TPI – Teoretyczne podstawy informatyki

NOTATKI DO EGZAMINU 2025

VURTZEL MORAN

SPIS TREŚCI

Teoria Informacji

Teoria informacji – o co chodzi?

- Jest to dział nauki stworzony głównie przez **Claude'a E. Shannona**.
- Bada, **ile informacji** można **przesłać i z jaką dokładnością**, a także **jak dobrze** można te dane **skompresować**.
- Kluczowym pojęciem jest **entropia** – czyli **miara niepewności** danej wiadomości/zdarzenia.

Entropia (H(A))

- H(A) oznacza **poziom niepewności** na temat zdarzenia A.
- Jeśli p(A) = 1 (pewne zdarzenie), to H(A) = 0 (brak niepewności).
- Im rzadsze (mniej prawdopodobne) zdarzenie, tym **wyższa entropia** (bo jest bardziej „zaskakujące").
- Matematycznie:

$$H(A) = \log \frac{1}{p(A)} \quad (\text{zwykle w bitach, więc to } \log_2).$$

Informacja (I(A/B))

- Kiedy dostajemy komunikat **B**, który mówi nam coś o **A**, może to obniżyć (lub nie) niepewność A.
- H(A/B) oznacza **entropię (niepewność) zdarzenia A po** uwzględnieniu informacji B.
- **I(A/B)** to **ile nowej wiedzy** o A wnosi B:

$$I(A/B) = H(A) - H(A/B).$$

- Przykładowe przypadki:
  - **H(A/B) = 0** → komunikat B **całkowicie wyjaśnia** A (wszystko staje się jasne).
  - **H(A/B) = H(A)** → B **nic nie wnosi** (nie zmniejsza niepewności).
  - **H(A/B) > H(A)** → B tylko **miesza** i podnosi niepewność (np. jest sprzeczny z A).

Źródła informacji

1. **Dyskretne (ziarniste)**: mamy skończoną liczbę możliwych wiadomości (np. litery alfabetu, rzuty kostką).
2. **Binarne**: wszystko sprowadzone do „tak/nie" (informacja w bitach).

3. **Ciągłe**: wartości zmieniają się płynnie (np. temperatura, sygnały analogowe).

Redundancja

- **Redundancja** to „nadmiarowość” informacji (pewne powtórzenia czy niepotrzebne szczegóły).
- **Im wyższa redundancja**, tym łatwiej **odtworzyć** informację, nawet gdy jej część zginie (dużo „zapasowych” elementów).
- **Kompresja** (np. ZIP, 7-Zip) polega na **usuwaniu nadmiaru** tak, żeby plik zajął mniej miejsca.
- **R = L – H** (w pewnym uproszczeniu) może opisywać, ile w wiadomości jest tej „nadmiarowości”.

Twierdzenie Shannona

- Mówi o maksymalnej **przepustowości kanału** (C), czyli o **największej szybkości**, z jaką da się przesłać informacje **bez błędów** (lub z bardzo małym błędem).
- Jeśli chcemy wysyłać dane szybciej, niż pozwala na to **C**, pojawią się błędy, których **nie da** się w pełni skorygować.
- Czasem zapisuje się to jako $H \cdot v = C$:
    - **H** – entropia na symbol,
    - **v** – szybkość nadawania (symboli/s),
    - **C** – pojemność kanału (w bitach/s).

Teoria kodowania

---

1. Kodowanie – co to jest?

- **Kodowanie** to proces zamiany wiadomości na ciąg symboli (np. liczb, znaków), który łatwo przesłać przez kanał komunikacyjny.
- Przykład: chcemy przesłać wiadomość „Cześć” przez telefon. Kodowanie zamienia to na sygnały (np. cyfrowe lub dźwiękowe), które można łatwo przesłać i potem zrozumieć.
- Kluczowe etapy:
    - Koder (zamienia wiadomość na kod, np. 101010).
    - Kanał łączności (miejsce, gdzie sygnał jest przesyłany, np. sieć internetowa).
    - Dekoder (zamienia kod z powrotem na wiadomość, np. „Cześć”).

---

2. Czym jest kod?

- **Kod** to zbiór zasad i symboli, które pozwalają zamienić wiadomość na odpowiedni ciąg znaków.
- Składniki kodu:
    - ΩX: wszystkie możliwe wiadomości do zakodowania.
    - ΩU: alfabet kodu, czyli symbole używane (np. 0 i 1 w kodzie binarnym).
    - ΩB: zbiór gotowych kodów (np. 101, 110 itp.).
- **Podstawa kodu**: ile różnych symboli jest w alfabecie (np. binarny = 2 symbole).

---

3. Własności kodów

- **Niesosobliwy**: każdy wyraz kodowy jest unikalny, czyli różni się od innych (np. 101 ≠ 110).
- **Jednoznacznie dekodowalny**: kod można łatwo odczytać bez konfliktów (każdy ciąg ma tylko jedno znaczenie).
- **Jednoznacznie dekodowalny bez opóźnień**: można odczytać kod na bieżąco, bo żaden kod nie jest początkiem innego (np. 101 ≠ 1010 – brak prefiksów).
- **Przecinkowy**: kody są oddzielone czymś, np. pauzą w alfabecie Morse’a (••-• || ••---).

---

4. Średnia prędkość transmisji

- **Prędkość przesyłania symboli**:

$$W = \frac{1}{T_u}$$ gdzie Tu to czas wysłania jednego symbolu (np. 1 ms na 1 bit). Im krótszy czas, tym szybciej.

- **Prędkość przesyłania wiadomości**:

$$V = \frac{W}{L}$$ gdzie L to długość wiadomości. Skracanie kodów (np. używanie krótszych ciągów dla częstych liter) zwiększa szybkość.

---

5. Sprawność kodu

- **Sprawność kodu** mówi, jak efektywny jest nasz kod.

$$\eta = \frac{H}{L}$$ H: entropia (minimalna teoretyczna długość kodu),

L: rzeczywista długość.

  - Jeśli η=1, kod jest idealny.
  - Wyższa sprawność oznacza krótsze kody i mniejszą redundancję (nadmiarowość).

---

6. Nierówność Krafta

- To matematyczna zasada mówiąca, czy kod da się jednoznacznie odczytać bez opóźnień.
- Wzór: $$\sum_{i=1}^{q} q^{-L_i} \leq 1$$ gdzie q to liczba symboli w alfabecie, a Li to długość kodów.
- Jeśli warunek jest spełniony, kod jest jednoznacznie dekodowalny i nie ma prefiksów.

---

7. Kodowanie Shannona

1. Porządkujemy prawdopodobieństwa (największe na górze).
2. Obliczamy długość każdego kodu:

$$L_i = \lceil \log_2 \frac{1}{p(x_i)} \rceil$$ Przypisujemy binarne wartości do każdego kodu, zaczynając od 0.

- Jest to prosty sposób na efektywne kodowanie, ale nie zawsze najkrótszy.

---

8. Kodowanie Fano

1. Układamy prawdopodobieństwa od największego do najmniejszego.
2. Szukamy punktu podziału, gdzie suma $p(x_i)$ powyżej i poniżej jest najbliższa.
3. Górną grupę oznaczamy 0, dolną 1.

- To metoda prostsza od Shannona i często daje krótsze kody.

---

9. Kodowanie Huffmana

1. Porządkujemy prawdopodobieństwa (największe na górze).
2. Łączymy dwie najmniejsze wartości i zapisujemy sumę.
3. Powtarzamy krok 2, aż zostaną dwie wartości.
4. Tworzymy kod: 0 dla górnej gałęzi, 1 dla dolnej.

- Najbardziej efektywne kodowanie, generuje najkrótsze możliwe kody.

---

10. Wnioski z twierdzenia Shannona

1. **Dobrze dobrane kody** zwiększają efektywność transmisji.
2. Maksymalna prędkość przesyłu jest możliwa, gdy używamy optymalnych kodów.
3. Konstrukcja kodów idealnych jest trudna.
4. **Dodanie redundancji** (nadmiaru danych) pomaga wykrywać i korygować błędy.

---

11. Kody zabezpieczające

- To kody, które pozwalają wykrywać lub naprawiać błędy.
- **Typy kodów**:
    - **Rekurencyjne**: oparte na powtarzających się zasadach.
    - **Blokowe**:
        - **Rozdzielne**: zawierają dodatkowe symbole kontrolne (np. do wykrycia błędów).
        - **Nierozdzielne**: brak kontroli.
- Przykłady:
    - **Kod Gray'a**: zmienia tylko jeden bit między sąsiednimi kodami.
    - **Kody paskowe**: używane w sklepach (UPC, EAN).

12. Odległość Hamminga

- **Odległość Hamminga** to liczba różnych bitów między dwoma kodami.
    - Przykład:
      Kod 1: 001101
      Kod 2: 011110
      Odległość Hamminga = 3.
- Określa, jak bardzo kod jest odporny na błędy:
    - Aby **wykryć** m błędów: $d \geq m + 1$
    - Aby **skorygować** m błędów: $d \geq 2m + 1$

Teoria języków i gramatyk formalnych

Wprowadzenie i Definicje

- **Cel**: Gramatyki formalne służą do precyzyjnego opisu struktury języków, co jest kluczowe w informatyce, zwłaszcza przy tworzeniu kompilatorów i interpreterów. Określają, które ciągi znaków (napisy) są „poprawne” w danym języku.

- **Definicja Gramatyki**: Gramatyka formalna jest definiowana jako czwórka (N, T, P, S):
  - N (Symbole Nieterminalne): Reprezentują kategorie składniowe (np. „wyrażenie arytmetyczne”, „instrukcja warunkowa”). Są to symbole, które podlegają dalszemu rozwijaniu. Przykład: <wyrażenie>, <instrukcja>.
  - T (Symbole Terminalne): To podstawowe elementy języka, „cegiełki”, z których budowane są napisy. Są to symbole, których nie da się już dalej rozłożyć. Przykład: a, 1, +, *, if, while.
  - P (Reguły Produkcji): Określają, jak symbole nieterminalne mogą być zastępowane przez ciągi symboli terminalnych i/lub nieterminalnych. Mają postać $A \rightarrow \alpha$, gdzie $A \in N$ i $\alpha \in (N \cup T)^*$. Przykład: <wyrażenie> → <wyrażenie> + <czynnik>.
  - S (Symbol Startu): To wyróżniony symbol nieterminalny, od którego zaczyna się proces generowania napisów języka. Reprezentuje on najwyższy poziom abstrakcji w gramatyce.

Notacje BNF i EBNF

- **BNF (Backus-Naur Form**): To notacja metajęzykowa używana do opisu gramatyk bezkontekstowych. Używa symboli takich jak ::= (oznacza „jest definiowane jako”) i | (oznacza „lub”). Przykład:
  <czynnik> ::= <identyfikator> | <liczba> | (<wyrażenie>)

- **EBNF (Extended Backus-Naur Form)**: To rozszerzona wersja BNF, która dodaje dodatkowe operatory ułatwiające zapis gramatyk, np.:
  - []: Opcjonalność (element może występować 0 lub 1 raz).
  - {}: Powtórzenie (element może występować 0 lub więcej razy).
  - Przykład: <lista_identyfikatorów> ::= <identyfikator> { , <identyfikator> }.

Metody Analizy Składniowej (Sprawdzania Poprawności)

- **Analiza zstępująca (Generacyjna)**: Proces rozpoczyna się od symbolu startowego (S) i poprzez stosowanie reguł produkcji próbuje się wyprowadzić dany napis. Jeśli jest to możliwe, napis jest poprawny.

- **Analiza wstępująca (Redukcyjna):** Proces rozpoczyna się od danego napisu i poprzez odwracanie reguł produkcji próbuje się go zredukować do symbolu startowego (S). Jeśli jest to możliwe, napis jest poprawny.

Przykład

Rozważmy prostą gramatykę dla wyrażeń arytmetycznych:

- N = {<wyrażenie>, <czynnik>}
- T = {a, +, *}
- S = <wyrażenie>
- P = {
    - <wyrażenie> → <wyrażenie> + <czynnik>
    - <wyrażenie> → <czynnik>
    - <czynnik> → a }

Napis „a + a * a” jest poprawny w tej gramatyce.

Hierarchia Chomsky'ego

Hierarchia Chomsky'ego to sposób na uporządkowanie gramatyk (czyli zbiorów reguł, które opisują języki). Dzieli je na cztery typy, od najprostszych do najtrudniejszych.

1. Gramatyki Regularne (Typ 3) – Super Proste

- **Charakterystyka:** Najbardziej ograniczone. Wyobraź sobie, że to jak automat, który ma tylko skończoną liczbę „stanów” i przeskakuje między nimi czytając po kolei litery.
- **Reguły:** Muszą być bardzo proste, np. A → aB (zamień A na a i B) albo A → a (zamień A na a). Czyli po lewej stronie JEDEN symbol, a po prawej co najwyżej JEDEN symbol plus litera.
- **Komentarz:** To tak jakbyś miał instrukcję „idź prosto, a potem skręć w lewo albo w prawo”. Nie możesz się cofać ani robić skomplikowanych manewrów. Dobre do opisywania np. formatów numerów telefonów.

2. Gramatyki Bezkontekstowe (Typ 2) – Trochę Bardziej Skomplikowane

- **Charakterystyka:** Tutaj już jest większa swoboda. Możesz zamieniać symbole niezależnie od tego, co jest obok nich.
- **Reguły:** Po lewej stronie JEDEN symbol, a po prawej COKOLWIEK. Np. A → aBcDe.
- **Komentarz:** To jakbyś miał instrukcję „zamień ten element na coś innego, nie ważne co jest dookoła”. Dobre do opisywania np. struktury kodu programów (np. nawiasy muszą się zgadzać).

3. Gramatyki Kontekstowe (Typ 1) – Jeszcze Trudniejsze

- **Charakterystyka:** Tutaj ważny jest kontekst, czyli to, co jest dookoła symbolu, który chcesz zamienić.

- **Reguły:** Trudniejsze do zapisania, ale ogólnie chodzi o to, że możesz zamienić coś tylko wtedy, gdy jest to otoczone konkretnymi symbolami.
- **Komentarz:** To jakbyś miał instrukcję „jeśli ten element jest otoczony tymi dwoma innymi elementami, to go zamień na coś tam". Rzadziej używane w praktyce, ale teoretycznie ważne.

4. Gramatyki Nieograniczone (Typ 0) – Bez Ograniczeń

- **Charakterystyka:** Tutaj nie ma żadnych ograniczeń. Możesz robić co chcesz.
- **Reguły:** Dowolne kombinacje symboli po obu stronach.
- **Komentarz:** To jakbyś miał instrukcję „rób co chcesz, nie ma żadnych reguł". Bardzo potężne, ale też bardzo trudne do analizy.

Podsumowanie w skrócie:

- **Typ 3 (Regularne):** Najprostsze, mało możliwości, ale łatwe do analizy.
- **Typ 2 (Bezkontekstowe):** Bardziej złożone, dobre do opisu struktury programów.
- **Typ 1 (Kontekstowe):** Uwzględniają kontekst, rzadko używane.
- **Typ 0 (Nieograniczone):** Najbardziej ogólne, ale trudne do analizy.

Parser

Parser to program w kompilatorze, który sprawdza, czy kod jest napisany „poprawnie" pod względem gramatyki języka.

Co robi parser?

- **Dostaje na wejściu:** Kod programu (tekst).
- **Sprawdza:** Czy kod jest zgodny z regułami języka (np. czy nawiasy są dobrze sparowane, czy instrukcje mają poprawną formę).
- **Daje na wyjściu:** Informację, czy kod jest OK, albo listę błędów. Czasem też tworzy specjalne „drzewo", które pokazuje strukturę kodu.

Jak działa (w skrócie)?

- **Automat + Pamięć (Stos):** Parser działa trochę jak automat, który „czyta" kod po kawałku i zapamiętuje pewne rzeczy w specjalnej „pamięci" (stos).
- **Instrukcje (Tablica):** Ma „instrukcje", co robić, gdy zobaczy dany symbol (np. słowo kluczowe, operator). Te instrukcje są zapisane w tablicy.
- **Akcje:** Wykonuje akcje takie jak:
    - **Przesuń (Shift):** „Przeczytaj kolejny element kodu".
    - **Zredukuj (Reduce):** „Rozpoznaj konstrukcję w kodzie" (np. „to jest wyrażenie arytmetyczne").
    - **Zaakceptuj (Accept):** „Kod jest OK!".
    - **Błąd (Error):** „Coś jest źle w kodzie!".

Wyobraź sobie, że parser to taki „policjant gramatyczny” dla kodu. Sprawdza, czy wszystko jest zgodnie z przepisami języka programowania. Jeśli znajdzie błąd, to go zgłasza. Działa trochę jak czytanie ze zrozumieniem – analizuje, co autor miał na myśli (czyli co napisał w kodzie) i czy to ma sens.

Parser = Sprawdzanie, czy kod jest „poprawny gramatycznie”.

Teoria automatów

Automat Skończony

Automat skończony to "maszyna" z ograniczoną pamięcią (stanami), która reaguje na wejścia. *(To taki prosty komputer, który pamięta tylko kilka rzeczy i reaguje na to, co mu podajemy.)*

- **Ma:**
    - **Stany (Q):** Miejsca, w których może być automat. *(To jak różne "tryby" działania, np. "czekam", "działam", "skończyłem".)*
    - **Wejścia (T):** Symbole, które "czyta" automat. *(To np. litery, cyfry, znaki, które dostaje automat do przetworzenia.)*
    - **Instrukcje (P):** Mówią, co zrobić po przeczytaniu danego symbolu. *(To zbiór reguł: "Jak dostanę 'a', to idę do stanu 'X'", "Jak dostanę '1', to idę do stanu 'Y'".)*
    - **Start (q_0):** Stan, w którym automat zaczyna. *(To "punkt startowy" automatu.)*
    - **Koniec (F):** Stany, w których automat "akceptuje" wejście. *(To stany, w których automat mówi: "OK, to wejście jest poprawne".)*
- **Działa:** Dostaje wejście, zmienia stan zgodnie z instrukcjami, akceptuje, jeśli skończy w stanie "koniec". *(Dostaje dane, "przechodzi" po stanach zgodnie z regułami i na końcu decyduje, czy dane są OK.)*
- **Rodzaje:**
    - Kombinacyjny (wyjście zależy tylko od wejścia). *(Bardzo proste: co dostaje na wejściu, to od razu daje na wyjściu.)*
    - Sekwencyjny (wyjście zależy od wejścia *i* stanu). *(Bardziej skomplikowane: to, co daje na wyjściu, zależy od tego, co dostał na wejściu i w jakim "trybie" był wcześniej.)*
- **Wyjście (typy automatów):**
    - Mealy'ego (od wejścia i stanu). *(To, co "wypisuje" automat, zależy od tego, co dostał i w jakim "trybie" jest.)*
    - Moore'a (tylko od stanu). *(To, co "wypisuje" automat, zależy tylko od tego, w jakim "trybie" jest.)*

- **Powiązanie:** Automat skończony = Gramatyka regularna. *(Automaty skończone i gramatyki regularne opisują te same rodzaje "języków".)*

Przykład: Parzysta liczba zer (0 i 1)

- Stany: q_0 (parzysta), q_1 (nieparzysta). *(Dwa "tryby": "widziałem parzystą liczbę zer" i "widziałem nieparzystą liczbę zer".)*
- Wejścia: 0, 1. *(Automat "czyta" zera i jedynki.)*
- Start: q_0. *(Zaczynamy od "widziałem parzystą liczbę zer".)*
- Koniec: q_0. *(Akceptujemy tylko te ciągi, w których jest parzysta liczba zer.)*
- Instrukcje: 0 zmienia stan ($q_0 \leftrightarrow q_1$), 1 nic nie zmienia. *(Jak dostaniemy zero, to zmieniamy "tryb" (z parzystej na nieparzystą i odwrotnie). Jak dostaniemy jedynkę, to nic się nie dzieje.)*

Wejście "11010": $q_0 \rightarrow q_0 \rightarrow q_0 \rightarrow q_1 \rightarrow q_1 \rightarrow q_0$ (AKCEPTACJA). *(Zaczynamy w "parzystej", dostajemy dwie jedynki (nic się nie dzieje), dostajemy zero (zmieniamy na "nieparzystą"), dostajemy jedynkę (nic się nie dzieje), dostajemy zero (wracamy do "parzystej"). Na końcu jesteśmy w "parzystej", więc akceptujemy.)*

Automat ze Stosem (AzS) i Maszyna Turinga (MT)

Automat ze Stosem (AzS): (Q, T, Z, P, z, q, F) *(To "maszyna" z pamięcią w postaci stosu – jak stos talerzy: dodajemy na górę i zdejmujemy z góry.)*

  - Q – zbiór stanów. *(Różne "tryby" działania.)*
  - T – alfabet terminalny. *(Symbole, które "czyta" automat.)*
  - Z – alfabet na stosie. *(Symbole, które może przechowywać na stosie.)*
  - P – funkcja przejść. *(Instrukcje, co robić w zależności od stanu i symbolu ze stosu.)*
  - z – symbol na dnie stosu. *(Symbol na samym spodzie stosu, od którego zaczynamy.)*
  - q – stan początkowy. *(Stan, w którym automat zaczyna.)*
  - F – zbiór stanów końcowych. *(Stany, w których automat akceptuje wejście.)*

- **Automat ze stosem = gramatyka bezkontekstowa.** *(AzS potrafi "rozpoznawać" języki opisane gramatykami bezkontekstowymi, czyli takie, w których reguły nie zależą od kontekstu.)*
- **Maszyna Turinga (MT): (Q, T, Z, R, q, F)** *(To "najpotężniejsza" z tych "maszyn". Ma nieskończoną taśmę, na której może czytać i zapisywać dane.)*
  - Q – zbiór stanów. *(Tak samo jak w AzS.)*
  - T – alfabet terminalny. *(Tak samo jak w AzS.)*

    - Z – alfabet na stosie + znak pusty. *(Podobnie jak w AzS, ale dodatkowo ma "pusty znak", oznaczający brak symbolu.)*
    - R – lista rozkazów. *(Instrukcje, co robić: czytać, pisać, przesuwać się po taśmie, zmieniać stan.)*
    - q – stan początkowy. *(Tak samo jak w AzS.)*
    - F – zbiór stanów końcowych. *(Tak samo jak w AzS.)*
- **Może zrealizować każdy algorytm.** *(MT potrafi wykonać każde obliczenie, które da się opisać algorytmem. Jest modelem "prawdziwego" komputera.)*
- **Automat MT = gramatyka kontekstowa.** *(MT potrafi "rozpoznawać" języki opisane gramatykami kontekstowymi, czyli takie, w których reguły mogą zależeć od kontekstu.)*

Przykłady Działania:

Przykład AzS: Sprawdzanie, czy ciąg ma postać "0ⁿ1ⁿ" (np. 0011, 000111)

- **Stos:** Będziemy odkładać na stos symbol 'X' za każde przeczytane '0'.
- **Działanie:**
    - Czytamy '0' – odkładamy 'X' na stos.
    - Czytamy '1' – zdejmujemy 'X' ze stosu.
    - Na końcu stos musi być pusty, żeby ciąg był zaakceptowany.

Przykład:

- **Wejście: "0011"**
    - Czytamy '0', odkładamy 'X'. Stos: X
    - Czytamy '0', odkładamy 'X'. Stos: XX
    - Czytamy '1', zdejmujemy 'X'. Stos: X
    - Czytamy '1', zdejmujemy 'X'. Stos: pusty – **AKCEPTACJA**
- **Wejście: "001"**
    - Czytamy '0', odkładamy 'X'. Stos: X
    - Czytamy '0', odkładamy 'X'. Stos: XX
    - Czytamy '1', zdejmujemy 'X'. Stos: X
    - Koniec wejścia, stos nie jest pusty – **ODRZUCENIE**

Przykład MT: Zwiększanie liczby binarnej o 1

- **Taśma:** Nieskończona z lewej i prawej strony.

- **Działanie (uproszczone):**
    - Przesuwamy głowicę na koniec liczby (w prawo).
    - Jeśli ostatnia cyfra to 0, zmieniamy ją na 1 i koniec.
    - Jeśli ostatnia cyfra to 1, zmieniamy ją na 0 i przesuwamy się w lewo. Powtarzamy ten krok.

Przykład:

- **Wejście na taśmie: "...0011..."** (gdzie "..." oznacza nieskończone zera po obu stronach)
    - Przesuwamy się na prawy koniec: "...0011..."
    - Ostatnia cyfra to 1, zmieniamy na 0 i przesuwamy w lewo: "...0010..."
    - Ostatnia cyfra to 1, zmieniamy na 0 i przesuwamy w lewo: "...0000..."
    - Ostatnia cyfra to 0, zmieniamy na 1 i koniec: "...0100..."

Modele maszyn cyfrowych

Maszyna Turinga

Maszyna Turinga to teoretyczny model komputera z:

- **Alfabetem:** Zbiór symboli do czytania/pisania. *(Znaki, z których korzysta.)*
- **Stanami:** "Tryby" działania. *(Różne etapy obliczeń.)*
- **Taśmą:** Nieskończona "kartka" do zapisu danych. *(Pamięć.)*
- **Głowicą:** "Ramię" do czytania/pisania po taśmie. *(Procesor.)*
- **Regułami:** Mówią, co robić w danym stanie i dla danego symbolu. *(Program.)*

Deterministyczna: Tylko jedno możliwe działanie dla danego stanu i symbolu. *(Jasne instrukcje.)*

Uniwersalna MT: Może symulować każdą inną MT (najlepszy wynik: 4 symbole i 6 stanów). *(Nawet prosta MT może być bardzo potężna.)*

Maszyna von Neumanna

To model budowy komputera z:

- **Jednostką Centralną (CPU):** "Mózg" (Procesor, Arytmometr, Pamięć RAM).
- **Pamięcią Zewnętrzną:** Magazyn danych (np. dysk).
- **Wejściem/Wyjściem (I/O):** Komunikacja ze światem (np. klawiatura, monitor).

Pamięć RAM: Ma adresy, CPU ma rejestry (A, LR, R, Σ).

PMC

- **Rozkaz:** Binarna instrukcja z częścią operacyjną (co robić) i adresową (na czym działać). *(Instrukcja dla komputera.)*
- **Arytmometr (ALU):** Liczy (sumator), przechowuje wynik (akumulator) i znak (rejestr znaków). *("Kalkulator" procesora.)*
- **Procesor:** Steruje, który rozkaz wykonać (licznik rozkazów/LR, rejestr rozkazów). *("Szef" komputera.)*

Jeszcze krócej z komentarzem:

Rozkaz -> Arytmometr (liczy) -> Procesor (steruje). *(Komputer pobiera instrukcję, wykonuje ją i idzie do następnej.)*

Cykl Pracy PMC

PMC jest maszyną *jednoadresową*. Oznacza to, że w jednym rozkazie można odwołać się tylko do jednego adresu pamięci.

Cykl pracy PMC (dwa etapy):

1. **Faza przygotowawcza:** Pobranie rozkazu z pamięci (adres wskazuje Licznik Rozkazów/LR) i umieszczenie go w Rejestrze Rozkazów. *(Komputer "czyta" instrukcję, co ma zrobić.)*
2. **Faza wykonawcza:** Wykonanie rozkazu. Licznik Rozkazów (LR) jest zwiększany o 1 (żeby wskazywał na następny rozkaz), *chyba że* rozkazem był skok (wtedy LR przyjmuje nową wartość, wskazaną w rozkazie). *(Komputer "robi" to, co kazała mu instrukcja. Potem "patrzy", co ma robić dalej.)*

Rodzaje rozkazów (przykłady):

- **Sterujące:**
    - Skok bezwarunkowy (SK): Zmiana kolejności wykonywania rozkazów na inny adres. *(Idź do innego miejsca w programie.)*
    - Skok warunkowy (SM): Skok tylko wtedy, gdy spełniony jest warunek (np. liczba w akumulatorze jest ujemna). *(Idź do innego miejsca w programie, jeśli coś jest prawdą.)*
    - Skocz i Stop (SS): Zakończenie programu. *(Koniec pracy.)*
- **Arytmetyczne:** Dodawanie (DO), Odejmowanie (OD), Przetwarzanie Binarno-Dziesiętne (PBD). *(Podstawowe operacje matematyczne.)*
- **Przesyłania:** Przesłanie (PA) z akumulatora do pamięci, Umieszczenie w Akumulatorze (UA) z pamięci do akumulatora. *(Przenoszenie danych między "kalkulatorem" a "pamięcią".)*

- **Dla urządzeń zewnętrznych:** Czytanie, Drukowanie, Wyszukiwanie papieru (w starszych drukarkach). *(Obsługa urządzeń, z którymi komunikuje się komputer.)*

Elementy algorytmiki

Algorytm

Algorytm to przepis na rozwiązanie problemu. Mówi krok po kroku, co trzeba zrobić, żeby osiągnąć cel.

Metody zapisu algorytmu (jak zapisać przepis):

- **Język naturalny:** Opis słowny, tak jak w zwykłym tekście. *(Np. "Idź do sklepu, kup chleb.")*
- **Pseudokod:** Uproszczony język programowania, zrozumiały dla ludzi. *(Np. "JEŻELI pada deszcz, TO weź parasol, W PRZECIWNYM RAZIE idź bez parasola.")*
- **Schemat blokowy:** Graficzny sposób przedstawienia algorytmu za pomocą bloków i strzałek. *(Obrazkowy "przepis".)*

Cechy algorytmów (co sprawia, że przepis jest dobry):

- **Ogólność:** Algorytm powinien działać dla różnych danych wejściowych. *(Powinien działać niezależnie od tego, jaki chleb chcemy kupić.)*
- **Jednoznaczność:** Każdy krok algorytmu musi być precyzyjnie określony. *(Nie może być wątpliwości, do którego sklepu iść.)*
- **Skończoność:** Algorytm musi zakończyć działanie po skończonej liczbie kroków. *(Zakupy kiedyś muszą się skończyć.)*

Struktury sterujące (jak kontrolować przebieg przepisu):

- **Sekwencja:** Kolejne kroki wykonywane jeden po drugim. *(Najpierw idziemy do sklepu, potem kupujemy chleb, potem wracamy do domu.)*
- **Selekcja (instrukcja warunkowa):** Wybór jednego z kilku wariantów w zależności od warunku. *(Jeśli jest promocja na bułki, to kupujemy też bułki, w przeciwnym razie kupujemy tylko chleb.)*
- **Pętla:** Powtarzanie pewnych kroków, dopóki spełniony jest warunek. *(Chodzimy po sklepie, dopóki nie znajdziemy chleba.)*
- **Rekurencja:** Algorytm odwołuje się sam do siebie. *(Trudniejsze do wytłumaczenia na przykładzie zakupów, ale ogólnie – funkcja wywołuje samą siebie, np. obliczanie silni liczby.)*

Ważna informacja: Nie ma ograniczeń co do złożoności algorytmu. Może być bardzo prosty lub bardzo skomplikowany.

Poprawność Algorytmu

Mówimy o tym, czy algorytm działa tak, jak powinien.

Rodzaje poprawności:

- **Częściowa:** Jeśli algorytm *w ogóle* się zakończy, to da *dobry* wynik. Ale *nie zawsze* musi się zakończyć. *(Jak przepis na ciasto, który działa, jeśli go dokończymy, ale możemy zapomnieć o jakimś kroku.)*
- **Całkowita:** Algorytm *zawsze* się zakończy i *zawsze* da *dobry* wynik. *(Idealny przepis, który zawsze działa.)*

Metoda Floyda (sposób na sprawdzenie poprawności):

Używa *asercji indukcyjnych* (twierdzeń, które muszą być prawdziwe, żeby algorytm był poprawny). To takie "punkty kontrolne" w algorytmie. Sprawdzamy, czy w tych punktach wszystko gra.

- **Asercja początkowa:** Dane na wejściu są poprawne. *(Mamy dobre składniki na ciasto.)*
- **Asercja końcowa:** Wynik jest taki, jakiego oczekiwaliśmy. *(Ciasto wyszło smaczne i wygląda tak, jak powinno.)*
- **Asercje wewnątrz algorytmu:** Sprawdzamy, czy w trakcie działania algorytmu wszystko idzie zgodnie z planem. *(Sprawdzamy, czy dobrze odmierzyliśmy składniki, czy ciasto się piecze, jak powinno itp.)*

Złożoność Algorytmu

Złożoność mówi nam, ile zasobów (czasu i pamięci) potrzebuje algorytm w zależności od wielkości danych wejściowych (oznaczanych jako N).

Rodzaje złożoności:

- **Czasowa (T):** Ile czasu zajmuje działanie algorytmu. *(Jak długo piecze się ciasto.)*
- **Pamięciowa (M):** Ile pamięci potrzebuje algorytm. *(Ile miejsca zajmują składniki i naczynia.)*
- **Średnia:** Średni czas działania dla typowych danych. *(Zazwyczaj ciasto piecze się tyle i tyle.)*
- **Pesymistyczna:** Czas działania w najgorszym możliwym przypadku. *(Jak bardzo może się przedłużyć pieczenie, np. gdy piekarnik jest słaby.)*

Przykłady złożoności (notacja "O" – duże O):

- **Wyszukiwanie liniowe:** T = O(N). Czas rośnie proporcjonalnie do liczby elementów (N). *(Szukanie igły w stogu siana – im większy stóg, tym dłużej szukamy.)*
- **Wyszukiwanie binarne:** T = O(log N). Czas rośnie bardzo wolno wraz z liczbą elementów. *(Szukanie słowa w słowniku – dzielimy słownik na pół i szukamy w odpowiedniej połowie, potem znowu dzielimy itd. – bardzo szybko znajdujemy szukane słowo.)*
- **k-krotne zagnieżdżenie pętli:** O(N^k)Czas rośnie bardzo szybko wraz z liczbą zagnieżdżeń (k). *(Kilka pętli jedna w drugiej – bardzo czasochłonne.)*

- **MINMAX (iteracyjnie):** T = O(N) = 2N. Czas rośnie proporcjonalnie do N. *(Znajdowanie minimum i maksimum w zbiorze danych.)*
- **MINMAX (rekurencyjnie):** T = O(N) = 3/2 N - 2. Podobnie jak iteracyjnie, liniowa zależność.
- **Sortowanie listy (N elementów):** O(N * log N). Dość efektywne sortowanie.
- **Wieża Hanoi:** O(2^n). Czas rośnie wykładniczo – bardzo szybko wraz z liczbą krążków. *(Przekładanie krążków z jednego drążka na drugi – im więcej krążków, tym kosmicznie więcej ruchów.)*
- **Istotne procesy decyzyjne:** O(N!). Czas rośnie silnie – ekstremalnie szybko. *(Problem komiwojażera – szukanie najkrótszej trasy, która odwiedza wszystkie miasta.)*
- **Arytmetyka Pressburgera** O(2^{2^N})Czas rośnie podwójnie wykładniczo – niewyobrażalnie szybko.

Algorytmy – Dodatkowe Pojęcia

- **Wyszukiwanie heurystyczne:** "Zgadywanie" rozwiązania na podstawie reguł i wskazówek. *("Dobre" zgadywanie, oparte na wiedzy o problemie, a nie losowe.)* To nie gwarantuje znalezienia najlepszego rozwiązania, ale często pozwala znaleźć "dobre" rozwiązanie w rozsądnym czasie, szczególnie dla trudnych problemów. *(Np. szukanie drogi w labiryncie – idziemy w kierunku wyjścia, a nie losowo.)*

- **Ograniczenie górne:** Najgorsza znana złożoność danego problemu. *("Wiemy, że na pewno nie będzie gorzej niż to.")* To złożoność najlepszego dotychczas znalezionego algorytmu dla danego problemu.

- **Ograniczenie dolne:** Najmniejsza możliwa złożoność, jaką *może* mieć algorytm rozwiązujący dany problem. *("Wiemy, że na pewno nie da się tego zrobić szybciej.")* To teoretyczna granica, często trudna do udowodnienia.

- **Luka algorytmiczna:** Różnica między ograniczeniem górnym a dolnym. *(Przestrzeń do ulepszeń.)* Jeśli istnieje luka, oznacza to, że potencjalnie można znaleźć lepszy algorytm.

- **Problem algorytmicznie otwarty:** Istnieje luka algorytmiczna. *(Nie znamy optymalnego rozwiązania.)*

- **Problem algorytmicznie zamknięty:** Ograniczenie górne i dolne są równe. *(Znamy optymalne rozwiązanie – nie da się go już ulepszyć pod względem złożoności.)*

Klasy Problemów Obliczeniowych

- **Problemy P (Wielomianowe):** To problemy "łatwe" do rozwiązania. Istnieje algorytm, który rozwiązuje je w czasie wielomianowym (czyli czas rozwiązania rośnie "rozsądnie" wraz z rozmiarem danych). *(Np. sortowanie listy liczb – da się to zrobić w miarę szybko, nawet dla dużych list.)*
- **Problemy NP (Niedeterministycznie Wielomianowe):** To problemy, dla których *potrafimy szybko sprawdzić*, czy dane rozwiązanie jest poprawne. Ale *niekoniecznie potrafimy szybko znaleźć* to rozwiązanie. *(Np. problem komiwojażera – trudno znaleźć najkrótszą trasę, ale łatwo sprawdzić, czy dana trasa jest krótsza od innej.)*
- **Problemy NPC (NP-zupełne):** To "najtrudniejsze" problemy z klasy NP. Jeśli udałoby się znaleźć szybki algorytm dla *jednego* problemu NPC, to automatycznie mielibyśmy szybkie algorytmy dla *wszystkich* problemów NP. *(To takie "uniwersalne" problemy trudne.)*
- **Teza Churcha-Turinga (CT):** Mówi, że każdy algorytm można zrealizować na Maszynie Turinga (teoretyczny model komputera). W praktyce oznacza to, że wszystkie komputery i języki programowania są równoważne pod względem możliwości obliczeniowych (jeśli mamy nieograniczony czas i pamięć). *(Niezależnie od tego, jakiego komputera użyjemy, możemy rozwiązać te same problemy, choć z różną efektywnością.)*
- **Problemy nierozstrzygalne:** To problemy, dla których *nie istnieje żaden algorytm*, który by je rozwiązywał. *(Niektórych problemów po prostu nie da się rozwiązać za pomocą komputera.)*

  Podsumowanie

  Wiemy, że P jest podzbiorem NP (czyli każdy problem z P jest też problemem z NP). Pytanie, czy P=NP, jest jednym z najważniejszych otwartych problemów informatyki. Jeśli okazałoby się, że P=NP, to wiele problemów, które dziś uważamy za trudne, okazałoby się łatwe do rozwiązania.