu z języków programowania, dane opierają się na obiektach
- noSQL:
    - Dokumentowa: dane zawarte są w dokumentach i przechowywane jako JSON / BSON (binarny JSON). Przypominają gotową formę, która jest wysyłana przez API. Dokumenty mają klucz i bazy danych wspierają język zapytań, pozwalający przesłać tylko potrzebne informacje. Grupowanie dokumentów w wiadra, kolekcje itd. Dokumenty mogą mieć i tak różne pola typowo
    - Grafowa: składa się z węzłów i połączeń, a same połączenia również są ważną częścią bazy danych. Przydatne w przypadku skomplikowanych struktur danych
    - Klucz-wartość: każdy klucz jest unikalny, i wartość jest szybko wyszukiwalna. Czyli tabela z kolumnami klucz i wartość. Hashmapa. Koszyk zakupów, informacje o sesji
    - Rodzina kolumn: przechowuje dane po kolumnach zamiast po wierszach, przydatne, gdy np. analiza wykorzystuje tylko część kolumn, albo jak struktura jest rozproszona i szerokie wiersze
- Relacyjny

Relacyjna baza danych to fundament nowoczesnych baz danych, wykorzystując język SQL w jego różnych wariantach, aby większość produktów IT działała.

Opiera się na relacjach i związkach między nimi. Relacja to jest tabela, czyli struktura zawierająca atrybuty (kolumny) oraz wiersze (krotki). Nazwa atrybutu w skali tabeli musi być unikalna i mieć ustalony typ danych. Kolejność atrybutów jest bez znaczenia. Każdy wiersz opisuje wszystkie atrybuty w relacji. Superklucz to zbiór atrybutów identyfikujących wiersz, klucz kandydujący to jeden z superkluczy, klucz główny to zazwyczaj 1-2 kolumny identyfikujące wiersz powstaje z klucza kandydującego. Związki między tabelami polegają na kolumnach odwołujących się do kluczy głównych innych tabel (klucze obce).

Stosuje się 3 różne typy związków:

- Jeden do jeden (1-1) - max jedna tabela do max jednej tabeli (w przypadku braku połączenia wykorzystać NULL), np. users: user_id, name; user_profile: profile_id; unique user_id, profile
- Jeden do wielu (1-N) - max jedna tabela do 0-wielu tabel. Czyli users: user_id, name; transactions: transaction_id, user_id, money...
- Wiele do wielu (N-N) - wiele tabel może być połączone do wielu tabel. Realizowane przez użycie tabeli asocjacyjnej z dwoma kolumnami - klucze A i B. Typowy przykład to studenci i kursy, czyli students: student_id; courses: course_id, student_course: student_id, course_id

Postać normalna służy do tego, aby przekształcić źle zorganizowaną bazę danych z redundancjami/anomaliami w prostą w utrzymaniu i usunięciu redundancji. Aby być w x NF, trzeba być w x-1 NF

Postacie normalne:

1. Pierwsza postać normalna wymaga użycia klucza głównego, dane atomowe, brak atrybutów wielowartościowych
2. Druga postać normalna: wydzielenie kolumn do tabel, które zależne są w całości tylko od części klucza złożonego, na przykład źle orders: product_id, buyer_id, x buyer_name, x product_price. Jeśli klucz to tylko jedna kolumna, to 1 NF = 2 NF.
3. Trzecia postać normalna: wydzielić każdą kolumnę zależną od innej kolumny niż klucz do osobnych tabel (przechodnia zależność), np. źle employees: employee_id, department_id, department_name. Brak zależności przechodnich.

Jest więcej postaci normalnych (4 NF, 5 NF), ale deweloperzy zazwyczaj ograniczają się do trzech. Trzeba też pamiętać, że niekiedy warto niestety zdenormalizować krytyczne dla wydajności tabele, ale takie wypadki trzeba dokładnie badać. Ale często przyspiesza to przez lepsze indeksy, mniej danych do przetworzenia

Transakcje:

Niektóre czynności na bazie danych relacyjnej powinny być wykonane w całości albo w ogóle. Na przykład kupując coś, ale po drodze okaże się, że jednak brakuje pieniędzy, odrzuca się całą transakcję, nie wykonując żadnej czynności.

CAP: Consistency, Availability, Partition tolerance (odporność na podział sieci). W systemie rozproszonym da się spełnić max 2 z 3. SQL zawsze wybiera C + A/P, NoSQL raczej AP

ACID (SQL):

- atomic - każda operacja to osobny byt i wszystko albo nic
- consistency (spójność) - każdy stan i przejście są poprawne

- izolacja - równolegle uruchomione transakcje nie wpływają na siebie i są izolowane (jakby sekwencyjne). Są 4 poziomy izolacji, aby zapobiec brudnemu czy niepowtarzalnemu odczytowi (od najsłabszego do najmocniejszego, ale najwolniejszego):
    1. Read Uncommited
    2. Read Commited. Domyślny w większości SQL
    3. Repeatable Read. Domyślny np. w MySQL
    4. Serializable - prawdziwie sekwencyjne. Bardzo wolne MVCC: czytający nie blokują piszących, a piszący czytających (parę wersji baz danych)
- durability (trwałość) - dane są permanentne i na dysk, nawet w przypadku awarii

BASE (noSQL):

- Basically Available - bez izolacji, priorytet na odpowiedzi, nawet nie w pełni poprawne
- Soft State - nie jest ciągle spójne
- Eventually consistent - ale w końcu będzie spójne

25. Język SQL. Charakterystyka. Podjęzyki

Język SQL (Structured Query Language) jest używany w praktycznie każdej relacyjnej bazie danych. SQL jest językiem deklaratywnym wysokiego poziomu – użytkownik opisuje co chce uzyskać, a silnik bazy decyduje jak to wykonać, korzystając z optymalizatora i indeksów. Charakteryzuje się prostą, czytelną składnią.

Typowa kwerenda SQL to

```sql
SELECT user_name AS "First name", department_name
FROM user
LEFT JOIN department ON department.department_id = user.department_id
WHERE age > 60
```

Podstawowe elementy języka to polecenia, klauzule, wyrażenia, predykaty, średniki kończące polecenie, komentarze (uważać na SQL Injection przy nieparametryzowanych zapytaniach)

Język wspiera użycie indeksów (drzewa binarne, bo wspierają <, > i =), optymalizator zapytań + execution plan automatycznie optymalizuje kwerendy. Język ma wiele wariantów, różniących się lekko składnią, wspieranymi typami i konkretnymi bazami danych, w których działają. Jest wykorzystywany wyłącznie do komunikacji z relacyjną bazą danych.

Dodatkowe cechy: ograniczenie do tylko określonych typów danych (bez np. list - 1 NF) typu INT, VARCHAR, CHAR, NUMERIC, FLOAT, DATE + wiele więcej

Łączenie tabel polega na użycia słowa kluczowego `JOIN`, z możliwościami użycia `LEFT JOIN` (wszystko z lewej nawet jeśli null), `RIGHT JOIN` (analogicznie w prawo), `INNER JOIN` (spełnia warunek po obu stronach). Wielkość liter słów kluczowych czy wcięcia nie mają znaczenia. Można aliasować nazwy używając `AS 'xyz'`. Najczęściej używany jest FROM aby wybrać tabelę, SELECT aby wybrać zwrócone atrybuty, WHERE aby filtrować, GROUP BY aby grupować, HAVING aby filtrować grupy, wcześniej omówione JOIN.

Prócz samego wybierania danych, SQL wspiera dodawanie wierszy, usuwanie, aktualizowanie, dodawanie tabel, tworzenie baz danych, użytkowników, zarządzanie bezpieczeństwem, transakcjami i wiele więcej

SQL ma parę podjęzyków, każdy odpowiedzialny za część funkcjonalności.

- DQL (query) - SELECT + klauzule jak WHERE, GROUP, HAVING - używane do kwerendowania danych
- DML (manipulation) - INSERT, UPDATE, DELETE - używane do tworzenia, aktualizowania i usuwania wierszy. TRUNCATE: szybki DELETE bez logowania
- DDL (definition) - CREATE, DROP, ALTER - zarządzanie strukturami jak tabele, indeksy
- DCL (control) - GRANT, REVOKE - zarządzanie użytkownikami i ich dostępami
- TCL (transaction control) - COMMIT, ROLLBACK - zarządzanie transakcjami

26. Modele cyklu życia oprogramowania

Modele cyklu życia oprogramowania dotyczą filozofii wobec podziału zadań dotyczących tworzenia oprogramowania na części składowe. Typowo dotyczą one podziału na etapy jak planowanie, implementacja, testy, a także poziom współpracy z biznesem czy podział na dostarczane produkty.

Pełny cykl życia oprogramowania dotyczy nie tylko fazy wytwarzania, ale składa się z następujących etapów:

1. Inicjacja - koncepcja, analiz biznesowa, wykonalności
2. Wytwarzanie - development
3. Eksploatacja i utrzymanie - długie i drogie. Naprawianie błędów, udoskonalanie, portowanie do nowych wersji, prewencja zestarzenia się
4. Wygaszanie - migracje danych, utylizacja, powiadomienie użytkowników

Modele SDLC (Software Development Life Cycle)

- Waterfall (kaskadowy/wodospad) - tradycyjny model wytwarzania oprogramowania. Dzieli się na ściśle określone etapy jak zbieranie wymagań, analiza wymagań, projektowanie, implementacja, testy i wdrożenie. Model jest ciekawy, ale trudny do wykonania w praktyce przez wiele powodów. Głównym powodem jest to, jak zmienne są wymagania - zbieranie ich to długi proces, wymagania zawsze zmieniają się z czasem, niezrozumienie potrzeb biznesowych kończy się niepraktycznym produktem. Innym powodem jest to, że traci się czas przez tak sztywny podział na etapy, które potem i tak są choć częściowo mieszane ze sobą. Dziś dumnie odchodzi się od waterfalla, ale czysty waterfall jest po prostu niemożliwy do spełnienia w rzeczywistości. Dodatkową praktyczną wadą jest możliwość poczucia klienta, że jest odsunięty od projektu, a po miesiącach może oczekiwać czegoś innego. Waterfall ma sens w systemach krytycznych lub tych, gdzie zmiany są bardzo drogie.
- Iteracyjny - najpierw ogólna analiza wymagań, a potem osobne waterfalle dla dostarczania produktu w iteracjach
- Spiralny - próba formalizacji podejścia iteracyjnego. Dodaje analizę ryzyka w każdej iteracji (monitorując uwagi użytkownika). Cykliczne powtarzanie planowania, analizy ryzyka, konstrukcja (mały waterfall), ocena przez klienta
- V - waterfall, ale z rozbudowaną fazą testów z podziałem na testy modułów, integracyjne, walidacyjne i akceptacji, co zapewnia znacznie wyższą jakość produktu.
- Prototypowy - dostarczamy prototyp za prototypem (szybko stworzone przybliżenie produktu o niskiej jakości), aby na końcu użyć innego modelu do stworzenia porządnej aplikacji. Wymagania będą wtedy bardzo dojrzałe i gotowe do implementacji

W 2001 powstał manifest Agile (zwinny), w którym twórcy zaznaczyli 4 punkty, wynikające z ich doświadczenia i zrozumienia, że wymagania będą się ciągle zmieniać, a spełnienie ich w ostatecznej formie da klientowi większą konkurencyjność.

Oto punkty:

- LUDZIE ponad procesy i narzędzia
- DZIAŁAJĄCY SOFTWARE ponad dokumentacją
- WSPÓŁPRACA Z KLIENTEM ponad renegocjacją kontraktu
- AKCEPTACJA ZMIANY zamiast wykonywania planu

Idea jest taka, że elementy po prawej są ważne, ale po lewej ważniejsze.

Scrum był z jedną z metod, która istniała przed manifestem Agile, ale po nim została uznana jako dobra, generalna implementacja Agile. Zakłada przyrostowe dostarczanie produktu klientowi, przydzielając do najbliższego sprintu (1-4 tygodnie) zadania do zrobienia. Występuje Scrum Master starający się usuwać blokady i wspierać zespół. Produkt Backlog to uporządkowana lista wymagań (user stories). Do każdego sprintu występuje planowanie sprintu (stworzenie Sprint Backlog) i recenzja oraz retrospekcja sprintu. Codziennie jest Daily Scrum (standup/daily), gdzie zespół się synchronizuje. Niestety, scrum czasem degeneruje się do micromanagement'u deweloperów.

Scrum, jak i agile, charakteryzuje się częstym feedbackiem od biznesu, szybką reakcją na zmiany wymagań, nacisk na działające oprogramowanie i samoorganizację zespołu.

27. Metodyki wytwarzania oprogramowania

Metodyki wytwarzania oprogramowania to całe filozofie inżynierii oprogramowania, czyli proces obejmujący wszystkie czynności systematycznie wykonywane w celu wytworzenia oprogramowania. Każda organizacja ma unikalne szczegóły wpływające na metodykę. Metodyki stosuje się, aby stworzyć jakościowy produkt, zmniejszyć ryzyko niepowodzenia, zwiększyć dogłębne zrozumienie projektu i umożliwić pracę zespołową. Każda metodyka ma następujące cechy:

- Zakres: które fazy wytwarzania oprogramowania są objęte metodyką, a także jakie role i aktywności są definiowane
- Rozmiar: liczba elementów kontrolowanych przez metodykę (dokumentacja, opisy technik, miary jakości)
- Ceremoniał (stopień formalizacji) - jak ważne i precyzyjne są dokumenty wytwarzane podczas wytwarzania oprogramowania
- Komunikacja z klientem: jak przebiega
- Widzialność: łatwość oceny, czy projekt jest wykonywany zgodnie z metodyką
- Waga: iloczyn rozmiaru i ceremoniału. Podział na lekkie/zwinne i ciężkie.
- Paradygmat: strukturalny czy obiektowy, ale to bardziej historycznie.

Metodyki ciężkie charakteryzują się dużą liczbą ról, artefaktów, dokumentacji. Proces wytwórczy bardzo ważny. Wymaga wysokiej dyscypliny. I najważniejsze: mniejsza podatność na zmiany wymagań.

Przykłady metodyk ciężkich:

- RUP (rational unified process) - iteracyjna i przyrostowa metodyka w IBM. Skupia się na wysokiej jakości wykonania, architektura oparta o komponenty, iteracyjne opracowywanie produktu, kojarzy mi się z UML. Dzieli się na 4 fazy (po każdej przekazanie klientowi):
    1. rozpoczęcie - sformułowanie zadania biznesowego i opracowanie wstępnego modelu przypadków użycia
    2. opracowanie - opracowanie architektury systemu, użytkowników, ról, plan całego projektu

3. konstrukcja - budowa komponentów
4. przekazanie - szkolenie użytkowników, testy akceptacyjne

Metodyki lekkie/zwinne powstały w sprzeciwie do ciężkich. W manifeście Agile z 2001 roku wymieniono 4 punkty:

- LUDZIE I INTERAKCJE ponad procesy i narzędzia
- DZIAŁAJĄCY SOFTWARE ponad skrupulatną dokumentację
- WSPÓŁPRACA Z KLIENTEM zamiast renegocjacji kontraktu
- AKCEPTACJA ZMIAN zamiast wykonywania planu

Przykłady metodyk lekkich

- AUP (agile unified process) - przemienienie RUP w agile. Cykl życia sekwencyjny w długiej perspektywie, iteracyjny w małej. Opuszczenie części artefaktów i ról z RUP
- Extreme programming - do małych/średnich projektów o wysokim ryzyku, gdzie nie wiadomo, jak dokładnie i czy da się dostarczyć rozwiązanie, tylko cześć wymagań. Pominięcie ceremoniałów i dokumentacji. Komunikacja ustna. Artefakty = kod + testy. Prosty projekt, ciągłe testowanie, standardy kodowania, ciągły kontakt z klientem. Dla programisty: tdd (test driven development, nie ufaj testowi, który nigdy nie był fałszywy), pair programming. Dla zespołu: continuous integration, collective code.
- Scrum - nie jset pełną metodyką wytwarzania, narzuca jedynie sposób organizacji pracy. Podział projektu na sprinty (1-4 tygodnie). Efektem sprintu jest namacalna nowa wersja z nowymi funkcjonalnościami. Product backlog - user stories czekające na implementację. Sprint Backlog - zadania do zrobienia przez sprint. Scrum Master - pilnuje poprawnego wykonywania scrum i rozwiązuje konflikty. Sprint planning - spotkanie planujące sprint. Sprint Retrospective - retrospekcja, sprint review - recenzja. Ważnym elementem są Daily Scrum do synchronizacji. Ciągły kontakt z klientem i przywitanie zmian z otwartymi ramionami.

28. Zastosowanie list, zbiorów i słowników w języku Python

Listy, zbiory i słowniki to podstawowe struktury danych w Pythonie.

Lista: tablica ze zmienną długością, przechowująca obiekty. Każdy typ w Pythonie jest obiektem, a listy nie mają przypisanego typu, więc można mieszać niepowiązane ze sobą typy w liście (choć trzeba pamiętać, że technicznie są powiązane, bo każde dziedziczy po podstawowym typie Object). Listę w Pythonie deklaruje się bardzo prosto: jako dwa nawiasy kwadratowe, oraz opcjonalnie elementy po przecinku. Np.

```
moja_lista = [1, "pies", 3]
```

Różne struktury danych można również castować do listy i zapisać je w takiej formie. Aby wytworzyć listę populowaną liczbami od 1 do 100 z krokiem 5, można napisać

```
moja_lista_2 = list(range(1, 100, 5))
```

Listy stosuje się do przechowania obiektów w odpowiedniej kolejności, bez zwracania uwagi na cechy obiektów (takie jak ich typ czy to, że są duplikaty).

Można stosować podstawowy operatory na dwóch listach, np. wygodnym sposobem dodania list jest

```
list_1 = [3, 4]
list_2 = [1, 2]
list_2 += list_1
```

Lista implementuje również operatory mutacji jej, jak append do dodania elementu na końcu, pop usuwające element o danym indeksie (lub na końcu), remove usuwające dany element i więcej. Na listach można również wywołać funkcję sorted() i dostać wynik posortowany wg. naszej lambdy. Listę można również skonstruować używając składni typu

```
[i * i / 2 for i in range(0, 10)]
```

Dostęp po indeksie to O(1).

Zbiory (set): kolejna struktura danych kolekcyjna. Tworzy się metodą set, przyjmującą kolekcję. Jest nieposortowana i nie przyjmuje duplikatów, to znaczy dodanie elementu do set doda go tylko, jeśli się tam nie znajduje. Trzeba nadpisać magiczne metody `__eq__` i `__hash__`, jeśli chcemy traktować 2 instancje klasy o tych atrybutach jako równe, bo dla instancji klas porównanie będzie nie na podstawie ich wartości, a tego, czy reference'ują ten sam obiekt. Szybkie sprawdzanie, czy obiekt jest w set: O(1). Listę czasem przemienia się w set chociaż na chwilę, by pozbyć się duplikatów. Można nawet pisać

```
list_1 = list(set(list_1))
```

Słownik: zbiór danych typu unikatowy klucz -> wartość. Metoda dict lub użycie `{ "Polska": "polacy", "Niemcy": "niemcy" }`. Bardzo szybki dostęp do uzyskania wartości dla klucza dzięki implementacji hash mapy - optymistycznie O(1), pesymistycznie O(n). Aby hashmapa działała, klucze muszą być niemutowalne - hashowalne. Często używany w zadaniach algorytmicznych do zwiększenia optymalizacji. Z `collections` można zaimportować `defaultdict`, aby przypisać domyślną wartość, dla kluczy których nie ma

```
d = defaultdict(list)
print(d["moj-klucz"])
> []
```

Bez obaw, że klucza wcześniej nie było. Parę metod iteracji z uwagi na jej charakter, można

```
for value in my_dict.values():
  ...
for key, value in my_dict.items():
  ...
```

note: kolizje hashy rzadkie, ale jak są, to wywołanie __eq__

29. Różnice i podobieństwa języków Java i Python

Python i Java to języki wysokiego poziomu, wieloplatformowe i używane na całym świecie przez miliony ludzi. Mają dużo różnic i podobieństw.

Podobieństwa:

- Wysokiego poziomu - nie pozwalają na zarządzanie programiście pamięcią
- Oba mają Garbage Collector do usuwania nieużywanych adresów pamięci
- Wieloplatformowe, wykorzystując przejściowe etapy między kodem maszynowym a własnym kodem (o nich później w różnicach)
- Silne typowanie - zmienne mają typy i nie są niejawnie konwertowane na inne, jak np. w JavaScript

Różnice:

- Typowanie statyczne dla Javy - zmienne mają określone typy, typowanie dynamiczne dla Pythona - typ zmiennych może się zmieniać, przez co mogą wystąpić błędy przy operacjach niedozwolonych na danym typie. W Pythonie dodano typowanie - można ale nie trzeba przydzielić zmiennym, funkcjom, parametrom dozwolony typ/typy, ale jest to tylko informacja do sprawdzenia przy statycznej analizie kodu - nie ma wpływu na działanie
- "boilerplate" kodu - dla Javy przyjęło się pisać objętościowo dużo kodu, oraz samo napisanie tych samych funkcji zajmuje więcej miejsca, niż dla Pythona. Dodatkowo Python ma dużo bibliotek i bogatą bibliotekę standardową, umożliwiającą skracanie niektórych wyrażeń. Ogólnie czas wytwarzania oprogramowania jest krótszy w Pythonie, również dzięki specyfikacji wybieranych frameworków
- Paradygmat: Java mocno obiektowy, i ma bogaty system słów kluczowych dla klas, dziedziczenia, interfejsów, modyfikatory dostępu itd. Python wspiera różne paradygmaty: również bardzo często jest wykorzystywany obiektowo, ale też funkcyjnie czy ogólnie imperatywnie. Python jest prostszy, jeśli chodzi o obiektowość - nie ma np. interfejsów, a multiinheritance - wielokrotne dziedziczenie. Występuje duck typing i można zaimportować klasy abstrakcyjne. Chociaż takie abstrakcje mogą być mniej intuicyjne dla programisty. Nowocześnie Java wprowadziła elementy funkcyjne, jak lambdy.
- Java jest kompilowana do kodu bajtowego, wykonywany przez maszynę wirtualną JVM. Python jest skryptowy - interpretowany. Też jest kompilowany do kodu bajtowego, ale nie jako jawny proces przed uruchomieniem
- Pamięć: inicjalizacja JVM zajmuje dużo pamięci, ale potem reszta programu stosunkowo mniej niż Python
- Szybkość: Java jest znacznie szybsza dzięki kompilacji do JIT. Aby przyspieszyć Python, można napisać biblioteki w językach niższego poziomu, jak np. C
- Wielowątkowość: Java wspiera prawdziwy multithreading, a w Pythonie jest GIL ograniczający możliwość wątków działających równoległe na wielu rdzeniach w jednym procesie

Zastosowania: obie do backendu. Python do data science, ML, nauki, skryptów. Java, poza backendem, do skomplikowanych systemów jak bankowe, też okazjonalnie do aplikacji desktopowych i mobilnych.

30. Zasady programowania równoległego w języku skryptowym Python

Proces a wątek: proces to program z własną przestrzenią adresową. Wątki działają w ramach jednego procesu, mogą współużywać pamięć i może być ich wiele dla procesu. Stworzenie wątku dla komputera ma

mniejszy narzut niż procesu, ale oba mają różne zastosowania. Dzięki wątkom i procesom możemy osiągnąć szybszą i bardziej responsywną aplikację, przez zrównoleglenie obliczeń, czytania pliku czy odpowiedzi na requesty HTTP.

GIL (global interpreter lock) - specyficzne dla Python. Sprawia, że naraz tylko 1 wątek może wykonywać kod Python. Dlatego używanie wielu wątków w Python nie przyspieszy programu, jeśli jedyne, co wątki robią, to wykonywanie kodu Python przez CPU. Przyspieszy, gdy limitem jest z zewnątrz, np. GPU, operacje I/O. Dlatego w Python wątki przydają się do operacji I/O (dysk, sieć, baza danych), a procesy do zadań CPU-intensive.

Główne zagrożenia

- Jeden z nich to równoległy dostęp do pliku, gdy różne wątki/procesy czytają i nadpisują naraz plik. Przez to, na przykład od momentu czytania pliku do zapisania danych mógł on się zmienić.
- Race condition występuje, gdy poprawny wynik zależy od poprawnej kolejności wykonywania zadań przez wątki/procesy. Programista może źle przewidzieć, kiedy które procesy/wątki się skończą, przez co program zadziała w sposób nieprzewidziany. Dodatkowo wątki dzielą pamięć, i jest ryzyko nadpisanie globalnych stanów przez nie w sposób nieprzewidziany.
- Deadlock (zakleszczenie): wątki blokują się nawzajem, oczekując na zasoby wzajemnie zajęte.

Do programowania równoległego w Pythonie wykorzystuje się 3 biblioteki: threading, asyncio i multiprocessing

- Threading / concurrent.futures.ThreadPoolExecutor: wątki. Executor: przypisujemy maksymalną liczbę zadań, wysyłamy zadanie przez e.submit albo e.map dla wykonania funkcji dla listy np.
- Asyncio: wielozadaniowość kooperacyjna. Czyli chodzi o kontrolowanie przepływu sterowania programu tak, aby nie zatrzymywał się tylko na wykonywaniu pewnego zadania, a ignorował np. wybranie opcji przez użytkownika. Czyli jeden wątek, pętla zdarzeń, zadania same oddają sterowanie (bo czekają na coś). Lżejsze niż wątki/procesy, ale nie wykonuje kodu równolegle
- Multiprocessing / concurrent.futures.ProcesPoolExecutor: procesy. Executor: przypisujemy maksymalną liczbę zadań, wysyłamy zadanie przez e.submit albo e.map dla wykonania funkcji dla listy np. Aby przekazać dane między procesami, używa się Queue/Pipe, serializujących dane

Aby uniknąć wyścigów, stosuje się:

- lock - zapewnia wyłączny dostęp do zasobu
- queue - bezpieczna współbieżnie metoda przekazywania danych
- semaphore - ogranicza liczbę wątków mających dostęp do zasobu

31. UML jako język specyfikacji projektu. Diagramy i ich zastosowanie

UML (Unified Modelling Language) to język pół-formalny, służący do modelowania różnych systemów informatycznych, mający wspomóc projektowanie systemu i służyć jako dokumentacja. Półformalny, ponieważ jest to bardziej zbiór narzędzi z mniej lub bardziej ogólnymi instrukcjami, jak je wykorzystać. Ujednolicił sposób opisywania dziedziny problemu. Reprezentacja graficzna wielu diagramów pozwala szybko wykryć pewne problemy. UML to nie język programowania, jest abstrakcyjny i agnostyczny od języka, ale niektóre diagramy najlepiej pasują do paradygmatu obiektowego.

UML wykorzystuje bardzo wiele diagramów, o różnym poziomie szczegółowości / spojrzenia na system. Najważniejszy podział to na statyczne (opis struktury danych, użytkowników, itd.) i behawioralny (interakcje

między elementami, rozkład w czasie, logika)

Najważniejsze diagramy statyczne:

- Diagram klas: powiązany z paradygmatem obiektowym. Przekłada się w dużej mierze na klasy, jakie będą w projekcie i w bazie danych, a także na relacje między nimi. Bardzo ważny
    - miedzy obiektami występują różne relacje/związki. Na każdym końcu mają mnogość relacji (1 użytkownik ma wiele zakupów). Klasy mogą być powiązane (asocjacja). Szczególne przypadki asocjacji to agregacja (obiekt wykorzystuje inny, ale mogą istnieć osobno) i kompozycja (cykl życia obiektu jest zależny od innego - rodzica, usuwasz rodzica - usuwasz childa)
    - często podaje się, prócz nazw klas, atrybuty, metody, można podać typy, modyfikatory dostępu i wiele więcej
    - generalizacja = dziedziczenie
    - klasy asocjacyjne
- Diagram obiektów: szczegółowe modelowaniu możliwych instancji obiektów dla klas z diagramu klas
- Diagram pakietów: logiczna struktura systemu (namespaces)
- Diagram wdrożenia: fizyczna struktura systemu, przydatny dla DevOps, jak wdrożyć projekt, z jakich elementów składa się technicznie

Diagramy behawioralne:

- Diagram przypadków użycia jest bardzo przydatny dla biznesu, jako diagram nietechniczny. Pokazuje, jacy aktorzy (rodzaj użytkownika) mogą osiągnąć jakie cele w systemie. Powinien być mało techniczny, a każdy cel powinien przynosić wartość biznesową użytkownika (jakiś zysk dla niego)
- Diagram interakcji: ważna podkategoria diagramów behawioralnych. Modeluje przepływ sterowania z naciskiem na komunikację między obiektami.
    - Diagram sekwencji: jak dany przypadek użycia może być technicznie spełniony, często przez konkretne warstwy w architekturze oprogramowania
- Diagram aktywności: przedstawia sekwencyjne i współbieżne przepływy sterowania i danych między czynnościami. Przypomina diagram blokowy
- Diagram maszyny stanów: opisuje możliwe stany obiektu i jego przejścia (transakcja - opłacony, do opłacenia, odrzucony...)

Minusem UML jest to, że różne osoby mogą mieć przeciwstawne wizje, jak diagramy powinny być modelowane. Diagramy UML zajmują dodatkowo dużo czasu do wykonania. W procesie modelowania można odkryć różne informacje o wymaganiach biznesowych, ale w przypadku faktycznie nowych wymagań, występuje potrzeba modyfikacji diagramu. W dzisiejszych projektach o metodyce zwinnej wykorzystuje się tylko część diagramów, często o niskiej formalności

32. Wzorce architektoniczne i projektowe – klasyfikacja, przykłady, zastosowania

Wzorce to sprawdzone sposoby na rozwiązanie problemów w inżynierii oprogramowania. Wprowadzając abstrakcje lub używając standardowych wzorców, inni deweloperzy nie muszą analizować implementacji, albo znają dobrze te konkretne wzorce.

Wzorce dzielimy na 2 główne kategorie. Architektoniczne dotyczą całej bazy kodu, i tworzą jasną i przewidywalną strukturę dla modułów / całego projektu. Wzorce projektowe rozwiązują problemy na poziomie klas, wprowadzając abstrakcję, aby schować implementację, pomagając z zależnościami i więcej.

Przykłady architektoniczne:

- Fizyczna
    - Klient - baza danych: frontend (klient) ma prostą/średnio skomplikowaną logikę, i ma bezpośredni kontakt z bazą danych. Ok do małych projektów, CRUD-owych
    - Architektura trójwarstwowa (zazwyczaj monolit): klient - logika biznesowa - baza danych: frontend nie ma żadnej logiki biznesowej, tylko wyświetla informacje od backendu (logiki biznesowej). Backend stoi między klientem a bazą danych - w nim znajduje się cała logika biznesowa, endpointy, komunikatory z bazą danych. Baza danych stoi jako osobny byt od backendu, odpowiadając na jego żądania. Uniwersalne i bardzo szeroko stosowane podejście. Nadaje się do małych i średnich projektów, przy dużych mogą pojawić się problemy z wydajnością i koordynacją dużego zespołu.
    - Mikroserwisy: odpowiedź na problemy monolitu. Dzieli system na serwisy działające osobno i komunikujące się ze sobą przez HTTP/kolejki. Serwisy mogą być skalowane przy integracji z chmurą, dzięki czemu system powinien lepiej odpowiadać na dużo zapytań dla konkretnego części aplikacji. Zazwyczaj każdy serwis ma własną bazę danych i są utrzymywane przez różne zespoły. Dobre do dużych, skomplikowanych systemów. Minusem jest skomplikowana infrastruktura, opóźnienia sieciowe między serwisami, mniejsza spójność danych. Utrudnione debugowanie, problem spójności danych między serwisami
- Logiczne
    - Warstwowa: system dzieli się na warstwy, gdzie pierwsza to faza prezentacji, jedyna dostępna dla użytkownika. Charakteryzuje się zależnościami w dół, czyli np. baza danych może istnieć bez żadnych wyższych warstw.
    - Heksagonalna: istnieje centralny punkt projektu (z logiką biznesową), który wystawia porty (interfejsy). Te interfejsy są implementowane przez inne projekty, np. frontend, bazę danych, inne serwisy. Łatwa wymiana projektów
    - Interaktywne
        - MVC - model, view, controller. Podstawowy wzorzec logiczny. Model: dane + logika biznesowa. View: np. HTML. Controller: odbiera input, aktualizuje model i wybiera widok.
            - MVC pasywny: model zmienia się tylko pod wpływem działań użytkownika
            - MVC aktywny: model sam może się zmienić (powiadamia kontroler o tym, Obserwator)
        - MVP - prezenter zamiast controllera - view dostaje model tylko od prezentera
        - MVVM: data binding view z viewmodel. View pokazuje dane, zmiana danych w ViewModel update'uje model, który update'uje ViewModel. ViewModel update'uje View.
        - Model - view - viewmodel - controller: z Ten Square Games, używany w Unity/GameDev. Model przychodzi z bazy danych i jest readonly. Controller ma model, i zawsze nasłuchuje na jego zmiany. Controller wystawia eventy, dzięki czemu jak user coś kliknie w view, to controller reaguje. View zarządza renderowaniem na podstawie viewmodelu, który dostaje od controllera. Viewmodel to model przetłumaczony na tylko dane potrzebne view, w odpowiednim formacie.

Przykłady projektowe:

- Kreacyjne:
    - Abstrakcyjna fabryka: klasa abstrakcyjna, np. ButtonFactory. Mamy różne buttony dla Mac i Windows, to tworzymy MacButtonFactory i WindowsButtonFactory, zwracające różne rzeczy. Ale widok nie widzi implementacji
    - Singleton: często definiowany jako zły wzorzec - anti-pattern. Singleton sprawia, że klasa udostępnia statyczne pole Instance z instancją siebie, dzięki czemu dostęp jest ekstremalnie prosty. Ale tak jak service locator, ukrywa on, ile klasy mają zależności. Bardzo łatwo mieszać wtedy zależności, mocno skomplikować flow logiki i utrudnić/uniemożliwić testowanie.
        - Dependency injection: odpowiedź na singleton. Jawne przekazywanie zależności w konstruktorze klas. Łatwo można wtedy testować klasy, przekazując mockowane zależności.
    - Builder: ułatwia budowanie obiektów, oferując metody, aby za pomocą serii wywołań metod wybrać dokładnie funkcjonalności danego obiektu.
- Strukturalne:
    - Adapter: serwis dostosowuje się do innego schematu, tworząc adapter, gdzie "tłumaczymy" metody z nowego schematu do starego.
    - Fasada: serwis ukrywa złożoność systemu za prostym interfejsem
    - Decorator: funkcja opakowywuje funkcję (albo klasa klasę bez dziedziczenia)
- Behawioralne:
    - Obserwator: inna klasa subskrybuje na zmiany innej, np. jak w TSG Controller nasłuchuje na zmiany modelu
    - Polecenie: klasa zamiast osobnych metod na funkcjonalności, ma 1 metodę otrzymującą polecenie danego typu + dane
    - Strategy: jasno deklarowane różne strategie. Przydatne np. w algorytmach Tabu Search różne heurystyki

33. Metody ochrony danych

Ochrona danych (treści przechowywanych, przetwarzanych i przesyłanych w systemie) to jedna z podstaw cyberbezpieczeństwa w inżynierii oprogramowania.

Ochrona danych opiera się na 3 filarach - triada CIA (Confidentiality, Integrity, Availability):

- Poufność: całkowity brak dostępu do danych dla osób nieupoważnionych. Zapewniane przez szyfrowanie w spoczynku i tranzycie oraz kontrolę dostępu.
- Integralność: brak możliwości modyfikacji danych bez odpowiednich uprawnień. Zapewniane przez funkcje skrótu, sumy kontrolne, MAC
- Dostępność: system działa i jest dostępny dla użytkowników, kiedy tego potrzebują. Zapewniamy redundancją, monitoringiem oraz skalowalnością. Zagrożenie: DDoS, awarie sprzętowe

Jedną z podstaw ochrony danych jest rodzina rozwiązań MAC. Mając wiadomość i klucz, możemy porównać sygnaturę (skrót) wiadomości wygenerowaną przez klienta z obliczonym skrót dla wiadomości w surowej formie z kluczem, aby być pewnym, że korzystamy z tego samego klucza oraz że wiadomość nie została zmodyfikowana.
Implementacją MAC jest HMAC, korzystający z hash (funkcji skrótu). Popularnym algorytmem jest np. SHA-256. Nie można użyć bezpośrednio na samej wiadomości, trzeba użyć na połączeniu wiadomości i klucza. A dokładniej przez pewne matematyczne własności, przez przeprowadzenie na nich paru operacji, w tym XOR. Z HMAC korzysta np. JWT, służący do autentykacji użytkowników w aplikacjach mobilnych.

Broni to przed aktywnym atakiem man in the middle, czyli nie problem braku łączności, a problem kogoś specjalnie zmieniającego komunikaty i liczącego skrót.

Podstawowe pojęcia:

- autentykacja (uwierzytelnienie) - potwierdzenia, że klient jest konkretnym użytkownikiem
- autoryzacja - potwierdzenie, że użytkownik ma dostęp do funkcji
- hash: jednokierunkowana funkcja skrótu. Danych nie da się w żaden sposób "odszyfrować"
- szyfrowanie/deszyfrowanie - wiadomość o tekście jawnym można zaszyfrować tak, aby tylko osoba z tym samym kluczem i algorytmem mogła ją odszyfrować. Broni to przed pasywnym atakiem man in the middle
- metody ochrony komunikacji: TLS/SSL (HTTPS - standard) oraz IPSec (często w VPN, warstwa sieci)
- niezaprzeczalność: nadawca nie może wyprzeć się komunikatu. Logi audytowe

Tylko po pozytywnej autentykacji i autoryzacji można przystąpić do przetwarzania zapytania. Należy pamiętać, że użytkownicy internetu mogą wysłać nam dowolne zapytanie oraz sprawdzić kod JavaScript w przeglądarce (a nawet wyłączyć wykonywanie go).

CORS: z jakiej domeny może być request.

CSRF: używanie tokenów bez wiedzy użytkownika

34. Podstawowe algorytmy kryptograficzne

Dzięki kryptografii jesteśmy w stanie przesyłać dane przez internet, gdzie na każdym kroku ktoś mógłby podsłuchiwać i sprawdzać wrażliwe dane (Man in the middle).

Aby zapewnić bezpieczeństwo projektowi, nie należy stosować security przez obscurity, a używać silnych, sprawdzonych, open source algorytmów z poprawnymi kluczami. Trzeba pamiętać, że aplikacje są narażone na dekompilacje, a JavaScript w przeglądarce jest wolno dostępny i może być debugowany. Dodatkowo, każdy może spreparować dowolne zapytanie HTTP.

Podział na kategorie:

- Symetryczne: algorytmy te wykorzystują ten sam klucz do szyfrowania i deszyfrowania danych. Najczęściej używane, oraz osiągają wysoką wydajność nawet dla dużych bloków danych. Dzielimy je na algorytmy operujące na bitach i blokach (współcześnie częściej używane). Trzeba uważać na to, aby algorytm nie ujawnił struktury danych (np. kontury tekstury) przez szyfrowanie bloków w sposób przewidywalny.
    - DES - przestarzały, przez zbyt małą długość klucza
    - AES - standardowy i wszędzie używany. Nie jest podatny na problemy z rozpoznaniem struktury danych, gdy używany jest w trybie działania CBC (w przeciwieństwie do ECB)
- Asymetryczne: A ma klucz prywatny i publiczny. Publiczny każdy zna, i B wyśle zapytanie do A, używając jego klucza publicznego. Ale klucz publiczny powstał jednostronnie na bazie prywatnego, więc tylko A może odkodować wiadomość. W HTTPS używane do bezpiecznej wymiany symetrycznych kluczy, aby komunikacja przebiegała szybko. Znacznie wolniejszy od alternatyw, dlatego stosuje się go w sposób ograniczony.
    - RSA - dobry algorytm, używany szeroko na całym świecie. Opiera się na trudnej faktoryzacji dużych liczb
    - Diffie-Hellman - bezpieczne uzgodnienie klucza symetrycznego przez niezaufany kanał

- Skrótu: jednostronna generacja skrótu/hashu wiadomości. Nie da się uzyskać wiadomości ze skrótu, zresztą skrót ma określoną liczbę bitów, a wiadomość mogłaby być bardzo duża. Używana do ukrywania haseł w bazie danych czy weryfikacji, czy plik pobrany z internetu nie został zmieniony względem oryginału. Zmiana nawet 1 bitu sprawi, że hash będzie zupełnie inny. Odporny na kolizje, trudno znaleźć 2 teksty o tym samym hashu. W bazach danych nie przechowujemy jawnie hasła, tylko skrót z niego
    - MD5 - przestarzały, podatny na kolizje
    - SHA-2 i SHA-3 - bezpieczne

Rainbow tables: time-memory trade off. Polega na użyciu gotowych tablic skrótów do szybkiego łamania haseł przez hakera. Skuteczne tylko na czyste funkcje hashujące. Nie zadziała, gdy dodamy sól, czyli taki sam tekst do każdego obliczania hasha hasła dla danego użytkownika.

35. Wielowymiarowe modelowanie danych (transakcyjne i analityczne systemy danych, rodzaje wielowymiarowych struktur OLAP)

Systemy danych dzielą się na transakcyjne (OLTP - online transaction processing) i analityczne (OLAP - online analitycal processing)

- Transakcyjne dotyczą systemu danych zorientowanych procesowo (transakcjach), operacjach CRUD, wspierający ciągłe funkcjonowanie systemu. Na przykład, system danych dla sklepu internetowego: danych ciągle przybywa, ciągle są aktualizowane i usuwane. Dane historyczne są przechowywane w ograniczonej formie. Przechowuje dane elementarne. Stosunkowo niewielka ilość danych

- Analityczne dotyczą systemu danych zorientowanych na analizie danych historycznych. Zazwyczaj agregują dane z wielu źródeł, i oferują prosty sposób na złożoną analizę faktów w czasie i innych wymiarach. Składa się z danych historycznych (i technicznie może z bieżących), jest zazwyczaj aktualizowana rzadko, często w zautomatyzowany sposób ETL (Extract, Transform, Load). Dane są zorientowane tematycznie. Przechowywane są dane elementarne i obliczone (sumy, średnie). Bardzo złożone zapytania. Ilość danych jest ogromna

Analityczne struktury danych dzielą dane na 2 osobne grupy: fakty i wymiary. Dla faktu mierzone są miary, które zawsze są numeryczne. Np. faktem może być sprzedać produktu, miarą liczba sprzedanych sztuk i zysk. Wymiary istnieją w relacji do faktów, np. sprzedaż wystąpiła w określonym kwartale danego roku (wymiar Czas), dla danego produktu (Produkt) w określonym państwie (Geografia) itd.
Analityczne struktury danych służą wyłącznie do analizy. Aby przyspieszyć proces, warto denormalizować tabele dla szybszych i prostszych zapytań. Dodatkowo, systemy transakcyjne nie są przystosowane do analizy, więc wykorzystywanie ich do analizy zmniejszyłoby prędkość działania np. sklepu internetowego = utrata pieniędzy.

Są 3 rodzaje modeli analitycznych OLAP:

- ROLAP - relacyjne: typowe dane relacyjne, jak SQL. Mniej miejsca, ale wolniejsze. Tabele faktów połączone z tabelami wymiarów przez klucze obce
- MOLAP - wielowymiarowy: podział na kostki. Szybkie, ale zajmuje więcej miejsca. Zawiera atrybuty wymiarów i faktów na różnym poziomie agregacji
- HOLAP - hybrydowe: łączy podejście relacyjne i wielowymiarowe. Agregacje w formacie wielowymiarowym, dane zarządzane relacyjnie. Wydajne i zajmuje mniej miejsca

Schematy logiczne ROLAP:

- Gwiazdy - ta z tabelą faktów na środku
- Snowflake - normalizacja gwiazdy, czyli wymiary mają wymiary
- Konstelacje - wiele tabel faktów, z własnymi i dzielonymi wymiarami

Do przechowywania danych analitycznych służy hurtownia danych.
Hurtownia to tematycznie zorientowana, zintegrowana (wiele źródeł), chronologiczna i trwała kolekcja danych do wspomagania procesów podejmowania decyzji.

Architektury hurtowni danych:

- Scentralizowana: jedna, fizyczna baza danych
- Federacyjna - wirtualne bazy danych, tak więc wymaga pobrania od nich danych przy kwerendach
- Warstwowa - dane przechodzą przez warstwy hurtowni tematycznych o coraz wyższym stopniu agregacji. Wysoka wydajność

36. Proces ETL

Hurtownia danych to zorientowana tematycznie, zintegrowana (wiele źródeł), chronologiczna i trwała kolekcja danych o charakterze wspomagającym analizę i procesy podejmowania decyzji.

ETL zasila hurtownię danych danymi. Jest to zautomatyzowany proces, złożony z wielu kroków, dzielących się na 3 kategorie: Extract, Transform, Load. Każdy taki krok należy jasno nazwać. Można ustawić, czy kroki będą wykonywać się równolegle, i co, jeśli w poprzednim wystąpi błąd. Można sterować kontrolą: pętle, ify.

- Extract: etap pozyskania danych z heterogenicznych źródeł. Najprostsze to pliki jak Excel, csv, ale zdecydowanie częściej dane pochodzą z zewnętrznych systemów (SAP, ERP), baz danych, czy hurtowni danych na niższym poziomie. Dane należy opisać i zapisać w poprawnym formacie. Po ekstrakcji dane trafiają do pośredniej Staging Area, aby nie obciążać systemów źródłowych
- Transform: często najbardziej pracochłonna część. Dane należy przeczyścić, zintegrować (standaryzować) i zagregować ze sobą dane z różnych źródeł. Zastosowanie logiki biznesowej. Często wykorzystuje się fuzzy matching, aby mr i mr. połączyć w jeden atrybut, ale trzeba uważać, żeby nie połączyło Ireland i Iceland. Transformacje mogą być bardzo różne, ale każdy krok warto jasno opisać. Filtrowanie danych, dzielenie i łączenie kolumn, transpozycje. Dokumentacja w postaci mapy logicznej danych: jak konkretne pole ze źródła ma trafić do konkretnej kolumny
- Load: załadowanie danych do tabeli wymiarów, a potem faktów. Dlatego, że klucze obce muszą istnieć, zanim wstawimy fakt. Ładowanie może być pełne (wszystko od zera) lub przyrostowe (tylko nowe fakty). Z przyrostowym ciężej zadbać, żeby wszystkie zmodyfikowane wiersze zostały zaktualizowane

Alternatywą dla ETL jest proces ELT. Jest to więc zintegrowanie źródeł danych (E) w jednej bazie np. Data Lake (L), i dopiero na niej wykonywanie (T) w bazie docelowej. Dzięki temu wykorzystujemy moc obliczeniową docelowej bazy danych, a nie serwera ETL

38. Metody przetwarzania wiedzy w systemach ekspertowych

System ekspertowy to program komputerowy, naśladujący proces eksperta podejmującego decyzję. Jest wyjaśnialny i zasady są oddzielone od silnika wnioskowania.

Architektura systemów ekspertowych:

- Baza wiedzy: reguły i fakty wprowadzone do systemu
- Silnik wnioskowania - oddzielony od danych i reguł wnioskowania
- Moduł pozyskiwania wiedzy - interfejs do wprowadzania danych przez eksperta
- Moduł wyjaśniający - dzięki temu jest wyjaśnialny, w przeciwieństwie do sieci neuronowych (black box)

Reprezentacje wiedzy:

- Reguły produkcji (if/then): struktura jeżeli warunek, to akcja/fakt
- Logika predykatów - formalny zapis matematyczny
- Ramy - struktury obiektowe (prototypy) posiadające sloty (atrybuty). Paradygmat obiektowy, szczegółowy opis systemów, wbudowana logika i wartości domyślne
- Sieć semantyczna (graf - węzły i krawędzie). Np. węzeł ptak i skrzydła połączone krawędzią ma_część

Dzięki temu, że silnik jest oddzielony od reguł, reguły można dowolnie modyfikować bez potrzeby rekompilacji programu
Można wykorzystywać logikę rozmytą (fuzzy logic). Logika rozmyta to nie jest prawdopodobieństwo, a bardziej procent przynależności do pewnego zbioru. Np. 50% burzy może oznaczać zwykły deszcz. Dzięki temu można używać współczynników pewności zamiast tylko prawda/fałsz

Systemy ekspertowe wykorzystuje się do zaumatyzowanego podejmowania decyzji i wsparcia podejmowania decyzji przez ludzi. Zależne są tylko od logicznych reguł, więc powinny być bardziej obiektywne (w praktyce zależy to oczywiście od postaci reguł).

System ekspertowy może wnioskować do przodu (forward chaining) i do tyłu (backwards chaining)

- Wnioskowanie do przodu to pozyskiwanie nowych faktów z danych. Np. mamy logi systemowe, i generujemy z nich wnioski i akcje.
- Wnioskowanie do tyłu to próba potwierdzenia/odrzucenia hipotezy. Np. mamy hipotezę, czy system jest bezpieczny, i sprawdzamy w tył, czy firewalle są włączone itd.

39. Wnioskowanie w logice niemonotonicznej – zadanie planowania

Logika niemonotoniczna różni się od logiki monotonicznej tym, że w miarę jej wykonywania stan wiedzy może się zmieniać - niektóre fakty stają się fałszywe, a inne prawdziwe. W logice monotonicznej stan wiedzy tylko przyrasta. System jest niekomutatywny, co oznacza, że kolejność wykonywania operacji ma ogromne znaczenie.

Klasycznym zastosowaniem wnioskowania w logice niemonotonicznej jest zadanie planowania (np. trasy logistycznej - VRP czy robota). W miarę wykonywania na nim operacji wg. reguł, stan się zmienia. Takie zadanie składa się z trzech elementów:

- Stan początkowy (stan wszystkich zmiennych, np. robot znajduje się w punkcie A oraz piłka znajduje się w punkcie A)
- Cel / stan końcowy (np. piłka znajduje się w punkcie B)
- Operatory / akcje - zmieniają stan na podstawie dozwolonych operacji, np. robot podnosi piłkę. Każda operacja ma preconditions (np. robot i piłka są w tym samym miejscu), oraz postconditions, czyli jak stan zostanie zmieniony (np. piłka teraz jest trzymana w ręce robota). Dokładniej, jest to lista dopisków (nowe fakty) i lista skreśleń (usuwanie faktów)

Do wykonywania takich zadań w standardowy sposób używa się STRIPS. Są różne algorytmy wykonania problemu opisanego w STRIPS, i niektóre standardy pozwalają np. na dodanie wagi operacjom, dzięki czemu nie tylko planujemy pewne rozwiązanie, ale szukamy też optymalnego.

Zadania planowania można wykonywać na dwa sposoby:

- W przód: zaczynamy w stanie początkowym, i mutujemy go operacjami, aż dojdziemy do stanu końcowego
- W tył: zaczynamy w stanie końcowym, i mutujemy go operacjami, aż dojdziemy do stanu początkowego. Niekiedy prostszy, gdyż opis świata w stanie początkowym może być skomplikowany, a cel zazwyczaj jest prosty - np. do piłki w bramce może doprowadzić tylko kopnięcie.

Należy uważać na pętle (powrót do pewnego poprzedniego stanu), wykrywać i pomijać.
Innym problemem jest problem ramowy, czyli jak zaprezentować to, co nie zmienia się w trakcie akcji. W praktyce zakładamy, że wszystko, czego operacja jawnie nie zmienia, pozostaje takie samo.

Dodatkowo, zazwyczaj występuje założenie zamkniętego świata - czyli wszystko, czego nie wiemy, jest uznawane za fałsz. To ważne założenie w STRIPS